

# FieldTalk Modbus Master C++ Library Software manual

Library version 2.13.3



---

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Library Structure . . . . .	1
<b>2</b>	<b>What You should know about Modbus</b>	<b>3</b>
2.1	Some Background . . . . .	3
2.2	Technical Information . . . . .	3
2.2.1	The Protocol Functions . . . . .	3
2.2.2	How Slave Devices are identified . . . . .	4
2.2.3	The Register Model and Data Tables . . . . .	4
2.2.4	Data Encoding . . . . .	5
2.2.5	Register and Discrete Numbering Scheme . . . . .	6
2.2.6	The ASCII Protocol . . . . .	7
2.2.7	The RTU Protocol . . . . .	7
2.2.8	The MODBUS/TCP Protocol . . . . .	7
<b>3</b>	<b>Installation and Source Code Compilation</b>	<b>9</b>
3.1	Windows Platforms: Unpacking and Preparation . . . . .	9
3.2	Linux, UNIX and QNX Platforms: Unpacking and Compiling the Source . . . . .	9
3.3	Specific Platform Notes . . . . .	11
3.3.1	VxWorks . . . . .	11
<b>4</b>	<b>Linking your Applications against the Library</b>	<b>12</b>
4.1	Windows Platforms: Compiling and Linking Applications . . . . .	12
4.2	Linux, UNIX and QNX Platforms: Compiling and Linking Applications . . . . .	13
<b>5</b>	<b>How to integrate the Protocol in your Application</b>	<b>15</b>
5.1	Using Serial Protocols . . . . .	15
5.2	Using MODBUS/TCP Protocol . . . . .	17
<b>6</b>	<b>Design Background</b>	<b>20</b>
<b>7</b>	<b>Module Documentation</b>	<b>21</b>
7.1	Data and Control Functions for all Modbus Protocol Flavours . . . . .	21
7.2	Serial Protocols . . . . .	22
7.2.1	Detailed Description . . . . .	22
7.3	IP based Protocols . . . . .	22

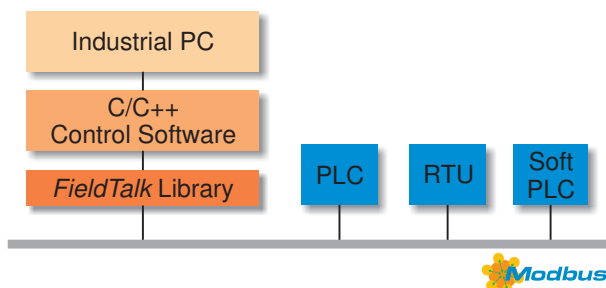
7.3.1	Detailed Description . . . . .	23
7.4	Error Management . . . . .	23
7.4.1	Detailed Description . . . . .	26
7.4.2	Macro Definition Documentation . . . . .	26
7.4.3	Function Documentation . . . . .	31
7.5	Device and Vendor Specific Modbus Functions . . . . .	31
7.5.1	Detailed Description . . . . .	31
7.5.2	Function Documentation . . . . .	31
<b>8</b>	<b>C++ Class Documentation</b>	<b>34</b>
8.1	MbusRtuMasterProtocol Class Reference . . . . .	34
8.1.1	Detailed Description . . . . .	38
8.1.2	Member Enumeration Documentation . . . . .	39
8.1.3	Member Function Documentation . . . . .	39
8.2	MbusAsciiMasterProtocol Class Reference . . . . .	73
8.2.1	Detailed Description . . . . .	78
8.2.2	Member Enumeration Documentation . . . . .	78
8.2.3	Member Function Documentation . . . . .	79
8.3	MbusElamMasterProtocol Class Reference . . . . .	112
8.3.1	Detailed Description . . . . .	117
8.3.2	Member Enumeration Documentation . . . . .	117
8.3.3	Member Function Documentation . . . . .	118
8.4	MbusTcpMasterProtocol Class Reference . . . . .	151
8.4.1	Detailed Description . . . . .	156
8.4.2	Member Function Documentation . . . . .	156
8.5	MbusRtuOverTcpMasterProtocol Class Reference . . . . .	190
8.5.1	Detailed Description . . . . .	195
8.5.2	Member Function Documentation . . . . .	195
8.6	MbusAsciiOverTcpMasterProtocol Class Reference . . . . .	230
8.6.1	Detailed Description . . . . .	235
8.6.2	Member Function Documentation . . . . .	235
8.7	MbusUdpMasterProtocol Class Reference . . . . .	270
8.7.1	Detailed Description . . . . .	275
8.7.2	Member Function Documentation . . . . .	275
<b>9</b>	<b>License</b>	<b>309</b>

<b>10 Support</b>	<b>312</b>
<b>11 Notices</b>	<b>313</b>



# 1 Introduction

The *FieldTalk*™ Modbus® Master C++ Library provides connectivity to Modbus slave compatible devices and applications.



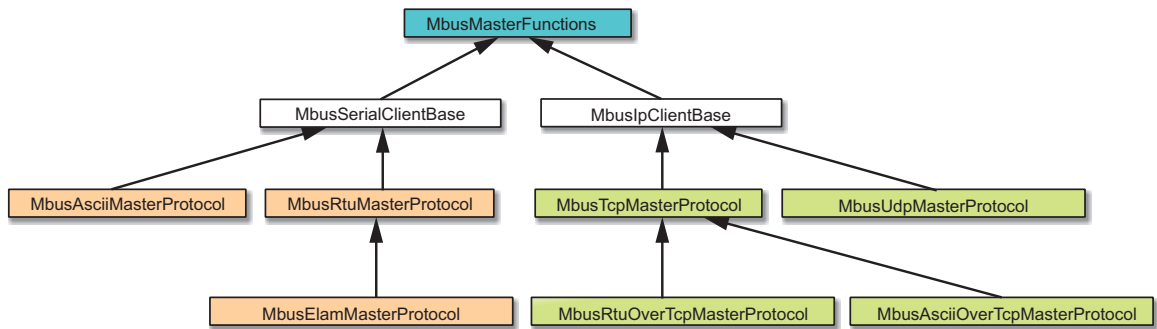
Typical applications are Modbus based Supervisory Control and Data Acquisition Systems (SCADA), Modbus data concentrators, Modbus gateways, User Interfaces and Factory Information Systems (FIS).

Features:

- Robust design suitable for real-time and industrial applications
- Full implementation of Bit Access and 16 Bits Access Function Codes as well as a subset of the most commonly used Diagnostics Function Codes
- Standard Modbus bit and 16-bit integer data types (coils, discretes & registers)
- Support for 32-bit integer, modulo-10000 and float data types, including Daniel/Enron protocol extensions
- Configurable word alignment for 32-bit types (big-endian, little-endian)
- Support of Broadcasting
- Failure and transmission counters
- Transmission and connection time-out supervision
- Detailed transmission and protocol failure reporting using error codes

## 1.1 Library Structure

The library's API is organised into one class for each Modbus protocol flavour and a common base class, which applies to all Modbus protocol flavours. Because the two serial-line protocols Modbus ASCII and Modbus RTU share some common code, an intermediate base class implements the functions specific to the serial protocols.



The base class MbusMasterFunctions contains all protocol unspecific functions, in particular the data and control functions defined by Modbus. All Modbus protocol flavours inherit from this base class.

The class MbusAsciiMasterProtocol implements the Modbus ASCII protocol, the class MbusRtuMasterProtocol implements the Modbus RTU protocol. The class MbusTcpMasterProtocol implements the MODBUS/TCP protocol and the class MbusRtuOverTcpMasterProtocol implements the Encapsulated Modbus RTU master protocol (also known as RTU over TCP or RTU/IP).

In order to use one of the four Modbus protocols, the desired Modbus protocol flavour class has to be instantiated:

```
MbusRtuMasterProtocol mbusProtocol;
```

After a protocol object has been declared and opened, data and control functions can be used:

```
mbusProtocol.writeSingleRegister(slaveId, startRef, 1234);
```



## 2 What You should know about Modbus

### 2.1 Some Background

The Modbus protocol family was originally developed by Schneider Automation Inc. as an industrial network for their Modicon programmable controllers.

Since then the Modbus protocol family has been established as vendor-neutral and open communication protocols, suitable for supervision and control of automation equipment.

### 2.2 Technical Information

Modbus is a master/slave protocol with half-duplex transmission.

One master and up to 247 slave devices can exist per network.

The protocol defines framing and message transfer as well as data and control functions.

The protocol does not define a physical network layer. Modbus works on different physical network layers. The ASCII and RTU protocol operate on RS-232, RS-422 and RS-485 physical networks. The Modbus/TCP protocol operates on all physical network layers supporting TCP/IP. This comprises 10BASE-T and 100BASE-T LANs as well as serial PPP and SLIP network layers.

#### Note

To utilise the multi-drop feature of Modbus, you need a multi-point network like RS-485. In order to access a RS-485 network, you will need a protocol converter which automatically switches between sending and transmitting operation. However some industrial hardware platforms have an embedded RS-485 line driver and support enabling and disabling of the RS-485 transmitter via the RTS signal. FieldTalk supports this RTS driven RS-485 mode.

#### 2.2.1 The Protocol Functions

Modbus defines a set of data and control functions to perform data transfer, slave diagnostic and PLC program download.

FieldTalk implements the most commonly used functions for data transfer as well as some diagnostic functions. The functions to perform PLC program download and other device specific functions are outside the scope of FieldTalk.

All Bit Access and 16 Bits Access Modbus Function Codes have been implemented. In addition the most frequently used Diagnostics Function Codes have been implemented. This rich function set enables a user to solve nearly every Modbus data transfer problem.

The following table lists the available Modbus Function Codes in this library:

Function Code	Current Terminology	Classic Terminology
<b>Bit Access</b>		
1	Read Coils	Read Coil Status

Function Code	Current Terminology	Classic Terminology
2	Read Discrete Inputs	Read Input Status
5	Write Single Coil	Force Single Coil
15 (0F hex)	Write Multiple Coils	Force Multiple Coils
<b>16 Bits Access</b>		
3	Read Multiple Registers	Read Holding Registers
4	Read Input Registers	Read Input Registers
6	Write Single Register	Preset Single Register
16 (10 Hex)	Write Multiple Registers	Preset Multiple Registers
22 (16 hex)	Mask Write Register	Mask Write 4X Register
23 (17 hex)	Read/Write Multiple Registers	Read/Write 4X Registers
<b>Record Access</b>		
20 (14 hex)	Read File Record	Read General Reference
21 (15 hex)	Write File Record	Write General Reference
<b>Diagnostics</b>		
7	Read Exception Status	Read Exception Status
8 subcode 00	Diagnostics - Return Query Data	Diagnostics - Return Query Data
8 subcode 01	Diagnostics - Restart Communications Option	Diagnostics - Restart Communications Option
43 (2B hex) subcode 14 (0E hex)	Read Device Identification	
<b>Vendor Specific</b>		
Advantech	Send/Receive ADAM 5000/6000 ASCII commands	

### 2.2.2 How Slave Devices are identified

A slave device is identified with its unique address identifier. Valid address identifiers supported are 1 to 247. Some library functions also extend the slave ID to 255, please check the individual function's documentation.

Some Modbus functions support broadcasting. With functions supporting broadcasting, a master can send broadcasts to all slave devices of a network by using address identifier 0. Broadcasts are unconfirmed, there is no guarantee of message delivery. Therefore broadcasts should only be used for uncritical data like time synchronisation.

### 2.2.3 The Register Model and Data Tables

The Modbus data functions are based on a register model. A register is the smallest addressable entity with Modbus.

The register model is based on a series of tables which have distinguishing characteristics. The four tables are:

Table	Classic Terminology	Modicon Register Table	Characteristics
Discrete outputs	Coils	0:00000	Single bit, alterable by an application program, read-write
Discrete inputs	Inputs	1:00000	Single bit, provided by an I/O system, read-only
Input registers	Input registers	3:00000	16-bit quantity, provided by an I/O system, read-only
Output registers	Holding registers	4:00000	16-bit quantity, alterable by an application program, read-write

The Modbus protocol defines these areas very loose. The distinction between inputs and outputs and bit-addressable and register-addressable data items does not imply any slave specific behaviour. It is very common that slave devices implement all tables as overlapping memory area.

For each of those tables, the protocol allows a maximum of 65536 data items to be accessed. It is slave dependant, which data items are accessible by a master. Typically a slave implements only a small memory area, for example of 1024 bytes, to be accessed.

## 2.2.4 Data Encoding

Classic Modbus defines only two elementary data types. The discrete type and the register type. A discrete type represents a bit value and is typically used to address output coils and digital inputs of a PLC. A register type represents a 16-bit integer value. Some manufacturers offer a special protocol flavour with the option of a single register representing one 32-bit value.

All Modbus data function are based on the two elementary data types. These elementary data types are transferred in big-endian byte order.

Based on the elementary 16-bit register, any bulk information of any type can be exchanged as long as that information can be represented as a contiguous block of 16-bit registers. The protocol itself does not specify how 32-bit data and bulk data like strings is structured. Data representation depends on the slave's implementation and varies from device to device.

It is very common to transfer 32-bit float values and 32-bit integer values as pairs of two consecutive 16-bit registers in little-endian word order. However some manufacturers like Daniel and Enron implement an enhanced flavour of Modbus which supports 32-bit wide register transfers. FielTalk supports Daniel/Enron 32-bit wide register transfers.

The FieldTalk Modbus Master Library defines functions for the most common tasks like:

- Reading and Writing bit values
- Reading and Writing 16-bit integers

- Reading and Writing 32-bit integers as two consecutive registers
- Reading and Writing 32-bit floats as two consecutive registers
- Reading and Writing 32-bit integers using Daniel/Enron single register transfers
- Reading and Writing 32-bit floats using Daniel/Enron single register transfers
- Configuring the word order and representation for 32-bit values

### 2.2.5 Register and Discrete Numbering Scheme

Modicon PLC registers and discretets are addressed by a memory type and a register number or a discrete number, e.g. 4:00001 would be the first reference of the output registers.

The type offset which selects the Modicon register table must not be passed to the FieldTalk functions. The register table is selected by choosing the corresponding function call as the following table illustrates.

Master Function Call	Modicon Register Table
readCoils(), writeCoil(), forceMultipleCoils()	0:00000
readInputDiscretets	1:00000
readInputRegisters()	3:00000
writeMultipleRegisters(), readMultipleRegisters(), writeSingleRegister(), maskWriteRegister(), readWriteRegisters()	4:00000

Modbus registers are numbered starting from 1. This is different to the conventional programming logic where the first reference is addressed by 0.

Modbus discretets are numbered starting from 1 which addresses the most significant bit in a 16-bit word. This is very different to the conventional programming logic where the first reference is addressed by 0 and the least significant bit is bit 0.

The following table shows the correlation between Discrete Numbers and Bit Numbers:

Modbus Number	Discrete	Bit Number	Modbus Number	Discrete	Bit Number
1		15 (hex 0x8000)	9		7 (hex 0x0080)
2		14 (hex 0x4000)	10		6 (hex 0x0040)
3		13 (hex 0x2000)	11		5 (hex 0x0020)
4		12 (hex 0x1000)	12		4 (hex 0x0010)
5		11 (hex 0x0800)	13		3 (hex 0x0008)
6		10 (hex 0x0400)	14		2 (hex 0x0004)
7		9 (hex 0x0200)	15		1 (hex 0x0002)
8		8 (hex 0x0100)	16		0 (hex 0x0001)

When exchanging register number and discrete number parameters with FieldTalk func-

tions and methods you have to use the Modbus register and discrete numbering scheme. (Internally the functions will deduct 1 from the start register value before transmitting the value to the slave device.)

## 2.2.6 The ASCII Protocol

The ASCII protocol uses an hexadecimal ASCII encoding of data and a 8 bit checksum. The message frames are delimited with a ':' character at the beginning and a carriage return/linefeed sequence at the end.

The ASCII messaging is less efficient and less secure than the RTU messaging and therefore it should only be used to talk to devices which don't support RTU. Another application of the ASCII protocol are communication networks where the RTU messaging is not applicable because characters cannot be transmitted as a continuous stream to the slave device.

The ASCII messaging is state-less. There is no need to open or close connections to a particular slave device or special error recovery procedures.

A transmission failure is indicated by not receiving a reply from the slave. In case of a transmission failure, a master simply repeats the message. A slave which detects a transmission failure will discard the message without sending a reply to the master.

## 2.2.7 The RTU Protocol

The RTU protocol uses binary encoding of data and a 16 bit CRC check for detection of transmission errors. The message frames are delimited by a silent interval of at least 3.5 character transmission times before and after the transmission of the message.

When using RTU protocol it is very important that messages are sent as continuous character stream without gaps. If there is a gap of more than 3.5 character times while receiving the message, a slave device will interpret this as end of frame and discard the bytes received.

The RTU messaging is state-less. There is no need to open or close connections to a particular slave device or special error recovery procedures.

A transmission failure is indicated by not receiving a reply from the slave. In case of a transmission failure, a master simply repeats the message. A slave which detects a transmission failure will discard the message without sending a reply to the master.

## 2.2.8 The MODBUS/TCP Protocol

MODBUS/TCP is a TCP/IP based variant of the Modbus RTU protocol. It covers the use of Modbus messaging in an 'Intranet' or 'Internet' environment.

The MODBUS/TCP protocol uses binary encoding of data and TCP/IP's error detection mechanism for detection of transmission errors.

In contrast to the ASCII and RTU protocols MODBUS/TCP is a connection oriented protocol. It allows concurrent connections to the same slave as well as concurrent connections to multiple slave devices.

In case of a TCP/IP time-out or a protocol failure, a master shall close and re-open the connection and then repeat the message.

## 3 Installation and Source Code Compilation

### 3.1 Windows Platforms: Unpacking and Preparation

1. Download and save the zip archive into a project directory.
2. Uncompress the archive using unzip or another zip tool of your choice:

```
# unzip FT-MBMP-WIN-ALL.2.9.0.zip
```

The archive will create the following directory structure in your project directory:

```
myprj
|
+-- fieldtalk
|
+-- doc
+-- include
+-- lib
+-- src
+-- samples
+-- Visual Studio
```

3. The library is ready to be used.
4. Optionally re-compile from source:

The Windows Editions do come with pre-compiled static libraries for Visual C++ and do not require compilation from source code. However there may be cases where re-compilation is desired.

To re-compile, open the Visual Studio\mbusmaster\_win.vcxproj solution file with Visual Studio 2019. The project file will also work with older Visual Studio versions. The library will be compiled into one of the following sub-directories of your project directory:

Platform	Library Directory
Windows 64-bit, Visual Studio 2019	lib\win\x64\Release
Windows 32-bit, Visual Studio 2019	lib\win\Win32\Release
Windows CE, Visual Studio 2005 or 2008	lib\wce\\$(Platform)\Release

### 3.2 Linux, UNIX and QNX Platforms: Unpacking and Compiling the Source

1. Download and save the zipped tarball into your project directory.
2. Uncompress the zipped tarball using gzip:

```
# gunzip FT-MBMP-??-ALL.2.9.0.tar.gz
```

3. Untar the tarball

```
# tar xf FT-MBMP-??-ALL.2.9.0.tar
```

The tarball will create the following directory structure in your project directory:

```
myprj
|
+-- fieldtalk
|
+-- doc
+-- include
+-- src
+-- samples
```

4. Compile the library from the source code. Enter the FieldTalk src directory and run make:

```
# cd fieldtalk/src
# make
```

Note: Previous versions used a shell script to build, this version now uses the make utility to build.

The make file tries to detect your host platform and executes the compiler and linker commands targeting the host platform. The compiler and linker configurations are contained in the platform make files in the makefiles folder.

To cross-compile, pass the basename of the cross-compilation make file as parameter:

```
# make arm-linux-gnueabihf
# make qnx6-ppc1e
```

The cross compilation make files can be found in the makefiles directory and you can add more by copying and editing the supplied files to match your toolchain setting.

5. The library will be compiled into one of the following platform specific sub-directories:

Platform	Library Directory
Linux	lib/linux
QNX 6	lib/qnx6
Solaris	lib/solaris
HP-UX	lib/hpux
IBM AIX	lib/aix

Your directory structure looks now like:

```
myprj
|
+-- fieldtalk
|
+-- doc
+-- src
+-- include
+-- samples
+-- lib
|
+-- {platform} (exact name depends on platform)
```

6. The library is ready to be used.



## 3.3 Specific Platform Notes

### 3.3.1 VxWorks

There is no make file or script supplied for VxWorks because VxWorks applications and libraries are best compiled from the Tornado IDE.

To compile and link your applications against the FieldTalk library, add all the \*.c and \*.cpp files supplied in the src, src/hmlib/common, src/hmlib/posix4 and src/hmlib/vxworks to your project.

## 4 Linking your Applications against the Library

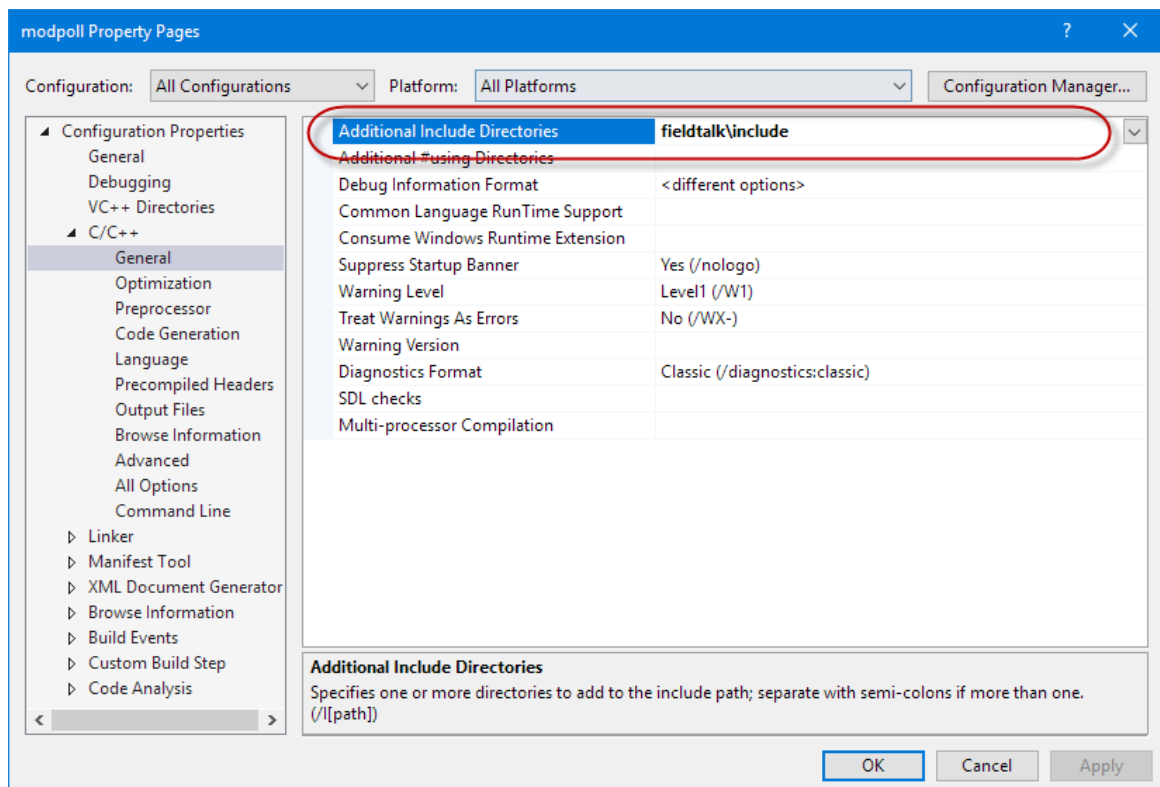
### 4.1 Windows Platforms: Compiling and Linking Applications

Let's assume the following project directory structure:

```
myprj
|
+-- myapp.cpp
+-- fieldtalk
   |-- include
   +-- lib
      |
      +-- win
         |
         +-- Win32
            |
            +-- Release
```

Add the library's include directory to the compiler's include path.

Visual Studio Example:



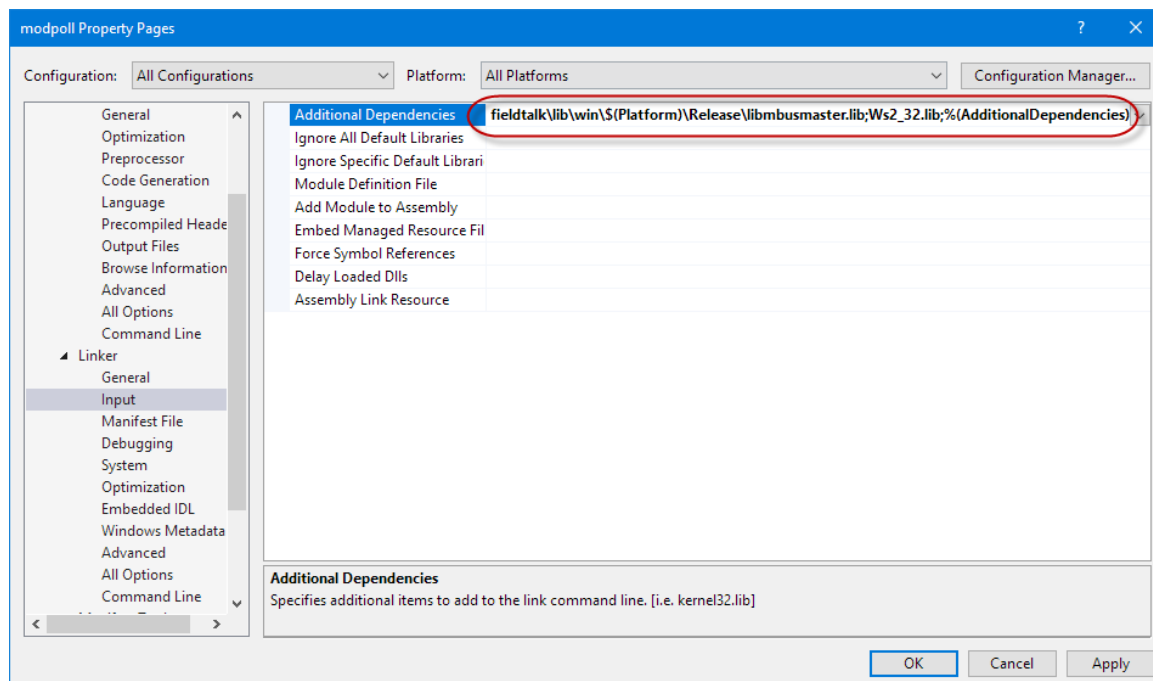
Visual C++ command line Example:

```
cl -I fieldtalk/include -c myapp.cpp
```

Add the file name of the library to the dependency list passed to the linker. Make sure the library chosen matches your CPU architecture (32-bit vs 64-bit). Use the Win32 library

for 32-bit code and the x64 library for 64-bit code. Within Visual Studio you can use the macro to automatically select the correct library architecture. You also must pass the Winsock2 standard library `Ws2_32.lib` as additional dependency to the linker.

Visual Studio Example:



Visual C++ command line Example:

```
cl -Fe myapp myapp.obj fieldtalk/lib/win/Win32/Release/libbusmaster.lib Ws2_32.lib
```

## 4.2 Linux, UNIX and QNX Platforms: Compiling and Linking Applications

Let's assume the following project directory structure:

```
myprj
|
+-- myapp.cpp
+-- fieldtalk
|
|   +-- include
|   +-- lib
|   |
|   +-- linux      (exact name depends on your platform)
```

Add the library's include directory to the compiler's include path.

Example:

```
c++ -I fieldtalk/include -c myapp.cpp
```

Add the file name of the library to the file list passed to the linker.

Example:

```
c++ -o myapp myapp.o fieldtalk/lib/linux/libmbusmaster.a
```

# 5 How to integrate the Protocol in your Application

## 5.1 Using Serial Protocols

Let's assume we want to talk to a Modbus slave device with slave address 1.

The registers for reading are in the reference range 4:00100 to 4:00119 and the registers for writing are in the range 4:00200 to 4:00219. The discretes for reading are in the reference range 0:00010 to 0:00019 and the discretes for writing are in the range 0:00020 to 0:00029.

1. Include the library header files

```
#include "MbusRtuMasterProtocol.hpp"
```

2. Device data profile definition

Define the data sets which reflects the slave's data profile by type and size:

```
short readRegSet[20];
short writeRegSet[20];
int readBitSet[20];
int writeBitSet[20];
```

If you are using floats instead of 16-bit shorts define:

```
float readFloatSet[10];
float writeFloatSet[10];
```

Note that because a float occupies two 16-bit registers the array size is half the size it would be for 16-bit shorts!

If you are using 32-bit ints instead of 16-bit shorts define:

```
long readLongSet[10];
long writeLongSet[10];
```

Note that because a long occupies two 16-bit registers the array size is half the size it would be for 16-bit shorts!

3. Declare and instantiate a protocol object

```
MbusRtuMasterProtocol mbusProtocol;
```

4. Open the protocol

```
int result;

result = mbusProtocol.openProtocol(portName,
                                   19200L, // Baudrate
                                   8,      // Databits
                                   1,      // Stopbits
```

```
                2);    // Even parity
if (result != FTALK_SUCCESS)
{
    fprintf(stderr, "Error opening protocol: %s!\n",
              getBusProtocolErrorText(result));
    exit(EXIT_FAILURE);
}
```

## 5. Perform the data transfer functions

- To read register values:

```
mbusProtocol.readMultipleRegisters(1, 100, readRegSet,
                                   sizeof(readRegSet) / sizeof(short));
```

- To write a single register value:

```
mbusProtocol.writeSingleRegister(1, 200, 1234);
```

- To write multiple register values:

```
mbusProtocol.writeMultipleRegisters(1, 200, writeRegSet,
                                     sizeof(writeRegSet) / sizeof(short));
```

- To read discrete values:

```
mbusProtocol.readCoils(1, 10, readBitSet, sizeof(readBitSet) / sizeof(int));
```

- To write a single discrete value:

```
mbusProtocol.writeCoil(1, 20, 1);
```

- To write multiple discrete values:

```
mbusProtocol.forceMultipleCoils(1, 20, sizeof(writeBitSet) / sizeof(int));
```

- To read float values:

```
mbusProtocol.readMultipleFloats(1, 100, readFloatSet,
                                 sizeof(readFloatSet) / sizeof(float));
```

- To read long integer values:

```
mbusProtocol.readMultipleLongInts(1, 100, readLongSet,
                                   sizeof(readLongSet) / sizeof(long));
```

## 6. Close the protocol port if not needed any more

```
mbusProtocol.closeProtocol();
```

## 7. Error Handling

Serial protocol errors like slave device failures, transmission failures, checksum errors and time-outs return an error code. The following code snippet can handle and report these errors:

```

int result;

result = mbusProtocol.readMultipleRegisters(1, 100, dataSetArray, 10);
if (result != FTALK_SUCCESS)
{
    fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));
    // Stop for fatal errors
    if (!(result & FTALK_BUS_PROTOCOL_ERROR_CLASS))
        return;
}

```

An automatic retry mechanism is available and can be enabled with `mbusProtocol.setRetryCnt(3)` before opening the protocol port.

## 5.2 Using MODBUS/TCP Protocol

Let's assume we want to talk to a Modbus slave device with unit address 1 and IP address 10.0.0.11.

The registers for reading are in the reference range 4:00100 to 4:00119 and the registers for writing are in the range 4:00200 to 4:00219. The discrettes for reading are in the reference range 0:00010 to 0:00019 and the discrettes for writing are in the range 0:00020 to 0:00029.

### 1. Include the library header files

```
#include "MbusTcpMasterProtocol.hpp"
```

### 2. Device data profile definition

Define the data sets which reflects the slave's data profile by type and size:

```

short readRegSet[20];
short writeRegSet[20];
int readBitSet[10];
int writeBitSet[10];

```

If you are using floats instead of 16-bit shorts define:

```

float readFloatSet[10];
float writeFloatSet[10];

```

Note that because a float occupies two 16-bit registers the array size is half the size it would be for 16-bit shorts!

If you are using 32-bit ints instead of 16-bit shorts define:

```

long readLongSet[10];
long writeLongSet[10];

```

Note that because a long occupies two 16-bit registers the array size is half the size it would be for 16-bit shorts!

### 3. Declare and instantiate a protocol object

```
MbusTcpMasterProtocol mbusProtocol;
```

#### 4. Open the protocol

```
mbusProtocol.openProtocol("10.0.0.11");
```

#### 5. Perform the data transfer functions

- To read register values:

```
mbusProtocol.readMultipleRegisters(1, 100, readRegSet,  
                                   sizeof(readRegSet) / sizeof(short));
```

- To write a single register value:

```
mbusProtocol.writeSingleRegister(1, 200, 1234);
```

- To write multiple register values:

```
mbusProtocol.writeMultipleRegisters(1, 200, writeRegSet,  
                                     sizeof(writeRegSet) / sizeof(short));
```

- To read discrete values:

```
mbusProtocol.readCoils(1, 10, readBitSet, sizeof(readBitSet) / sizeof(int));
```

- To write a single discrete value:

```
mbusProtocol.writeCoil(1, 20, 1);
```

- To write multiple discrete values:

```
mbusProtocol.forceMultipleCoils(1, 20, writeBitSet,  
                                 sizeof(writeBitSet) / sizeof(int));
```

- To read float values:

```
mbusProtocol.readMultipleFloats(1, 100, readFloatSet,  
                                  sizeof(readFloatSet) / sizeof(float));
```

- To read long integer values:

```
mbusProtocol.readMultipleLongInts(1, 100, readLongSet,  
                                   sizeof(readLongSet) / sizeof(long));
```

#### 6. Close the connection if not needed any more

```
mbusProtocol.closeProtocol();
```

#### 7. Error Handling

TCP/IP protocol errors like slave failures, TCP/IP connection failures and time-outs return an error code. The following code snippet can handle these errors:



```
int result;

result = mbusProtocol.readMultipleRegisters(1, 100, dataSetArray, 10);
if (result != FTALK_SUCCESS)
{
    fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));
    // Stop for fatal errors
    if (!(result & FTALK_BUS_PROTOCOL_ERROR_CLASS))
        return;
}
```

If the method returns `FTALK_CONNECTION_WAS_CLOSED`, it signals that the the TCP/IP connection was lost or closed by the remote end. Before using further data and control functions the connection has to be re-opened succesfully.

## 6 Design Background

FieldTalk is based on a programming language neutral but object oriented design model.

This design approach enables us to offer the protocol stack for the languages C++, C#, Visual Basic .NET, Java and Object Pascal while maintaining similar functionality.

The C++ editions of the protocol stack have also been designed to support multiple operating system and compiler platforms, including real-time operating systems. In order to support this multi-platform approach, the C++ editions are built around a lightweight OS abstraction layer called *HMLIB*.

During the course of implementation, the usability in automation, control and other industrial environments was always kept in mind.

## 7 Module Documentation

### 7.1 Data and Control Functions for all Modbus Protocol Flavours

This Modbus protocol library implements the most commonly used data functions as well as some control functions. The functions to perform PLC program download and other device specific functions are outside the scope of this library.

All Bit Access and 16 Bits Access Modbus Function Codes have been implemented. In addition the most frequently used Diagnostics Function Codes have been implemented. This rich function set enables a user to solve nearly every Modbus data transfer problem.

The following table lists the supported Modbus function codes:

Function Code	Current Terminology	Classic Terminology
<b>Bit Access</b>		
1	Read Coils	Read Coil Status
2	Read Discrete Inputs	Read Input Status
5	Write Single Coil	Force Single Coil
15 (0F hex)	Write Multiple Coils	Force Multiple Coils
<b>16 Bits Access</b>		
3	Read Multiple Registers	Read Holding Registers
4	Read Input Registers	Read Input Registers
6	Write Single Register	Preset Single Register
16 (10 Hex)	Write Multiple Registers	Preset Multiple Registers
22 (16 hex)	Mask Write Register	Mask Write 4X Register
23 (17 hex)	Read/Write Multiple Registers	Read/Write 4X Registers
<b>Record Access</b>		
20 (14 hex)	Read File Record	Read General Reference
21 (15 hex)	Write File Record	Write General Reference
<b>Diagnostics</b>		
7	Read Exception Status	Read Exception Status
8 subcode 00	Diagnostics - Return Query Data	Diagnostics - Return Query Data
8 subcode 01	Diagnostics - Restart Communications Option	Diagnostics - Restart Communications Option
43 (2B hex) subcode 14 (0E hex)	Read Device Identification	
<b>Vendor Specific</b>		
Advantech	Send/Receive ADAM 5000/6000 ASCII commands	

### Remarks

When passing register numbers and discrete numbers to FieldTalk library functions you have to use the the Modbus register and discrete numbering scheme. See Register and Discrete Numbering Scheme. (Internally the functions will deduct 1 from the start register value before transmitting the value to the slave device.)

Using multiple instances of a MbusMaster... class enables concurrent protocol transfer on different communication channels (e.g. multiple TCP/IP sessions in separate threads or multiple COM ports in separate threads).

## 7.2 Serial Protocols

### Classes

- class MbusRtuMasterProtocol  
*Modbus RTU Master Protocol class.*
- class MbusAsciiMasterProtocol  
*Modbus ASCII Master Protocol class.*
- class MbusElamMasterProtocol  
*Extended Lufkin Automation Modbus Master Protocol.*

### 7.2.1 Detailed Description

The two classic serial Modbus protocol flavours RTU and ASCII are implemented in the MbusRtuMasterProtocol and MbusAsciiMasterProtocol classes.

The popular vendor specific Extended Lufkin Automation Modbus Master (ELAM) protocol is also available as class MbusElamMasterProtocol. This proprietary Modbus extension allows addressing of up to 2295 slave units and the retrieval of up to 2500 registers for Modbus functions 3 and 4.

These classes provide functions to open and to close serial port as well as data and control functions which can be used at any time after a protocol has been opened. The data and control functions are organized into different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Modbus Protocol Flavours.

Using multiple instances of a MbusRtuMasterProtocol or MbusAsciiMasterProtocol class enables concurrent protocol transfers on multiple COM ports (they should be executed in separate threads).

See sections The RTU Protocol and The ASCII Protocol for some background information about the two serial Modbus protocols.

See section Using Serial Protocols for an example how to use the MbusRtuMasterProtocol class.

## 7.3 IP based Protocols

## Classes

- class MbusTcpMasterProtocol  
*MODBUS/TCP Master Protocol class.*
- class MbusRtuOverTcpMasterProtocol  
*Encapsulated Modbus RTU Master Protocol class.*
- class MbusElamOverTcpMasterProtocol  
*Encapsulated Extended Lufkin Automation Modbus Master Protocol.*
- class MbusAsciiOverTcpMasterProtocol  
*Modbus ASCII over TCP Master Protocol class.*
- class MbusUdpMasterProtocol  
*MODBUS/UDP Master Protocol class.*

### 7.3.1 Detailed Description

The library provides several flavours of IP based Modbus protocols.

The MODBUS/TCP master protocol is implemented in the class MbusTcpMasterProtocol and is the only IP based protocol officially specified by the Modbus organisation.

In addition to MODBUS/TCP, the library offers implementations of both serial protocols RTU and ASCII transported over TCP streams. These are implemented in the classes MbusRtuOverTcpMasterProtocol and MbusAsciiOverTcpMasterProtocol.

Also an implementation for MODBUS/TCP packets transported via UDP is available in form of the class MbusUdpMasterProtocol.

All classes provide functions to establish and to close a TCP/IP connection to the slave as well as data and control functions which can be used after a connection to a slave device has been established successfully. For a more detailed description of the data and control functions see section Data and Control Functions for all Modbus Protocol Flavours.

Using multiple instances of a MbusTcpMasterProtocol class enables concurrent protocol transfers using multiple TCP/IP sessions. They should be executed in separate threads.

See section The MODBUS/TCP Protocol for some background information about MODBUS/TCP.

See section Using MODBUS/TCP Protocol for an example how to use the MbusTcpMasterProtocol class.

## 7.4 Error Management

### Macros

- #define FTALK\_SUCCESS 0  
*Operation was successful.*
- #define FTALK\_ILLEGAL\_ARGUMENT\_ERROR 1  
*Illegal argument error.*
- #define FTALK\_ILLEGAL\_STATE\_ERROR 2

*Illegal state error.*

- #define FTALK\_EVALUATION\_EXPIRED 3  
*Evaluation expired.*
- #define FTALK\_NO\_DATA\_TABLE\_ERROR 4  
*No data table configured.*
- #define FTALK\_ILLEGAL\_SLAVE\_ADDRESS 5  
*Slave address 0 illegal for serial protocols.*
- #define FTALK\_INSUFFICIENT\_BUFFER 6  
*Size of response buffer insufficient.*

## Functions

- const TCHAR \* getBusProtocolErrorText (int errCode)  
*Translates a numeric error code into a description string.*

## Fatal I/O Errors

Errors of this class signal a problem in conjunction with the I/O system.

If errors of this class occur, the operation must be aborted and the protocol closed.

- #define FTALK\_IO\_ERROR\_CLASS 64  
*I/O error class.*
- #define FTALK\_IO\_ERROR 65  
*I/O error.*
- #define FTALK\_OPEN\_ERR 66  
*Port or socket open error.*
- #define FTALK\_PORT\_ALREADY\_OPEN 67  
*Serial port already open.*
- #define FTALK\_TCPIP\_CONNECT\_ERR 68  
*TCP/IP connection error.*
- #define FTALK\_CONNECTION\_WAS\_CLOSED 69  
*Remote peer closed TCP/IP connection.*
- #define FTALK\_SOCKET\_LIB\_ERROR 70  
*Socket library error.*
- #define FTALK\_PORT\_ALREADY\_BOUND 71  
*TCP port already bound.*
- #define FTALK\_LISTEN\_FAILED 72  
*Listen failed.*
- #define FTALK\_FILEDES\_EXCEEDED 73  
*File descriptors exceeded.*
- #define FTALK\_PORT\_NO\_ACCESS 74  
*No permission to access serial port or TCP port.*
- #define FTALK\_PORT\_NOT\_AVAIL 75

---

*TCP port not available.*

- `#define FTALK_LINE_BUSY_ERROR 76`

*Serial line busy/noisy.*

## Communication Errors

Errors of this class indicate either communication faults or Modbus exceptions reported by the slave device.

- `#define FTALK_BUS_PROTOCOL_ERROR_CLASS 128`

*Fieldbus protocol error class.*

- `#define FTALK_CHECKSUM_ERROR 129`

*Checksum error.*

- `#define FTALK_INVALID_FRAME_ERROR 130`

*Invalid frame error.*

- `#define FTALK_INVALID_REPLY_ERROR 131`

*Invalid reply error.*

- `#define FTALK_REPLY_TIMEOUT_ERROR 132`

*Reply time-out.*

- `#define FTALK_SEND_TIMEOUT_ERROR 133`

*Send time-out.*

- `#define FTALK_INVALID_MBAP_ID 134`

*Invalid MPAB identifier.*

- `#define FTALK_LINE_ERROR 135`

*Serial line error.*

- `#define FTALK_OVERRUN_ERROR 136`

*Serial buffer overrun.*

- `#define FTALK_MBUS_EXCEPTION_RESPONSE 160`

*Modbus exception response.*

- `#define FTALK_MBUS_ILLEGAL_FUNCTION_RESPONSE 161`

*Illegal Function exception response.*

- `#define FTALK_MBUS_ILLEGAL_ADDRESS_RESPONSE 162`

*Illegal Data Address exception response.*

- `#define FTALK_MBUS_ILLEGAL_VALUE_RESPONSE 163`

*Illegal Data Value exception response.*

- `#define FTALK_MBUS_SLAVE_FAILURE_RESPONSE 164`

*Slave Device Failure exception response.*

- `#define FTALK_MBUS_GW_PATH_UNAVAIL_RESPONSE 170`

*Gateway Path Unavailable exception response.*

- `#define FTALK_MBUS_GW_TARGET_FAIL_RESPONSE 171`

*Gateway Target Device Failed exception response.*

## 7.4.1 Detailed Description

This module documents all the exception classes, error and return codes reported by the various library functions.

## 7.4.2 Macro Definition Documentation

**FTALK\_SUCCESS** `#define FTALK_SUCCESS 0`

Operation was successful.

This return codes indicates no error.

**FTALK\_ILLEGAL\_ARGUMENT\_ERROR** `#define FTALK_ILLEGAL_ARGUMENT_ERROR 1`

Illegal argument error.

A parameter passed to the function returning this error code is invalid or out of range.

**FTALK\_ILLEGAL\_STATE\_ERROR** `#define FTALK_ILLEGAL_STATE_ERROR 2`

Illegal state error.

The function is called in a wrong state. This return code is returned by all functions if the protocol has not been opened successfully yet.

**FTALK\_EVALUATION\_EXPIRED** `#define FTALK_EVALUATION_EXPIRED 3`

Evaluation expired.

This version of the library is a function limited evaluation version and has now expired.

**FTALK\_NO\_DATA\_TABLE\_ERROR** `#define FTALK_NO_DATA_TABLE_ERROR 4`

No data table configured.

The slave has been started without adding a data table. A data table must be added by either calling `addDataTable` or passing it as a constructor argument.

**FTALK\_ILLEGAL\_SLAVE\_ADDRESS** `#define FTALK_ILLEGAL_SLAVE_ADDRESS 5`

Slave address 0 illegal for serial protocols.

A slave address or unit ID of 0 is used as broadcast address for ASCII and RTU protocol and therefor illegal.

**FTALK\_INSUFFICIENT\_BUFFER** `#define FTALK_INSUFFICIENT_BUFFER 6`

Size of response buffer insufficient.



The received response was larger than the buffer provided. This error only applies to function codes with a variable response length.

**FTALK\_IO\_ERROR\_CLASS** #define FTALK\_IO\_ERROR\_CLASS 64

I/O error class.

Errors of this class signal a problem in conjunction with the I/O system.

**FTALK\_IO\_ERROR** #define FTALK\_IO\_ERROR 65

I/O error.

The underlying I/O system reported an error.

**FTALK\_OPEN\_ERR** #define FTALK\_OPEN\_ERR 66

Port or socket open error.

The TCP/IP socket or the serial port could not be opened. In case of a serial port it indicates that the serial port does not exist on the system.

**FTALK\_PORT\_ALREADY\_OPEN** #define FTALK\_PORT\_ALREADY\_OPEN 67

Serial port already open.

The serial port defined for the open operation is already opened by another application.

**FTALK\_TCPIP\_CONNECT\_ERR** #define FTALK\_TCPIP\_CONNECT\_ERR 68

TCP/IP connection error.

Signals that the TCP/IP connection could not be established. Typically this error occurs when a host does not exist on the network or the IP address or host name is wrong. The remote host must also listen on the appropriate port.

**FTALK\_CONNECTION\_WAS\_CLOSED** #define FTALK\_CONNECTION\_WAS\_CLOSED 69

Remote peer closed TCP/IP connection.

Signals that the TCP/IP connection was closed by the remote peer or is broken. One reason can be that the Slave Address does not match and was rejected.

**FTALK\_SOCKET\_LIB\_ERROR** #define FTALK\_SOCKET\_LIB\_ERROR 70

Socket library error.

The TCP/IP socket library (e.g. WINSOCK) could not be loaded or the DLL is missing or not installed.

**FTALK\_PORT\_ALREADY\_BOUND** #define FTALK\_PORT\_ALREADY\_BOUND 71

TCP port already bound.

Indicates that the specified TCP port cannot be bound. The port might already be taken by another application or hasn't been released yet by the TCP/IP stack for re-use.

**FTALK\_LISTEN\_FAILED** #define FTALK\_LISTEN\_FAILED 72

Listen failed.

The listen operation on the specified TCP port failed..

**FTALK\_FILEDES\_EXCEEDED** #define FTALK\_FILEDES\_EXCEEDED 73

File descriptors exceeded.

Maximum number of usable file descriptors exceeded.

**FTALK\_PORT\_NO\_ACCESS** #define FTALK\_PORT\_NO\_ACCESS 74

No permission to access serial port or TCP port.

You don't have permission to access the serial port or TCP port. Run the program as root. If the error is related to a serial port, change the access privilege. If it is related to TCP/IP use TCP port number which is outside the IPPORT\_RESERVED range.

**FTALK\_PORT\_NOT\_AVAIL** #define FTALK\_PORT\_NOT\_AVAIL 75

TCP port not available.

The specified TCP port is not available on this machine.

**FTALK\_LINE\_BUSY\_ERROR** #define FTALK\_LINE\_BUSY\_ERROR 76

Serial line busy/noisy.

The serial line is receiving characters or noise despite being in a state where there should be no traffic.

**FTALK\_BUS\_PROTOCOL\_ERROR\_CLASS** #define FTALK\_BUS\_PROTOCOL\_ERROR\_CLASS 128

Fieldbus protocol error class.

Signals that a fieldbus protocol related error has occurred. This class is the general class of errors produced by failed or interrupted data transfer functions. It is also produced when receiving invalid frames or exception responses.

**FTALK\_CHECKSUM\_ERROR** #define FTALK\_CHECKSUM\_ERROR 129

Checksum error.

Signals that the checksum of a received frame is invalid. A poor data link typically causes this error.

**FTALK\_INVALID\_FRAME\_ERROR** #define FTALK\_INVALID\_FRAME\_ERROR 130

Invalid frame error.

Signals that a received frame does not correspond either by structure or content to the specification or does not match a previously sent query frame. A poor data link typically causes this error.

**FTALK\_INVALID\_REPLY\_ERROR** #define FTALK\_INVALID\_REPLY\_ERROR 131

Invalid reply error.

Signals that a received reply does not correspond to the specification.

**FTALK\_REPLY\_TIMEOUT\_ERROR** #define FTALK\_REPLY\_TIMEOUT\_ERROR 132

Reply time-out.

Signals that a fieldbus data transfer timed out. This can occur if the slave device does not reply in time or does not reply at all. A wrong unit address will also cause this error. In some occasions this exception is also produced if the characters received don't constitute a complete frame.

**FTALK\_SEND\_TIMEOUT\_ERROR** #define FTALK\_SEND\_TIMEOUT\_ERROR 133

Send time-out.

Signals that a fieldbus data send timed out. This can only occur if the handshake lines are not properly set.

**FTALK\_INVALID\_MBAP\_ID** #define FTALK\_INVALID\_MBAP\_ID 134

Invalid MPAB identifier.

Either the protocol or transaction identifier in the reply is incorrect. A slave device must return the identifiers received from the master.

**FTALK\_LINE\_ERROR** #define FTALK\_LINE\_ERROR 135

Serial line error.

A receive error was detected by the UART. This can be a parity error, character overrun or frame error.

**FTALK\_OVERRUN\_ERROR** #define FTALK\_OVERRUN\_ERROR 136

Serial buffer overrun.

More characters have been received than expected.

**FTALK\_MBUS\_EXCEPTION\_RESPONSE** #define FTALK\_MBUS\_EXCEPTION\_RESPONSE 160

Modbus exception response.

Signals that a Modbus exception response was received. Exception responses are sent by a slave device instead of a normal response message if it received the query message correctly but cannot handle the query. This error usually occurs if a master queried an invalid or non-existing data address or if the master used a Modbus function, which is not supported by the slave device.

**FTALK\_MBUS\_ILLEGAL\_FUNCTION\_RESPONSE** #define FTALK\_MBUS\_ILLEGAL\_FUNCTION\_RESPONSE 161

Illegal Function exception response.

Signals that an Illegal Function exception response (code 01) was received. This exception response is sent by a slave device instead of a normal response message if a master sent a Modbus function, which is not supported by the slave device.

**FTALK\_MBUS\_ILLEGAL\_ADDRESS\_RESPONSE** #define FTALK\_MBUS\_ILLEGAL\_ADDRESS\_RESPONSE 162

Illegal Data Address exception response.

Signals that an Illegal Data Address exception response (code 02) was received. This exception response is sent by a slave device instead of a normal response message if a master queried an invalid or non-existing data address.

**FTALK\_MBUS\_ILLEGAL\_VALUE\_RESPONSE** #define FTALK\_MBUS\_ILLEGAL\_VALUE\_RESPONSE 163

Illegal Data Value exception response.

Signals that a Illegal Value exception response was (code 03) received. This exception response is sent by a slave device instead of a normal response message if a master sent a data value, which is not an allowable value for the slave device.

**FTALK\_MBUS\_SLAVE\_FAILURE\_RESPONSE** #define FTALK\_MBUS\_SLAVE\_FAILURE\_RESPONSE 164

Slave Device Failure exception response.

Signals that a Slave Device Failure exception response (code 04) was received. This exception response is sent by a slave device instead of a normal response message if an unrecoverable error occurred while processing the requested action. This response is also sent if the request would generate a response whose size exceeds the allowable data size.

**FTALK\_MBUS\_GW\_PATH\_UNAVAIL\_RESPONSE** #define FTALK\_MBUS\_GW\_PATH\_UNAVAIL\_RESPONSE 170

Gateway Path Unavailable exception response.

Signals that a Gateway Path Unavailable exception response (code 0A) was received. This exception is typically sent by gateways if the gateway was unable to establish a connection with the target device.

**FTALK\_MBUS\_GW\_TARGET\_FAIL\_RESPONSE** #define FTALK\_MBUS\_GW\_TARGET\_FAIL\_RESPONSE 171

Gateway Target Device Failed exception response.

Signals that a Gateway Target Device failed exception response (code 0B) was received. This exception is typically sent by gateways if the gateway was unable to receive a response from the target device. Usually means that the device is not present on the network.

### 7.4.3 Function Documentation

```
getBusProtocolErrorText() const TCHAR* getBusProtocolErrorText (
    int errCode )
```

Translates a numeric error code into a description string.

#### Parameters

<i>errCode</i>	FieldTalk error code
----------------	----------------------

#### Returns

Error text string

## 7.5 Device and Vendor Specific Modbus Functions

### Custom Function Codes

- int customFunction (int slaveAddr, int functionCode, void \*requestData, size\_t requestLen, void \*responseData, size\_t \*responseLenPtr)
  - User Defined Function Code*
  - This method can be used to implement User Defined Function Codes.*

### Advantec ADAM 5000/6000 Series Commands

- int adamSendReceiveAsciiCmd (const char \*const commandSz, char \*responseSz)
  - Send/Receive ADAM 5000/6000 ASCII command.*

#### 7.5.1 Detailed Description

Some device specific or vendor specific functions and enhancements are supported.

#### 7.5.2 Function Documentation

```
customFunction() int customFunction (
    int slaveAddr,
    int functionCode,
    void * requestData,
    size_t requestLen,
    void * responseData,
    size_t * responseLenPtr )
```

### User Defined Function Code

This method can be used to implement User Defined Function Codes.

The caller has only to pass the user data to this function. The assembly of the Modbus frame (the so called ADU) including checksums, slave address and function code and subsequently the transmission, is taken care of by this method.

The modbus specification reserves function codes 65-72 and 100-110 for user defined functions.

### Note

Modbus functions usually have an implied response length and therefore the number of bytes expected to be received is known at the time when sending the request. In case of a custom Modbus function with an open or unknown response length, this function can not be used.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>functionCode</i>	Custom function code to be used for Modbus transaction (1-127)
<i>requestData</i>	Pointer to data sent as request (not including slave address or function code)
<i>requestLen</i>	Length of request data structure (0-252)
<i>responseData</i>	Pointer to data structure which holds response data
<i>responseLenPtr</i>	Length of response data buffer (0-252). This value will be adjusted should the actual response data be less. A FTALK_INSUFFICIENT_BUFFER error code will be returned in case the allocated buffer size was insufficient.

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
adamSendReceiveAsciiCmd() int adamSendReceiveAsciiCmd (
    const char *const commandSz,
    char * responseSz )
```

Send/Receive ADAM 5000/6000 ASCII command.

Sends an ADAM 5000/6000 ASCII command to the device and receives the reply as ASCII string. (e.g. "\$01M" to retrieve the module name)

#### Parameters

<i>commandSz</i>	Buffer which holds command string. Must not be longer than 255 characters.
<i>responseSz</i>	Buffer which holds response string. Must be a buffer of 256 bytes. A possible trailing CR is removed.

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

References FTALK\_CONNECTION\_WAS\_CLOSED, FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_IO\_ERROR, and FTALK\_SUCCESS.

## 8 C++ Class Documentation

### 8.1 MbusRtuMasterProtocol Class Reference

Modbus RTU Master Protocol class.

#### Public Types

- enum { SER\_DATABITS\_7 = 7, SER\_DATABITS\_8 = 8 }
- enum { SER\_STOPBITS\_1 = 1, SER\_STOPBITS\_2 = 2 }
- enum { SER\_PARITY\_NONE = 0, SER\_PARITY\_EVEN = 2, SER\_PARITY\_ODD = 1 }

#### Public Member Functions

- MbusRtuMasterProtocol ()  
*Constructs a MbusRtuMasterProtocol object and initialises its data.*
- virtual int openProtocol (const char \*const portName, long baudRate, int dataBits, int stopBits, int parity)  
*Opens a Modbus RTU protocol and the associated serial port with specific port parameters.*
- virtual void closeProtocol ()  
*Closes an open protocol including any associated communication resources (com ports or sockets).*
- virtual int isOpen ()
- virtual int enableRs485Mode (int rtsDelay)  
*Enables RS485 mode.*

#### Bit Access

Table 0:00000 (Coils) and Table 1:0000 (Input Status)

- int readCoils (int slaveAddr, int startRef, int bitArr[], int refCnt)  
*Modbus function 1, Read Coil Status/Read Coils.*
- int readInputDiscretes (int slaveAddr, int startRef, int bitArr[], int refCnt)  
*Modbus function 2, Read Inputs Status/Read Input Discretes.*
- int writeCoil (int slaveAddr, int bitAddr, int bitVal)  
*Modbus function 5, Force Single Coil/Write Coil.*
- int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[], int refCnt)  
*Modbus function 15 (0F Hex), Force Multiple Coils.*



## 16-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleRegisters (int slaveAddr, int startRef, short regArr[], int refCnt)`  
*Modbus function 3, Read Holding Registers/Read Multiple Registers.*
- `int readInputRegisters (int slaveAddr, int startRef, short regArr[], int refCnt)`  
*Modbus function 4, Read Input Registers.*
- `int writeSingleRegister (int slaveAddr, int regAddr, short regVal)`  
*Modbus function 6, Preset Single Register/Write Single Register.*
- `int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[], int refCnt)`  
*Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.*
- `int maskWriteRegister (int slaveAddr, int regAddr, short andMask, short orMask)`  
*Modbus function 22 (16 Hex), Mask Write Register.*
- `int readWriteRegisters (int slaveAddr, int readRef, short readArr[], int readCnt, int writeRef, const short writeArr[], int writeCnt)`  
*Modbus function 23 (17 Hex), Read/Write Registers.*

## 32-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleLongInts (int slaveAddr, int startRef, int int32Arr[], int refCnt)`  
*Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.*
- `int readInputLongInts (int slaveAddr, int startRef, int int32Arr[], int refCnt)`  
*Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.*
- `int writeMultipleLongInts (int slaveAddr, int startRef, const int int32Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.*
- `int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[], int refCnt)`  
*Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.*
- `int readInputFloats (int slaveAddr, int startRef, float float32Arr[], int refCnt)`  
*Modbus function 4 for 32-bit float data types, Read Input Registers as float data.*
- `int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.*
- `int readMultipleMod10000 (int slaveAddr, int startRef, int int32Arr[], int refCnt)`  
*Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/↔ Read Multiple Registers as modulo-10000 long int data.*
- `int readInputMod10000 (int slaveAddr, int startRef, int int32Arr[], int refCnt)`  
*Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.*
- `int writeMultipleMod10000 (int slaveAddr, int startRef, const int int32Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.*

## 64-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readHoldingRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[ ], int refCnt)`  
*Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int readInputRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[ ], int refCnt)`  
*Modbus function 4 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int writeMultipleRegistersI64 (int slaveAddr, int startRef, const int64_t int64Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.*
- `int readHoldingRegistersF64 (int slaveAddr, int startRef, double double64Arr[ ], int refCnt)`  
*Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.*
- `int readInputRegistersF64 (int slaveAddr, int startRef, double double64Arr[ ], int refCnt)`  
*Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.*
- `int writeMultipleRegistersF64 (int slaveAddr, int startRef, const double double64Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.*

## File Record Access

- `int readFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, short recordArr[ ], int recordCnt)`  
*Modbus function 20, Read File Record.*
- `int readFileRecord (int slaveAddr, FileSubRequest subReqArr[ ], int subReqCnt)`  
*Modbus function 20, Read File Record.*
- `int writeFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, const short recordArr[ ], int recordCnt)`  
*Modbus function 21, Write File Record.*
- `int writeFileRecord (int slaveAddr, const FileSubRequest subReqArr[ ], int subReqCnt)`  
*Modbus function 21, Write File Record.*

## Diagnostics

- `int readExceptionStatus (int slaveAddr, unsigned char *statusBytePtr)`  
*Modbus function 7, Read Exception Status.*
- `int returnQueryData (int slaveAddr, const unsigned char queryArr[ ], unsigned char echoArr[ ], int len)`  
*Modbus function code 8, sub-function 00, Return Query Data.*
- `int restartCommunicationsOption (int slaveAddr, int clearEventLog)`  
*Modbus function code 8, sub-function 01, Restart Communications Option.*
- `int readDeviceIdentification (int slaveAddr, int accessType, int objId, DeviceIdObjectList *objListPtr)`  
*Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.*

---

## Custom Function Codes

- `int customFunction (int slaveAddr, int functionCode, void *requestData, size_t requestLen, void *responseData, size_t *responseLenPtr)`

*User Defined Function Code*

*This method can be used to implement User Defined Function Codes.*

## Protocol Configuration

- `int setTimeout (int timeOut)`  
*Configures time-out.*
- `int getTimeout ()`  
*Returns the time-out value.*
- `int setPollDelay (int pollDelay)`  
*Configures poll delay.*
- `int getPollDelay ()`  
*Returns the poll delay time.*
- `int setRetryCnt (int retryCnt)`  
*Configures the automatic retry setting.*
- `int getRetryCnt ()`  
*Returns the automatic retry count.*

## Transmission Statistic Functions

- `long getTotalCounter ()`  
*Returns how often a message transfer has been executed.*
- `void resetTotalCounter ()`  
*Resets total message transfer counter.*
- `long getSuccessCounter ()`  
*Returns how often a message transfer was successful.*
- `void resetSuccessCounter ()`  
*Resets successful message transfer counter.*

## Slave Configuration

- `void configureBigEndianInts ()`  
*Configures 32-bit int data type functions to do a word swap.*
- `int configureBigEndianInts (int slaveAddr)`  
*Enables int data type functions to do a word swap on a per slave basis.*
- `void configureLittleEndianInts ()`  
*Configures 32-bit int data type functions not to do a word swap.*
- `int configureLittleEndianInts (int slaveAddr)`  
*Disables word swapping for int data type functions on a per slave basis.*

- void configureleeeeFloats ()  
*Configures float data type functions not to do a word swap.*
- int configureleeeeFloats (int slaveAddr)  
*Disables float data type functions to do a word swap on a per slave basis.*
- void configureSwappedFloats ()  
*Configures float data type functions to do a word swap.*
- int configureSwappedFloats (int slaveAddr)  
*Enables float data type functions to do a word swap on a per slave basis.*
- void configureStandard32BitMode ()  
*Configures all slaves for Standard 32-bit Mode.*
- int configureStandard32BitMode (int slaveAddr)  
*Configures a slave for Standard 32-bit Register Mode.*
- void configureEnron32BitMode ()  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- int configureEnron32BitMode (int slaveAddr)  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- void configureCountFromOne ()  
*Configures the reference counting scheme to start with one for all slaves.*
- int configureCountFromOne (int slaveAddr)  
*Configures a slave's reference counting scheme to start with one.*
- void configureCountFromZero ()  
*Configures the reference counting scheme to start with zero for all slaves.*
- int configureCountFromZero (int slaveAddr)  
*Configures a slave's reference counting scheme to start with zero.*

## Utility Functions

- static const TCHAR \* getPackageVersion ()  
*Returns the library version number.*

### 8.1.1 Detailed Description

Modbus RTU Master Protocol class.

This class realizes the Modbus RTU master protocol. It provides functions to open and to close serial port as well as data and control functions which can be used at any time after the protocol has been opened. The data and control functions are organized into different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Modbus Protocol Flavours.

It is possible to instantiate multiple instances of this class for establishing multiple connections on different serial ports (They should be executed in separate threads).

#### See also

Data and Control Functions for all Modbus Protocol Flavours, Serial Protocols  
MbusRtuOverTcpMasterProtocol

## 8.1.2 Member Enumeration Documentation

**anonymous enum** anonymous enum [inherited]

Enumerator

SER_DATABITS↔ _7	7 data bits
SER_DATABITS↔ _8	8 data bits

**anonymous enum** anonymous enum [inherited]

Enumerator

SER_STOPBITS↔ _1	1 stop bit
SER_STOPBITS↔ _2	2 stop bits

**anonymous enum** anonymous enum [inherited]

Enumerator

SER_PARITY_NONE	No parity.
SER_PARITY_EVEN	Even parity.
SER_PARITY_ODD	Odd parity.

## 8.1.3 Member Function Documentation

**openProtocol()** int openProtocol (  
     const char \*const *portName*,  
     long *baudRate*,  
     int *dataBits*,

```
int stopBits,  
int parity ) [virtual]
```

Opens a Modbus RTU protocol and the associated serial port with specific port parameters.

This function opens the serial port. After a port has been opened, data and control functions can be used.

#### Note

The default time-out for the data transfer is 1000 ms.

The default poll delay is 0 ms.

Automatic retries are switched off (retry count is 0).

The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.

#### Parameters

<i>portName</i>	Serial port identifier (e.g. "COM1", "/dev/ser1" or "/dev/ttyS0")
<i>baudRate</i>	The port baudRate in bps (typically 1200 - 115200, maximum value depends on UART hardware)
<i>dataBits</i>	Must be SER_DATABITS_8 for RTU
<i>stopBits</i>	SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits
<i>parity</i>	SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_EVEN: even parity

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

Reimplemented from MbusSerialClientBase.

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, MbusSerialClientBase::openProtocol(), and MbusSerialClientBase::SER\_DATABITS\_8.

```
isOpen() int isOpen ( ) [virtual], [inherited]
```

**Deprecated** Returns whether the protocol is open or not.

This simply returns the state of an internal flag whether openProtocol() has been called or not.

#### Note

This function does not report the state of the underlying transport network, in particular it does not report whether the TCP connection is still in 'established' state or in fully 'closed' state.

Return values

<i>true</i>	= open
<i>false</i>	= closed

Implements MbusMasterFunctions.

```
enableRs485Mode() int enableRs485Mode (
    int rtsDelay ) [virtual], [inherited]
```

Enables RS485 mode.

In RS485 mode the RTS signal can be used to enable and disable the transmitter of a RS232/RS485 converter. The RTS signal is asserted before sending data. It is cleared after the transmit buffer has been emptied and in addition the specified delay time has elapsed. The delay time is necessary because even the transmit buffer is already empty, the UART's FIFO will still contain unsent characters.

#### Warning

The use of RTS controlled RS232/RS485 converters should be avoided if possible. It is difficult to determine the exact time when to switch off the transmitter with non real-time operating systems like Windows and Linux. If it is switched off too early characters might still sit in the FIFO or the transmit register of the UART and these characters will be lost. Hence the slave will not recognize the message. On the other hand if it is switched off too late then the slave's message is corrupted and the master will not recognize the message.

#### Remarks

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

#### Note

A protocol must be closed in order to configure it.

#### Parameters

<i>rtsDelay</i>	Delay time in ms (Range: 0 - 100000) which applies after the transmit buffer is empty. 0 disables this mode.
-----------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
----------------------	---------

## Return values

<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ OR	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusSerialClientBase::isOpen()`.

```
readCoils() int readCoils (  
    int slaveAddr,  
    int startRef,  
    int bitArr[],  
    int refCnt ) [inherited]
```

Modbus function 1, Read Coil Status/Read Coils.

Reads the contents of the discrete outputs (coils, 0:00000 table).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

**Returns**

*FTALK\_SUCCESS* on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputDiscretes() int readInputDiscretes (  
    int slaveAddr,  
    int startRef,  
    int bitArr[],  
    int refCnt ) [inherited]
```

Modbus function 2, Read Inputs Status/Read Input Discretes.

Reads the contents of the discrete inputs (input status, 1:00000 table).



### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeCoil() int writeCoil (
    int slaveAddr,
    int bitAddr,
    int bitVal ) [inherited]
```

Modbus function 5, Force Single Coil/Write Coil.

Sets a single discrete output variable (coil, 0:00000 table) to either ON or OFF.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>bitAddr</i>	Coil address (Range: 1 - 65536)
<i>bitVal</i>	true sets, false clears discrete output variable

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
forceMultipleCoils() int forceMultipleCoils (  
    int slaveAddr,  
    int startRef,  
    const int bitArr[],  
    int refCnt ) [inherited]
```

Modbus function 15 (0F Hex), Force Multiple Coils.

Writes binary values into a sequence of discrete outputs (coils, 0:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which contains the data to be sent
<i>refCnt</i>	Number of coils to be written (Range: 1-1968)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleRegisters() int readMultipleRegisters (  
    int slaveAddr,  
    int startRef,  
    short regArr[],  
    int refCnt ) [inherited]
```

Modbus function 3, Read Holding Registers/Read Multiple Registers.

Reads the contents of the output registers (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputRegisters() int readInputRegisters (
    int slaveAddr,
    int startRef,
    short regArr[],
    int refCnt ) [inherited]
```

Modbus function 4, Read Input Registers.

Read the contents of the input registers (3:00000 table).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read.
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeSingleRegister() int writeSingleRegister (
    int slaveAddr,
    int regAddr,
    short regVal ) [inherited]
```

Modbus function 6, Preset Single Register/Write Single Register.

Writes a value into a single output register (holding register, 4:00000 reference).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>regVal</i>	Data to be sent

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
writeMultipleRegisters() int writeMultipleRegisters (  
    int slaveAddr,  
    int startRef,  
    const short regArr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.

Writes values into a sequence of output registers (holding registers, 4:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer with the data to be sent.
<i>refCnt</i>	Number of registers to be written (Range: 1-123)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

```
maskWriteRegister() int maskWriteRegister (
    int slaveAddr,
    int regAddr,
    short andMask,
    short orMask ) [inherited]
```

Modbus function 22 (16 Hex), Mask Write Register.

Masks bits according to an AND & an OR mask into a single output register (holding register, 4:00000 reference). Masking is done as follows: result = (currentVal AND andMask) OR (orMask AND (NOT andMask))

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>andMask</i>	Mask to be applied as a logic AND to the register
<i>orMask</i>	Mask to be applied as a logic OR to the register

#### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readWriteRegisters() int readWriteRegisters (
    int slaveAddr,
    int readRef,
    short readArr[],
    int readCnt,
    int writeRef,
    const short writeArr[],
    int writeCnt ) [inherited]
```

Modbus function 23 (17 Hex), Read/Write Registers.

Combines reading and writing of the output registers in one transaction (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>readRef</i>	Start register for reading (Range: 1 - 65536)
<i>readArr</i>	Buffer which will contain the data read
<i>readCnt</i>	Number of registers to be read (Range: 1-125)
<i>writeRef</i>	Start register for writing (Range: 1 - 65536)

**Parameters**

<i>writeArr</i>	Buffer with data to be sent
<i>writeCnt</i>	Number of registers to be written (Range: 1-121)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleLongInts() int readMultipleLongInts (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into 32-bit long int values.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputLongInts() int readInputLongInts (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) into 32-bit long int values.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleLongInts() int writeMultipleLongInts (
    int slaveAddr,
    int startRef,
    const int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/↔ Write Multiple Registers with long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleFloats() int readMultipleFloats (  
    int slaveAddr,  
    int startRef,  
    float float32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into float values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)



### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readInputFloats() int readInputFloats (
    int slaveAddr,
    int startRef,
    float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit float data types, Read Input Registers as float data.

Reads the contents of pairs of consecutive input registers (3:00000 table) into float values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleFloats() int writeMultipleFloats (
    int slaveAddr,
    int startRef,
    const float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

Writes float values into pairs of output registers (holding registers, 4:00000 table).

#### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleMod10000() int readMultipleMod10000 (  
    int slaveAddr,  
    int startRef,  
    int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value into 32-bit int values and performs number format conversion.

#### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readInputMod10000() int readInputMod10000 (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) representing a modulo-10000 long int value into 32-bit long int values and performs number format conversion.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleMod10000() int writeMultipleMod10000 (  
    int slaveAddr,  
    int startRef,  
    const int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value and performs number format conversion.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values values to be sent (Range: 1-61)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersI64() int readHoldingRegistersI64 (  
    int slaveAddr,  
    int startRef,  
    int64_t int64Arr[],  
    int refCnt ) [inherited]
```

Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.

Reads the contents of quartets of consecutive holding registers (4:00000 table) into 64-bit integer values.

#### Remarks

Depending on the 32-bit Mode setting, an `int64_t` will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
readInputRegistersI64() int readInputRegistersI64 (
    int slaveAddr,
    int startRef,
    int64_t int64Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 64-bit integer data, Read Holding Registers as `int64_t` type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit integer values.

#### Remarks

Depending on the 32-bit Mode setting, an `int64_t` will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleRegistersI64() int writeMultipleRegistersI64 (  
    int slaveAddr,  
    int startRef,  
    const int64_t int64Arr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.

Writes for 64-bit integer values into quartets of holding registers (4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersF64() int readHoldingRegistersF64 (  
    int slaveAddr,
```

```

int startRef,
double double64Arr[],
int refCnt ) [inherited]

```

Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (4:00000 table) into 64-bit floating point values.

### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```

readInputRegistersF64() int readInputRegistersF64 (
    int slaveAddr,
    int startRef,
    double double64Arr[],
    int refCnt ) [inherited]

```

Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit floating point values.

### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleRegistersF64() int writeMultipleRegistersF64 (
    int slaveAddr,
    int startRef,
    const double double64Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.

Writes for 64-bit floating point values into quartets of holding registers (4:00000 table).

**Remarks**

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.



**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readFileRecord() [1/2] int readFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 20, Read File Record.

Performs a file record read for a single subrequest.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to read from, 0-based numbering.
<i>recordArr</i>	Buffer which will be filled with the data read
<i>recordCnt</i>	Number of records to read (Range 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readFileRecord() [2/2] int readFileRecord (
    int slaveAddr,
    FileSubRequest subReqArr[],
    int subReqCnt ) [inherited]
```

Modbus function 20, Read File Record.

Performs a file record read for a list of subrequests.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to read (Range 1-35)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK↔\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
writeFileRecord() [1/2] int writeFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    const short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a single subrequest.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to write to, 0-based numbering.
<i>recordArr</i>	Buffer with the data to be sent.
<i>recordCnt</i>	Number of records to write (Range 1-122)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeFileRecord() [2/2] int writeFileRecord (
    int slaveAddr,
    const FileSubRequest subReqArr[],
    int subReqCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a list of subrequests.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to write (Range 1-27)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK↔\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
readExceptionStatus() int readExceptionStatus (
    int slaveAddr,
    unsigned char * statusBytePtr ) [inherited]
```

Modbus function 7, Read Exception Status.

Reads the eight exception status coils within the slave device.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>statusBytePtr</i>	Slave status byte. The meaning of this status byte is slave specific and varies from device to device.

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

```
returnQueryData() int returnQueryData (  
    int slaveAddr,  
    const unsigned char queryArr[],  
    unsigned char echoArr[],  
    int len ) [inherited]
```

Modbus function code 8, sub-function 00, Return Query Data.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>queryArr</i>	Buffer with data to be sent
<i>echoArr</i>	Buffer which will contain the data read
<i>len</i>	Number of bytes send sent and read back

**Returns**

FTALK\_SUCCESS on success, FTALK\_INVALID\_REPLY\_ERROR if reply does not match query data or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
restartCommunicationsOption() int restartCommunicationsOption (  
    int slaveAddr,  
    int clearEventLog ) [inherited]
```

Modbus function code 8, sub-function 01, Restart Communications Option.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>clearEventLog</i>	Flag when set to one clears in addition the slave's communication even log.

### Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readDeviceIdentification() int readDeviceIdentification (
    int slaveAddr,
    int accessType,
    int objId,
    DeviceIdObjectList * objListPtr ) [inherited]
```

Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.

This function allows a master to retrieve various objects with meta information about a slave device. The data type of the objects returned is a 0-terminated ASCII string.

Object Id	Object Name / Description
0x00	VendorName
0x01	ProductCode
0x02	MajorMinorRevision
0x03	VendorUrl
0x04	ProductName
0x05	ModelName
0x06	UserApplicationName
0x07 - 0x7F	<i>Reserved</i>
0x80 - 0xFF	Vendor specific private objects

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>accessType</i>	Category type (1=only mandatory basic objects, 2=all regular objects, 3=include vendor specific objects, 4=single object)
<i>objId</i>	An object ID from above table

## Parameters

<i>objListPtr</i>	A pointer to a object list container which will receive the retrieved objects.
-------------------	--

## Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

## Note

No broadcast supported

Example retrieving all Device ID strings of basic device identification level (level 1):

```
DeviceIdObjectList objs;
int objId = 0; // Start with object 0
do
{
    result = mbusProtocol.readDeviceIdentification(1, 1, objId, &objs);

    if (result != FTALK_SUCCESS)
    {
        fprintf(stderr, "Error: %s!\n", getBusProtocolErrorText(result));
        exit(EXIT_FAILURE);
    }
    else
    {
        size_t objLen;
        char* objData;

        while (objs.getNext(&objId, &objData, &objLen))
        {
            printf("obj[%d] = '%s', len = %u\n", objId, objData, objLen);
        }
    }
}
while (objs.moreToFollow());
```

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_FRAME\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
setTimeout() int setTimeout (
                    int msTime ) [inherited]
```

Configures time-out.

This function sets the operation or socket time-out to the specified value.

## Remarks

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

**Note**

A protocol must be closed in order to configure it.

**Parameters**

<i>msTime</i>	Timeout value in ms (Range: 1 - 100000)
---------------	---

**Return values**

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getTimeout()** `int getTimeout ( ) [inline], [inherited]`

Returns the time-out value.

**Remarks**

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

**Returns**

Timeout value in ms

**setPollDelay()** `int setPollDelay (`  
`int msTime ) [inherited]`

Configures poll delay.

This function sets the delay time which applies between two consecutive Modbus read/write. A value of 0 disables the poll delay.

**Remarks**

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

**Note**

A protocol must be closed in order to configure it.

**Parameters**

<i>msTime</i>	Delay time in ms (Range: 0 - 100000), 0 disables poll delay
---------------	---

## Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getPollDelay()** `int getPollDelay ( ) [inline], [inherited]`

Returns the poll delay time.

**Returns**

Delay time in ms, 0 if poll delay is switched off

**setRetryCnt()** `int setRetryCnt ( int retries ) [inherited]`

Configures the automatic retry setting.

A value of 0 disables any automatic retries.

**Note**

A protocol must be closed in order to configure it.

**Parameters**

<i>retries</i>	Retry count (Range: 0 - 10), 0 disables retries
----------------	---

## Return values

<i>FTALK_SUCCESS</i>	Success
----------------------	---------



Return values

<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ OR	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References `FTALK_ILLEGAL_ARGUMENT_ERROR`, `FTALK_ILLEGAL_STATE_ERROR`, `FTALK_↔SUCCESS`, and `MbusMasterFunctions::isOpen()`.

**getRetryCnt()** `int getRetryCnt ( ) [inline], [inherited]`

Returns the automatic retry count.

#### Returns

Retry count

**getTotalCounter()** `long getTotalCounter ( ) [inline], [inherited]`

Returns how often a message transfer has been executed.

#### Returns

Counter value

**getSuccessCounter()** `long getSuccessCounter ( ) [inline], [inherited]`

Returns how often a message transfer was successful.

#### Returns

Counter value

**configureBigEndianInts()** `[1/2] void configureBigEndianInts ( ) [inherited]`

Configures 32-bit int data type functions to do a word swap.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian slave.

**configureBigEndianInts()** [2/2] int configureBigEndianInts (  
int *slaveAddr* ) [inherited]

Enables int data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian machine.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureLittleEndianInts()** [1/2] void configureLittleEndianInts ( ) [inherited]

Configures 32-bit int data type functions not to do a word swap.

This is the default.

**configureLittleEndianInts()** [2/2] int configureLittleEndianInts (  
int *slaveAddr* ) [inherited]

Disables word swapping for int data type functions on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit data in little little-endian word order.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureIeeeFloats()** [1/2] void configureIeeeFloats ( ) [inherited]

Configures float data type functions not to do a word swap.

This is the default.

**configureIeeeFloats()** [2/2] int configureIeeeFloats (

```
int slaveAddr ) [inherited]
```

Disables float data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit floats in little little-endian word order which is the most common case.

### Remarks

This is the default mode

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

```
configureSwappedFloats() [1/2] void configureSwappedFloats ( ) [inherited]
```

Configures float data type functions to do a word swap.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

```
configureSwappedFloats() [2/2] int configureSwappedFloats (
    int slaveAddr ) [inherited]
```

Enables float data type functions to do a word swap on a per slave basis.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureStandard32BitMode()** [1/2] void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

#### Note

This function call also re-configures the endianness to default little-endian for 32-bit values!

#### Remarks

This is the default mode

**configureStandard32BitMode()** [2/2] int configureStandard32BitMode ( int *slaveAddr* ) [inherited]

Configures a slave for Standard 32-bit Register Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERROR</i> ↔ <i>OR</i>	Argument out of range

#### Remarks

This is the default mode

#### Note

This function call also re-configures the endianness to default little-endian for 32-bit values!

References `MbusMasterFunctions::configureIEEEFloats()`, `MbusMasterFunctions::configureLittleEndianInts()`, `FTALK_ILLEGAL_ARGUMENT_ERROR`, and `FTALK_SUCCESS`.

**configureEnron32BitMode()** [1/2] `void configureEnron32BitMode ( )` [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

#### Note

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

**configureEnron32BitMode()** [2/2] `int configureEnron32BitMode ( int slaveAddr )` [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<code>FTALK_SUCCESS</code>	Success
<code>FTALK_ILLEGAL_ARGUMENT_ERROR</code> OR	Argument out of range

#### Note

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

References `MbusMasterFunctions::configureBigEndianInts()`, `MbusMasterFunctions::configureSwappedFloats()`, `FTALK_ILLEGAL_ARGUMENT_ERROR`, and `FTALK_SUCCESS`.

**configureCountFromOne()** [1/2] `void configureCountFromOne ( )` [inherited]

Configures the reference counting scheme to start with one for all slaves.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Remarks

This is the default mode

**configureCountFromOne()** [2/2] `int configureCountFromOne (`  
`int slaveAddr ) [inherited]`

Configures a slave's reference counting scheme to start with one.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

#### Remarks

This is the default mode

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureCountFromZero()** [1/2] `void configureCountFromZero ( ) [inherited]`

Configures the reference counting scheme to start with zero for all slaves.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

This renders the first register to be #0 for all slaves.

**configureCountFromZero()** [2/2] `int configureCountFromZero (`  
`int slaveAddr ) [inherited]`

Configures a slave's reference counting scheme to start with zero.

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**getPackageVersion()** `const TCHAR * getPackageVersion ( ) [static], [inherited]`

Returns the library version number.

#### Returns

Library version string

## 8.2 MbusAsciiMasterProtocol Class Reference

Modbus ASCII Master Protocol class.

### Public Types

- enum { SER\_DATABITS\_7 = 7, SER\_DATABITS\_8 = 8 }
- enum { SER\_STOPBITS\_1 = 1, SER\_STOPBITS\_2 = 2 }
- enum { SER\_PARITY\_NONE = 0, SER\_PARITY\_EVEN = 2, SER\_PARITY\_ODD = 1 }

### Public Member Functions

- MbusAsciiMasterProtocol ()  
*Constructs a MbusAsciiMasterProtocol object and initialises its data.*
- virtual int openProtocol (const char \*const portName, long baudRate, int dataBits, int stopBits, int parity)  
*Opens a serial Modbus protocol and the associated serial port with specific port parameters.*
- virtual void closeProtocol ()  
*Closes an open protocol including any associated communication resources (com ports or sockets).*
- virtual int isOpen ()
- virtual int enableRs485Mode (int rtsDelay)  
*Enables RS485 mode.*

### Bit Access

Table 0:00000 (Coils) and Table 1:0000 (Input Status)

- int readCoils (int slaveAddr, int startRef, int bitArr[], int refCnt)  
*Modbus function 1, Read Coil Status/Read Coils.*
- int readInputDiscretes (int slaveAddr, int startRef, int bitArr[], int refCnt)  
*Modbus function 2, Read Inputs Status/Read Input Discretes.*
- int writeCoil (int slaveAddr, int bitAddr, int bitVal)

*Modbus function 5, Force Single Coil/Write Coil.*

- `int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)`  
*Modbus function 15 (0F Hex), Force Multiple Coils.*

## 16-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 3, Read Holding Registers/Read Multiple Registers.*
- `int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 4, Read Input Registers.*
- `int writeSingleRegister (int slaveAddr, int regAddr, short regVal)`  
*Modbus function 6, Preset Single Register/Write Single Register.*
- `int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)`  
*Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.*
- `int maskWriteRegister (int slaveAddr, int regAddr, short andMask, short orMask)`  
*Modbus function 22 (16 Hex), Mask Write Register.*
- `int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)`  
*Modbus function 23 (17 Hex), Read/Write Registers.*

## 32-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.*
- `int readInputLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.*
- `int writeMultipleLongInts (int slaveAddr, int startRef, const int int32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.*
- `int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.*
- `int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit float data types, Read Input Registers as float data.*
- `int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.*
- `int readMultipleMod10000 (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/↔ Read Multiple Registers as modulo-10000 long int data.*



- `int readInputMod10000 (int slaveAddr, int startRef, int int32Arr[], int refCnt)`  
*Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.*
- `int writeMultipleMod10000 (int slaveAddr, int startRef, const int int32Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.*

## 64-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readHoldingRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int readInputRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int writeMultipleRegistersI64 (int slaveAddr, int startRef, const int64_t int64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.*
- `int readHoldingRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.*
- `int readInputRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.*
- `int writeMultipleRegistersF64 (int slaveAddr, int startRef, const double double64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.*

## File Record Access

- `int readFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, short recordArr[], int recordCnt)`  
*Modbus function 20, Read File Record.*
- `int readFileRecord (int slaveAddr, FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 20, Read File Record.*
- `int writeFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, const short recordArr[], int recordCnt)`  
*Modbus function 21, Write File Record.*
- `int writeFileRecord (int slaveAddr, const FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 21, Write File Record.*

## Diagnostics

- `int readExceptionStatus (int slaveAddr, unsigned char *statusBytePtr)`  
*Modbus function 7, Read Exception Status.*
- `int returnQueryData (int slaveAddr, const unsigned char queryArr[], unsigned char echoArr[], int len)`  
*Modbus function code 8, sub-function 00, Return Query Data.*
- `int restartCommunicationsOption (int slaveAddr, int clearEventLog)`  
*Modbus function code 8, sub-function 01, Restart Communications Option.*
- `int readDeviceIdentification (int slaveAddr, int accessType, int objId, DeviceIdObjectList *objListPtr)`  
*Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.*

## Custom Function Codes

- `int customFunction (int slaveAddr, int functionCode, void *requestData, size_t requestLen, void *responseData, size_t *responseLenPtr)`  
*User Defined Function Code*  
*This method can be used to implement User Defined Function Codes.*

## Protocol Configuration

- `int setTimeout (int timeOut)`  
*Configures time-out.*
- `int getTimeout ()`  
*Returns the time-out value.*
- `int setPollDelay (int pollDelay)`  
*Configures poll delay.*
- `int getPollDelay ()`  
*Returns the poll delay time.*
- `int setRetryCnt (int retryCnt)`  
*Configures the automatic retry setting.*
- `int getRetryCnt ()`  
*Returns the automatic retry count.*

## Transmission Statistic Functions

- `long getTotalCounter ()`  
*Returns how often a message transfer has been executed.*
- `void resetTotalCounter ()`  
*Resets total message transfer counter.*
- `long getSuccessCounter ()`  
*Returns how often a message transfer was successful.*
- `void resetSuccessCounter ()`  
*Resets successful message transfer counter.*

## Slave Configuration

- `void configureBigEndianInts ()`  
*Configures 32-bit int data type functions to do a word swap.*
- `int configureBigEndianInts (int slaveAddr)`  
*Enables int data type functions to do a word swap on a per slave basis.*
- `void configureLittleEndianInts ()`  
*Configures 32-bit int data type functions not to do a word swap.*
- `int configureLittleEndianInts (int slaveAddr)`  
*Disables word swapping for int data type functions on a per slave basis.*
- `void configureleeeeFloats ()`  
*Configures float data type functions not to do a word swap.*
- `int configureleeeeFloats (int slaveAddr)`  
*Disables float data type functions to do a word swap on a per slave basis.*
- `void configureSwappedFloats ()`  
*Configures float data type functions to do a word swap.*
- `int configureSwappedFloats (int slaveAddr)`  
*Enables float data type functions to do a word swap on a per slave basis.*
- `void configureStandard32BitMode ()`  
*Configures all slaves for Standard 32-bit Mode.*
- `int configureStandard32BitMode (int slaveAddr)`  
*Configures a slave for Standard 32-bit Register Mode.*
- `void configureEnron32BitMode ()`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `int configureEnron32BitMode (int slaveAddr)`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `void configureCountFromOne ()`  
*Configures the reference counting scheme to start with one for all slaves.*
- `int configureCountFromOne (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with one.*
- `void configureCountFromZero ()`  
*Configures the reference counting scheme to start with zero for all slaves.*
- `int configureCountFromZero (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with zero.*

## Utility Functions

- `static const TCHAR * getPackageVersion ()`  
*Returns the library version number.*

## 8.2.1 Detailed Description

Modbus ASCII Master Protocol class.

This class realizes the Modbus ASCII master protocol. It provides functions to open and to close serial port as well as data and control functions which can be used at any time after the protocol has been opened. The data and control functions are organized into different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Modbus Protocol Flavours.

It is possible to instantiate multiple instances of this class for establishing multiple connections on different serial ports (They should be executed in separate threads).

### See also

Data and Control Functions for all Modbus Protocol Flavours, Serial Protocols  
MbusAsciiOverTcpMasterProtocol

## 8.2.2 Member Enumeration Documentation

**anonymous enum** anonymous enum [inherited]

Enumerator

SER_DATABITS↔ _7	7 data bits
SER_DATABITS↔ _8	8 data bits

**anonymous enum** anonymous enum [inherited]

Enumerator

SER_STOPBITS↔ _1	1 stop bit
SER_STOPBITS↔ _2	2 stop bits

**anonymous enum** anonymous enum [inherited]

Enumerator

SER_PARITY_NONE	No parity.
SER_PARITY_EVEN	Even parity.
SER_PARITY_ODD	Odd parity.

## 8.2.3 Member Function Documentation

```
openProtocol() int openProtocol (
    const char *const portName,
    long baudRate,
    int dataBits,
    int stopBits,
    int parity ) [virtual], [inherited]
```

Opens a serial Modbus protocol and the associated serial port with specific port parameters.

This function opens the serial port. After a port has been opened, data and control functions can be used.

### Note

The default time-out for the data transfer is 1000 ms.

The default poll delay is 0 ms.

Automatic retries are switched off (retry count is 0).

The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.

### Parameters

<i>portName</i>	Serial port identifier (e.g. "COM1", "/dev/ser1" or "/dev/ttyS0")
<i>baudRate</i>	The port baudRate in bps (typically 1200 - 115200, maximum value depends on UART hardware)
<i>dataBits</i>	SER_DATABITS_7: 7 data bits (ASCII protocol only), SER_DATABITS_8: data bits
<i>stopBits</i>	SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits
<i>parity</i>	SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_EVEN: even parity

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

Reimplemented in MbusRtuMasterProtocol.

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_ILLEGAL\_STATE\_ERROR, FTALK↔\_OPEN\_ERR, FTALK\_PORT\_ALREADY\_OPEN, FTALK\_PORT\_NO\_ACCESS, FTALK\_SUCCESS, and MbusSerialClientBase::isOpen().

**isOpen()** `int isOpen ( ) [virtual], [inherited]`

**Deprecated** Returns whether the protocol is open or not.

This simply returns the state of an internal flag whether openProtocol() has been called or not.

#### Note

This function does not report the state of the underlying transport network, in particular it does not report whether the TCP connection is still in 'established' state or in fully 'closed' state.

Return values

<i>true</i>	= open
<i>false</i>	= closed

Implements MbusMasterFunctions.

**enableRs485Mode()** `int enableRs485Mode ( int rtsDelay ) [virtual], [inherited]`

Enables RS485 mode.

In RS485 mode the RTS signal can be used to enable and disable the transmitter of a RS232/RS485 converter. The RTS signal is asserted before sending data. It is cleared after the transmit buffer has been emptied and in addition the specified delay time has elapsed. The delay time is necessary because even the transmit buffer is already empty, the UART's FIFO will still contain unsent characters.

#### Warning

The use of RTS controlled RS232/RS485 converters should be avoided if possible. It is difficult to determine the exact time when to switch off the transmitter with non real-time operating systems like Windows and Linux. If it is switched off too early characters might still sit in the FIFO or the transmit register of the UART and these characters will be lost. Hence the slave will not recognize the message. On the other hand if it is switched off too late then the slave's message is corrupted and the master will not recognize the message.

### Remarks

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

### Note

A protocol must be closed in order to configure it.

### Parameters

<i>rtsDelay</i>	Delay time in ms (Range: 0 - 100000) which applies after the transmit buffer is empty. 0 disables this mode.
-----------------	--

### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusSerialClientBase::isOpen()`.

```
readCoils() int readCoils (
    int slaveAddr,
    int startRef,
    int bitArr[],
    int refCnt ) [inherited]
```

Modbus function 1, Read Coil Status/Read Coils.

Reads the contents of the discrete outputs (coils, 0:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

### Returns

*FTALK\_SUCCESS* on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

**readInputDiscretes()** `int readInputDiscretes (`  
    `int slaveAddr,`  
    `int startRef,`  
    `int bitArr[],`  
    `int refCnt )` [inherited]

Modbus function 2, Read Inputs Status/Read Input Discretes.

Reads the contents of the discrete inputs (input status, 1:00000 table).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

**writeCoil()** `int writeCoil (`  
    `int slaveAddr,`  
    `int bitAddr,`  
    `int bitVal )` [inherited]

Modbus function 5, Force Single Coil/Write Coil.

Sets a single discrete output variable (coil, 0:00000 table) to either ON or OFF.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>bitAddr</i>	Coil address (Range: 1 - 65536)
<i>bitVal</i>	true sets, false clears discrete output variable



### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
forceMultipleCoils() int forceMultipleCoils (
    int slaveAddr,
    int startRef,
    const int bitArr[],
    int refCnt ) [inherited]
```

Modbus function 15 (0F Hex), Force Multiple Coils.

Writes binary values into a sequence of discrete outputs (coils, 0:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which contains the data to be sent
<i>refCnt</i>	Number of coils to be written (Range: 1-1968)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleRegisters() int readMultipleRegisters (
    int slaveAddr,
    int startRef,
    short regArr[],
    int refCnt ) [inherited]
```

Modbus function 3, Read Holding Registers/Read Multiple Registers.

Reads the contents of the output registers (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
readInputRegisters() int readInputRegisters (  
    int slaveAddr,  
    int startRef,  
    short regArr[],  
    int refCnt ) [inherited]
```

Modbus function 4, Read Input Registers.

Read the contents of the input registers (3:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read.
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
writeSingleRegister() int writeSingleRegister (
    int slaveAddr,
    int regAddr,
    short regVal ) [inherited]
```

Modbus function 6, Preset Single Register/Write Single Register.

Writes a value into a single output register (holding register, 4:00000 reference).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>regVal</i>	Data to be sent

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
writeMultipleRegisters() int writeMultipleRegisters (
    int slaveAddr,
    int startRef,
    const short regArr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.

Writes values into a sequence of output registers (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer with the data to be sent.
<i>refCnt</i>	Number of registers to be written (Range: 1-123)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

```
maskWriteRegister() int maskWriteRegister (
    int slaveAddr,
    int regAddr,
    short andMask,
    short orMask ) [inherited]
```

Modbus function 22 (16 Hex), Mask Write Register.

Masks bits according to an AND & an OR mask into a single output register (holding register, 4:00000 reference). Masking is done as follows: result = (currentVal AND andMask) OR (orMask AND (NOT andMask))

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>andMask</i>	Mask to be applied as a logic AND to the register
<i>orMask</i>	Mask to be applied as a logic OR to the register

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readWriteRegisters() int readWriteRegisters (
    int slaveAddr,
    int readRef,
    short readArr[],
    int readCnt,
    int writeRef,
    const short writeArr[],
    int writeCnt ) [inherited]
```

Modbus function 23 (17 Hex), Read/Write Registers.

Combines reading and writing of the output registers in one transaction (holding registers, 4:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>readRef</i>	Start register for reading (Range: 1 - 65536)
<i>readArr</i>	Buffer which will contain the data read
<i>readCnt</i>	Number of registers to be read (Range: 1-125)
<i>writeRef</i>	Start register for writing (Range: 1 - 65536)
<i>writeArr</i>	Buffer with data to be sent
<i>writeCnt</i>	Number of registers to be written (Range: 1-121)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleLongInts() int readMultipleLongInts (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into 32-bit long int values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputLongInts() int readInputLongInts (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data. Reads the contents of pairs of consecutive input registers (3:00000 table) into 32-bit long int values.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleLongInts() int writeMultipleLongInts (
    int slaveAddr,
    int startRef,
```

```

const int int32Arr[],
int refCnt ) [inherited]

```

Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/↔ Write Multiple Registers with long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```

readMultipleFloats() int readMultipleFloats (
    int slaveAddr,
    int startRef,
    float float32Arr[],
    int refCnt ) [inherited]

```

Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into float values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputFloats() int readInputFloats (
    int slaveAddr,
    int startRef,
    float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit float data types, Read Input Registers as float data.

Reads the contents of pairs of consecutive input registers (3:00000 table) into float values.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.



**Note**

No broadcast supported

```
writeMultipleFloats() int writeMultipleFloats (
    int slaveAddr,
    int startRef,
    const float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

Writes float values into pairs of output registers (holding registers, 4:00000 table).

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleMod10000() int readMultipleMod10000 (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value into 32-bit int values and performs number format conversion.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputMod10000() int readInputMod10000 (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) representing a modulo-10000 long int value into 32-bit long int values and performs number format conversion.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleMod10000() int writeMultipleMod10000 (
    int slaveAddr,
    int startRef,
    const int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value and performs number format conversion.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersI64() int readHoldingRegistersI64 (
    int slaveAddr,
```

```
int startRef,
int64_t int64Arr[],
int refCnt ) [inherited]
```

Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.

Reads the contents of quartets of consecutive holding registers (4:00000 table) into 64-bit integer values.

**Remarks**

Depending on the 32-bit Mode setting, an int64\_t will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputRegistersI64() int readInputRegistersI64 (
int slaveAddr,
int startRef,
int64_t int64Arr[],
int refCnt ) [inherited]
```

Modbus function 4 for 64-bit integer data, Read Holding Registers as int64\_t type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit integer values.

**Remarks**

Depending on the 32-bit Mode setting, an int64\_t will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleRegistersI64() int writeMultipleRegistersI64 (
    int slaveAddr,
    int startRef,
    const int64_t int64Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.

Writes for 64-bit integer values into quartets of holding registers (4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersF64() int readHoldingRegistersF64 (  
    int slaveAddr,  
    int startRef,  
    double double64Arr[],  
    int refCnt ) [inherited]
```

Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (4:00000 table) into 64-bit floating point values.

**Remarks**

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputRegistersF64() int readInputRegistersF64 (  
    int slaveAddr,  
    int startRef,  
    double double64Arr[],  
    int refCnt ) [inherited]
```

Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit floating point values.

#### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
writeMultipleRegistersF64() int writeMultipleRegistersF64 (
    int slaveAddr,
    int startRef,
    const double double64Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.

Writes for 64-bit floating point values into quartets of holding registers (4:00000 table).

#### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readFileRecord() [1/2] int readFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 20, Read File Record.  
 Performs a file record read for a single subrequest.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to read from, 0-based numbering.
<i>recordArr</i>	Buffer which will be filled with the data read
<i>recordCnt</i>	Number of records to read (Range 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readFileRecord() [2/2] int readFileRecord (
    int slaveAddr,
```



```
FileSubRequest subReqArr[],
int subReqCnt ) [inherited]
```

Modbus function 20, Read File Record.

Performs a file record read for a list of subrequests.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to read (Range 1-35)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK↔\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
writeFileRecord() [1/2] int writeFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    const short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a single subrequest.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to write to, 0-based numbering.
<i>recordArr</i>	Buffer with the data to be sent.
<i>recordCnt</i>	Number of records to write (Range 1-122)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeFileRecord() [2/2] int writeFileRecord (
    int slaveAddr,
    const FileSubRequest subReqArr[],
    int subReqCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a list of subrequests.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReqCnt</i>	Number of subrequests to write (Range 1-27)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
readExceptionStatus() int readExceptionStatus (
    int slaveAddr,
    unsigned char * statusBytePtr ) [inherited]
```

Modbus function 7, Read Exception Status.

Reads the eight exception status coils within the slave device.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>statusBytePtr</i>	Slave status byte. The meaning of this status byte is slave specific and varies from device to device.

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

```
returnQueryData() int returnQueryData (
    int slaveAddr,
    const unsigned char queryArr[],
    unsigned char echoArr[],
    int len ) [inherited]
```

Modbus function code 8, sub-function 00, Return Query Data.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>queryArr</i>	Buffer with data to be sent
<i>echoArr</i>	Buffer which will contain the data read
<i>len</i>	Number of bytes send sent and read back

### Returns

FTALK\_SUCCESS on success, FTALK\_INVALID\_REPLY\_ERROR if reply does not match query data or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

**restartCommunicationsOption()** int restartCommunicationsOption (  
int *slaveAddr*,  
int *clearEventLog* ) [inherited]

Modbus function code 8, sub-function 01, Restart Communications Option.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>clearEventLog</i>	Flag when set to one clears in addition the slave's communication even log.

#### Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

#### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

**readDeviceIdentification()** int readDeviceIdentification (  
int *slaveAddr*,  
int *accessType*,  
int *objId*,  
DeviceIdObjectList \* *objListPtr* ) [inherited]

Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.

This function allows a master to retrieve various objects with meta information about a slave device. The data type of the objects returned is a 0-terminated ASCII string.

Object Id	Object Name / Description
0x00	VendorName
0x01	ProductCode
0x02	MajorMinorRevision
0x03	VendorUrl
0x04	ProductName
0x05	ModelName
0x06	UserApplicationName
0x07 - 0x7F	<i>Reserved</i>
0x80 - 0xFF	Vendor specific private objects

## Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>accessType</i>	Category type (1=only mandatory basic objects, 2=all regular objects, 3=include vendor specific objects, 4=single object)
<i>objId</i>	An object ID from above table
<i>objListPtr</i>	A pointer to a object list container which will receive the retrieved objects.

## Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

## Note

No broadcast supported

Example retrieving all Device ID strings of basic device identification level (level 1):

```
DeviceIdObjectList objs;
int objId = 0; // Start with object 0
do
{
    result = mbusProtocol.readDeviceIdentification(1, 1, objId, &objs);

    if (result != FTALK_SUCCESS)
    {
        fprintf(stderr, "Error: %s!\n", getBusProtocolErrorText(result));
        exit(EXIT_FAILURE);
    }
    else
    {
        size_t objLen;
        char* objData;

        while (objs.getNext(&objId, &objData, &objLen))
        {
            printf("obj[%d] = '%s', len = %u\n", objId, objData, objLen);
        }
    }
}
while (objs.moreToFollow());
```

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_FRAME\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
setTimeout() int setTimeout (
    int msTime ) [inherited]
```

Configures time-out.

This function sets the operation or socket time-out to the specified value.

**Remarks**

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

**Note**

A protocol must be closed in order to configure it.

**Parameters**

<i>msTime</i>	Timeout value in ms (Range: 1 - 100000)
---------------	---

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getTimeout()** `int getTimeout ( ) [inline], [inherited]`

Returns the time-out value.

**Remarks**

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

**Returns**

Timeout value in ms

**setPollDelay()** `int setPollDelay ( int msTime ) [inherited]`

Configures poll delay.

This function sets the delay time which applies between two consecutive Modbus read/write. A value of 0 disables the poll delay.

### Remarks

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

### Note

A protocol must be closed in order to configure it.

### Parameters

<i>msTime</i>	Delay time in ms (Range: 0 - 100000), 0 disables poll delay
---------------	---

### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ OR	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getPollDelay()** `int getPollDelay ( ) [inline], [inherited]`

Returns the poll delay time.

### Returns

Delay time in ms, 0 if poll delay is switched off

**setRetryCnt()** `int setRetryCnt ( int retries ) [inherited]`

Configures the automatic retry setting.

A value of 0 disables any automatic retries.

### Note

A protocol must be closed in order to configure it.

### Parameters

<i>retries</i>	Retry count (Range: 0 - 10), 0 disables retries
----------------	---

### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getRetryCnt()** `int getRetryCnt ( ) [inline], [inherited]`

Returns the automatic retry count.

### Returns

Retry count

**getTotalCounter()** `long getTotalCounter ( ) [inline], [inherited]`

Returns how often a message transfer has been executed.

### Returns

Counter value

**getSuccessCounter()** `long getSuccessCounter ( ) [inline], [inherited]`

Returns how often a message transfer was successful.

### Returns

Counter value



**configureBigEndianInts()** [1/2] void configureBigEndianInts ( ) [inherited]

Configures 32-bit int data type functions to do a word swap.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian slave.

**configureBigEndianInts()** [2/2] int configureBigEndianInts (   
 int *slaveAddr* ) [inherited]

Enables int data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian machine.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureLittleEndianInts()** [1/2] void configureLittleEndianInts ( ) [inherited]

Configures 32-bit int data type functions not to do a word swap.

This is the default.

**configureLittleEndianInts()** [2/2] int configureLittleEndianInts (   
 int *slaveAddr* ) [inherited]

Disables word swapping for int data type functions on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit data in little little-endian word order.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureIeeeFloats()** [1/2] void configureIeeeFloats ( ) [inherited]

Configures float data type functions not to do a word swap.  
This is the default.

**configureIeeeFloats()** [2/2] int configureIeeeFloats (   
int *slaveAddr* ) [inherited]

Disables float data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit floats in little little-endian word order which is the most common case.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureSwappedFloats()** [1/2] void configureSwappedFloats ( ) [inherited]

Configures float data type functions to do a word swap.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

#### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

**configureSwappedFloats()** [2/2] int configureSwappedFloats (   
int *slaveAddr* ) [inherited]

Enables float data type functions to do a word swap on a per slave basis.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

#### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureStandard32BitMode()** [1/2] void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

### Note

This function call also re-configures the endianness to default little-endian for 32-bit values!

### Remarks

This is the default mode

**configureStandard32BitMode()** [2/2] int configureStandard32BitMode ( int *slaveAddr* ) [inherited]

Configures a slave for Standard 32-bit Register Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range

### Remarks

This is the default mode

**Note**

This function call also re-configures the endianness to default little-endian for 32-bit values!

References MbusMasterFunctions::configureIEEEFloats(), MbusMasterFunctions::configureLittleEndianInts(), FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureEnron32BitMode()** [1/2] void configureEnron32BitMode ( ) [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

**Note**

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

**configureEnron32BitMode()** [2/2] int configureEnron32BitMode ( int *slaveAddr* ) [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

**Return values**

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERROR</i> OR	Argument out of range

**Note**

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

References MbusMasterFunctions::configureBigEndianInts(), MbusMasterFunctions::configureSwappedFloats(), FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureCountFromOne()** [1/2] void configureCountFromOne ( ) [inherited]

Configures the reference counting scheme to start with one for all slaves.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Remarks

This is the default mode

**configureCountFromOne()** [2/2] int configureCountFromOne (   
 int *slaveAddr* ) [inherited]

Configures a slave's reference counting scheme to start with one.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

#### Remarks

This is the default mode

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureCountFromZero()** [1/2] void configureCountFromZero ( ) [inherited]

Configures the reference counting scheme to start with zero for all slaves.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

This renders the first register to be #0 for all slaves.

**configureCountFromZero()** [2/2] int configureCountFromZero (   
 int *slaveAddr* ) [inherited]

Configures a slave's reference counting scheme to start with zero.

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**getPackageVersion()** `const TCHAR * getPackageVersion ( ) [static], [inherited]`

Returns the library version number.

#### Returns

Library version string

## 8.3 MbusElamMasterProtocol Class Reference

Extended Lufkin Automation Modbus Master Protocol.

### Public Types

- enum { SER\_DATABITS\_7 = 7, SER\_DATABITS\_8 = 8 }
- enum { SER\_STOPBITS\_1 = 1, SER\_STOPBITS\_2 = 2 }
- enum { SER\_PARITY\_NONE = 0, SER\_PARITY\_EVEN = 2, SER\_PARITY\_ODD = 1 }

### Public Member Functions

- MbusElamMasterProtocol ()  
*Constructs a MbusElamMasterProtocol object and initialises its data.*
- virtual int openProtocol (const char \*const portName, long baudRate, int dataBits, int stopBits, int parity)  
*Opens a Modbus RTU protocol and the associated serial port with specific port parameters.*
- virtual void closeProtocol ()  
*Closes an open protocol including any associated communication resources (com ports or sockets).*
- virtual int isOpen ()
- virtual int enableRs485Mode (int rtsDelay)  
*Enables RS485 mode.*

### Bit Access

Table 0:00000 (Coils) and Table 1:0000 (Input Status)

- int readCoils (int slaveAddr, int startRef, int bitArr[], int refCnt)  
*Modbus function 1, Read Coil Status/Read Coils.*
- int readInputDiscretes (int slaveAddr, int startRef, int bitArr[], int refCnt)  
*Modbus function 2, Read Inputs Status/Read Input Discretes.*
- int writeCoil (int slaveAddr, int bitAddr, int bitVal)

*Modbus function 5, Force Single Coil/Write Coil.*

- `int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)`  
*Modbus function 15 (0F Hex), Force Multiple Coils.*

## 16-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 3, Read Holding Registers/Read Multiple Registers.*
- `int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 4, Read Input Registers.*
- `int writeSingleRegister (int slaveAddr, int regAddr, short regVal)`  
*Modbus function 6, Preset Single Register/Write Single Register.*
- `int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)`  
*Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.*
- `int maskWriteRegister (int slaveAddr, int regAddr, short andMask, short orMask)`  
*Modbus function 22 (16 Hex), Mask Write Register.*
- `int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)`  
*Modbus function 23 (17 Hex), Read/Write Registers.*

## 32-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.*
- `int readInputLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.*
- `int writeMultipleLongInts (int slaveAddr, int startRef, const int int32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.*
- `int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.*
- `int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit float data types, Read Input Registers as float data.*
- `int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.*
- `int readMultipleMod10000 (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/↔ Read Multiple Registers as modulo-10000 long int data.*

- `int readInputMod10000 (int slaveAddr, int startRef, int int32Arr[], int refCnt)`  
*Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.*
- `int writeMultipleMod10000 (int slaveAddr, int startRef, const int int32Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.*

## 64-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readHoldingRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int readInputRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int writeMultipleRegistersI64 (int slaveAddr, int startRef, const int64_t int64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.*
- `int readHoldingRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.*
- `int readInputRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.*
- `int writeMultipleRegistersF64 (int slaveAddr, int startRef, const double double64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.*

## File Record Access

- `int readFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, short recordArr[], int recordCnt)`  
*Modbus function 20, Read File Record.*
- `int readFileRecord (int slaveAddr, FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 20, Read File Record.*
- `int writeFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, const short recordArr[], int recordCnt)`  
*Modbus function 21, Write File Record.*
- `int writeFileRecord (int slaveAddr, const FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 21, Write File Record.*



## Diagnostics

- `int readExceptionStatus (int slaveAddr, unsigned char *statusBytePtr)`  
*Modbus function 7, Read Exception Status.*
- `int returnQueryData (int slaveAddr, const unsigned char queryArr[], unsigned char echoArr[], int len)`  
*Modbus function code 8, sub-function 00, Return Query Data.*
- `int restartCommunicationsOption (int slaveAddr, int clearEventLog)`  
*Modbus function code 8, sub-function 01, Restart Communications Option.*
- `int readDeviceIdentification (int slaveAddr, int accessType, int objId, DeviceIdObjectList *objListPtr)`  
*Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.*

## Custom Function Codes

- `int customFunction (int slaveAddr, int functionCode, void *requestData, size_t requestLen, void *responseData, size_t *responseLenPtr)`  
*User Defined Function Code*  
*This method can be used to implement User Defined Function Codes.*

## Protocol Configuration

- `int setTimeout (int timeOut)`  
*Configures time-out.*
- `int getTimeout ()`  
*Returns the time-out value.*
- `int setPollDelay (int pollDelay)`  
*Configures poll delay.*
- `int getPollDelay ()`  
*Returns the poll delay time.*
- `int setRetryCnt (int retryCnt)`  
*Configures the automatic retry setting.*
- `int getRetryCnt ()`  
*Returns the automatic retry count.*

## Transmission Statistic Functions

- `long getTotalCounter ()`  
*Returns how often a message transfer has been executed.*
- `void resetTotalCounter ()`  
*Resets total message transfer counter.*
- `long getSuccessCounter ()`  
*Returns how often a message transfer was successful.*
- `void resetSuccessCounter ()`  
*Resets successful message transfer counter.*

## Slave Configuration

- `void configureBigEndianInts ()`  
*Configures 32-bit int data type functions to do a word swap.*
- `int configureBigEndianInts (int slaveAddr)`  
*Enables int data type functions to do a word swap on a per slave basis.*
- `void configureLittleEndianInts ()`  
*Configures 32-bit int data type functions not to do a word swap.*
- `int configureLittleEndianInts (int slaveAddr)`  
*Disables word swapping for int data type functions on a per slave basis.*
- `void configureleeeeFloats ()`  
*Configures float data type functions not to do a word swap.*
- `int configureleeeeFloats (int slaveAddr)`  
*Disables float data type functions to do a word swap on a per slave basis.*
- `void configureSwappedFloats ()`  
*Configures float data type functions to do a word swap.*
- `int configureSwappedFloats (int slaveAddr)`  
*Enables float data type functions to do a word swap on a per slave basis.*
- `void configureStandard32BitMode ()`  
*Configures all slaves for Standard 32-bit Mode.*
- `int configureStandard32BitMode (int slaveAddr)`  
*Configures a slave for Standard 32-bit Register Mode.*
- `void configureEnron32BitMode ()`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `int configureEnron32BitMode (int slaveAddr)`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `void configureCountFromOne ()`  
*Configures the reference counting scheme to start with one for all slaves.*
- `int configureCountFromOne (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with one.*
- `void configureCountFromZero ()`  
*Configures the reference counting scheme to start with zero for all slaves.*
- `int configureCountFromZero (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with zero.*

## Utility Functions

- `static const TCHAR * getPackageVersion ()`  
*Returns the library version number.*

### 8.3.1 Detailed Description

Extended Lufkin Automation Modbus Master Protocol.

This class realizes the Extended Lufkin Automation (ELAM) Modbus protocol. This proprietary Modbus extension allows addressing of up to 2295 slave units and the retrieval of up to 2500 registers for Modbus functions 3 and 4.

Its implementation is based on the specification "ELAM Extended Lufkin Automation Modbus Version 1.01" published by LUFKIN Automation. The ELAM multiple instruction requests extensions are not implemented.

Tests showed the following size limits with a LUFKIN SAM Well Manager device:

Coils: 1992 for read Registers: 2500 to read, 60 for write

#### See also

Data and Control Functions for all Modbus Protocol Flavours, Serial Protocols  
MbusRtuMasterProtocol

### 8.3.2 Member Enumeration Documentation

**anonymous enum** anonymous enum [inherited]

Enumerator

SER_DATABITS↔ _7	7 data bits
SER_DATABITS↔ _8	8 data bits

**anonymous enum** anonymous enum [inherited]

Enumerator

SER_STOPBITS↔ _1	1 stop bit
SER_STOPBITS↔ _2	2 stop bits

**anonymous enum** anonymous enum [inherited]

## Enumerator

SER_PARITY_NONE	No parity.
SER_PARITY_EVEN	Even parity.
SER_PARITY_ODD	Odd parity.

### 8.3.3 Member Function Documentation

```
openProtocol() int openProtocol (  
    const char *const portName,  
    long baudRate,  
    int dataBits,  
    int stopBits,  
    int parity ) [virtual], [inherited]
```

Opens a Modbus RTU protocol and the associated serial port with specific port parameters.

This function opens the serial port. After a port has been opened, data and control functions can be used.

**Note**

The default time-out for the data transfer is 1000 ms.

The default poll delay is 0 ms.

Automatic retries are switched off (retry count is 0).

The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.

**Parameters**

<i>portName</i>	Serial port identifier (e.g. "COM1", "/dev/ser1" or "/dev/ttyS0")
<i>baudRate</i>	The port baudRate in bps (typically 1200 - 115200, maximum value depends on UART hardware)
<i>dataBits</i>	Must be SER_DATABITS_8 for RTU
<i>stopBits</i>	SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits
<i>parity</i>	SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_EVEN: even parity

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

Reimplemented from MbusSerialClientBase.

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, MbusSerialClientBase::openProtocol(), and MbusSerialClientBase::SER\_DATABITS\_8.

**isOpen()** `int isOpen ( ) [virtual], [inherited]`

**Deprecated** Returns whether the protocol is open or not.

This simply returns the state of an internal flag whether openProtocol() has been called or not.

#### Note

This function does not report the state of the underlying transport network, in particular it does not report whether the TCP connection is still in 'established' state or in fully 'closed' state.

Return values

<i>true</i>	= open
<i>false</i>	= closed

Implements MbusMasterFunctions.

**enableRs485Mode()** `int enableRs485Mode ( int rtsDelay ) [virtual], [inherited]`

Enables RS485 mode.

In RS485 mode the RTS signal can be used to enable and disable the transmitter of a RS232/RS485 converter. The RTS signal is asserted before sending data. It is cleared after the transmit buffer has been emptied and in addition the specified delay time has elapsed. The delay time is necessary because even the transmit buffer is already empty, the UART's FIFO will still contain unsent characters.

#### Warning

The use of RTS controlled RS232/RS485 converters should be avoided if possible. It is difficult to determine the exact time when to switch off the transmitter with non real-time operating systems like Windows and Linux. If it is switched off too early characters might still sit in the FIFO or the transmit register of the UART and these characters will be lost. Hence the slave will not recognize the message. On the other hand if it is switched off too late then the slave's message is corrupted and the master will not recognize the message.

### Remarks

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

### Note

A protocol must be closed in order to configure it.

### Parameters

<i>rtsDelay</i>	Delay time in ms (Range: 0 - 100000) which applies after the transmit buffer is empty. 0 disables this mode.
-----------------	--

### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusSerialClientBase::isOpen()`.

```
readCoils() int readCoils (  
    int slaveAddr,  
    int startRef,  
    int bitArr[],  
    int refCnt ) [inherited]
```

Modbus function 1, Read Coil Status/Read Coils.

Reads the contents of the discrete outputs (coils, 0:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

### Returns

*FTALK\_SUCCESS* on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputDiscretes() int readInputDiscretes (
    int slaveAddr,
    int startRef,
    int bitArr[],
    int refCnt ) [inherited]
```

Modbus function 2, Read Inputs Status/Read Input Discretes.

Reads the contents of the discrete inputs (input status, 1:00000 table).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeCoil() int writeCoil (
    int slaveAddr,
    int bitAddr,
    int bitVal ) [inherited]
```

Modbus function 5, Force Single Coil/Write Coil.

Sets a single discrete output variable (coil, 0:00000 table) to either ON or OFF.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>bitAddr</i>	Coil address (Range: 1 - 65536)
<i>bitVal</i>	true sets, false clears discrete output variable

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
forceMultipleCoils() int forceMultipleCoils (  
    int slaveAddr,  
    int startRef,  
    const int bitArr[],  
    int refCnt ) [inherited]
```

Modbus function 15 (0F Hex), Force Multiple Coils.

Writes binary values into a sequence of discrete outputs (coils, 0:00000 table).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which contains the data to be sent
<i>refCnt</i>	Number of coils to be written (Range: 1-1968)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleRegisters() int readMultipleRegisters (  
    int slaveAddr,  
    int startRef,  
    short regArr[],  
    int refCnt ) [inherited]
```



Modbus function 3, Read Holding Registers/Read Multiple Registers.

Reads the contents of the output registers (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
readInputRegisters() int readInputRegisters (
    int slaveAddr,
    int startRef,
    short regArr[],
    int refCnt ) [inherited]
```

Modbus function 4, Read Input Registers.

Read the contents of the input registers (3:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read.
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
writeSingleRegister() int writeSingleRegister (
    int slaveAddr,
    int regAddr,
    short regVal ) [inherited]
```

Modbus function 6, Preset Single Register/Write Single Register.

Writes a value into a single output register (holding register, 4:00000 reference).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>regVal</i>	Data to be sent

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
writeMultipleRegisters() int writeMultipleRegisters (
    int slaveAddr,
    int startRef,
    const short regArr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.

Writes values into a sequence of output registers (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer with the data to be sent.
<i>refCnt</i>	Number of registers to be written (Range: 1-123)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

```
maskWriteRegister() int maskWriteRegister (
    int slaveAddr,
    int regAddr,
    short andMask,
    short orMask ) [inherited]
```

Modbus function 22 (16 Hex), Mask Write Register.

Masks bits according to an AND & an OR mask into a single output register (holding register, 4:00000 reference). Masking is done as follows: result = (currentVal AND andMask) OR (orMask AND (NOT andMask))

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>andMask</i>	Mask to be applied as a logic AND to the register
<i>orMask</i>	Mask to be applied as a logic OR to the register

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readWriteRegisters() int readWriteRegisters (
    int slaveAddr,
    int readRef,
    short readArr[],
    int readCnt,
    int writeRef,
    const short writeArr[],
    int writeCnt ) [inherited]
```

Modbus function 23 (17 Hex), Read/Write Registers.

Combines reading and writing of the output registers in one transaction (holding registers, 4:00000 table).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>readRef</i>	Start register for reading (Range: 1 - 65536)
<i>readArr</i>	Buffer which will contain the data read
<i>readCnt</i>	Number of registers to be read (Range: 1-125)
<i>writeRef</i>	Start register for writing (Range: 1 - 65536)
<i>writeArr</i>	Buffer with data to be sent
<i>writeCnt</i>	Number of registers to be written (Range: 1-121)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleLongInts() int readMultipleLongInts (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into 32-bit long int values.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readInputLongInts() int readInputLongInts (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) into 32-bit long int values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleLongInts() int writeMultipleLongInts (
    int slaveAddr,
    int startRef,
```

```
const int int32Arr[],  
int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/↔ Write Multiple Registers with long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleFloats() int readMultipleFloats (  
    int slaveAddr,  
    int startRef,  
    float float32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into float values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readInputFloats() int readInputFloats (
    int slaveAddr,
    int startRef,
    float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit float data types, Read Input Registers as float data.

Reads the contents of pairs of consecutive input registers (3:00000 table) into float values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleFloats() int writeMultipleFloats (  
    int slaveAddr,  
    int startRef,  
    const float float32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

Writes float values into pairs of output registers (holding registers, 4:00000 table).

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleMod10000() int readMultipleMod10000 (  
    int slaveAddr,  
    int startRef,  
    int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.



Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value into 32-bit int values and performs number format conversion.

#### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
readInputMod10000() int readInputMod10000 (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) representing a modulo-10000 long int value into 32-bit long int values and performs number format conversion.

#### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleMod10000() int writeMultipleMod10000 (  
    int slaveAddr,  
    int startRef,  
    const int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value and performs number format conversion.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersI64() int readHoldingRegistersI64 (  
    int slaveAddr,
```

```

int startRef,
int64_t int64Arr[],
int refCnt ) [inherited]

```

Modbus function 3 for 64-bit integer data, Read Holding Registers as `int64_t` type.

Reads the contents of quartets of consecutive holding registers (4:00000 table) into 64-bit integer values.

### Remarks

Depending on the 32-bit Mode setting, an `int64_t` will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```

readInputRegistersI64() int readInputRegistersI64 (
int slaveAddr,
int startRef,
int64_t int64Arr[],
int refCnt ) [inherited]

```

Modbus function 4 for 64-bit integer data, Read Holding Registers as `int64_t` type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit integer values.

### Remarks

Depending on the 32-bit Mode setting, an `int64_t` will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleRegistersI64() int writeMultipleRegistersI64 (  
    int slaveAddr,  
    int startRef,  
    const int64_t int64Arr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.

Writes for 64-bit integer values into quartets of holding registers (4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersF64() int readHoldingRegistersF64 (
    int slaveAddr,
    int startRef,
    double double64Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (4:00000 table) into 64-bit floating point values.

**Remarks**

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputRegistersF64() int readInputRegistersF64 (
    int slaveAddr,
    int startRef,
    double double64Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit floating point values.

**Remarks**

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleRegistersF64( int writeMultipleRegistersF64 (
    int slaveAddr,
    int startRef,
    const double double64Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.

Writes for 64-bit floating point values into quartets of holding registers (4:00000 table).

**Remarks**

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readFileRecord() [1/2] int readFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 20, Read File Record.

Performs a file record read for a single subrequest.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to read from, 0-based numbering.
<i>recordArr</i>	Buffer which will be filled with the data read
<i>recordCnt</i>	Number of records to read (Range 1-125)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readFileRecord() [2/2] int readFileRecord (
    int slaveAddr,
```

```
FileSubRequest subReqArr[],
int subReqCnt ) [inherited]
```

Modbus function 20, Read File Record.

Performs a file record read for a list of subrequests.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to read (Range 1-35)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK↔\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
writeFileRecord() [1/2] int writeFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    const short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a single subrequest.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to write to, 0-based numbering.
<i>recordArr</i>	Buffer with the data to be sent.
<i>recordCnt</i>	Number of records to write (Range 1-122)



### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeFileRecord() [2/2] int writeFileRecord (
    int slaveAddr,
    const FileSubRequest subReqArr[],
    int subReqCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a list of subrequests.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to write (Range 1-27)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK↔\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
readExceptionStatus() int readExceptionStatus (
    int slaveAddr,
    unsigned char * statusBytePtr ) [inherited]
```

Modbus function 7, Read Exception Status.

Reads the eight exception status coils within the slave device.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>statusBytePtr</i>	Slave status byte. The meaning of this status byte is slave specific and varies from device to device.

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

```
returnQueryData() int returnQueryData (  
    int slaveAddr,  
    const unsigned char queryArr[],  
    unsigned char echoArr[],  
    int len ) [inherited]
```

Modbus function code 8, sub-function 00, Return Query Data.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>queryArr</i>	Buffer with data to be sent
<i>echoArr</i>	Buffer which will contain the data read
<i>len</i>	Number of bytes send sent and read back

### Returns

FTALK\_SUCCESS on success, FTALK\_INVALID\_REPLY\_ERROR if reply does not match query data or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
restartCommunicationsOption() int restartCommunicationsOption (
    int slaveAddr,
    int clearEventLog ) [inherited]
```

Modbus function code 8, sub-function 01, Restart Communications Option.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>clearEventLog</i>	Flag when set to one clears in addition the slave's communication even log.

#### Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

#### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readDeviceIdentification() int readDeviceIdentification (
    int slaveAddr,
    int accessType,
    int objId,
    DeviceIdObjectList * objListPtr ) [inherited]
```

Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.

This function allows a master to retrieve various objects with meta information about a slave device. The data type of the objects returned is a 0-terminated ASCII string.

Object Id	Object Name / Description
0x00	VendorName
0x01	ProductCode
0x02	MajorMinorRevision
0x03	VendorUrl
0x04	ProductName
0x05	ModelName
0x06	UserApplicationName
0x07 - 0x7F	<i>Reserved</i>
0x80 - 0xFF	Vendor specific private objects

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>accessType</i>	Category type (1=only mandatory basic objects, 2=all regular objects, 3=include vendor specific objects, 4=single object)
<i>objId</i>	An object ID from above table
<i>objListPtr</i>	A pointer to a object list container which will receive the retrieved objects.

### Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

### Note

No broadcast supported

Example retrieving all Device ID strings of basic device identification level (level 1):

```
DeviceIdObjectList objs;
int objId = 0; // Start with object 0
do
{
    result = mbusProtocol.readDeviceIdentification(1, 1, objId, &objs);

    if (result != FTALK_SUCCESS)
    {
        fprintf(stderr, "Error: %s!\n", getBusProtocolErrorText(result));
        exit(EXIT_FAILURE);
    }
    else
    {
        size_t objLen;
        char* objData;

        while (objs.getNext(&objId, &objData, &objLen))
        {
            printf("obj[%d] = '%s', len = %u\n", objId, objData, objLen);
        }
    }
}
while (objs.moreToFollow());
```

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_FRAME\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
setTimeout() int setTimeout (
    int msTime ) [inherited]
```

Configures time-out.

This function sets the operation or socket time-out to the specified value.

### Remarks

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

### Note

A protocol must be closed in order to configure it.

### Parameters

<i>msTime</i>	Timeout value in ms (Range: 1 - 100000)
---------------	---

### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getTimeout()** `int getTimeout ( ) [inline], [inherited]`

Returns the time-out value.

### Remarks

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

### Returns

Timeout value in ms

**setPollDelay()** `int setPollDelay (`  
`int msTime ) [inherited]`

Configures poll delay.

This function sets the delay time which applies between two consecutive Modbus read/write. A value of 0 disables the poll delay.

### Remarks

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

### Note

A protocol must be closed in order to configure it.

### Parameters

<i>msTime</i>	Delay time in ms (Range: 0 - 100000), 0 disables poll delay
---------------	---

### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getPollDelay()** `int getPollDelay ( ) [inline], [inherited]`

Returns the poll delay time.

### Returns

Delay time in ms, 0 if poll delay is switched off

**setRetryCnt()** `int setRetryCnt (`  
`int retries ) [inherited]`

Configures the automatic retry setting.

A value of 0 disables any automatic retries.

### Note

A protocol must be closed in order to configure it.

### Parameters

<i>retries</i>	Retry count (Range: 0 - 10), 0 disables retries
----------------	---

### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getRetryCnt()** `int getRetryCnt ( ) [inline], [inherited]`

Returns the automatic retry count.

### Returns

Retry count

**getTotalCounter()** `long getTotalCounter ( ) [inline], [inherited]`

Returns how often a message transfer has been executed.

### Returns

Counter value

**getSuccessCounter()** `long getSuccessCounter ( ) [inline], [inherited]`

Returns how often a message transfer was successful.

### Returns

Counter value

**configureBigEndianInts()** [1/2] void configureBigEndianInts ( ) [inherited]

Configures 32-bit int data type functions to do a word swap.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian slave.

**configureBigEndianInts()** [2/2] int configureBigEndianInts (   
int *slaveAddr* ) [inherited]

Enables int data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian machine.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureLittleEndianInts()** [1/2] void configureLittleEndianInts ( ) [inherited]

Configures 32-bit int data type functions not to do a word swap.

This is the default.

**configureLittleEndianInts()** [2/2] int configureLittleEndianInts (   
int *slaveAddr* ) [inherited]

Disables word swapping for int data type functions on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit data in little little-endian word order.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.



**configureIeeeFloats()** [1/2] void configureIeeeFloats ( ) [inherited]

Configures float data type functions not to do a word swap.  
This is the default.

**configureIeeeFloats()** [2/2] int configureIeeeFloats (   
int *slaveAddr* ) [inherited]

Disables float data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit floats in little little-endian word order which is the most common case.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureSwappedFloats()** [1/2] void configureSwappedFloats ( ) [inherited]

Configures float data type functions to do a word swap.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

#### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

**configureSwappedFloats()** [2/2] int configureSwappedFloats (   
int *slaveAddr* ) [inherited]

Enables float data type functions to do a word swap on a per slave basis.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

#### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureStandard32BitMode()** [1/2] void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

### Note

This function call also re-configures the endianness to default little-endian for 32-bit values!

### Remarks

This is the default mode

**configureStandard32BitMode()** [2/2] int configureStandard32BitMode ( int *slaveAddr* ) [inherited]

Configures a slave for Standard 32-bit Register Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERROR</i> OR	Argument out of range

### Remarks

This is the default mode

**Note**

This function call also re-configures the endianness to default little-endian for 32-bit values!

References `MbusMasterFunctions::configureIEEEFloats()`, `MbusMasterFunctions::configureLittleEndianInts()`, `FTALK_ILLEGAL_ARGUMENT_ERROR`, and `FTALK_SUCCESS`.

**configureEnron32BitMode()** [1/2] `void configureEnron32BitMode ( )` [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

**Note**

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

**configureEnron32BitMode()** [2/2] `int configureEnron32BitMode ( int slaveAddr )` [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERROR</i> OR	Argument out of range

**Note**

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

References `MbusMasterFunctions::configureBigEndianInts()`, `MbusMasterFunctions::configureSwappedFloats()`, `FTALK_ILLEGAL_ARGUMENT_ERROR`, and `FTALK_SUCCESS`.

**configureCountFromOne()** [1/2] void configureCountFromOne ( ) [inherited]

Configures the reference counting scheme to start with one for all slaves.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Remarks

This is the default mode

**configureCountFromOne()** [2/2] int configureCountFromOne (   
int *slaveAddr* ) [inherited]

Configures a slave's reference counting scheme to start with one.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

#### Remarks

This is the default mode

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureCountFromZero()** [1/2] void configureCountFromZero ( ) [inherited]

Configures the reference counting scheme to start with zero for all slaves.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

This renders the first register to be #0 for all slaves.

**configureCountFromZero()** [2/2] int configureCountFromZero (   
int *slaveAddr* ) [inherited]

Configures a slave's reference counting scheme to start with zero.

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**getPackageVersion()** `const TCHAR * getPackageVersion ( ) [static], [inherited]`

Returns the library version number.

#### Returns

Library version string

## 8.4 MbusTcpMasterProtocol Class Reference

MODBUS/TCP Master Protocol class.

### Public Member Functions

- `MbusTcpMasterProtocol ()`  
*Constructs a MbusTcpMasterProtocol object and initialises its data.*
- `int openProtocol (const char *const hostName)`  
*Connects to a MODBUS/TCP slave.*
- `int setPort (unsigned short portNo)`  
*Sets the TCP port number to be used by the protocol.*
- `int openProtocol (const char *const hostName, const char *const interfacelp)`  
*Opens a connection to a slave device.*
- `virtual void closeProtocol ()`  
*Closes a TCP/IP connection to a slave and releases any system resources associated with the connection.*
- `virtual int isOpen ()`
- `int setClosingTimeout (int msTime)`  
*Applies a time-out to socket closure and makes closeProtocol() wait for the server to acknowledge closing before potentially opening a new one.*
- `unsigned short getPort ()`  
*Returns the TCP port number used by the protocol.*

### Bit Access

Table 0:0000 (Coils) and Table 1:0000 (Input Status)

- `int readCoils (int slaveAddr, int startRef, int bitArr[], int refCnt)`  
*Modbus function 1, Read Coil Status/Read Coils.*
- `int readInputDiscretes (int slaveAddr, int startRef, int bitArr[], int refCnt)`  
*Modbus function 2, Read Inputs Status/Read Input Discretes.*
- `int writeCoil (int slaveAddr, int bitAddr, int bitVal)`

*Modbus function 5, Force Single Coil/Write Coil.*

- `int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)`  
*Modbus function 15 (0F Hex), Force Multiple Coils.*

## 16-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 3, Read Holding Registers/Read Multiple Registers.*
- `int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 4, Read Input Registers.*
- `int writeSingleRegister (int slaveAddr, int regAddr, short regVal)`  
*Modbus function 6, Preset Single Register/Write Single Register.*
- `int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)`  
*Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.*
- `int maskWriteRegister (int slaveAddr, int regAddr, short andMask, short orMask)`  
*Modbus function 22 (16 Hex), Mask Write Register.*
- `int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)`  
*Modbus function 23 (17 Hex), Read/Write Registers.*

## 32-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.*
- `int readInputLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.*
- `int writeMultipleLongInts (int slaveAddr, int startRef, const int int32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.*
- `int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.*
- `int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit float data types, Read Input Registers as float data.*
- `int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.*
- `int readMultipleMod10000 (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/↔ Read Multiple Registers as modulo-10000 long int data.*

- `int readInputMod10000 (int slaveAddr, int startRef, int int32Arr[], int refCnt)`  
*Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.*
- `int writeMultipleMod10000 (int slaveAddr, int startRef, const int int32Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.*

## 64-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readHoldingRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int readInputRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int writeMultipleRegistersI64 (int slaveAddr, int startRef, const int64_t int64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.*
- `int readHoldingRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.*
- `int readInputRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.*
- `int writeMultipleRegistersF64 (int slaveAddr, int startRef, const double double64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.*

## File Record Access

- `int readFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, short recordArr[], int recordCnt)`  
*Modbus function 20, Read File Record.*
- `int readFileRecord (int slaveAddr, FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 20, Read File Record.*
- `int writeFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, const short recordArr[], int recordCnt)`  
*Modbus function 21, Write File Record.*
- `int writeFileRecord (int slaveAddr, const FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 21, Write File Record.*

## Diagnostics

- `int readExceptionStatus (int slaveAddr, unsigned char *statusBytePtr)`  
*Modbus function 7, Read Exception Status.*
- `int returnQueryData (int slaveAddr, const unsigned char queryArr[], unsigned char echoArr[], int len)`  
*Modbus function code 8, sub-function 00, Return Query Data.*
- `int restartCommunicationsOption (int slaveAddr, int clearEventLog)`  
*Modbus function code 8, sub-function 01, Restart Communications Option.*
- `int readDeviceIdentification (int slaveAddr, int accessType, int objId, DeviceIdObjectList *objListPtr)`  
*Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.*

## Custom Function Codes

- `int customFunction (int slaveAddr, int functionCode, void *requestData, size_t requestLen, void *responseData, size_t *responseLenPtr)`  
*User Defined Function Code*  
*This method can be used to implement User Defined Function Codes.*

## Protocol Configuration

- `int setTimeout (int timeOut)`  
*Configures time-out.*
- `int getTimeout ()`  
*Returns the time-out value.*
- `int setPollDelay (int pollDelay)`  
*Configures poll delay.*
- `int getPollDelay ()`  
*Returns the poll delay time.*
- `int setRetryCnt (int retryCnt)`  
*Configures the automatic retry setting.*
- `int getRetryCnt ()`  
*Returns the automatic retry count.*

## Transmission Statistic Functions

- `long getTotalCounter ()`  
*Returns how often a message transfer has been executed.*
- `void resetTotalCounter ()`  
*Resets total message transfer counter.*
- `long getSuccessCounter ()`  
*Returns how often a message transfer was successful.*
- `void resetSuccessCounter ()`  
*Resets successful message transfer counter.*



## Slave Configuration

- `void configureBigEndianInts ()`  
*Configures 32-bit int data type functions to do a word swap.*
- `int configureBigEndianInts (int slaveAddr)`  
*Enables int data type functions to do a word swap on a per slave basis.*
- `void configureLittleEndianInts ()`  
*Configures 32-bit int data type functions not to do a word swap.*
- `int configureLittleEndianInts (int slaveAddr)`  
*Disables word swapping for int data type functions on a per slave basis.*
- `void configureleeeeFloats ()`  
*Configures float data type functions not to do a word swap.*
- `int configureleeeeFloats (int slaveAddr)`  
*Disables float data type functions to do a word swap on a per slave basis.*
- `void configureSwappedFloats ()`  
*Configures float data type functions to do a word swap.*
- `int configureSwappedFloats (int slaveAddr)`  
*Enables float data type functions to do a word swap on a per slave basis.*
- `void configureStandard32BitMode ()`  
*Configures all slaves for Standard 32-bit Mode.*
- `int configureStandard32BitMode (int slaveAddr)`  
*Configures a slave for Standard 32-bit Register Mode.*
- `void configureEnron32BitMode ()`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `int configureEnron32BitMode (int slaveAddr)`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `void configureCountFromOne ()`  
*Configures the reference counting scheme to start with one for all slaves.*
- `int configureCountFromOne (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with one.*
- `void configureCountFromZero ()`  
*Configures the reference counting scheme to start with zero for all slaves.*
- `int configureCountFromZero (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with zero.*

## Utility Functions

- `static const TCHAR * getPackageVersion ()`  
*Returns the library version number.*

## Advantec ADAM 5000/6000 Series Commands

- `int adamSendReceiveAsciiCmd (const char *const commandSz, char *responseSz)`  
*Send/Receive ADAM 5000/6000 ASCII command.*

## 8.4.1 Detailed Description

MODBUS/TCP Master Protocol class.

This class realises the MODBUS/TCP master protocol. It provides functions to establish and to close a TCP/IP connection to the slave as well as data and control functions which can be used after a connection to a slave device has been established successfully. The data and control functions are organized into different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Modbus Protocol Flavours.

It is also possible to instantiate multiple instances of this class for establishing multiple connections to either the same or different hosts.

### See also

Data and Control Functions for all Modbus Protocol Flavours, IP based Protocols

## 8.4.2 Member Function Documentation

```
openProtocol() [1/2] int openProtocol (
    const char *const hostName )
```

Connects to a MODBUS/TCP slave.

This function establishes a logical network connection between master and slave. After a connection has been established data and control functions can be used. A TCP/IP connection should be closed if it is no longer needed.

### Note

The default time-out for the connection is 1000 ms.

### Parameters

<i>hostName</i>	String with IP address or host name
-----------------	-------------------------------------

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

The default TCP port number is 502.

```
setPort() int setPort (
    unsigned short portNo )
```

Sets the TCP port number to be used by the protocol.

Usually the port number remains unchanged from its default. In this case no call to this function is necessary. However if the port number has to be different, this function must be called *before* opening the connection with `openProtocol()`.

#### Parameters

<i>portNo</i>	Port number to be used when opening the connection
---------------	--

#### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_STATE_ERR</i> OR	Protocol already open

#### Note

Defaults to 502.

#### Remarks

Usually the port number remains unchanged and defaults to 502. In this case no call to this function is necessary. However if the port number has to be different from 502 this function must be called *before* opening the connection with `openProtocol()`.

```
openProtocol() [2/2] int openProtocol (
    const char *const hostName,
    const char *const interfaceIp ) [inherited]
```

Opens a connection to a slave device.

This function establishes a logical network connection between master and slave. After a connection has been established data and control functions can be used. A TCP/IP connection should be closed if it is no longer needed.

#### Note

The default time-out for the connection is 1000 ms.

#### Parameters

<i>hostName</i>	String with IP address or host name
<i>interfaceIp</i>	String with IP address for a specific host interface or NULL if connections are accepted on any interface*

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

References FTALK\_ILLEGAL\_STATE\_ERROR, FTALK\_TCPIP\_CONNECT\_ERR, and MbusIp↔ClientBase::isOpen().

**isOpen()** `int isOpen ( ) [virtual], [inherited]`

**Deprecated** Returns whether the protocol is open or not.

This simply returns the state of an internal flag whether openProtocol() has been called or not.

### Note

This function does not report the state of the underlying transport network, in particular it does not report whether the TCP connection is still in 'established' state or in fully 'closed' state.

Return values

<i>true</i>	= open
<i>false</i>	= closed

Implements MbusMasterFunctions.

**setClosingTimeout()** `int setClosingTimeout ( int msTime ) [inherited]`

Applies a time-out to socket closure and makes closeProtocol() wait for the server to acknowledge closing before potentially opening a new one.

### Note

A protocol must be closed in order to configure it.

### Parameters

<i>msTime</i>	Timeout value in ms (Range: 1 - 100000)
---------------	---

Return values

<i>FTALK_SUCCESS</i>	Success
----------------------	---------

Return values

<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ OR	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and *MbusIpClientBase::isOpen()*.

**getPort()** unsigned short getPort ( ) [inline], [inherited]

Returns the TCP port number used by the protocol.

#### Returns

Port number used by the protocol

**readCoils()** int readCoils (   
     int *slaveAddr*,   
     int *startRef*,   
     int *bitArr*[],   
     int *refCnt* ) [inherited]

Modbus function 1, Read Coil Status/Read Coils.

Reads the contents of the discrete outputs (coils, 0:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

#### Returns

*FTALK\_SUCCESS* on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
readInputDiscretes() int readInputDiscretes (  
    int slaveAddr,  
    int startRef,  
    int bitArr[],  
    int refCnt ) [inherited]
```

Modbus function 2, Read Inputs Status/Read Input Discretes.

Reads the contents of the discrete inputs (input status, 1:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
writeCoil() int writeCoil (  
    int slaveAddr,  
    int bitAddr,  
    int bitVal ) [inherited]
```

Modbus function 5, Force Single Coil/Write Coil.

Sets a single discrete output variable (coil, 0:00000 table) to either ON or OFF.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>bitAddr</i>	Coil address (Range: 1 - 65536)
<i>bitVal</i>	true sets, false clears discrete output variable

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
forceMultipleCoils() int forceMultipleCoils (
    int slaveAddr,
    int startRef,
    const int bitArr[],
    int refCnt ) [inherited]
```

Modbus function 15 (0F Hex), Force Multiple Coils.

Writes binary values into a sequence of discrete outputs (coils, 0:00000 table).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which contains the data to be sent
<i>refCnt</i>	Number of coils to be written (Range: 1-1968)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleRegisters() int readMultipleRegisters (
    int slaveAddr,
    int startRef,
    short regArr[],
    int refCnt ) [inherited]
```

Modbus function 3, Read Holding Registers/Read Multiple Registers.

Reads the contents of the output registers (holding registers, 4:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readInputRegisters() int readInputRegisters (  
    int slaveAddr,  
    int startRef,  
    short regArr[],  
    int refCnt ) [inherited]
```

Modbus function 4, Read Input Registers.

Read the contents of the input registers (3:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read.
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeSingleRegister() int writeSingleRegister (  
    int slaveAddr,
```



```
int regAddr,
short regVal ) [inherited]
```

Modbus function 6, Preset Single Register/Write Single Register.

Writes a value into a single output register (holding register, 4:00000 reference).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>regVal</i>	Data to be sent

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
writeMultipleRegisters() int writeMultipleRegisters (
    int slaveAddr,
    int startRef,
    const short regArr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.

Writes values into a sequence of output registers (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer with the data to be sent.
<i>refCnt</i>	Number of registers to be written (Range: 1-123)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

```
maskWriteRegister() int maskWriteRegister (
    int slaveAddr,
    int regAddr,
    short andMask,
    short orMask ) [inherited]
```

Modbus function 22 (16 Hex), Mask Write Register.

Masks bits according to an AND & an OR mask into a single output register (holding register, 4:00000 reference). Masking is done as follows: result = (currentVal AND andMask) OR (orMask AND (NOT andMask))

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>andMask</i>	Mask to be applied as a logic AND to the register
<i>orMask</i>	Mask to be applied as a logic OR to the register

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readWriteRegisters() int readWriteRegisters (
    int slaveAddr,
    int readRef,
    short readArr[],
    int readCnt,
    int writeRef,
    const short writeArr[],
    int writeCnt ) [inherited]
```

Modbus function 23 (17 Hex), Read/Write Registers.

Combines reading and writing of the output registers in one transaction (holding registers, 4:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>readRef</i>	Start register for reading (Range: 1 - 65536)
<i>readArr</i>	Buffer which will contain the data read
<i>readCnt</i>	Number of registers to be read (Range: 1-125)
<i>writeRef</i>	Start register for writing (Range: 1 - 65536)
<i>writeArr</i>	Buffer with data to be sent
<i>writeCnt</i>	Number of registers to be written (Range: 1-121)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleLongInts() int readMultipleLongInts (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into 32-bit long int values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readInputLongInts() int readInputLongInts (  
    int slaveAddr,  
    int startRef,  
    int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) into 32-bit long int values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleLongInts() int writeMultipleLongInts (  
    int slaveAddr,  
    int startRef,
```

```

const int int32Arr[],
int refCnt ) [inherited]

```

Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/↔ Write Multiple Registers with long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```

readMultipleFloats() int readMultipleFloats (
    int slaveAddr,
    int startRef,
    float float32Arr[],
    int refCnt ) [inherited]

```

Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into float values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputFloats() int readInputFloats (
    int slaveAddr,
    int startRef,
    float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit float data types, Read Input Registers as float data.  
 Reads the contents of pairs of consecutive input registers (3:00000 table) into float values.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleFloats() int writeMultipleFloats (
    int slaveAddr,
    int startRef,
    const float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

Writes float values into pairs of output registers (holding registers, 4:00000 table).

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleMod10000() int readMultipleMod10000 (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value into 32-bit int values and performs number format conversion.

#### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
readInputMod10000() int readInputMod10000 (  
    int slaveAddr,  
    int startRef,  
    int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) representing a modulo-10000 long int value into 32-bit long int values and performs number format conversion.

#### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)



### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleMod10000() int writeMultipleMod10000 (
    int slaveAddr,
    int startRef,
    const int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value and performs number format conversion.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersI64() int readHoldingRegistersI64 (
    int slaveAddr,
```

```
int startRef,
int64_t int64Arr[],
int refCnt ) [inherited]
```

Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.

Reads the contents of quartets of consecutive holding registers (4:00000 table) into 64-bit integer values.

**Remarks**

Depending on the 32-bit Mode setting, an int64\_t will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputRegistersI64() int readInputRegistersI64 (
int slaveAddr,
int startRef,
int64_t int64Arr[],
int refCnt ) [inherited]
```

Modbus function 4 for 64-bit integer data, Read Holding Registers as int64\_t type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit integer values.

**Remarks**

Depending on the 32-bit Mode setting, an int64\_t will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleRegistersI64() int writeMultipleRegistersI64 (
    int slaveAddr,
    int startRef,
    const int64_t int64Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.

Writes for 64-bit integer values into quartets of holding registers (4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersF64() int readHoldingRegistersF64 (  
    int slaveAddr,  
    int startRef,  
    double double64Arr[],  
    int refCnt ) [inherited]
```

Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (4:00000 table) into 64-bit floating point values.

**Remarks**

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputRegistersF64() int readInputRegistersF64 (  
    int slaveAddr,  
    int startRef,  
    double double64Arr[],  
    int refCnt ) [inherited]
```

Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit floating point values.

#### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
writeMultipleRegistersF64() int writeMultipleRegistersF64 (
    int slaveAddr,
    int startRef,
    const double double64Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.

Writes for 64-bit floating point values into quartets of holding registers (4:00000 table).

#### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readFileRecord() [1/2] int readFileRecord (  
    int slaveAddr,  
    int refType,  
    int fileNo,  
    int recordNo,  
    short recordArr[],  
    int recordCnt ) [inherited]
```

Modbus function 20, Read File Record.

Performs a file record read for a single subrequest.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to read from, 0-based numbering.
<i>recordArr</i>	Buffer which will be filled with the data read
<i>recordCnt</i>	Number of records to read (Range 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readFileRecord() [2/2] int readFileRecord (  
    int slaveAddr,
```

```
FileSubRequest subReqArr[],
int subReqCnt ) [inherited]
```

Modbus function 20, Read File Record.

Performs a file record read for a list of subrequests.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to read (Range 1-35)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK↔\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
writeFileRecord() [1/2] int writeFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    const short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a single subrequest.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to write to, 0-based numbering.
<i>recordArr</i>	Buffer with the data to be sent.
<i>recordCnt</i>	Number of records to write (Range 1-122)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeFileRecord() [2/2] int writeFileRecord (
    int slaveAddr,
    const FileSubRequest subReqArr[],
    int subReqCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a list of subrequests.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to write (Range 1-27)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK↔\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
readExceptionStatus() int readExceptionStatus (
    int slaveAddr,
    unsigned char * statusBytePtr ) [inherited]
```

Modbus function 7, Read Exception Status.

Reads the eight exception status coils within the slave device.



### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>statusBytePtr</i>	Slave status byte. The meaning of this status byte is slave specific and varies from device to device.

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

```
returnQueryData() int returnQueryData (
    int slaveAddr,
    const unsigned char queryArr[],
    unsigned char echoArr[],
    int len ) [inherited]
```

Modbus function code 8, sub-function 00, Return Query Data.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>queryArr</i>	Buffer with data to be sent
<i>echoArr</i>	Buffer which will contain the data read
<i>len</i>	Number of bytes send sent and read back

### Returns

FTALK\_SUCCESS on success, FTALK\_INVALID\_REPLY\_ERROR if reply does not match query data or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

**restartCommunicationsOption()** int restartCommunicationsOption (  
int *slaveAddr*,  
int *clearEventLog* ) [inherited]

Modbus function code 8, sub-function 01, Restart Communications Option.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>clearEventLog</i>	Flag when set to one clears in addition the slave's communication even log.

#### Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

#### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

**readDeviceIdentification()** int readDeviceIdentification (  
int *slaveAddr*,  
int *accessType*,  
int *objId*,  
DeviceIdObjectList \* *objListPtr* ) [inherited]

Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.

This function allows a master to retrieve various objects with meta information about a slave device. The data type of the objects returned is a 0-terminated ASCII string.

Object Id	Object Name / Description
0x00	VendorName
0x01	ProductCode
0x02	MajorMinorRevision
0x03	VendorUrl
0x04	ProductName
0x05	ModelName
0x06	UserApplicationName
0x07 - 0x7F	<i>Reserved</i>
0x80 - 0xFF	Vendor specific private objects

## Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>accessType</i>	Category type (1=only mandatory basic objects, 2=all regular objects, 3=include vendor specific objects, 4=single object)
<i>objId</i>	An object ID from above table
<i>objListPtr</i>	A pointer to a object list container which will receive the retrieved objects.

## Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

## Note

No broadcast supported

Example retrieving all Device ID strings of basic device identification level (level 1):

```
DeviceIdObjectList objs;
int objId = 0; // Start with object 0
do
{
    result = mbusProtocol.readDeviceIdentification(1, 1, objId, &objs);

    if (result != FTALK_SUCCESS)
    {
        fprintf(stderr, "Error: %s!\n", getBusProtocolErrorText(result));
        exit(EXIT_FAILURE);
    }
    else
    {
        size_t objLen;
        char* objData;

        while (objs.getNext(&objId, &objData, &objLen))
        {
            printf("obj[%d] = '%s', len = %u\n", objId, objData, objLen);
        }
    }
}
while (objs.moreToFollow());
```

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_FRAME\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
setTimeout() int setTimeout (
    int msTime ) [inherited]
```

Configures time-out.

This function sets the operation or socket time-out to the specified value.

### Remarks

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

### Note

A protocol must be closed in order to configure it.

### Parameters

<i>msTime</i>	Timeout value in ms (Range: 1 - 100000)
---------------	---

### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERROR</i> <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getTimeout()** `int getTimeout ( ) [inline], [inherited]`

Returns the time-out value.

### Remarks

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

### Returns

Timeout value in ms

**setPollDelay()** `int setPollDelay (`  
`int msTime ) [inherited]`

Configures poll delay.

This function sets the delay time which applies between two consecutive Modbus read/write. A value of 0 disables the poll delay.

### Remarks

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

### Note

A protocol must be closed in order to configure it.

### Parameters

<i>msTime</i>	Delay time in ms (Range: 0 - 100000), 0 disables poll delay
---------------	---

### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ OR	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getPollDelay()** `int getPollDelay ( ) [inline], [inherited]`

Returns the poll delay time.

### Returns

Delay time in ms, 0 if poll delay is switched off

**setRetryCnt()** `int setRetryCnt ( int retries ) [inherited]`

Configures the automatic retry setting.

A value of 0 disables any automatic retries.

### Note

A protocol must be closed in order to configure it.

### Parameters

<i>retries</i>	Retry count (Range: 0 - 10), 0 disables retries
----------------	---

### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getRetryCnt()** `int getRetryCnt ( ) [inline], [inherited]`

Returns the automatic retry count.

### Returns

Retry count

**getTotalCounter()** `long getTotalCounter ( ) [inline], [inherited]`

Returns how often a message transfer has been executed.

### Returns

Counter value

**getSuccessCounter()** `long getSuccessCounter ( ) [inline], [inherited]`

Returns how often a message transfer was successful.

### Returns

Counter value

**configureBigEndianInts()** [1/2] void configureBigEndianInts ( ) [inherited]

Configures 32-bit int data type functions to do a word swap.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian slave.

**configureBigEndianInts()** [2/2] int configureBigEndianInts (   
 int *slaveAddr* ) [inherited]

Enables int data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian machine.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureLittleEndianInts()** [1/2] void configureLittleEndianInts ( ) [inherited]

Configures 32-bit int data type functions not to do a word swap.

This is the default.

**configureLittleEndianInts()** [2/2] int configureLittleEndianInts (   
 int *slaveAddr* ) [inherited]

Disables word swapping for int data type functions on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit data in little little-endian word order.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureIeeeFloats()** [1/2] void configureIeeeFloats ( ) [inherited]

Configures float data type functions not to do a word swap.  
This is the default.

**configureIeeeFloats()** [2/2] int configureIeeeFloats (   
int *slaveAddr* ) [inherited]

Disables float data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit floats in little little-endian word order which is the most common case.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureSwappedFloats()** [1/2] void configureSwappedFloats ( ) [inherited]

Configures float data type functions to do a word swap.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

#### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

**configureSwappedFloats()** [2/2] int configureSwappedFloats (   
int *slaveAddr* ) [inherited]

Enables float data type functions to do a word swap on a per slave basis.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

#### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.



### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureStandard32BitMode()** [1/2] void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

### Note

This function call also re-configures the endianness to default little-endian for 32-bit values!

### Remarks

This is the default mode

**configureStandard32BitMode()** [2/2] int configureStandard32BitMode ( int *slaveAddr* ) [inherited]

Configures a slave for Standard 32-bit Register Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range

### Remarks

This is the default mode

**Note**

This function call also re-configures the endianness to default little-endian for 32-bit values!

References MbusMasterFunctions::configureIEEEFloats(), MbusMasterFunctions::configureLittleEndianInts(), FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureEnron32BitMode()** [1/2] void configureEnron32BitMode ( ) [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

**Note**

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

**configureEnron32BitMode()** [2/2] int configureEnron32BitMode ( int *slaveAddr* ) [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERROR</i> OR	Argument out of range

**Note**

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

References MbusMasterFunctions::configureBigEndianInts(), MbusMasterFunctions::configureSwappedFloats(), FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureCountFromOne()** [1/2] void configureCountFromOne ( ) [inherited]

Configures the reference counting scheme to start with one for all slaves.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Remarks

This is the default mode

**configureCountFromOne()** [2/2] int configureCountFromOne (   
 int *slaveAddr* ) [inherited]

Configures a slave's reference counting scheme to start with one.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

#### Remarks

This is the default mode

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureCountFromZero()** [1/2] void configureCountFromZero ( ) [inherited]

Configures the reference counting scheme to start with zero for all slaves.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

This renders the first register to be #0 for all slaves.

**configureCountFromZero()** [2/2] int configureCountFromZero (   
 int *slaveAddr* ) [inherited]

Configures a slave's reference counting scheme to start with zero.

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**getPackageVersion()** `const TCHAR * getPackageVersion ( ) [static], [inherited]`

Returns the library version number.

#### Returns

Library version string

## 8.5 MbusRtuOverTcpMasterProtocol Class Reference

Encapsulated Modbus RTU Master Protocol class.

### Public Member Functions

- `MbusRtuOverTcpMasterProtocol ( )`  
*Constructs a MbusRtuOverTcpMasterProtocol object and initialises its data.*
- `int openProtocol (const char *const hostName)`  
*Connects to a Modbus RTU slave via TCP/IP.*
- `int setPort (unsigned short portNo)`  
*Sets the TCP port number used to connect to the Modbus RTU slave device.*
- `int openProtocol (const char *const hostName, const char *const interfacelp)`  
*Opens a connection to a slave device.*
- `virtual void closeProtocol ( )`  
*Closes a TCP/IP connection to a slave and releases any system resources associated with the connection.*
- `virtual int isOpen ( )`
- `int setClosingTimeout (int msTime)`  
*Applies a time-out to socket closure and makes closeProtocol() wait for the server to acknowledge closing before potentially opening a new one.*
- `unsigned short getPort ( )`  
*Returns the TCP port number used by the protocol.*

### Bit Access

Table 0:0000 (Coils) and Table 1:0000 (Input Status)

- `int readCoils (int slaveAddr, int startRef, int bitArr[ ], int refCnt)`  
*Modbus function 1, Read Coil Status/Read Coils.*
- `int readInputDiscretes (int slaveAddr, int startRef, int bitArr[ ], int refCnt)`  
*Modbus function 2, Read Inputs Status/Read Input Discretes.*
- `int writeCoil (int slaveAddr, int bitAddr, int bitVal)`

*Modbus function 5, Force Single Coil/Write Coil.*

- `int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)`  
*Modbus function 15 (0F Hex), Force Multiple Coils.*

## 16-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 3, Read Holding Registers/Read Multiple Registers.*
- `int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 4, Read Input Registers.*
- `int writeSingleRegister (int slaveAddr, int regAddr, short regVal)`  
*Modbus function 6, Preset Single Register/Write Single Register.*
- `int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)`  
*Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.*
- `int maskWriteRegister (int slaveAddr, int regAddr, short andMask, short orMask)`  
*Modbus function 22 (16 Hex), Mask Write Register.*
- `int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)`  
*Modbus function 23 (17 Hex), Read/Write Registers.*

## 32-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.*
- `int readInputLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.*
- `int writeMultipleLongInts (int slaveAddr, int startRef, const int int32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.*
- `int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.*
- `int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit float data types, Read Input Registers as float data.*
- `int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.*
- `int readMultipleMod10000 (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/↔ Read Multiple Registers as modulo-10000 long int data.*

- `int readInputMod10000 (int slaveAddr, int startRef, int int32Arr[], int refCnt)`  
*Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.*
- `int writeMultipleMod10000 (int slaveAddr, int startRef, const int int32Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.*

## 64-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readHoldingRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int readInputRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int writeMultipleRegistersI64 (int slaveAddr, int startRef, const int64_t int64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.*
- `int readHoldingRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.*
- `int readInputRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.*
- `int writeMultipleRegistersF64 (int slaveAddr, int startRef, const double double64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.*

## File Record Access

- `int readFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, short recordArr[], int recordCnt)`  
*Modbus function 20, Read File Record.*
- `int readFileRecord (int slaveAddr, FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 20, Read File Record.*
- `int writeFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, const short recordArr[], int recordCnt)`  
*Modbus function 21, Write File Record.*
- `int writeFileRecord (int slaveAddr, const FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 21, Write File Record.*

## Diagnostics

- `int readExceptionStatus (int slaveAddr, unsigned char *statusBytePtr)`  
*Modbus function 7, Read Exception Status.*
- `int returnQueryData (int slaveAddr, const unsigned char queryArr[], unsigned char echoArr[], int len)`  
*Modbus function code 8, sub-function 00, Return Query Data.*
- `int restartCommunicationsOption (int slaveAddr, int clearEventLog)`  
*Modbus function code 8, sub-function 01, Restart Communications Option.*
- `int readDeviceIdentification (int slaveAddr, int accessType, int objId, DeviceIdObjectList *objListPtr)`  
*Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.*

## Custom Function Codes

- `int customFunction (int slaveAddr, int functionCode, void *requestData, size_t requestLen, void *responseData, size_t *responseLenPtr)`  
*User Defined Function Code*  
*This method can be used to implement User Defined Function Codes.*

## Protocol Configuration

- `int setTimeout (int timeOut)`  
*Configures time-out.*
- `int getTimeout ()`  
*Returns the time-out value.*
- `int setPollDelay (int pollDelay)`  
*Configures poll delay.*
- `int getPollDelay ()`  
*Returns the poll delay time.*
- `int setRetryCnt (int retryCnt)`  
*Configures the automatic retry setting.*
- `int getRetryCnt ()`  
*Returns the automatic retry count.*

## Transmission Statistic Functions

- `long getTotalCounter ()`  
*Returns how often a message transfer has been executed.*
- `void resetTotalCounter ()`  
*Resets total message transfer counter.*
- `long getSuccessCounter ()`  
*Returns how often a message transfer was successful.*
- `void resetSuccessCounter ()`  
*Resets successful message transfer counter.*

## Slave Configuration

- `void configureBigEndianInts ()`  
*Configures 32-bit int data type functions to do a word swap.*
- `int configureBigEndianInts (int slaveAddr)`  
*Enables int data type functions to do a word swap on a per slave basis.*
- `void configureLittleEndianInts ()`  
*Configures 32-bit int data type functions not to do a word swap.*
- `int configureLittleEndianInts (int slaveAddr)`  
*Disables word swapping for int data type functions on a per slave basis.*
- `void configureleeeeFloats ()`  
*Configures float data type functions not to do a word swap.*
- `int configureleeeeFloats (int slaveAddr)`  
*Disables float data type functions to do a word swap on a per slave basis.*
- `void configureSwappedFloats ()`  
*Configures float data type functions to do a word swap.*
- `int configureSwappedFloats (int slaveAddr)`  
*Enables float data type functions to do a word swap on a per slave basis.*
- `void configureStandard32BitMode ()`  
*Configures all slaves for Standard 32-bit Mode.*
- `int configureStandard32BitMode (int slaveAddr)`  
*Configures a slave for Standard 32-bit Register Mode.*
- `void configureEnron32BitMode ()`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `int configureEnron32BitMode (int slaveAddr)`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `void configureCountFromOne ()`  
*Configures the reference counting scheme to start with one for all slaves.*
- `int configureCountFromOne (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with one.*
- `void configureCountFromZero ()`  
*Configures the reference counting scheme to start with zero for all slaves.*
- `int configureCountFromZero (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with zero.*

## Utility Functions

- `static const TCHAR * getPackageVersion ()`  
*Returns the library version number.*



## 8.5.1 Detailed Description

Encapsulated Modbus RTU Master Protocol class.

This class realises the Encapsulated Modbus RTU master protocol. This protocol is also known as RTU over TCP or RTU/IP and used for example by ISaGRAF Soft-PLCs. This class provides functions to establish and to close a TCP/IP connection to the slave as well as data and control functions which can be used after a connection to a slave device has been established successfully. The data and control functions are organized different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Modbus Protocol Flavours.

It is also possible to instantiate multiple instances of this class for establishing multiple connections to either the same or different hosts.

### See also

Data and Control Functions for all Modbus Protocol Flavours, IP based Protocols  
MbusRtuMasterProtocol

## 8.5.2 Member Function Documentation

```
openProtocol() [1/2] int openProtocol (  
    const char *const hostName )
```

Connects to a Modbus RTU slave via TCP/IP.

This function establishes a logical network connection between master and slave. After a connection has been established data and control functions can be used. A TCP/IP connection should be closed if it is no longer needed.

### Note

The default time-out for the connection is 1000 ms.

### Parameters

<i>hostName</i>	String with IP address or host name
-----------------	-------------------------------------

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

The default TCP port number is 1100.

**setPort()** int setPort (   
 unsigned short *portNo* )

Sets the TCP port number used to connect to the Modbus RTU slave device.

Usually the port number remains unchanged from its default. In this case no call to this function is necessary. However if the port number has to be different, this function must be called *before* opening the connection with openProtocol().

#### Parameters

<i>port</i> ↔ <i>No</i>	Port number to be used when opening the connection
----------------------------	--

#### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_STATE_ERR</i> ↔ <i>OR</i>	Protocol already open

#### Note

Defaults to 1100.

**openProtocol()** [2/2] int openProtocol (   
 const char \*const *hostName*,   
 const char \*const *interfaceIp* ) [inherited]

Opens a connection to a slave device.

This function establishes a logical network connection between master and slave. After a connection has been established data and control functions can be used. A TCP/IP connection should be closed if it is no longer needed.

#### Note

The default time-out for the connection is 1000 ms.

#### Parameters

<i>hostName</i>	String with IP address or host name
<i>interfaceIp</i>	String with IP address for a specific host interface or NULL if connections are accepted on any interface*

#### Returns

*FTALK\_SUCCESS* on success or error code. See Error Management for a list of error codes.

References FTALK\_ILLEGAL\_STATE\_ERROR, FTALK\_TCPIP\_CONNECT\_ERR, and MbusIp↔ ClientBase::isOpen().

**isOpen()** `int isOpen ( ) [virtual], [inherited]`

**Deprecated** Returns whether the protocol is open or not.

This simply returns the state of an internal flag whether openProtocol() has been called or not.

#### Note

This function does not report the state of the underlying transport network, in particular it does not report whether the TCP connection is still in 'established' state or in fully 'closed' state.

Return values

<i>true</i>	= open
<i>false</i>	= closed

Implements MbusMasterFunctions.

**setClosingTimeout()** `int setClosingTimeout ( int msTime ) [inherited]`

Applies a time-out to socket closure and makes closeProtocol() wait for the server to acknowledge closing before potentially opening a new one.

#### Note

A protocol must be closed in order to configure it.

#### Parameters

<i>msTime</i>	Timeout value in ms (Range: 1 - 100000)
---------------	---

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_ILLEGAL\_STATE\_ERROR, FTALK\_↔ SUCCESS, and MbusIpClientBase::isOpen().

**getPort()** unsigned short getPort ( ) [inline], [inherited]

Returns the TCP port number used by the protocol.

#### Returns

Port number used by the protocol

**readCoils()** int readCoils (   
    int *slaveAddr*,   
    int *startRef*,   
    int *bitArr*[],   
    int *refCnt* ) [inherited]

Modbus function 1, Read Coil Status/Read Coils.

Reads the contents of the discrete outputs (coils, 0:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

**readInputDiscretes()** int readInputDiscretes (   
    int *slaveAddr*,   
    int *startRef*,   
    int *bitArr*[],   
    int *refCnt* ) [inherited]

Modbus function 2, Read Inputs Status/Read Input Discretes.

Reads the contents of the discrete inputs (input status, 1:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeCoil() int writeCoil (  
                int slaveAddr,  
                int bitAddr,  
                int bitVal ) [inherited]
```

Modbus function 5, Force Single Coil/Write Coil.

Sets a single discrete output variable (coil, 0:00000 table) to either ON or OFF.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>bitAddr</i>	Coil address (Range: 1 - 65536)
<i>bitVal</i>	true sets, false clears discrete output variable

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
forceMultipleCoils() int forceMultipleCoils (
    int slaveAddr,
    int startRef,
    const int bitArr[],
    int refCnt ) [inherited]
```

Modbus function 15 (0F Hex), Force Multiple Coils.

Writes binary values into a sequence of discrete outputs (coils, 0:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which contains the data to be sent
<i>refCnt</i>	Number of coils to be written (Range: 1-1968)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleRegisters() int readMultipleRegisters (
    int slaveAddr,
    int startRef,
    short regArr[],
    int refCnt ) [inherited]
```

Modbus function 3, Read Holding Registers/Read Multiple Registers.

Reads the contents of the output registers (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputRegisters() int readInputRegisters (
    int slaveAddr,
    int startRef,
    short regArr[],
    int refCnt ) [inherited]
```

Modbus function 4, Read Input Registers.

Read the contents of the input registers (3:00000 table).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read.
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeSingleRegister() int writeSingleRegister (
    int slaveAddr,
    int regAddr,
    short regVal ) [inherited]
```

Modbus function 6, Preset Single Register/Write Single Register.

Writes a value into a single output register (holding register, 4:00000 reference).



### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>regVal</i>	Data to be sent

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
writeMultipleRegisters() int writeMultipleRegisters (
    int slaveAddr,
    int startRef,
    const short regArr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.

Writes values into a sequence of output registers (holding registers, 4:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer with the data to be sent.
<i>refCnt</i>	Number of registers to be written (Range: 1-123)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

```
maskWriteRegister() int maskWriteRegister (
    int slaveAddr,
    int regAddr,
    short andMask,
    short orMask ) [inherited]
```

Modbus function 22 (16 Hex), Mask Write Register.

Masks bits according to an AND & an OR mask into a single output register (holding register, 4:00000 reference). Masking is done as follows: result = (currentVal AND andMask) OR (orMask AND (NOT andMask))

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>andMask</i>	Mask to be applied as a logic AND to the register
<i>orMask</i>	Mask to be applied as a logic OR to the register

#### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readWriteRegisters() int readWriteRegisters (
    int slaveAddr,
    int readRef,
    short readArr[],
    int readCnt,
    int writeRef,
    const short writeArr[],
    int writeCnt ) [inherited]
```

Modbus function 23 (17 Hex), Read/Write Registers.

Combines reading and writing of the output registers in one transaction (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>readRef</i>	Start register for reading (Range: 1 - 65536)
<i>readArr</i>	Buffer which will contain the data read
<i>readCnt</i>	Number of registers to be read (Range: 1-125)
<i>writeRef</i>	Start register for writing (Range: 1 - 65536)

### Parameters

<i>writeArr</i>	Buffer with data to be sent
<i>writeCnt</i>	Number of registers to be written (Range: 1-121)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleLongInts() int readMultipleLongInts (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into 32-bit long int values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputLongInts() int readInputLongInts (  
    int slaveAddr,  
    int startRef,  
    int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) into 32-bit long int values.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleLongInts() int writeMultipleLongInts (  
    int slaveAddr,  
    int startRef,  
    const int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/↔ Write Multiple Registers with long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleFloats() int readMultipleFloats (
    int slaveAddr,
    int startRef,
    float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into float values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputFloats() int readInputFloats (
    int slaveAddr,
    int startRef,
    float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit float data types, Read Input Registers as float data. Reads the contents of pairs of consecutive input registers (3:00000 table) into float values.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleFloats() int writeMultipleFloats (
    int slaveAddr,
    int startRef,
    const float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

Writes float values into pairs of output registers (holding registers, 4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleMod10000() int readMultipleMod10000 (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value into 32-bit int values and performs number format conversion.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputMod10000() int readInputMod10000 (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) representing a modulo-10000 long int value into 32-bit long int values and performs number format conversion.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.



**Note**

No broadcast supported

```
writeMultipleMod10000() int writeMultipleMod10000 (
    int slaveAddr,
    int startRef,
    const int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value and performs number format conversion.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values values to be sent (Range: 1-61)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersI64() int readHoldingRegistersI64 (
    int slaveAddr,
    int startRef,
    int64_t int64Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.

Reads the contents of quartets of consecutive holding registers (4:00000 table) into 64-bit integer values.

#### Remarks

Depending on the 32-bit Mode setting, an `int64_t` will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
readInputRegistersI64() int readInputRegistersI64 (
    int slaveAddr,
    int startRef,
    int64_t int64Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 64-bit integer data, Read Holding Registers as `int64_t` type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit integer values.

#### Remarks

Depending on the 32-bit Mode setting, an `int64_t` will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleRegistersI64() int writeMultipleRegistersI64 (
    int slaveAddr,
    int startRef,
    const int64_t int64Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.

Writes for 64-bit integer values into quartets of holding registers (4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersF64() int readHoldingRegistersF64 (
    int slaveAddr,
```

```
int startRef,
double double64Arr[],
int refCnt ) [inherited]
```

Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (4:00000 table) into 64-bit floating point values.

**Remarks**

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputRegistersF64() int readInputRegistersF64 (
int slaveAddr,
int startRef,
double double64Arr[],
int refCnt ) [inherited]
```

Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit floating point values.

**Remarks**

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64</i> ↔ <i>Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleRegistersF64() int writeMultipleRegistersF64 (
    int slaveAddr,
    int startRef,
    const double double64Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.

Writes for 64-bit floating point values into quartets of holding registers (4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64</i> ↔ <i>Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readFileRecord() [1/2] int readFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 20, Read File Record.

Performs a file record read for a single subrequest.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to read from, 0-based numbering.
<i>recordArr</i>	Buffer which will be filled with the data read
<i>recordCnt</i>	Number of records to read (Range 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readFileRecord() [2/2] int readFileRecord (
    int slaveAddr,
    FileSubRequest subReqArr[],
    int subReqCnt ) [inherited]
```

Modbus function 20, Read File Record.

Performs a file record read for a list of subrequests.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to read (Range 1-35)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK↔\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
writeFileRecord() [1/2] int writeFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    const short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a single subrequest.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to write to, 0-based numbering.
<i>recordArr</i>	Buffer with the data to be sent.
<i>recordCnt</i>	Number of records to write (Range 1-122)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeFileRecord() [2/2] int writeFileRecord (
    int slaveAddr,
    const FileSubRequest subReqArr[],
    int subReqCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a list of subrequests.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReqCnt</i>	Number of subrequests to write (Range 1-27)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
readExceptionStatus() int readExceptionStatus (
    int slaveAddr,
    unsigned char * statusBytePtr ) [inherited]
```

Modbus function 7, Read Exception Status.

Reads the eight exception status coils within the slave device.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>statusBytePtr</i>	Slave status byte. The meaning of this status byte is slave specific and varies from device to device.



### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

```
returnQueryData() int returnQueryData (
    int slaveAddr,
    const unsigned char queryArr[],
    unsigned char echoArr[],
    int len ) [inherited]
```

Modbus function code 8, sub-function 00, Return Query Data.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>queryArr</i>	Buffer with data to be sent
<i>echoArr</i>	Buffer which will contain the data read
<i>len</i>	Number of bytes send sent and read back

### Returns

FTALK\_SUCCESS on success, FTALK\_INVALID\_REPLY\_ERROR if reply does not match query data or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
restartCommunicationsOption() int restartCommunicationsOption (
    int slaveAddr,
    int clearEventLog ) [inherited]
```

Modbus function code 8, sub-function 01, Restart Communications Option.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>clearEventLog</i>	Flag when set to one clears in addition the slave's communication even log.

**Returns**

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readDeviceIdentification() int readDeviceIdentification (
    int slaveAddr,
    int accessType,
    int objId,
    DeviceIdObjectList * objListPtr ) [inherited]
```

Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.

This function allows a master to retrieve various objects with meta information about a slave device. The data type of the objects returned is a 0-terminated ASCII string.

Object Id	Object Name / Description
0x00	VendorName
0x01	ProductCode
0x02	MajorMinorRevision
0x03	VendorUrl
0x04	ProductName
0x05	ModelName
0x06	UserApplicationName
0x07 - 0x7F	Reserved
0x80 - 0xFF	Vendor specific private objects

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>accessType</i>	Category type (1=only mandatory basic objects, 2=all regular objects, 3=include vendor specific objects, 4=single object)
<i>objId</i>	An object ID from above table

## Parameters

<i>objListPtr</i>	A pointer to a object list container which will receive the retrieved objects.
-------------------	--

## Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

## Note

No broadcast supported

Example retrieving all Device ID strings of basic device identification level (level 1):

```
DeviceIdObjectList objs;
int objId = 0; // Start with object 0
do
{
    result = mbusProtocol.readDeviceIdentification(1, 1, objId, &objs);

    if (result != FTALK_SUCCESS)
    {
        fprintf(stderr, "Error: %s!\n", getBusProtocolErrorText(result));
        exit(EXIT_FAILURE);
    }
    else
    {
        size_t objLen;
        char* objData;

        while (objs.getNext(&objId, &objData, &objLen))
        {
            printf("obj[%d] = '%s', len = %u\n", objId, objData, objLen);
        }
    }
}
while (objs.moreToFollow());
```

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_FRAME\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
setTimeout() int setTimeout (
    int msTime ) [inherited]
```

Configures time-out.

This function sets the operation or socket time-out to the specified value.

## Remarks

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

**Note**

A protocol must be closed in order to configure it.

**Parameters**

<i>msTime</i>	Timeout value in ms (Range: 1 - 100000)
---------------	---

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and *MbusMasterFunctions::isOpen()*.

**getTimeout()** `int getTimeout ( ) [inline], [inherited]`

Returns the time-out value.

**Remarks**

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

**Returns**

Timeout value in ms

**setPollDelay()** `int setPollDelay ( int msTime ) [inherited]`

Configures poll delay.

This function sets the delay time which applies between two consecutive Modbus read/write. A value of 0 disables the poll delay.

**Remarks**

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

**Note**

A protocol must be closed in order to configure it.

**Parameters**

<i>msTime</i>	Delay time in ms (Range: 0 - 100000), 0 disables poll delay
---------------	---

## Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getPollDelay()** `int getPollDelay ( ) [inline], [inherited]`

Returns the poll delay time.

**Returns**

Delay time in ms, 0 if poll delay is switched off

**setRetryCnt()** `int setRetryCnt ( int retries ) [inherited]`

Configures the automatic retry setting.

A value of 0 disables any automatic retries.

**Note**

A protocol must be closed in order to configure it.

**Parameters**

<i>retries</i>	Retry count (Range: 0 - 10), 0 disables retries
----------------	---

## Return values

<i>FTALK_SUCCESS</i>	Success
----------------------	---------

Return values

<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ OR	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getRetryCnt()** `int getRetryCnt ( ) [inline], [inherited]`

Returns the automatic retry count.

#### Returns

Retry count

**getTotalCounter()** `long getTotalCounter ( ) [inline], [inherited]`

Returns how often a message transfer has been executed.

#### Returns

Counter value

**getSuccessCounter()** `long getSuccessCounter ( ) [inline], [inherited]`

Returns how often a message transfer was successful.

#### Returns

Counter value

**configureBigEndianInts()** `[1/2] void configureBigEndianInts ( ) [inherited]`

Configures 32-bit int data type functions to do a word swap.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian slave.

**configureBigEndianInts()** [2/2] `int configureBigEndianInts (`  
`int slaveAddr )` [inherited]

Enables int data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian machine.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureLittleEndianInts()** [1/2] `void configureLittleEndianInts ( )` [inherited]

Configures 32-bit int data type functions not to do a word swap.

This is the default.

**configureLittleEndianInts()** [2/2] `int configureLittleEndianInts (`  
`int slaveAddr )` [inherited]

Disables word swapping for int data type functions on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit data in little little-endian word order.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureIeeeFloats()** [1/2] `void configureIeeeFloats ( )` [inherited]

Configures float data type functions not to do a word swap.

This is the default.

**configureIeeeFloats()** [2/2] `int configureIeeeFloats (`

`int slaveAddr ) [inherited]`

Disables float data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit floats in little little-endian word order which is the most common case.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureSwappedFloats()** [1/2] `void configureSwappedFloats ( ) [inherited]`

Configures float data type functions to do a word swap.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

#### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

**configureSwappedFloats()** [2/2] `int configureSwappedFloats ( int slaveAddr ) [inherited]`

Enables float data type functions to do a word swap on a per slave basis.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

#### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--



References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureStandard32BitMode()** [1/2] void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

#### Note

This function call also re-configures the endianness to default little-endian for 32-bit values!

#### Remarks

This is the default mode

**configureStandard32BitMode()** [2/2] int configureStandard32BitMode ( int *slaveAddr* ) [inherited]

Configures a slave for Standard 32-bit Register Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range

#### Remarks

This is the default mode

#### Note

This function call also re-configures the endianness to default little-endian for 32-bit values!

References MbusMasterFunctions::configureIEEEFloats(), MbusMasterFunctions::configureLittleEndianInts(), FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureEnron32BitMode()** [1/2] void configureEnron32BitMode ( ) [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

**Note**

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

**configureEnron32BitMode()** [2/2] int configureEnron32BitMode ( int *slaveAddr* ) [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERROR</i> OR	Argument out of range

**Note**

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

References MbusMasterFunctions::configureBigEndianInts(), MbusMasterFunctions::configureSwappedFloats(), FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureCountFromOne()** [1/2] void configureCountFromOne ( ) [inherited]

Configures the reference counting scheme to start with one for all slaves.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Remarks

This is the default mode

```
configureCountFromOne() [2/2] int configureCountFromOne (
    int slaveAddr ) [inherited]
```

Configures a slave's reference counting scheme to start with one.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

#### Remarks

This is the default mode

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

```
configureCountFromZero() [1/2] void configureCountFromZero ( ) [inherited]
```

Configures the reference counting scheme to start with zero for all slaves.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

This renders the first register to be #0 for all slaves.

```
configureCountFromZero() [2/2] int configureCountFromZero (
    int slaveAddr ) [inherited]
```

Configures a slave's reference counting scheme to start with zero.

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**getPackageVersion()** `const TCHAR * getPackageVersion ( ) [static], [inherited]`

Returns the library version number.

#### Returns

Library version string

## 8.6 MbusAsciiOverTcpMasterProtocol Class Reference

Modbus ASCII over TCP Master Protocol class.

### Public Member Functions

- `MbusAsciiOverTcpMasterProtocol ( )`  
*Constructs a MbusAsciiOverTcpMasterProtocol object and initialises its data.*
- `int openProtocol (const char *const hostName)`  
*Connects to a Modbus ASCII slave via TCP/IP.*
- `int setPort (unsigned short portNo)`  
*Sets the TCP port number used to connect to the Modbus ASCII slave device.*
- `int openProtocol (const char *const hostName, const char *const interfacelp)`  
*Opens a connection to a slave device.*
- `virtual void closeProtocol ( )`  
*Closes a TCP/IP connection to a slave and releases any system resources associated with the connection.*
- `virtual int isOpen ( )`
- `int setClosingTimeout (int msTime)`  
*Applies a time-out to socket closure and makes closeProtocol() wait for the server to acknowledge closing before potentially opening a new one.*
- `unsigned short getPort ( )`  
*Returns the TCP port number used by the protocol.*

### Bit Access

Table 0:0000 (Coils) and Table 1:0000 (Input Status)

- `int readCoils (int slaveAddr, int startRef, int bitArr[ ], int refCnt)`  
*Modbus function 1, Read Coil Status/Read Coils.*
- `int readInputDiscretets (int slaveAddr, int startRef, int bitArr[ ], int refCnt)`  
*Modbus function 2, Read Inputs Status/Read Input Discretets.*
- `int writeCoil (int slaveAddr, int bitAddr, int bitVal)`

*Modbus function 5, Force Single Coil/Write Coil.*

- `int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)`  
*Modbus function 15 (0F Hex), Force Multiple Coils.*

## 16-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 3, Read Holding Registers/Read Multiple Registers.*
- `int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 4, Read Input Registers.*
- `int writeSingleRegister (int slaveAddr, int regAddr, short regVal)`  
*Modbus function 6, Preset Single Register/Write Single Register.*
- `int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)`  
*Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.*
- `int maskWriteRegister (int slaveAddr, int regAddr, short andMask, short orMask)`  
*Modbus function 22 (16 Hex), Mask Write Register.*
- `int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)`  
*Modbus function 23 (17 Hex), Read/Write Registers.*

## 32-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.*
- `int readInputLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.*
- `int writeMultipleLongInts (int slaveAddr, int startRef, const int int32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.*
- `int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.*
- `int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit float data types, Read Input Registers as float data.*
- `int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.*
- `int readMultipleMod10000 (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/↔ Read Multiple Registers as modulo-10000 long int data.*

- `int readInputMod10000 (int slaveAddr, int startRef, int int32Arr[], int refCnt)`  
*Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.*
- `int writeMultipleMod10000 (int slaveAddr, int startRef, const int int32Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.*

## 64-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readHoldingRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int readInputRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int writeMultipleRegistersI64 (int slaveAddr, int startRef, const int64_t int64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.*
- `int readHoldingRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.*
- `int readInputRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.*
- `int writeMultipleRegistersF64 (int slaveAddr, int startRef, const double double64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.*

## File Record Access

- `int readFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, short recordArr[], int recordCnt)`  
*Modbus function 20, Read File Record.*
- `int readFileRecord (int slaveAddr, FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 20, Read File Record.*
- `int writeFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, const short recordArr[], int recordCnt)`  
*Modbus function 21, Write File Record.*
- `int writeFileRecord (int slaveAddr, const FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 21, Write File Record.*

## Diagnostics

- `int readExceptionStatus (int slaveAddr, unsigned char *statusBytePtr)`  
*Modbus function 7, Read Exception Status.*
- `int returnQueryData (int slaveAddr, const unsigned char queryArr[], unsigned char echoArr[], int len)`  
*Modbus function code 8, sub-function 00, Return Query Data.*
- `int restartCommunicationsOption (int slaveAddr, int clearEventLog)`  
*Modbus function code 8, sub-function 01, Restart Communications Option.*
- `int readDeviceIdentification (int slaveAddr, int accessType, int objId, DeviceIdObjectList *objListPtr)`  
*Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.*

## Custom Function Codes

- `int customFunction (int slaveAddr, int functionCode, void *requestData, size_t requestLen, void *responseData, size_t *responseLenPtr)`  
*User Defined Function Code*  
*This method can be used to implement User Defined Function Codes.*

## Protocol Configuration

- `int setTimeout (int timeOut)`  
*Configures time-out.*
- `int getTimeout ()`  
*Returns the time-out value.*
- `int setPollDelay (int pollDelay)`  
*Configures poll delay.*
- `int getPollDelay ()`  
*Returns the poll delay time.*
- `int setRetryCnt (int retryCnt)`  
*Configures the automatic retry setting.*
- `int getRetryCnt ()`  
*Returns the automatic retry count.*

## Transmission Statistic Functions

- `long getTotalCounter ()`  
*Returns how often a message transfer has been executed.*
- `void resetTotalCounter ()`  
*Resets total message transfer counter.*
- `long getSuccessCounter ()`  
*Returns how often a message transfer was successful.*
- `void resetSuccessCounter ()`  
*Resets successful message transfer counter.*

## Slave Configuration

- `void configureBigEndianInts ()`  
*Configures 32-bit int data type functions to do a word swap.*
- `int configureBigEndianInts (int slaveAddr)`  
*Enables int data type functions to do a word swap on a per slave basis.*
- `void configureLittleEndianInts ()`  
*Configures 32-bit int data type functions not to do a word swap.*
- `int configureLittleEndianInts (int slaveAddr)`  
*Disables word swapping for int data type functions on a per slave basis.*
- `void configureleeeeFloats ()`  
*Configures float data type functions not to do a word swap.*
- `int configureleeeeFloats (int slaveAddr)`  
*Disables float data type functions to do a word swap on a per slave basis.*
- `void configureSwappedFloats ()`  
*Configures float data type functions to do a word swap.*
- `int configureSwappedFloats (int slaveAddr)`  
*Enables float data type functions to do a word swap on a per slave basis.*
- `void configureStandard32BitMode ()`  
*Configures all slaves for Standard 32-bit Mode.*
- `int configureStandard32BitMode (int slaveAddr)`  
*Configures a slave for Standard 32-bit Register Mode.*
- `void configureEnron32BitMode ()`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `int configureEnron32BitMode (int slaveAddr)`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `void configureCountFromOne ()`  
*Configures the reference counting scheme to start with one for all slaves.*
- `int configureCountFromOne (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with one.*
- `void configureCountFromZero ()`  
*Configures the reference counting scheme to start with zero for all slaves.*
- `int configureCountFromZero (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with zero.*

## Utility Functions

- `static const TCHAR * getPackageVersion ()`  
*Returns the library version number.*



## 8.6.1 Detailed Description

Modbus ASCII over TCP Master Protocol class.

This class realises the Modbus ASCII master protocol using TCP as transport layer. This class provides functions to establish and to close a TCP/IP connection to the slave as well as data and control functions which can be used after a connection to a slave device has been established successfully. The data and control functions are organized different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Modbus Protocol Flavours.

It is also possible to instantiate multiple instances of this class for establishing multiple connections to either the same or different hosts.

### See also

Data and Control Functions for all Modbus Protocol Flavours, IP based Protocols  
MbusAsciiMasterProtocol

## 8.6.2 Member Function Documentation

```
openProtocol() [1/2] int openProtocol (  
    const char *const hostName )
```

Connects to a Modbus ASCII slave via TCP/IP.

This function establishes a logical network connection between master and slave. After a connection has been established data and control functions can be used. A TCP/IP connection should be closed if it is no longer needed.

### Note

The default time-out for the connection is 1000 ms.

### Parameters

<i>hostName</i>	String with IP address or host name
-----------------	-------------------------------------

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

The default TCP port number is Telnet port 23.

**setPort()** int setPort (   
 unsigned short *portNo* )

Sets the TCP port number used to connect to the Modbus ASCII slave device.

Usually the port number remains unchanged from its default. In this case no call to this function is necessary. However if the port number has to be different, this function must be called *before* opening the connection with openProtocol().

#### Parameters

<i>port</i> ↔ <i>No</i>	Port number to be used when opening the connection
----------------------------	--

#### Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_STATE_ERR</i> ↔ <i>OR</i>	Protocol already open

#### Note

Defaults to 23 (Telnet port).

**openProtocol()** [2/2] int openProtocol (   
 const char \*const *hostName*,   
 const char \*const *interfaceIp* ) [inherited]

Opens a connection to a slave device.

This function establishes a logical network connection between master and slave. After a connection has been established data and control functions can be used. A TCP/IP connection should be closed if it is no longer needed.

#### Note

The default time-out for the connection is 1000 ms.

#### Parameters

<i>hostName</i>	String with IP address or host name
<i>interfaceIp</i>	String with IP address for a specific host interface or NULL if connections are accepted on any interface*

#### Returns

*FTALK\_SUCCESS* on success or error code. See Error Management for a list of error codes.

References FTALK\_ILLEGAL\_STATE\_ERROR, FTALK\_TCPIP\_CONNECT\_ERR, and MbusIp↔ ClientBase::isOpen().

**isOpen()** int isOpen ( ) [virtual], [inherited]

**Deprecated** Returns whether the protocol is open or not.

This simply returns the state of an internal flag whether openProtocol() has been called or not.

#### Note

This function does not report the state of the underlying transport network, in particular it does not report whether the TCP connection is still in 'established' state or in fully 'closed' state.

Return values

<i>true</i>	= open
<i>false</i>	= closed

Implements MbusMasterFunctions.

**setClosingTimeout()** int setClosingTimeout ( int *msTime* ) [inherited]

Applies a time-out to socket closure and makes closeProtocol() wait for the server to acknowledge closing before potentially opening a new one.

#### Note

A protocol must be closed in order to configure it.

#### Parameters

<i>msTime</i>	Timeout value in ms (Range: 1 - 100000)
---------------	---

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_ILLEGAL\_STATE\_ERROR, FTALK\_↔ SUCCESS, and MbusIpClientBase::isOpen().

**getPort()** unsigned short getPort ( ) [inline], [inherited]

Returns the TCP port number used by the protocol.

#### Returns

Port number used by the protocol

**readCoils()** int readCoils (   
    int *slaveAddr*,   
    int *startRef*,   
    int *bitArr*[],   
    int *refCnt* ) [inherited]

Modbus function 1, Read Coil Status/Read Coils.

Reads the contents of the discrete outputs (coils, 0:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

**readInputDiscretes()** int readInputDiscretes (   
    int *slaveAddr*,   
    int *startRef*,   
    int *bitArr*[],   
    int *refCnt* ) [inherited]

Modbus function 2, Read Inputs Status/Read Input Discretes.

Reads the contents of the discrete inputs (input status, 1:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeCoil() int writeCoil (  
                int slaveAddr,  
                int bitAddr,  
                int bitVal ) [inherited]
```

Modbus function 5, Force Single Coil/Write Coil.

Sets a single discrete output variable (coil, 0:00000 table) to either ON or OFF.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>bitAddr</i>	Coil address (Range: 1 - 65536)
<i>bitVal</i>	true sets, false clears discrete output variable

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
forceMultipleCoils() int forceMultipleCoils (
    int slaveAddr,
    int startRef,
    const int bitArr[],
    int refCnt ) [inherited]
```

Modbus function 15 (0F Hex), Force Multiple Coils.

Writes binary values into a sequence of discrete outputs (coils, 0:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which contains the data to be sent
<i>refCnt</i>	Number of coils to be written (Range: 1-1968)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleRegisters() int readMultipleRegisters (
    int slaveAddr,
    int startRef,
    short regArr[],
    int refCnt ) [inherited]
```

Modbus function 3, Read Holding Registers/Read Multiple Registers.

Reads the contents of the output registers (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputRegisters() int readInputRegisters (
    int slaveAddr,
    int startRef,
    short regArr[],
    int refCnt ) [inherited]
```

Modbus function 4, Read Input Registers.

Read the contents of the input registers (3:00000 table).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read.
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeSingleRegister() int writeSingleRegister (
    int slaveAddr,
    int regAddr,
    short regVal ) [inherited]
```

Modbus function 6, Preset Single Register/Write Single Register.

Writes a value into a single output register (holding register, 4:00000 reference).



### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>regVal</i>	Data to be sent

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
writeMultipleRegisters() int writeMultipleRegisters (
    int slaveAddr,
    int startRef,
    const short regArr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.

Writes values into a sequence of output registers (holding registers, 4:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer with the data to be sent.
<i>refCnt</i>	Number of registers to be written (Range: 1-123)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

```
maskWriteRegister() int maskWriteRegister (
    int slaveAddr,
    int regAddr,
    short andMask,
    short orMask ) [inherited]
```

Modbus function 22 (16 Hex), Mask Write Register.

Masks bits according to an AND & an OR mask into a single output register (holding register, 4:00000 reference). Masking is done as follows: result = (currentVal AND andMask) OR (orMask AND (NOT andMask))

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>andMask</i>	Mask to be applied as a logic AND to the register
<i>orMask</i>	Mask to be applied as a logic OR to the register

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readWriteRegisters() int readWriteRegisters (
    int slaveAddr,
    int readRef,
    short readArr[],
    int readCnt,
    int writeRef,
    const short writeArr[],
    int writeCnt ) [inherited]
```

Modbus function 23 (17 Hex), Read/Write Registers.

Combines reading and writing of the output registers in one transaction (holding registers, 4:00000 table).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>readRef</i>	Start register for reading (Range: 1 - 65536)
<i>readArr</i>	Buffer which will contain the data read
<i>readCnt</i>	Number of registers to be read (Range: 1-125)
<i>writeRef</i>	Start register for writing (Range: 1 - 65536)

### Parameters

<i>writeArr</i>	Buffer with data to be sent
<i>writeCnt</i>	Number of registers to be written (Range: 1-121)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleLongInts() int readMultipleLongInts (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into 32-bit long int values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputLongInts() int readInputLongInts (  
    int slaveAddr,  
    int startRef,  
    int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) into 32-bit long int values.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleLongInts() int writeMultipleLongInts (  
    int slaveAddr,  
    int startRef,  
    const int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/↔ Write Multiple Registers with long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleFloats() int readMultipleFloats (
    int slaveAddr,
    int startRef,
    float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into float values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readInputFloats() int readInputFloats (  
    int slaveAddr,  
    int startRef,  
    float float32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit float data types, Read Input Registers as float data.

Reads the contents of pairs of consecutive input registers (3:00000 table) into float values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleFloats() int writeMultipleFloats (  
    int slaveAddr,  
    int startRef,  
    const float float32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

Writes float values into pairs of output registers (holding registers, 4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleMod10000() int readMultipleMod10000 (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value into 32-bit int values and performs number format conversion.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readInputMod10000() int readInputMod10000 (  
    int slaveAddr,  
    int startRef,  
    int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) representing a modulo-10000 long int value into 32-bit long int values and performs number format conversion.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.



**Note**

No broadcast supported

```
writeMultipleMod10000() int writeMultipleMod10000 (
    int slaveAddr,
    int startRef,
    const int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value and performs number format conversion.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values values to be sent (Range: 1-61)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersI64() int readHoldingRegistersI64 (
    int slaveAddr,
    int startRef,
    int64_t int64Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.

Reads the contents of quartets of consecutive holding registers (4:00000 table) into 64-bit integer values.

#### Remarks

Depending on the 32-bit Mode setting, an `int64_t` will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
readInputRegistersI64() int readInputRegistersI64 (  
    int slaveAddr,  
    int startRef,  
    int64_t int64Arr[],  
    int refCnt ) [inherited]
```

Modbus function 4 for 64-bit integer data, Read Holding Registers as `int64_t` type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit integer values.

#### Remarks

Depending on the 32-bit Mode setting, an `int64_t` will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleRegistersI64() int writeMultipleRegistersI64 (
    int slaveAddr,
    int startRef,
    const int64_t int64Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.

Writes for 64-bit integer values into quartets of holding registers (4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersF64() int readHoldingRegistersF64 (
    int slaveAddr,
```

```
int startRef,  
double double64Arr[],  
int refCnt ) [inherited]
```

Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (4:00000 table) into 64-bit floating point values.

#### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```
readInputRegistersF64() int readInputRegistersF64 (  
int slaveAddr,  
int startRef,  
double double64Arr[],  
int refCnt ) [inherited]
```

Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit floating point values.

#### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleRegistersF64() int writeMultipleRegistersF64 (
    int slaveAddr,
    int startRef,
    const double double64Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.

Writes for 64-bit floating point values into quartets of holding registers (4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readFileRecord() [1/2] int readFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 20, Read File Record.

Performs a file record read for a single subrequest.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to read from, 0-based numbering.
<i>recordArr</i>	Buffer which will be filled with the data read
<i>recordCnt</i>	Number of records to read (Range 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readFileRecord() [2/2] int readFileRecord (
    int slaveAddr,
    FileSubRequest subReqArr[],
    int subReqCnt ) [inherited]
```

Modbus function 20, Read File Record.

Performs a file record read for a list of subrequests.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to read (Range 1-35)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK↔\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
writeFileRecord() [1/2] int writeFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    const short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a single subrequest.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to write to, 0-based numbering.
<i>recordArr</i>	Buffer with the data to be sent.
<i>recordCnt</i>	Number of records to write (Range 1-122)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeFileRecord() [2/2] int writeFileRecord (  
    int slaveAddr,  
    const FileSubRequest subReqArr[],  
    int subReqCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a list of subrequests.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to write (Range 1-27)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK↔\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
readExceptionStatus() int readExceptionStatus (  
    int slaveAddr,  
    unsigned char * statusBytePtr ) [inherited]
```

Modbus function 7, Read Exception Status.

Reads the eight exception status coils within the slave device.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>statusBytePtr</i>	Slave status byte. The meaning of this status byte is slave specific and varies from device to device.



### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

```
returnQueryData() int returnQueryData (
    int slaveAddr,
    const unsigned char queryArr[],
    unsigned char echoArr[],
    int len ) [inherited]
```

Modbus function code 8, sub-function 00, Return Query Data.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>queryArr</i>	Buffer with data to be sent
<i>echoArr</i>	Buffer which will contain the data read
<i>len</i>	Number of bytes send sent and read back

### Returns

FTALK\_SUCCESS on success, FTALK\_INVALID\_REPLY\_ERROR if reply does not match query data or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
restartCommunicationsOption() int restartCommunicationsOption (
    int slaveAddr,
    int clearEventLog ) [inherited]
```

Modbus function code 8, sub-function 01, Restart Communications Option.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>clearEventLog</i>	Flag when set to one clears in addition the slave's communication even log.

**Returns**

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readDeviceIdentification() int readDeviceIdentification (
    int slaveAddr,
    int accessType,
    int objId,
    DeviceIdObjectList * objListPtr ) [inherited]
```

Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.

This function allows a master to retrieve various objects with meta information about a slave device. The data type of the objects returned is a 0-terminated ASCII string.

Object Id	Object Name / Description
0x00	VendorName
0x01	ProductCode
0x02	MajorMinorRevision
0x03	VendorUrl
0x04	ProductName
0x05	ModelName
0x06	UserApplicationName
0x07 - 0x7F	Reserved
0x80 - 0xFF	Vendor specific private objects

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>accessType</i>	Category type (1=only mandatory basic objects, 2=all regular objects, 3=include vendor specific objects, 4=single object)
<i>objId</i>	An object ID from above table

## Parameters

<i>objListPtr</i>	A pointer to a object list container which will receive the retrieved objects.
-------------------	--

## Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

## Note

No broadcast supported

Example retrieving all Device ID strings of basic device identification level (level 1):

```
DeviceIdObjectList objs;
int objId = 0; // Start with object 0
do
{
    result = mbusProtocol.readDeviceIdentification(1, 1, objId, &objs);

    if (result != FTALK_SUCCESS)
    {
        fprintf(stderr, "Error: %s!\n", getBusProtocolErrorText(result));
        exit(EXIT_FAILURE);
    }
    else
    {
        size_t objLen;
        char* objData;

        while (objs.getNext(&objId, &objData, &objLen))
        {
            printf("obj[%d] = '%s', len = %u\n", objId, objData, objLen);
        }
    }
}
while (objs.moreToFollow());
```

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_FRAME\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
setTimeout() int setTimeout (
    int msTime ) [inherited]
```

Configures time-out.

This function sets the operation or socket time-out to the specified value.

## Remarks

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

**Note**

A protocol must be closed in order to configure it.

**Parameters**

<i>msTime</i>	Timeout value in ms (Range: 1 - 100000)
---------------	---

## Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and *MbusMasterFunctions::isOpen()*.

**getTimeout()** `int getTimeout ( ) [inline], [inherited]`

Returns the time-out value.

**Remarks**

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

**Returns**

Timeout value in ms

**setPollDelay()** `int setPollDelay (`  
`int msTime ) [inherited]`

Configures poll delay.

This function sets the delay time which applies between two consecutive Modbus read/write. A value of 0 disables the poll delay.

**Remarks**

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

**Note**

A protocol must be closed in order to configure it.

**Parameters**

<i>msTime</i>	Delay time in ms (Range: 0 - 100000), 0 disables poll delay
---------------	---

## Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getPollDelay()** `int getPollDelay ( ) [inline], [inherited]`

Returns the poll delay time.

**Returns**

Delay time in ms, 0 if poll delay is switched off

**setRetryCnt()** `int setRetryCnt ( int retries ) [inherited]`

Configures the automatic retry setting.

A value of 0 disables any automatic retries.

**Note**

A protocol must be closed in order to configure it.

**Parameters**

<i>retries</i>	Retry count (Range: 0 - 10), 0 disables retries
----------------	---

## Return values

<i>FTALK_SUCCESS</i>	Success
----------------------	---------

Return values

<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getRetryCnt()** `int getRetryCnt ( ) [inline], [inherited]`

Returns the automatic retry count.

#### Returns

Retry count

**getTotalCounter()** `long getTotalCounter ( ) [inline], [inherited]`

Returns how often a message transfer has been executed.

#### Returns

Counter value

**getSuccessCounter()** `long getSuccessCounter ( ) [inline], [inherited]`

Returns how often a message transfer was successful.

#### Returns

Counter value

**configureBigEndianInts()** `[1/2] void configureBigEndianInts ( ) [inherited]`

Configures 32-bit int data type functions to do a word swap.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian slave.

**configureBigEndianInts()** [2/2] `int configureBigEndianInts ( int slaveAddr )` [inherited]

Enables int data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian machine.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureLittleEndianInts()** [1/2] `void configureLittleEndianInts ( )` [inherited]

Configures 32-bit int data type functions not to do a word swap.

This is the default.

**configureLittleEndianInts()** [2/2] `int configureLittleEndianInts ( int slaveAddr )` [inherited]

Disables word swapping for int data type functions on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit data in little little-endian word order.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureIeeeFloats()** [1/2] `void configureIeeeFloats ( )` [inherited]

Configures float data type functions not to do a word swap.

This is the default.

**configureIeeeFloats()** [2/2] `int configureIeeeFloats (`

`int slaveAddr ) [inherited]`

Disables float data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit floats in little little-endian word order which is the most common case.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureSwappedFloats()** [1/2] `void configureSwappedFloats ( ) [inherited]`

Configures float data type functions to do a word swap.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

#### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

**configureSwappedFloats()** [2/2] `int configureSwappedFloats ( int slaveAddr ) [inherited]`

Enables float data type functions to do a word swap on a per slave basis.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

#### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--



References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureStandard32BitMode()** [1/2] void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

#### Note

This function call also re-configures the endianness to default little-endian for 32-bit values!

#### Remarks

This is the default mode

**configureStandard32BitMode()** [2/2] int configureStandard32BitMode ( int *slaveAddr* ) [inherited]

Configures a slave for Standard 32-bit Register Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range

#### Remarks

This is the default mode

#### Note

This function call also re-configures the endianness to default little-endian for 32-bit values!

References `MbusMasterFunctions::configureIEEEFloats()`, `MbusMasterFunctions::configureLittleEndianInts()`, `FTALK_ILLEGAL_ARGUMENT_ERROR`, and `FTALK_SUCCESS`.

**configureEnron32BitMode()** [1/2] `void configureEnron32BitMode ( )` [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

#### Note

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

**configureEnron32BitMode()** [2/2] `int configureEnron32BitMode ( int slaveAddr )` [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERROR</i> OR	Argument out of range

#### Note

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

References `MbusMasterFunctions::configureBigEndianInts()`, `MbusMasterFunctions::configureSwappedFloats()`, `FTALK_ILLEGAL_ARGUMENT_ERROR`, and `FTALK_SUCCESS`.

**configureCountFromOne()** [1/2] `void configureCountFromOne ( )` [inherited]

Configures the reference counting scheme to start with one for all slaves.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Remarks

This is the default mode

```
configureCountFromOne() [2/2] int configureCountFromOne (
    int slaveAddr ) [inherited]
```

Configures a slave's reference counting scheme to start with one.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

#### Remarks

This is the default mode

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

```
configureCountFromZero() [1/2] void configureCountFromZero ( ) [inherited]
```

Configures the reference counting scheme to start with zero for all slaves.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

This renders the first register to be #0 for all slaves.

```
configureCountFromZero() [2/2] int configureCountFromZero (
    int slaveAddr ) [inherited]
```

Configures a slave's reference counting scheme to start with zero.

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**getPackageVersion()** `const TCHAR * getPackageVersion ( ) [static], [inherited]`

Returns the library version number.

#### Returns

Library version string

## 8.7 MbusUdpMasterProtocol Class Reference

MODBUS/UDP Master Protocol class.

### Public Member Functions

- `MbusUdpMasterProtocol ( )`  
*Constructs a MbusUdpMasterProtocol object and initialises its data.*
- `int openProtocol (const char *const hostName)`  
*Opens a connection to a slave device.*
- `int openProtocol (const char *const hostName, const char *const interfacelp)`  
*Opens a connection to a slave device.*
- `virtual void closeProtocol ( )`  
*Closes a TCP/IP connection to a slave and releases any system resources associated with the connection.*
- `virtual int isOpen ( )`
- `int setPort (unsigned short portNo)`  
*Sets the port number to be used by the protocol.*
- `int setClosingTimeout (int msTime)`  
*Applies a time-out to socket closure and makes closeProtocol() wait for the server to acknowledge closing before potentially opening a new one.*
- `unsigned short getPort ( )`  
*Returns the TCP port number used by the protocol.*

### Bit Access

Table 0:00000 (Coils) and Table 1:0000 (Input Status)

- `int readCoils (int slaveAddr, int startRef, int bitArr[ ], int refCnt)`  
*Modbus function 1, Read Coil Status/Read Coils.*
- `int readInputDiscretes (int slaveAddr, int startRef, int bitArr[ ], int refCnt)`  
*Modbus function 2, Read Inputs Status/Read Input Discretes.*
- `int writeCoil (int slaveAddr, int bitAddr, int bitVal)`

*Modbus function 5, Force Single Coil/Write Coil.*

- `int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)`  
*Modbus function 15 (0F Hex), Force Multiple Coils.*

## 16-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 3, Read Holding Registers/Read Multiple Registers.*
- `int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)`  
*Modbus function 4, Read Input Registers.*
- `int writeSingleRegister (int slaveAddr, int regAddr, short regVal)`  
*Modbus function 6, Preset Single Register/Write Single Register.*
- `int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)`  
*Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.*
- `int maskWriteRegister (int slaveAddr, int regAddr, short andMask, short orMask)`  
*Modbus function 22 (16 Hex), Mask Write Register.*
- `int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)`  
*Modbus function 23 (17 Hex), Read/Write Registers.*

## 32-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readMultipleLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.*
- `int readInputLongInts (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.*
- `int writeMultipleLongInts (int slaveAddr, int startRef, const int int32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.*
- `int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.*
- `int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)`  
*Modbus function 4 for 32-bit float data types, Read Input Registers as float data.*
- `int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.*
- `int readMultipleMod10000 (int slaveAddr, int startRef, int int32Arr[ ], int refCnt)`  
*Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/↔ Read Multiple Registers as modulo-10000 long int data.*

- `int readInputMod10000 (int slaveAddr, int startRef, int int32Arr[], int refCnt)`  
*Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.*
- `int writeMultipleMod10000 (int slaveAddr, int startRef, const int int32Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.*

## 64-bit Access

Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)

- `int readHoldingRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int readInputRegistersI64 (int slaveAddr, int startRef, int64_t int64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit integer data, Read Holding Registers as int64\_t type.*
- `int writeMultipleRegistersI64 (int slaveAddr, int startRef, const int64_t int64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.*
- `int readHoldingRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.*
- `int readInputRegistersF64 (int slaveAddr, int startRef, double double64Arr[], int refCnt)`  
*Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.*
- `int writeMultipleRegistersF64 (int slaveAddr, int startRef, const double double64Arr[], int refCnt)`  
*Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.*

## File Record Access

- `int readFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, short recordArr[], int recordCnt)`  
*Modbus function 20, Read File Record.*
- `int readFileRecord (int slaveAddr, FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 20, Read File Record.*
- `int writeFileRecord (int slaveAddr, int refType, int fileNo, int recordNo, const short recordArr[], int recordCnt)`  
*Modbus function 21, Write File Record.*
- `int writeFileRecord (int slaveAddr, const FileSubRequest subReqArr[], int subReqCnt)`  
*Modbus function 21, Write File Record.*

## Diagnostics

- `int readExceptionStatus (int slaveAddr, unsigned char *statusBytePtr)`  
*Modbus function 7, Read Exception Status.*
- `int returnQueryData (int slaveAddr, const unsigned char queryArr[], unsigned char echoArr[], int len)`  
*Modbus function code 8, sub-function 00, Return Query Data.*
- `int restartCommunicationsOption (int slaveAddr, int clearEventLog)`  
*Modbus function code 8, sub-function 01, Restart Communications Option.*
- `int readDeviceIdentification (int slaveAddr, int accessType, int objId, DeviceIdObjectList *objListPtr)`  
*Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.*

## Custom Function Codes

- `int customFunction (int slaveAddr, int functionCode, void *requestData, size_t requestLen, void *responseData, size_t *responseLenPtr)`  
*User Defined Function Code*  
*This method can be used to implement User Defined Function Codes.*

## Protocol Configuration

- `int setTimeout (int timeOut)`  
*Configures time-out.*
- `int getTimeout ()`  
*Returns the time-out value.*
- `int setPollDelay (int pollDelay)`  
*Configures poll delay.*
- `int getPollDelay ()`  
*Returns the poll delay time.*
- `int setRetryCnt (int retryCnt)`  
*Configures the automatic retry setting.*
- `int getRetryCnt ()`  
*Returns the automatic retry count.*

## Transmission Statistic Functions

- `long getTotalCounter ()`  
*Returns how often a message transfer has been executed.*
- `void resetTotalCounter ()`  
*Resets total message transfer counter.*
- `long getSuccessCounter ()`  
*Returns how often a message transfer was successful.*
- `void resetSuccessCounter ()`  
*Resets successful message transfer counter.*

## Slave Configuration

- `void configureBigEndianInts ()`  
*Configures 32-bit int data type functions to do a word swap.*
- `int configureBigEndianInts (int slaveAddr)`  
*Enables int data type functions to do a word swap on a per slave basis.*
- `void configureLittleEndianInts ()`  
*Configures 32-bit int data type functions not to do a word swap.*
- `int configureLittleEndianInts (int slaveAddr)`  
*Disables word swapping for int data type functions on a per slave basis.*
- `void configureleeeeFloats ()`  
*Configures float data type functions not to do a word swap.*
- `int configureleeeeFloats (int slaveAddr)`  
*Disables float data type functions to do a word swap on a per slave basis.*
- `void configureSwappedFloats ()`  
*Configures float data type functions to do a word swap.*
- `int configureSwappedFloats (int slaveAddr)`  
*Enables float data type functions to do a word swap on a per slave basis.*
- `void configureStandard32BitMode ()`  
*Configures all slaves for Standard 32-bit Mode.*
- `int configureStandard32BitMode (int slaveAddr)`  
*Configures a slave for Standard 32-bit Register Mode.*
- `void configureEnron32BitMode ()`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `int configureEnron32BitMode (int slaveAddr)`  
*Configures all slaves for Daniel/ENRON 32-bit Mode.*
- `void configureCountFromOne ()`  
*Configures the reference counting scheme to start with one for all slaves.*
- `int configureCountFromOne (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with one.*
- `void configureCountFromZero ()`  
*Configures the reference counting scheme to start with zero for all slaves.*
- `int configureCountFromZero (int slaveAddr)`  
*Configures a slave's reference counting scheme to start with zero.*

## Utility Functions

- `static const TCHAR * getPackageVersion ()`  
*Returns the library version number.*



## 8.7.1 Detailed Description

MODBUS/UDP Master Protocol class.

This class realises a Modbus client using MODBUS over UDP protocol variant. It provides functions to establish a UDP connection to the slave as well as data and control functions which can be used after a connection to a slave device has been established successfully. The data and control functions are organized into different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Modbus Protocol Flavours.

It is also possible to instantiate multiple instances of this class for establishing multiple connections to either the same or different hosts.

### See also

Data and Control Functions for all Modbus Protocol Flavours, IP based Protocols

## 8.7.2 Member Function Documentation

```
openProtocol() [1/2] int openProtocol (
    const char *const hostName ) [inherited]
```

Opens a connection to a slave device.

This function establishes a logical network connection between master and slave. After a connection has been established data and control functions can be used. A TCP/IP connection should be closed if it is no longer needed.

### Note

The default time-out for the connection is 1000 ms.

### Parameters

<i>hostName</i>	String with IP address or host name
-----------------	-------------------------------------

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

```
openProtocol() [2/2] int openProtocol (
    const char *const hostName,
    const char *const interfaceIp ) [inherited]
```

Opens a connection to a slave device.

This function establishes a logical network connection between master and slave. After a connection has been established data and control functions can be used. A TCP/IP connection should be closed if it is no longer needed.

#### Note

The default time-out for the connection is 1000 ms.

#### Parameters

<i>hostName</i>	String with IP address or host name
<i>interfaceIp</i>	String with IP address for a specific host interface or NULL if connections are accepted on any interface*

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

References FTALK\_ILLEGAL\_STATE\_ERROR, FTALK\_TCPIP\_CONNECT\_ERR, and MbusIp↔ClientBase::isOpen().

**isOpen()** int isOpen ( ) [virtual], [inherited]

**Deprecated** Returns whether the protocol is open or not.

This simply returns the state of an internal flag whether openProtocol() has been called or not.

#### Note

This function does not report the state of the underlying transport network, in particular it does not report whether the TCP connection is still in 'established' state or in fully 'closed' state.

Return values

<i>true</i>	= open
<i>false</i>	= closed

Implements MbusMasterFunctions.

**setPort()** int setPort (

`unsigned short portNo ) [inherited]`

Sets the port number to be used by the protocol.

Usually the port number remains unchanged from its default. In this case no call to this function is necessary. However if the port number has to be different, this function must be called *before* opening the connection with `openProtocol()`.

#### Parameters

<code>portNo</code>	Port number to be used when opening the connection
---------------------	--

Return values

<code>FTALK_SUCCESS</code>	Success
<code>FTALK_ILLEGAL_STATE_ERROR</code> OR	Protocol already open

References `FTALK_ILLEGAL_STATE_ERROR`, `FTALK_SUCCESS`, and `MbuspClientBase::isOpen()`.

**setClosingTimeout()** `int setClosingTimeout ( int msTime ) [inherited]`

Applies a time-out to socket closure and makes `closeProtocol()` wait for the server to acknowledge closing before potentially opening a new one.

#### Note

A protocol must be closed in order to configure it.

#### Parameters

<code>msTime</code>	Timeout value in ms (Range: 1 - 100000)
---------------------	---

Return values

<code>FTALK_SUCCESS</code>	Success
<code>FTALK_ILLEGAL_ARGUMENT_ERROR</code> OR	Argument out of range
<code>FTALK_ILLEGAL_STATE_ERROR</code>	Protocol is already open

References `FTALK_ILLEGAL_ARGUMENT_ERROR`, `FTALK_ILLEGAL_STATE_ERROR`, `FTALK_SUCCESS`, and `MbuspClientBase::isOpen()`.

**getPort()** unsigned short getPort ( ) [inline], [inherited]  
Returns the TCP port number used by the protocol.

#### Returns

Port number used by the protocol

**readCoils()** int readCoils (   
    int *slaveAddr*,  
    int *startRef*,  
    int *bitArr*[],  
    int *refCnt* ) [inherited]

Modbus function 1, Read Coil Status/Read Coils.

Reads the contents of the discrete outputs (coils, 0:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

**readInputDiscretes()** int readInputDiscretes (   
    int *slaveAddr*,  
    int *startRef*,  
    int *bitArr*[],  
    int *refCnt* ) [inherited]

Modbus function 2, Read Inputs Status/Read Input Discretes.

Reads the contents of the discrete inputs (input status, 1:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
------------------	--

### Parameters

<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which will contain the data read
<i>refCnt</i>	Number of coils to be read (Range: 1-2000)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeCoil() int writeCoil (
    int slaveAddr,
    int bitAddr,
    int bitVal ) [inherited]
```

Modbus function 5, Force Single Coil/Write Coil.

Sets a single discrete output variable (coil, 0:00000 table) to either ON or OFF.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>bitAddr</i>	Coil address (Range: 1 - 65536)
<i>bitVal</i>	true sets, false clears discrete output variable

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
forceMultipleCoils() int forceMultipleCoils (
    int slaveAddr,
```

```
int startRef,  
const int bitArr[],  
int refCnt ) [inherited]
```

Modbus function 15 (0F Hex), Force Multiple Coils.

Writes binary values into a sequence of discrete outputs (coils, 0:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>bitArr</i>	Buffer which contains the data to be sent
<i>refCnt</i>	Number of coils to be written (Range: 1-1968)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleRegisters() int readMultipleRegisters (  
int slaveAddr,  
int startRef,  
short regArr[],  
int refCnt ) [inherited]
```

Modbus function 3, Read Holding Registers/Read Multiple Registers.

Reads the contents of the output registers (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputRegisters() int readInputRegisters (
    int slaveAddr,
    int startRef,
    short regArr[],
    int refCnt ) [inherited]
```

Modbus function 4, Read Input Registers.

Read the contents of the input registers (3:00000 table).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer which will be filled with the data read.
<i>refCnt</i>	Number of registers to be read (Range: 1-125)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeSingleRegister() int writeSingleRegister (
    int slaveAddr,
    int regAddr,
    short regVal ) [inherited]
```

Modbus function 6, Preset Single Register/Write Single Register.

Writes a value into a single output register (holding register, 4:00000 reference).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>regVal</i>	Data to be sent

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
writeMultipleRegisters() int writeMultipleRegisters (  
    int slaveAddr,  
    int startRef,  
    const short regArr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex), Preset Multiple Registers/Write Multiple Registers.

Writes values into a sequence of output registers (holding registers, 4:00000 table).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start register (Range: 1 - 65536)
<i>regArr</i>	Buffer with the data to be sent.
<i>refCnt</i>	Number of registers to be written (Range: 1-123)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

```
maskWriteRegister() int maskWriteRegister (  
    int slaveAddr,  
    int regAddr,  
    short andMask,  
    short orMask ) [inherited]
```

Modbus function 22 (16 Hex), Mask Write Register.



Masks bits according to an AND & an OR mask into a single output register (holding register, 4:00000 reference). Masking is done as follows:  $\text{result} = (\text{currentVal AND andMask}) \text{ OR } (\text{orMask AND (NOT andMask)})$

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>regAddr</i>	Register address (Range: 1 - 65536)
<i>andMask</i>	Mask to be applied as a logic AND to the register
<i>orMask</i>	Mask to be applied as a logic OR to the register

#### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readWriteRegisters() int readWriteRegisters (
    int slaveAddr,
    int readRef,
    short readArr[],
    int readCnt,
    int writeRef,
    const short writeArr[],
    int writeCnt ) [inherited]
```

Modbus function 23 (17 Hex), Read/Write Registers.

Combines reading and writing of the output registers in one transaction (holding registers, 4:00000 table).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>readRef</i>	Start register for reading (Range: 1 - 65536)
<i>readArr</i>	Buffer which will contain the data read
<i>readCnt</i>	Number of registers to be read (Range: 1-125)
<i>writeRef</i>	Start register for writing (Range: 1 - 65536)
<i>writeArr</i>	Buffer with data to be sent
<i>writeCnt</i>	Number of registers to be written (Range: 1-121)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
readMultipleLongInts() int readMultipleLongInts (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into 32-bit long int values.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputLongInts() int readInputLongInts (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit long int data types, Read Input Registers as long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) into 32-bit long int values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleLongInts() int writeMultipleLongInts (
    int slaveAddr,
    int startRef,
    const int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit long int data types, Preset Multiple Registers/↔ Write Multiple Registers with long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleFloats() int readMultipleFloats (  
    int slaveAddr,  
    int startRef,  
    float float32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into float values.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
readInputFloats() int readInputFloats (
    int slaveAddr,
    int startRef,
    float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit float data types, Read Input Registers as float data.

Reads the contents of pairs of consecutive input registers (3:00000 table) into float values.

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-62)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleFloats() int writeMultipleFloats (
    int slaveAddr,
    int startRef,
    const float float32Arr[],
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

Writes float values into pairs of output registers (holding registers, 4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>float32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readMultipleMod10000() int readMultipleMod10000 (  
    int slaveAddr,  
    int startRef,  
    int int32Arr[],  
    int refCnt ) [inherited]
```

Modbus function 3 for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:0000 table) representing a modulo-10000 long int value into 32-bit int values and performs number format conversion.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readInputMod10000() int readInputMod10000 (
    int slaveAddr,
    int startRef,
    int int32Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive input registers (3:00000 table) representing a modulo-10000 long int value into 32-bit long int values and performs number format conversion.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of M10K integers to be read (Range: 1-62)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleMod10000() int writeMultipleMod10000 (
    int slaveAddr,
```

```
int startRef,  
const int int32Arr[],  
int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value and performs number format conversion.

### Remarks

Depending on the 32-bit Mode setting, an int will be transferred as two consecutive 16-bit registers (Standard) or as one 32-bit register (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int32Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values values to be sent (Range: 1-61)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersI64() int readHoldingRegistersI64 (  
int slaveAddr,  
int startRef,  
int64_t int64Arr[],  
int refCnt ) [inherited]
```

Modbus function 3 for 64-bit integer data, Read Holding Registers as int64\_t type.

Reads the contents of quartets of consecutive holding registers (4:00000 table) into 64-bit integer values.

### Remarks

Depending on the 32-bit Mode setting, an int64\_t will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).



### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readInputRegistersI64() int readInputRegistersI64 (
    int slaveAddr,
    int startRef,
    int64_t int64Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 64-bit integer data, Read Holding Registers as int64\_t type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit integer values.

### Remarks

Depending on the 32-bit Mode setting, an int64\_t will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeMultipleRegistersI64() int writeMultipleRegistersI64 (  
    int slaveAddr,  
    int startRef,  
    const int64_t int64Arr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit integer data, Write Multiple Registers as int64\_t type.

Writes for 64-bit integer values into quartets of holding registers (4:00000 table).

**Remarks**

Depending on the 32-bit Mode setting, an int will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>int64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readHoldingRegistersF64() int readHoldingRegistersF64 (  
    int slaveAddr,  
    int startRef,  
    double double64Arr[],  
    int refCnt ) [inherited]
```

Modbus function 3 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (4:00000 table) into 64-bit floating point values.

### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
readInputRegistersF64() int readInputRegistersF64 (
    int slaveAddr,
    int startRef,
    double double64Arr[],
    int refCnt ) [inherited]
```

Modbus function 4 for 64-bit floating point data, Read Input Registers as double type.

Reads the contents of quartets of consecutive input registers (3:00000 table) into 64-bit floating point values.

### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer which will be filled with the data read
<i>refCnt</i>	Number of values to be read (Range: 1-31)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

No broadcast supported

```
writeMultipleRegistersF64() int writeMultipleRegistersF64 (  
    int slaveAddr,  
    int startRef,  
    const double double64Arr[],  
    int refCnt ) [inherited]
```

Modbus function 16 (10 Hex) for 64-bit floating point data, Write Multiple Registers as double type.

Writes for 64-bit floating point values into quartets of holding registers (4:00000 table).

### Remarks

Depending on the 32-bit Mode setting, a double will be transferred as four consecutive 16-bit registers (Standard) or as two 32-bit registers (Daniel/Enron).

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 0 - 255)
<i>startRef</i>	Start reference (Range: 1 - 65536)
<i>double64Arr</i>	Buffer with the data to be sent
<i>refCnt</i>	Number of values to be sent (Range: 1-30)

### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

### Note

Broadcast supported for serial protocols

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readFileRecord() [1/2] int readFileRecord (  
    int slaveAddr,
```

```

int refType,
int fileNo,
int recordNo,
short recordArr[],
int recordCnt ) [inherited]

```

Modbus function 20, Read File Record.

Performs a file record read for a single subrequest.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to read from, 0-based numbering.
<i>recordArr</i>	Buffer which will be filled with the data read
<i>recordCnt</i>	Number of records to read (Range 1-125)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

```

readFileRecord() [2/2] int readFileRecord (
    int slaveAddr,
    FileSubRequest subReqArr[],
    int subReqCnt ) [inherited]

```

Modbus function 20, Read File Record.

Performs a file record read for a list of subrequests.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to read (Range 1-35)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
writeFileRecord() [1/2] int writeFileRecord (
    int slaveAddr,
    int refType,
    int fileNo,
    int recordNo,
    const short recordArr[],
    int recordCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a single subrequest.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>refType</i>	Reference type (Must be specified as 6).
<i>fileNo</i>	File number (Range 1-65535). 3 is the input registers file, 4 is the holding registers file.
<i>recordNo</i>	Record Number (Range: 0 - 65535). The first record to write to, 0-based numbering.
<i>recordArr</i>	Buffer with the data to be sent.
<i>recordCnt</i>	Number of records to write (Range 1-122)

**Returns**

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

```
writeFileRecord() [2/2] int writeFileRecord (
    int slaveAddr,
```

```
const FileSubRequest subReqArr[],
int subReqCnt ) [inherited]
```

Modbus function 21, Write File Record.

Performs a file record write for a list of subrequests.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255)
<i>subReqArr</i>	Array with list of subrequests
<i>subReq</i> ↔ <i>Cnt</i>	Number of subrequests to write (Range 1-27)

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

#### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, FTALK↔\_SUCCESS, and MbusMasterFunctions::FileSubRequest::recordCnt.

```
readExceptionStatus() int readExceptionStatus (
    int slaveAddr,
    unsigned char * statusBytePtr ) [inherited]
```

Modbus function 7, Read Exception Status.

Reads the eight exception status coils within the slave device.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>statusBytePtr</i>	Slave status byte. The meaning of this status byte is slave specific and varies from device to device.

#### Returns

FTALK\_SUCCESS on success or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

```
returnQueryData() int returnQueryData (  
    int slaveAddr,  
    const unsigned char queryArr[],  
    unsigned char echoArr[],  
    int len ) [inherited]
```

Modbus function code 8, sub-function 00, Return Query Data.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>queryArr</i>	Buffer with data to be sent
<i>echoArr</i>	Buffer which will contain the data read
<i>len</i>	Number of bytes send sent and read back

**Returns**

FTALK\_SUCCESS on success, FTALK\_INVALID\_REPLY\_ERROR if reply does not match query data or error code. See Error Management for a list of error codes.

**Note**

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
restartCommunicationsOption() int restartCommunicationsOption (  
    int slaveAddr,  
    int clearEventLog ) [inherited]
```

Modbus function code 8, sub-function 01, Restart Communications Option.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>clearEventLog</i>	Flag when set to one clears in addition the slave's communication even log.



### Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

### Note

No broadcast supported

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR.

```
readDeviceIdentification() int readDeviceIdentification (
    int slaveAddr,
    int accessType,
    int objId,
    DeviceIdObjectList * objListPtr ) [inherited]
```

Modbus function 43 (hex 2B) subfunction 14 (hex 0E), Read Device Identification.

This function allows a master to retrieve various objects with meta information about a slave device. The data type of the objects returned is a 0-terminated ASCII string.

Object Id	Object Name / Description
0x00	VendorName
0x01	ProductCode
0x02	MajorMinorRevision
0x03	VendorUrl
0x04	ProductName
0x05	ModelName
0x06	UserApplicationName
0x07 - 0x7F	<i>Reserved</i>
0x80 - 0xFF	Vendor specific private objects

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255 for serial, 0 - 255 for TCP)
<i>accessType</i>	Category type (1=only mandatory basic objects, 2=all regular objects, 3=include vendor specific objects, 4=single object)
<i>objId</i>	An object ID from above table
<i>objListPtr</i>	A pointer to a object list container which will receive the retrieved objects.

### Returns

FTALK\_SUCCESS on success. See Error Management for a list of error codes.

## Note

### No broadcast supported

Example retrieving all Device ID strings of basic device identification level (level 1):

```
DeviceIdObjectList objjs;
int objId = 0; // Start with object 0
do
{
    result = mbusProtocol.readDeviceIdentification(1, 1, objId, &objjs);

    if (result != FTALK_SUCCESS)
    {
        fprintf(stderr, "Error: %s!\n", getBusProtocolErrorText(result));
        exit(EXIT_FAILURE);
    }
    else
    {
        size_t objLen;
        char* objData;

        while (objjs.getNext(&objId, &objData, &objLen))
        {
            printf("obj[%d] = '%s', len = %u\n", objId, objData, objLen);
        }
    }
}
while (objjs.moreToFollow());
```

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, FTALK\_INVALID\_FRAME\_ERROR, FTALK\_INVALID\_REPLY\_ERROR, and FTALK\_SUCCESS.

```
setTimeout() int setTimeout (
                    int msTime ) [inherited]
```

Configures time-out.

This function sets the operation or socket time-out to the specified value.

## Remarks

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

## Note

A protocol must be closed in order to configure it.

## Parameters

<i>msTime</i>	Timeout value in ms (Range: 1 - 100000)
---------------	---

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References `FTALK_ILLEGAL_ARGUMENT_ERROR`, `FTALK_ILLEGAL_STATE_ERROR`, `FTALK_↔SUCCESS`, and `MbusMasterFunctions::isOpen()`.

**getTimeout()** `int getTimeout ( ) [inline], [inherited]`

Returns the time-out value.

#### Remarks

The time-out value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

#### Returns

Timeout value in ms

**setPollDelay()** `int setPollDelay (`  
`int msTime ) [inherited]`

Configures poll delay.

This function sets the delay time which applies between two consecutive Modbus read/write. A value of 0 disables the poll delay.

#### Remarks

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.

#### Note

A protocol must be closed in order to configure it.

#### Parameters

<i>msTime</i>	Delay time in ms (Range: 0 - 100000), 0 disables poll delay
---------------	---

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

**getPollDelay()** `int getPollDelay ( ) [inline], [inherited]`

Returns the poll delay time.

#### Returns

Delay time in ms, 0 if poll delay is switched off

**setRetryCnt()** `int setRetryCnt (`  
`int retries ) [inherited]`

Configures the automatic retry setting.

A value of 0 disables any automatic retries.

#### Note

A protocol must be closed in order to configure it.

#### Parameters

<i>retries</i>	Retry count (Range: 0 - 10), 0 disables retries
----------------	---

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range
<i>FTALK_ILLEGAL_STATE_ERROR</i>	Protocol is already open

References *FTALK\_ILLEGAL\_ARGUMENT\_ERROR*, *FTALK\_ILLEGAL\_STATE\_ERROR*, *FTALK\_↔SUCCESS*, and `MbusMasterFunctions::isOpen()`.

---

**getRetryCnt()** `int getRetryCnt ( ) [inline], [inherited]`

Returns the automatic retry count.

#### Returns

Retry count

**getTotalCounter()** `long getTotalCounter ( ) [inline], [inherited]`

Returns how often a message transfer has been executed.

#### Returns

Counter value

**getSuccessCounter()** `long getSuccessCounter ( ) [inline], [inherited]`

Returns how often a message transfer was successful.

#### Returns

Counter value

**configureBigEndianInts()** `[1/2] void configureBigEndianInts ( ) [inherited]`

Configures 32-bit int data type functions to do a word swap.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian slave.

**configureBigEndianInts()** `[2/2] int configureBigEndianInts ( int slaveAddr ) [inherited]`

Enables int data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. The data transfer functions operating upon 32-bit int data types can be configured to do a word swap which enables them to read 32-bit data correctly from a big-endian machine.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureLittleEndianInts()** [1/2] void configureLittleEndianInts ( ) [inherited]

Configures 32-bit int data type functions not to do a word swap.

This is the default.

**configureLittleEndianInts()** [2/2] int configureLittleEndianInts (   
int *slaveAddr* ) [inherited]

Disables word swapping for int data type functions on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit data in little little-endian word order.

#### Remarks

This is the default mode

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureIeeeFloats()** [1/2] void configureIeeeFloats ( ) [inherited]

Configures float data type functions not to do a word swap.

This is the default.

**configureIeeeFloats()** [2/2] int configureIeeeFloats (   
int *slaveAddr* ) [inherited]

Disables float data type functions to do a word swap on a per slave basis.

Modbus is using little-endian word order for 32-bit values. This setting assumes that the slave also serves 32-bit floats in little little-endian word order which is the most common case.

#### Remarks

This is the default mode

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureSwappedFloats()** [1/2] void configureSwappedFloats ( ) [inherited]

Configures float data type functions to do a word swap.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

**configureSwappedFloats()** [2/2] int configureSwappedFloats ( int *slaveAddr* ) [inherited]

Enables float data type functions to do a word swap on a per slave basis.

The data functions operating upon 32-bit float data types can be configured to do a word swap.

### Note

Most platforms store floats in IEEE 754 little-endian order which does not need a word swap.

### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureStandard32BitMode()** [1/2] void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

**Note**

This function call also re-configures the endianness to default little-endian for 32-bit values!

**Remarks**

This is the default mode

**configureStandard32BitMode()** [2/2] int configureStandard32BitMode (   
 int *slaveAddr* ) [inherited]

Configures a slave for Standard 32-bit Register Mode.

In Standard 32-bit Register Mode a 32-bit value is transmitted as two consecutive 16-bit Modbus registers.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range

**Remarks**

This is the default mode

**Note**

This function call also re-configures the endianness to default little-endian for 32-bit values!

References MbusMasterFunctions::configureIEEEFloats(), MbusMasterFunctions↔  
::configureLittleEndianInts(), FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureEnron32BitMode()** [1/2] void configureEnron32BitMode ( ) [inherited]

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.



**Note**

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

```
configureEnron32BitMode() [2/2] int configureEnron32BitMode (
    int slaveAddr ) [inherited]
```

Configures all slaves for Daniel/ENRON 32-bit Mode.

Some Modbus flavours like the Daniel/Enron protocol represent a 32-bit value using one 32-bit Modbus register instead of two 16-bit registers.

**Parameters**

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

Return values

<i>FTALK_SUCCESS</i>	Success
<i>FTALK_ILLEGAL_ARGUMENT_ERR</i> ↔ <i>OR</i>	Argument out of range

**Note**

This function call also re-configures the endianness to big-endian for 32-bit values as defined by the Daniel/ENRON protocol!

References `MbusMasterFunctions::configureBigEndianInts()`, `MbusMasterFunctions↔::configureSwappedFloats()`, `FTALK_ILLEGAL_ARGUMENT_ERROR`, and `FTALK_SUCCESS`.

```
configureCountFromOne() [1/2] void configureCountFromOne ( ) [inherited]
```

Configures the reference counting scheme to start with one for all slaves.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

**Remarks**

This is the default mode

```
configureCountFromOne() [2/2] int configureCountFromOne (
    int slaveAddr ) [inherited]
```

Configures a slave's reference counting scheme to start with one.

This renders the reference range to be 1 to 65536 (0x10000) and register #0 is an illegal register.

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

#### Remarks

This is the default mode

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**configureCountFromZero()** [1/2] void configureCountFromZero ( ) [inherited]

Configures the reference counting scheme to start with zero for all slaves.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

This renders the first register to be #0 for all slaves.

**configureCountFromZero()** [2/2] int configureCountFromZero (   
int *slaveAddr* ) [inherited]

Configures a slave's reference counting scheme to start with zero.

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535 (0xFFFF).

#### Parameters

<i>slaveAddr</i>	Modbus address of slave device or unit identifier (Range: 1 - 255). A value of zero configures the behaviour for broadcasting.
------------------	--

References FTALK\_ILLEGAL\_ARGUMENT\_ERROR, and FTALK\_SUCCESS.

**getPackageVersion()** const TCHAR \* getPackageVersion ( ) [static], [inherited]

Returns the library version number.

#### Returns

Library version string

## 9 License

### Library License

proconX Pty Ltd, Brisbane/Australia, ACN 104 080 935

Revision 4, October 2008

#### Definitions

"Software" refers to the collection of files and any part hereof, including, but not limited to, source code, programs, binary executables, object files, libraries, images, and scripts, which are distributed by proconX.

"Copyright Holder" is whoever is named in the copyright or copyrights for the Software.

"You" is you, if you are thinking about using, copying or distributing this Software or parts of it.

"Distributable Components" are dynamic libraries, shared libraries, class files and similar components supplied by proconX for redistribution. They must be listed in a "README" or "DEPLOY" file included with the Software.

"Application" pertains to Your product be it an application, applet or embedded software product.

---

#### License Terms

1. In consideration of payment of the licence fee and your agreement to abide by the terms and conditions of this licence, proconX grants You the following non-exclusive rights:
  - a. You may use the Software on one or more computers by a single person who uses the software personally;
  - b. You may use the Software nonsimultaneously by multiple people if it is installed on a single computer;
  - c. You may use the Software on a network, provided that the network is operated by the organisation who purchased the license and there is no concurrent use of the Software;
  - d. You may copy the Software for archival purposes.
2. You may reproduce and distribute, in executable form only, Applications linked with static libraries supplied as part of the Software and Applications incorporating dynamic libraries, shared libraries and similar components supplied as Distributable Components without royalties provided that:
  - a. You paid the license fee;
  - b. the purpose of distribution is to execute the Application;
  - c. the Distributable Components are not distributed or resold apart from the Application;
  - d. it includes all of the original Copyright Notices and associated Disclaimers;
  - e. it does not include any Software source code or part thereof.
3. If You have received this Software for the purpose of evaluation, proconX grants You a non-exclusive license to use the Software free of charge for the purpose of evaluating whether to purchase an ongoing license to use the Software. The evaluation period is limited to 30 days and does not include the right to reproduce and distribute Applications using the Software. At the end of the evaluation period, if You do not purchase a license, You must uninstall the Software from the computers or devices You installed

it on.

4. You are not required to accept this License, since You have not signed it. However, nothing else grants You permission to use or distribute the Software or its derivative works. These actions are prohibited by law if You do not accept this License. Therefore, by using or distributing the Software (or any work based on the Software), You indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or using the Software or works based on it.
5. You may not use the Software to develop products which can be used as a replacement or a directly competing product of this Software.
6. Where source code is provided as part of the Software, You may modify the source code for the purpose of improvements and defect fixes. If any modifications are made to any the source code, You will put an additional banner into the code which indicates that modifications were made by You.
7. You may not disclose the Software's software design, source code and documentation or any part thereof to any third party without the expressed written consent from proconX.
8. This License does not grant You any title, ownership rights, rights to patents, copyrights, trade secrets, trademarks, or any other rights in respect to the Software.
9. You may not use, copy, modify, sublicense, or distribute the Software except as expressly provided under this License. Any attempt otherwise to use, copy, modify, sublicense or distribute the Software is void, and will automatically terminate your rights under this License.
10. The License is not transferable without written permission from proconX.
11. proconX may create, from time to time, updated versions of the Software. Updated versions of the Software will be subject to the terms and conditions of this agreement and reference to the Software in this agreement means and includes any version update.
12. THERE IS NO WARRANTY FOR THE SOFTWARE, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING PROCONX, THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE SOFTWARE "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH YOU. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. ANY LIABILITY OF PROCONX WILL BE LIMITED EXCLUSIVELY TO REFUND OF PURCHASE PRICE. IN ADDITION, IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL PROCONX OR ITS PRINCIPALS, SHAREHOLDERS, OFFICERS, EMPLOYEES, AFFILIATES, CONTRACTORS, SUBSIDIARIES, PARENT ORGANIZATIONS AND ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE SOFTWARE AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE SOFTWARE TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
14. IN ADDITION, IN NO EVENT DOES PROCONX AUTHORIZE YOU TO USE THIS SOFTWARE IN APPLICATIONS OR SYSTEMS WHERE IT'S FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO RESULT IN A SIGNIFICANT PHYSICAL INJURY, OR IN LOSS OF LIFE. ANY SUCH USE BY YOU IS ENTIRELY AT YOUR OWN RISK, AND YOU AGREE TO HOLD PROCONX HARMLESS FROM ANY CLAIMS OR LOSSES RELATING TO SUCH UNAUTHORIZED USE.
15. This agreement constitutes the entire agreement between proconX

and You in relation to your use of the Software. Any change will be effective only if in writing signed by proconX and you.

16. This License is governed by the laws of Queensland, Australia, excluding choice of law rules. If any part of this License is found to be in conflict with the law, that part shall be interpreted in its broadest meaning consistent with the law, and no other parts of the License shall be affected.
-

## 10 Support

We provide electronic support and feedback for our FieldTalk products. Please use the Support web page at: <https://www.modbusdriver.com/support>

Your feedback is always welcome. It helps improving this product.

---

# 11 Notices

**Disclaimer:**

proconX Pty Ltd makes no warranty for the use of its products, other than those expressly contained in the Company's standard warranty which is detailed in the Terms and Conditions located on the Company's Website. The Company assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information contained herein. No licenses to patents or other intellectual property of proconX are granted by the Company in connection with the sale of proconX products, expressly or by implication. proconX products are not authorized for use as critical components in life support devices or systems.

This FieldTalk™ library was developed by:

proconX Pty Ltd, Australia.

Copyright © 2002-2025. All rights reserved.

proconX and FieldTalk are trademarks of proconX Pty Ltd. Modbus is a registered trademark of Schneider Automation Inc. All other product and brand names mentioned in this document may be trademarks or registered trademarks of their respective owners.