FieldTalk Modbus Master C++ Library

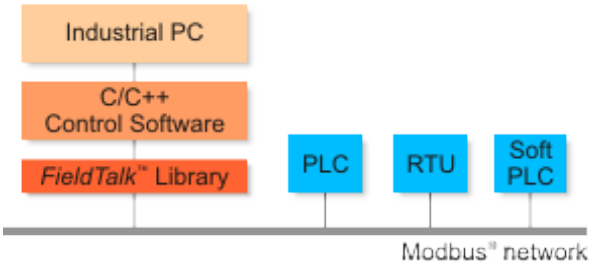
Software manual

Library version 2.6.4

Contents

1 Introduction

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



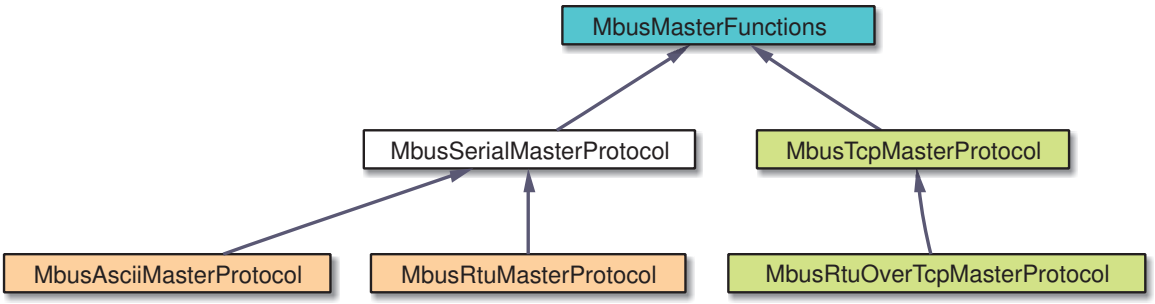
Typical applications are Modbus based Supervisory Control and Data Acquisition Systems (SCADA), Modbus data concentrators, Modbus gateways, User Interfaces and Factory Information Systems (FIS).

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** (p. 136) 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** (p. 84) implements the Modbus ASCII protocol, the class **MbusRtuMasterProtocol** (p. 57) implements the Modbus RTU protocol. The class **MbusTcpMasterProtocol** (p. 32) implements the MODBUS/TCP protocol and the class **MbusRtuOverTcpMasterProtocol** (p. 110) 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 compromises 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
Bit Access		
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
16 Bits Access		
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
Diagnostics		
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
Vendor Specific		
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 consective registers
- Reading and Writing 32-bit floats as two consective 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 discretes 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
readInputDiscretes	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 discretes 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 Discrete Number	Bit Number	Modbus Discrete Number	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 functions and methdos 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 continuos 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 Systems: Unpacking and Compiling the Source

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.2.6.0.zip
```

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

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

3. Compile the library from the source code.

To compile using command line tools, enter the FieldTalk src directory and run the make file.

If you are using Microsoft C++ and nmake:

```
# cd fieldtalk\src
# nmake
```

To compile using Visual Studio, open the supplied .sln solution files with Visual Studio 2003 or 2005.

4. The library will be compiled into one of the following sub-directories of your project directory:

Platform	Library Directory
Windows 32-bit Visual Studio 2003 or 2005	lib\win\win32\release
Windows CE Visual Studio 2005	lib\wce\[platformname]\release

Your directory structure looks now like:

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

```
+-- samples
+-+ lib
  |
  +-- win
  |   |
  |   +-- win32
  |       |
  |       +-- release
  |
  +-- wce
      |
      +-- [platformname]
          |
          +-- release
```

5. The library is ready to be used.

3.2 Specific Platform Notes

3.2.1 ucLinux

Instead of using the default Linux build script, use the make script with the patform.uclinux configuration file by passing uclinux as parameter:

```
./make uclinux
```

You can edit the architecture settings and CPU flags in platform.uclinux to suit your processor.

3.2.2 arm-linux cross tools

Instead of using the default Linux build script, use the make script with the patform.arm-linux configuration file by passing arm-linux as parameter:

```
./make arm-linux
```

3.2.3 QNX 4

In order to get proper control over Modebus timing, you have to adjust the system's clock rate. The standard ticksize is not suitable for Modbus RTU and needs to be adjusted. Configure the ticksize to be <= 1 ms.

3.2.4 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 Linux, UNIX and QNX Systems: Compiling and Linking Applications

Let's assume the following project directory structure:

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

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

```
c++ -Ifieldtalk/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
```

4.2 Windows Systems: Compiling and Linking Applications

Let's assume the following project directory structure:

```
myprj
|
+-- fieldtalk
    |
    +-- doc
    +-- samples
    +-- src
    +-- include
    +-+ lib
      |
      +-- win
          |
          +-- win32
              |
              +-- release
```

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

Visual C++ Example:

```
cl -Ifieldtalk/include -c myapp.cpp
```

Borland C++ Example:

```
bcc32 -Ifieldtalk/include -c myapp.cpp
```

Add the file name of the library to the file list passed to the linker. Visual C++ only: If you are using the Modbus/TCP protocol you have to add the Winsock2 library Ws2_32.lib.

Visual C++ Example:

```
cl -Fe myapp myapp.obj
   fieldtalk/lib/win/win32/release/libmbusmaster.lib Ws2_32.lib
```

4.3 Linux, UNIX and QNX Systems: 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.2.6.0.tar.gz
```

3. Untar the tarball

```
# tar xf FT-MBMP-??-ALL.2.6.0.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 call the make script:

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

The make shell script tries to detect your platform and executes the compiler and linker commands.

The compiler and linker configuration is contained in the file src/platform.

To cross-compile for ucLinux or arm-linux pass uclinux or arm-linux as a parameter to the the make script:

```
# ./make arm-linux
```

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

Platform	Library Directory
Linux	lib/linux
QNX 6	lib/qnx6
QNX 4	lib/qnx4
Irix	lib/irix
OSF1/True 64/Digital UNIX	lib/osf
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.

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 mutliple 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 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 "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 mutliple 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.

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
Bit Access		
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
16 Bits Access		
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
Diagnostics		
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
Vendor Specific		
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** (p. 6). (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 Device and Vendor Specific Modbus Functions

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

Advantec ADAM 5000/6000 Series Commands

- int **adamSendReceiveAsciiCmd** (const char ∗const commandSz, char ∗responseSz)

  Send/Receive ADAM 5000/6000 ASCII command.

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.

7.2.1 Detailed Description

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

7.2.2 Function Documentation

int adamSendReceiveAsciiCmd (const char ∗const *commandSz*, char ∗ *responseSz*) `[inherited]`

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:

commandSz Buffer which holds command string. Must not be longer than 255 characters.

responseSz 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

int customFunction (int *slaveAddr*, int *functionCode*, void * *requestData*, size_t *requestLen*, void * *responseData*, size_t * *responseLenPtr*) [inherited]

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 subsequentially 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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

functionCode Custom function code to be used for Modbus transaction (1-127)

requestData Pointer to data sent as request (not including slave address or function code)

requestLen Length of request data structure (0-252)

responseData Pointer to data structure which holds response data

responseLenPtr Length of response data (0-252). The number of bytes expected to be sent as response must be known when submitting the request.

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

7.3 TCP/IP Protocols

The library provides two flavours of TCP/IP based Modbus protocols.

Classes

- class **MbusTcpMasterProtocol**

  MODBUS/TCP Master Protocol class.

- class **MbusRtuOverTcpMasterProtocol**

  Encapsulated Modbus RTU Master Protocol class.

7.3.1 Detailed Description

The library provides two flavours of TCP/IP based Modbus protocols. The MODBUS/TCP master protocol is implemented in the class **MbusTcpMasterProtocol** (p. 32). The Encapsulated Modbus RTU master protocol is implemented in the class **MbusRtuOverTcpMasterProtocol** (p. 110).

Both classes 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. For a more detailed description of the data and control functions see section **Data and Control Functions for all Modbus Protocol Flavours** (p. 18).

Using multiple instances of a **MbusTcpMasterProtocol** (p. 32) class enables concurrent protocol transfers using multiple TCP/IP sessions. They should be executed in separate threads.

See section **The MODBUS/TCP Protocol** (p. 7) for some background information about MODBUS/TCP.

See section **Using MODBUS/TCP Protocol** (p. 15) for an example how to use the **MbusTcpMasterProtocol** (p. 32) class.

7.4 Serial Protocols

The two serial protocol flavours are implemented in the **MbusRtuMasterProtocol** (p. 57) and **MbusAsciiMasterProtocol** (p. 84) class.

Classes

- class **MbusRtuMasterProtocol**

  Modbus RTU Master Protocol class.

- class **MbusAsciiMasterProtocol**

  Modbus ASCII Master Protocol class.

7.4.1 Detailed Description

The two serial protocol flavours are implemented in the **MbusRtuMasterProtocol** (p. 57) and **MbusAsciiMasterProtocol** (p. 84) class. 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 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** (p. 18).

Using multiple instances of a **MbusRtuMasterProtocol** (p. 57) or **MbusAsciiMasterProtocol** (p. 84) class enables concurrent protocol transfers on multiple COM ports (They should be executed in separate threads).

See sections **The RTU Protocol** (p. 7) and **The ASCII Protocol** (p. 7) for some background information about the two serial Modbus protocols.

See section **Using Serial Protocols** (p. 13) for an example how to use the **MbusRtuMasterProtocol** (p. 57) class.

7.5 Encapsulated Modbus RTU Protocol

The Encapsulated Modbus RTU master protocol is implemented in the class **MbusRtuOverTcpMasterProtocol** (p. 110).

Classes

- class **MbusRtuOverTcpMasterProtocol**

  Encapsulated Modbus RTU Master Protocol class.

7.5.1 Detailed Description

The Encapsulated Modbus RTU master protocol is implemented in the class **MbusRtuOverTcpMasterProtocol** (p. 110). 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 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** (p. 18).

Using multiple instances of a **MbusRtuOverTcpMasterProtocol** (p. 110) class enables concurrent protocol transfers using multiple TCP/IP sessions (They should be executed in separate threads).

7.6 Error Management

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

Defines

- #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.

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.

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 indentifer.
- #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.

7.6.1 Detailed Description

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

7.6.2 Define Documentation

#define FTALK_SUCCESS 0

Operation was successful.

This return codes indicates no 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.

#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 succesfully yet.

#define FTALK_EVALUATION_EXPIRED 3

Evaluation expired.

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

#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.

#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.

#define FTALK_IO_ERROR_CLASS 64

I/O error class.

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

#define FTALK_IO_ERROR 65

I/O error.

The underlaying I/O system reported an error.

#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.

#define FTALK_PORT_ALREADY_OPEN 67

Serial port already open.

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

#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.

#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.

#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.

#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.

#define FTALK_LISTEN_FAILED 72

Listen failed.

The listen operation on the specified TCP port failed..

#define FTALK_FILEDES_EXCEEDED 73

File descriptors exceeded.

Maximum number of usable file descriptors exceeded.

#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.

#define FTALK_PORT_NOT_AVAIL 75

TCP port not available.

The specified TCP port is not available on this machine.

#define FTALK_LINE_BUSY_ERROR 76

Serial line busy.

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

#define FTALK_BUS_PROTOCOL_ERROR_CLASS 128

Fieldbus protocol error class.

Signals that a fieldbus protocol related error has occured. 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.

#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.

#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.

#define FTALK_INVALID_REPLY_ERROR 131

Invalid reply error.

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

#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 adress will also cause this error. In some occasions this exception is also produced if the characters received don't constitute a complete frame.

#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.

#define FTALK_INVALID_MBAP_ID 134

Invalid MPAB indentifer.

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

#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.

#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.

#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.

#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.

#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 occured while processing the requested action. This response is also sent if the request would generate a response whose size exceeds the allowable data size.

#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.

#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.

7.6.3 Function Documentation

const TCHAR* getBusProtocolErrorText (int *errCode*)

Translates a numeric error code into a description string.

Parameters:

errCode FieldTalk error code

Returns:

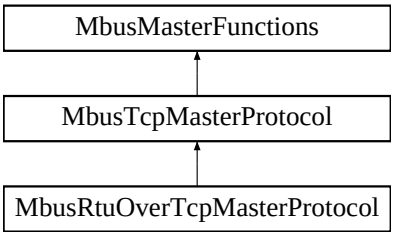
Error text string

8 C++ Class Documentation

8.1 MbusTcpMasterProtocol Class Reference

MODBUS/TCP Master Protocol class.

Inheritance diagram for MbusTcpMasterProtocol:



Public Member Functions

- **MbusTcpMasterProtocol** ()
  *Constructs a **MbusTcpMasterProtocol** (p. 32) object and initialises its data.*
- int **openProtocol** (const TCHAR ∗const hostName)
  Connects to a MODBUS/TCP slave.
- virtual void **closeProtocol** ()
  Closes a TCP/IP connection to a slave and releases any system resources associated with the connection.
- virtual int **isOpen** ()
  Returns whether the protocol is open or not.
- int **setPort** (unsigned short portNo)
  Sets the TCP port number to be used by the protocol.
- unsigned short **getPort** ()
  Returns the TCP port number used by the protocol.

Advantec ADAM 5000/6000 Series Commands

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

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.

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.

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 **configureIeeeFloats** ()

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

- int **configureIeeeFloats** (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/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 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** (p. 18).

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 (p. 18), **TCP/IP Protocols** (p. 20)
MbusMasterFunctions (p. 136), **MbusSerialMasterProtocol** (p. 160), **MbusRtuMasterProtocol** (p. 57), **MbusAsciiMasterProtocol** (p. 84), **MbusRtuOverTcpMasterProtocol** (p. 110)

8.1.2 Member Function Documentation

int openProtocol (const TCHAR *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.
The default TCP port number is 502.

Parameters:

hostName String with IP address or host name

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Reimplemented in **MbusRtuOverTcpMasterProtocol** (p. 115).

int isOpen ( ) [virtual]

Returns whether the protocol is open or not.

Return values:

true = open
false = closed

Implements **MbusMasterFunctions** (p. 159).

int setPort (unsigned short *portNo*)

Sets the TCP port number to be used by the protocol.

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().

Parameters:

portNo Port number to be used when opening the connection

Return values:

FTALK_SUCCESS Success
FTALK_ILLEGAL_STATE_ERROR Protocol already open

Reimplemented in **MbusRtuOverTcpMasterProtocol** (p. 116).

unsigned short getPort ( ) [inline]

Returns the TCP port number used by the protocol.

Returns:

Port number used by the protocol

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

bitAddr Coil address (Range: 1 - 65536)

bitVal true sets, false clears discrete output variable

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which contains the data to be sent

refCnt Number of coils to be written (Range: 1-1968)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer which will be filled with the data read

refCnt Number of registers to be read (Range: 1-125)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer which will be filled with the data read.

refCnt Number of registers to be read (Range: 1-125)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

regAddr Register address (Range: 1 - 65536)

regVal Data to be sent

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer with the data to be sent.

refCnt Number of registers to be written (Range: 1-123)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

regAddr Register address (Range: 1 - 65536)

andMask Mask to be applied as a logic AND to the register

orMask Mask to be applied as a logic OR to the register

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

readRef Start register for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read

readCnt Number of registers to be read (Range: 1-125)

writeRef Start register for writing (Range: 1 - 65536)

writeArr Buffer with data to be sent

writeCnt Number of registers to be written (Range: 1-121)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integers to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of float values to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of floats to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer with the data to be sent

refCnt Number of float values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integer values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

statusBytePtr 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

queryArr Buffer with data to be sent

echoArr Buffer which will contain the data read

len 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

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

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

clearEventLog Flag when set to one clears in addition the slave's communication even log.

Returns:

FTALK_SUCCESS on success. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

msTime Timeout value in ms (Range: 1 - 100000)

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

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

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getPollDelay ( ) [inline, inherited]

Returns the poll delay time.

Returns:

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

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:

retries Retry count (Range: 0 - 10), 0 disables retries

Return values:

FTALK_SUCCESS Success
FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range
FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getRetryCnt ( ) [inline, inherited]

Returns the automatic retry count.

Returns:

Retry count

long getTotalCounter ( ) [inline, inherited]

Returns how often a message transfer has been executed.

Returns:

Counter value

long getSuccessCounter ( ) [inline, inherited]

Returns how often a message transfer was successful.

Returns:

Counter value

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.

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:

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

void configureLittleEndianInts ( ) [inherited]

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

This is the default.

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:

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

void configureIeeeFloats ( ) [inherited]

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

This is the default.

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:

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

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.

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:

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

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.

Remarks:

This is the default mode

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Remarks:

This is the default mode

Note:

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

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.

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Note:

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

void configureCountFromOne ( ) [inherited]

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

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

Remarks:

This is the default mode

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 0x10000 and register #0 is an illegal register.

Parameters:

slaveAddr 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

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 0xFFFF.

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

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 0xFFFF.

Parameters:

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

const TCHAR * getPackageVersion ( ) [static, inherited]

Returns the library version number.

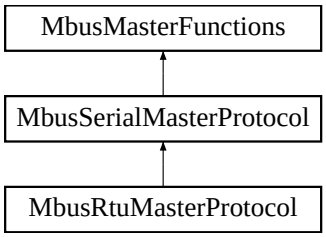
Returns:

Library version string

8.2 MbusRtuMasterProtocol Class Reference

Modbus RTU Master Protocol class.

Inheritance diagram for MbusRtuMasterProtocol:



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** (p. 57) object and initialises its data.*

- virtual int **openProtocol** (const TCHAR *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** ()

  Returns whether the protocol is open or not.

- 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.

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.

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 **configureIeeeFloats** ()
  Configures float data type functions not to do a word swap.
- int **configureIeeeFloats** (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 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 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** (p. 18).

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 (p. 18), **Serial Protocols** (p. 21)
MbusMasterFunctions (p. 136), **MbusSerialMasterProtocol** (p. 160), **MbusAsciiMasterProtocol** (p. 84), **MbusTcpMasterProtocol** (p. 32), **MbusRtuOverTcpMasterProtocol** (p. 110)

8.2.2 Member Enumeration Documentation

anonymous enum `[inherited]`

Enumerator:

SER_DATABITS_7 7 data bits

SER_DATABITS_8 8 data bits

anonymous enum [inherited]

Enumerator:

SER_STOPBITS_1 1 stop bit

SER_STOPBITS_2 2 stop bits

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

int openProtocol (const TCHAR *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:

portName Serial port identifier (e.g. "COM1", "/dev/ser1" or "/dev/ttyS0")

baudRate The port baudRate in bps (typically 1200 - 115200, maximum value depends on UART hardware)

dataBits Must be SER_DATABITS_8 for RTU

stopBits SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits

parity 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** (p. 22) for a list of error codes.

Reimplemented from **MbusSerialMasterProtocol** (p. 166).

int isOpen ( ) [virtual, inherited]

Returns whether the protocol is open or not.

Return values:

true = open

false = closed

Implements **MbusMasterFunctions** (p. 159).

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 to 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, it's scheduling priority and it's system timer resolution.

Note:

A protocol must be closed in order to configure it.

Parameters:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

bitAddr Coil address (Range: 1 - 65536)

bitVal true sets, false clears discrete output variable

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which contains the data to be sent

refCnt Number of coils to be written (Range: 1-1968)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer which will be filled with the data read

refCnt Number of registers to be read (Range: 1-125)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer which will be filled with the data read.

refCnt Number of registers to be read (Range: 1-125)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

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

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

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

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

regAddr Register address (Range: 1 - 65536)

andMask Mask to be applied as a logic AND to the register

orMask Mask to be applied as a logic OR to the register

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

readRef Start register for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read

readCnt Number of registers to be read (Range: 1-125)

writeRef Start register for writing (Range: 1 - 65536)

writeArr Buffer with data to be sent

writeCnt Number of registers to be written (Range: 1-121)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integers to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of float values to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of floats to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer with the data to be sent

refCnt Number of float values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integer values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

statusBytePtr 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

queryArr Buffer with data to be sent

echoArr Buffer which will contain the data read

len 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

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

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

clearEventLog Flag when set to one clears in addition the slave's communication even log.

Returns:

FTALK_SUCCESS on success. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

msTime Timeout value in ms (Range: 1 - 100000)

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

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

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getPollDelay ( ) [inline, inherited]

Returns the poll delay time.

Returns:

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

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:

retries Retry count (Range: 0 - 10), 0 disables retries

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getRetryCnt ( ) [inline, inherited]

Returns the automatic retry count.

Returns:

Retry count

long getTotalCounter ( ) [inline, inherited]

Returns how often a message transfer has been executed.

Returns:

Counter value

long getSuccessCounter ( ) [inline, inherited]

Returns how often a message transfer was successful.

Returns:

Counter value

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.

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:

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

void configureLittleEndianInts ( ) [inherited]

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

This is the default.

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:

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

void configureIeeeFloats ( ) [inherited]

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

This is the default.

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:

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

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.

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:

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

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.

Remarks:

This is the default mode

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Remarks:

This is the default mode

Note:

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

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.

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:

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

Return values:

FTALK_SUCCESS Success
FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Note:

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

void configureCountFromOne ( ) [inherited]

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

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

Remarks:

This is the default mode

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 0x10000 and register #0 is an illegal register.

Parameters:

slaveAddr 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

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 0xFFFF.

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

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 0xFFFF.

Parameters:

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

const TCHAR * getPackageVersion ( ) [static, inherited]

Returns the library version number.

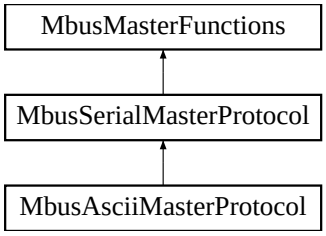
Returns:

Library version string

8.3 MbusAsciiMasterProtocol Class Reference

Modbus ASCII Master Protocol class.

Inheritance diagram for MbusAsciiMasterProtocol:



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** (p. 84) object and initialises its data.*

- virtual int **openProtocol** (const TCHAR *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** ()

  Returns whether the protocol is open or not.

- 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.

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.

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 **configureIeeeFloats** ()

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

- int **configureIeeeFloats** (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

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 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** (p. 18).

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 (p. 18), **Serial Protocols** (p. 21)
MbusMasterFunctions (p. 136), **MbusSerialMasterProtocol** (p. 160), **MbusRtuMasterProtocol** (p. 57) **MbusTcpMasterProtocol** (p. 32), **MbusRtuOverTcpMasterProtocol** (p. 110)

8.3.2 Member Enumeration Documentation

anonymous enum `[inherited]`

Enumerator:

SER_DATABITS_7 7 data bits

SER_DATABITS_8 8 data bits

anonymous enum `[inherited]`

Enumerator:

SER_STOPBITS_1 1 stop bit

SER_STOPBITS_2 2 stop bits

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

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

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:

portName Serial port identifier (e.g. "COM1", "/dev/ser1" or "/dev/ttyS0")

baudRate The port baudRate in bps (typically 1200 - 115200, maximum value depends on UART hardware)

dataBits SER_DATABITS_7: 7 data bits (ASCII protocol only), SER_DATABITS_8: data bits

stopBits SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits

parity 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** (p. 22) for a list of error codes.

Reimplemented from **MbusSerialMasterProtocol** (p. 166).

int isOpen ( ) [virtual, inherited]

Returns whether the protocol is open or not.

Return values:

true = open

false = closed

Implements **MbusMasterFunctions** (p. 159).

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 to 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, it's scheduling priority and it's system timer resolution.

Note:

A protocol must be closed in order to configure it.

Parameters:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

bitAddr Coil address (Range: 1 - 65536)

bitVal true sets, false clears discrete output variable

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which contains the data to be sent

refCnt Number of coils to be written (Range: 1-1968)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer which will be filled with the data read

refCnt Number of registers to be read (Range: 1-125)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

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

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

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

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer with the data to be sent.

refCnt Number of registers to be written (Range: 1-123)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

regAddr Register address (Range: 1 - 65536)

andMask Mask to be applied as a logic AND to the register

orMask Mask to be applied as a logic OR to the register

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

readRef Start register for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read

readCnt Number of registers to be read (Range: 1-125)

writeRef Start register for writing (Range: 1 - 65536)

writeArr Buffer with data to be sent

writeCnt Number of registers to be written (Range: 1-121)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integers to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of float values to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of floats to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer with the data to be sent

refCnt Number of float values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integer values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

statusBytePtr 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

queryArr Buffer with data to be sent

echoArr Buffer which will contain the data read

len 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

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

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

clearEventLog Flag when set to one clears in addition the slave's communication even log.

Returns:

FTALK_SUCCESS on success. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

msTime Timeout value in ms (Range: 1 - 100000)

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

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

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getPollDelay ( ) [inline, inherited]

Returns the poll delay time.

Returns:

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

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:

retries Retry count (Range: 0 - 10), 0 disables retries

Return values:

FTALK_SUCCESS Success
FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range
FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getRetryCnt ( ) [inline, inherited]

Returns the automatic retry count.

Returns:

Retry count

long getTotalCounter ( ) [inline, inherited]

Returns how often a message transfer has been executed.

Returns:

Counter value

long getSuccessCounter ( ) [inline, inherited]

Returns how often a message transfer was successful.

Returns:

Counter value

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.

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:

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

void configureLittleEndianInts ( ) [inherited]

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

This is the default.

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:

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

void configureIeeeFloats ( ) [inherited]

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

This is the default.

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:

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

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.

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:

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

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.

Remarks:

This is the default mode

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Remarks:

This is the default mode

Note:

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

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.

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Note:

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

void configureCountFromOne ( ) [inherited]

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

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

Remarks:

This is the default mode

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 0x10000 and register #0 is an illegal register.

Parameters:

slaveAddr 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

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 0xFFFF.

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

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 0xFFFF.

Parameters:

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

const TCHAR * getPackageVersion ( ) [static, inherited]

Returns the library version number.

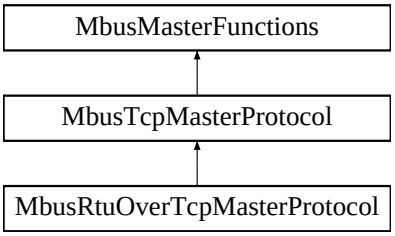
Returns:

Library version string

8.4 MbusRtuOverTcpMasterProtocol Class Reference

Encapsulated Modbus RTU Master Protocol class.

Inheritance diagram for MbusRtuOverTcpMasterProtocol:



Public Member Functions

- **MbusRtuOverTcpMasterProtocol** ()

  *Constructs a **MbusRtuOverTcpMasterProtocol** (p. 110) object and initialises its data.*

- int **openProtocol** (const TCHAR *const hostName)

  Connects to a Encapsulated Modbus RTU slave.

- int **setPort** (unsigned short portNo)

  Sets the TCP port number to be used by the protocol.

- virtual void **closeProtocol** ()

  Closes a TCP/IP connection to a slave and releases any system resources associated with the connection.

- virtual int **isOpen** ()

  Returns whether the protocol is open or not.

- 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.

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.

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 **configureIeeeFloats** ()
  Configures float data type functions not to do a word swap.
- int **configureIeeeFloats** (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.

8.4.1 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** (p. 18).

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 (p. 18), **TCP/IP Protocols** (p. 20)
MbusMasterFunctions (p. 136)
MbusSerialMasterProtocol (p. 160), **MbusRtuMasterProtocol** (p. 57), **MbusAsciiMasterProtocol** (p. 84)
MbusTcpMasterProtocol (p. 32)

8.4.2 Member Function Documentation

int openProtocol (const TCHAR *const *hostName*)

Connects to a Encapsulated Modbus RTU 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.
The default TCP port number is 1100.

Parameters:

hostName String with IP address or host name

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Reimplemented from **MbusTcpMasterProtocol** (p. 37).

int setPort (unsigned short *portNo*)

Sets the TCP port number to be used by the protocol.

Remarks:

Usually the port number remains unchanged and defaults to 1100. In this case no call to this function is necessary. However if the port number has to be different from 1100 this function must be called *before* opening the connection with openProtocol().

Parameters:

portNo Port number to be used when opening the connection

Return values:

FTALK_SUCCESS Success
FTALK_ILLEGAL_STATE_ERROR Protocol already open

Reimplemented from **MbusTcpMasterProtocol** (p. 38).

int isOpen ( ) [virtual, inherited]

Returns whether the protocol is open or not.

Return values:

true = open

false = closed

Implements **MbusMasterFunctions** (p. 159).

unsigned short getPort ( ) [inline, inherited]

Returns the TCP port number used by the protocol.

Returns:

Port number used by the protocol

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

bitAddr Coil address (Range: 1 - 65536)

bitVal true sets, false clears discrete output variable

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which contains the data to be sent

refCnt Number of coils to be written (Range: 1-1968)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer which will be filled with the data read

refCnt Number of registers to be read (Range: 1-125)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer which will be filled with the data read.

refCnt Number of registers to be read (Range: 1-125)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

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

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

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

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

regAddr Register address (Range: 1 - 65536)

andMask Mask to be applied as a logic AND to the register

orMask Mask to be applied as a logic OR to the register

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

readRef Start register for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read

readCnt Number of registers to be read (Range: 1-125)

writeRef Start register for writing (Range: 1 - 65536)

writeArr Buffer with data to be sent

writeCnt Number of registers to be written (Range: 1-121)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integers to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of float values to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of floats to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer with the data to be sent

refCnt Number of float values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integer values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

statusBytePtr 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

queryArr Buffer with data to be sent

echoArr Buffer which will contain the data read

len 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

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

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

clearEventLog Flag when set to one clears in addition the slave's communication even log.

Returns:

FTALK_SUCCESS on success. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

msTime Timeout value in ms (Range: 1 - 100000)

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

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

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getPollDelay ( ) [inline, inherited]

Returns the poll delay time.

Returns:

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

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:

retries Retry count (Range: 0 - 10), 0 disables retries

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getRetryCnt ( ) [inline, inherited]

Returns the automatic retry count.

Returns:

Retry count

long getTotalCounter ( ) [inline, inherited]

Returns how often a message transfer has been executed.

Returns:

Counter value

long getSuccessCounter ( ) [inline, inherited]

Returns how often a message transfer was successful.

Returns:

Counter value

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.

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:

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

void configureLittleEndianInts ( ) [inherited]

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

This is the default.

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:

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

void configureIeeeFloats ( ) [inherited]

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

This is the default.

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:

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

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.

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:

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

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.

Remarks:

This is the default mode

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Remarks:

This is the default mode

Note:

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

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.

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:

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

Return values:

FTALK_SUCCESS Success
FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Note:

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

void configureCountFromOne ( ) [inherited]

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

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

Remarks:

This is the default mode

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 0x10000 and register #0 is an illegal register.

Parameters:

slaveAddr 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

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 0xFFFF.

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

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 0xFFFF.

Parameters:

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

const TCHAR * getPackageVersion ( ) [static, inherited]

Returns the library version number.

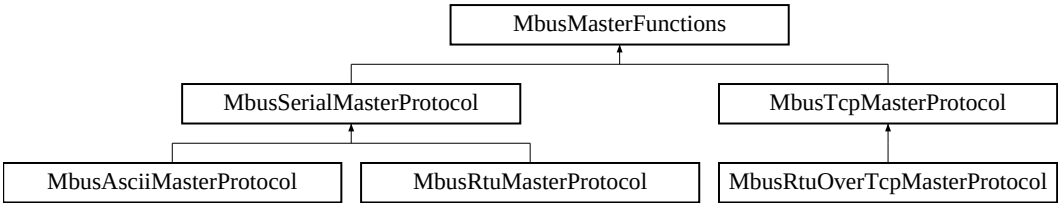
Returns:

Library version string

8.5 MbusMasterFunctions Class Reference

Base class which implements Modbus data and control functions.

Inheritance diagram for MbusMasterFunctions:



Public Member Functions

- virtual ~**MbusMasterFunctions** ()

  Destructor.

- virtual int **isOpen** ()=0

  Returns whether the protocol is open or not.

- virtual void **closeProtocol** ()=0

  Closes an open protocol including any associated communication resources (com ports or sockets).

Protected Member Functions

- **MbusMasterFunctions** ()

  *Constructs a **MbusMasterFunctions*** (p. 136) *object and initialises its data.*

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.

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.

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.
- void **configureLittleEndianInts** ()
  Configures 32-bit int data type functions not to do a word swap.
- void **configureIeeeFloats** ()
  Configures float data type functions not to do a word swap.
- void **configureSwappedFloats** ()
  Configures float data type functions to do a word swap.
- void **configureStandard32BitMode** ()
  Configures all slaves for Standard 32-bit Mode.
- void **configureEnron32BitMode** ()
  Configures all slaves for Daniel/ENRON 32-bit Mode.

- void **configureCountFromOne** ()
  Configures the reference counting scheme to start with one for all slaves.
- void **configureCountFromZero** ()
  Configures the reference counting scheme to start with zero for all slaves.
- int **configureBigEndianInts** (int slaveAddr)
  Enables int data type functions to do a word swap on a per slave basis.
- int **configureLittleEndianInts** (int slaveAddr)
  Disables word swapping for int data type functions on a per slave basis.
- int **configureIeeeFloats** (int slaveAddr)
  Disables float data type functions to do a word swap on a per slave basis.
- int **configureSwappedFloats** (int slaveAddr)
  Enables float data type functions to do a word swap on a per slave basis.
- int **configureStandard32BitMode** (int slaveAddr)
  Configures a slave for Standard 32-bit Register Mode.
- int **configureEnron32BitMode** (int slaveAddr)
  Configures all slaves for Daniel/ENRON 32-bit Mode.
- int **configureCountFromOne** (int slaveAddr)
  Configures a slave's reference counting scheme to start with one.
- 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

Base class which implements Modbus data and control functions. The functions provided by this base class apply to all protocol flavours via inheritance. For a more detailed description see section **Data and Control Functions for all Modbus Protocol Flavours** (p. 18).

See also:

Data and Control Functions for all Modbus Protocol Flavours (p. 18)

MbusSerialMasterProtocol (p. 160), **MbusRtuMasterProtocol** (p. 57) **MbusAsciiMasterProtocol** (p. 84), **MbusTcpMasterProtocol** (p. 32), **MbusRtuOverTcpMasterProtocol** (p. 110)

8.5.2 Constructor & Destructor Documentation

MbusMasterFunctions ( ) [protected]

Constructs a **MbusMasterFunctions** (p. 136) object and initialises its data.

It also detects the endianess of the machine it's running on and configures byte swapping if necessary.

~MbusMasterFunctions ( ) [virtual]

Destructor.

Does clean-up and closes an open protocol including any associated communication resources (serial ports or sockets).

8.5.3 Member Function Documentation

int readCoils (int *slaveAddr*, int *startRef*, int *bitArr[]*, int *refCnt*)

Modbus function 1, Read Coil Status/Read Coils.

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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

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

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

int writeCoil (int *slaveAddr*, int *bitAddr*, int *bitVal*)

Modbus function 5, Force Single Coil/Write Coil.

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

bitAddr Coil address (Range: 1 - 65536)

bitVal true sets, false clears discrete output variable

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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

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

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which contains the data to be sent

refCnt Number of coils to be written (Range: 1-1968)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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

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

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer which will be filled with the data read

refCnt Number of registers to be read (Range: 1-125)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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

Modbus function 4, Read Input Registers.

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer which will be filled with the data read.

refCnt Number of registers to be read (Range: 1-125)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

int writeSingleRegister (int *slaveAddr,* int *regAddr,* short *regVal*)

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

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

regAddr Register address (Range: 1 - 65536)

regVal Data to be sent

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer with the data to be sent.

refCnt Number of registers to be written (Range: 1-123)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

int maskWriteRegister (int *slaveAddr*, int *regAddr*, short *andMask*, short *orMask*)

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

regAddr Register address (Range: 1 - 65536)

andMask Mask to be applied as a logic AND to the register

orMask Mask to be applied as a logic OR to the register

Note:

No broadcast supported

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.

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

readRef Start register for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read

readCnt Number of registers to be read (Range: 1-125)

writeRef Start register for writing (Range: 1 - 65536)

writeArr Buffer with data to be sent

writeCnt Number of registers to be written (Range: 1-121)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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.

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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.

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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.

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integers to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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.

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of float values to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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.

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of floats to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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.

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer with the data to be sent

refCnt Number of float values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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.

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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.

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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.

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integer values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

int readExceptionStatus (int *slaveAddr,* unsigned char * *statusBytePtr*)

Modbus function 7, Read Exception Status.

Reads the eight exception status coils within the slave device.

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

statusBytePtr 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

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

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

queryArr Buffer with data to be sent

echoArr Buffer which will contain the data read

len 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

int restartCommunicationsOption (int *slaveAddr*, int *clearEventLog*)

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

clearEventLog Flag when set to one clears in addition the slave's communication even log.

Returns:

FTALK_SUCCESS on success. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

int setTimeout (int *msTime*)

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:

msTime Timeout value in ms (Range: 1 - 100000)

Return values:

FTALK_SUCCESS Success
FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range
FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getTimeout ( ) [inline]

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

int setPollDelay (int *msTime*)

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:

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

Return values:

FTALK_SUCCESS Success
FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range
FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getPollDelay ( ) [inline]

Returns the poll delay time.

Returns:

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

int setRetryCnt (int *retries*)

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:

retries Retry count (Range: 0 - 10), 0 disables retries

Return values:

FTALK_SUCCESS Success
FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range
FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getRetryCnt ( ) [inline]

Returns the automatic retry count.

Returns:

Retry count

long getTotalCounter ( ) [inline]

Returns how often a message transfer has been executed.

Returns:

Counter value

long getSuccessCounter ( ) [inline]

Returns how often a message transfer was successful.

Returns:

Counter value

void configureBigEndianInts ( )

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.

void configureLittleEndianInts ( )

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

This is the default.

void configureIeeeFloats ( )

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

This is the default.

void configureSwappedFloats ( )

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.

void configureStandard32BitMode ( )

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.

Remarks:

This is the default mode

void configureEnron32BitMode ( )

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.

void configureCountFromOne ( )

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

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

Remarks:

This is the default mode

void configureCountFromZero ( )

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

This renders the valid reference range to be 0 to 0xFFFF.

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

int configureBigEndianInts (int *slaveAddr*)

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:

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

int configureLittleEndianInts (int *slaveAddr*)

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:

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

int configureIeeeFloats (int *slaveAddr*)

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:

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

int configureSwappedFloats (int *slaveAddr*)

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:

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

int configureStandard32BitMode (int *slaveAddr*)

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Remarks:

This is the default mode

Note:

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

int configureEnron32BitMode (int *slaveAddr*)

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Note:

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

int configureCountFromOne (int *slaveAddr*)

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

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

Parameters:

slaveAddr 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

int configureCountFromZero (int *slaveAddr*)

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 0xFFFF.

Parameters:

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

virtual int isOpen ( ) `[pure virtual]`

Returns whether the protocol is open or not.

Return values:

true = open
false = closed

Implemented in **MbusTcpMasterProtocol** (p. 116), and **MbusSerialMasterProtocol** (p. 167).

const TCHAR * getPackageVersion ( ) [static]

Returns the library version number.

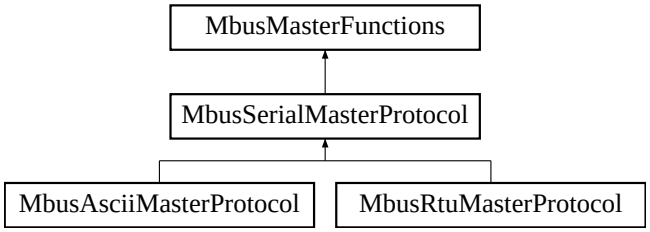
Returns:

Library version string

8.6 MbusSerialMasterProtocol Class Reference

Base class for serial serial master protocols.

Inheritance diagram for MbusSerialMasterProtocol:



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

- virtual int **openProtocol** (const TCHAR *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** ()

Returns whether the protocol is open or not.

- virtual int **enableRs485Mode** (int rtsDelay)
  Enables RS485 mode.

Protected Member Functions

- **MbusSerialMasterProtocol** ()
  *Constructs a **MbusSerialMasterProtocol*** (p. *160) object and initialises its data.*

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.

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.

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 **configureIeeeFloats** ()
  Configures float data type functions not to do a word swap.
- int **configureIeeeFloats** (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

Base class for serial serial master protocols. This base class realises the Modbus serial master protocols. 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 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** (p. 18).

It is possible to instantiate multiple instances 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 (p. 18), **Serial Protocols** (p. 21)
MbusMasterFunctions (p. 136), **MbusRtuMasterProtocol** (p. 57), **MbusAsciiMasterProtocol** (p. 84), **MbusTcpMasterProtocol** (p. 32), **MbusRtuOverTcpMasterProtocol** (p. 110)

8.6.2 Member Enumeration Documentation

anonymous enum

Enumerator:

SER_DATABITS_7 7 data bits

SER_DATABITS_8 8 data bits

anonymous enum

Enumerator:

SER_STOPBITS_1 1 stop bit

SER_STOPBITS_2 2 stop bits

anonymous enum

Enumerator:

SER_PARITY_NONE No parity.

SER_PARITY_EVEN Even parity.

SER_PARITY_ODD Odd parity.

8.6.3 Member Function Documentation

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

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:

portName Serial port identifier (e.g. "COM1", "/dev/ser1" or "/dev/ttyS0")

baudRate The port baudRate in bps (typically 1200 - 115200, maximum value depends on UART hardware)

dataBits SER_DATABITS_7: 7 data bits (ASCII protocol only), SER_DATABITS_8: data bits

stopBits SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits

parity 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** (p. 22) for a list of error codes.

Reimplemented in **MbusRtuMasterProtocol** (p. 63), and **MbusAsciiMasterProtocol** (p. 90).

int isOpen ( ) [virtual]

Returns whether the protocol is open or not.

Return values:

true = open

false = closed

Implements **MbusMasterFunctions** (p. 159).

int enableRs485Mode (int *rtsDelay*) [virtual]

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 to 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, it's scheduling priority and it's system timer resolution.

Note:

A protocol must be closed in order to configure it.

Parameters:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which will contain the data read

refCnt Number of coils to be read (Range: 1-2000)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

bitAddr Coil address (Range: 1 - 65536)

bitVal true sets, false clears discrete output variable

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

bitArr Buffer which contains the data to be sent

refCnt Number of coils to be written (Range: 1-1968)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer which will be filled with the data read

refCnt Number of registers to be read (Range: 1-125)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start register (Range: 1 - 65536)

regArr Buffer which will be filled with the data read.

refCnt Number of registers to be read (Range: 1-125)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

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

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

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

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

regAddr Register address (Range: 1 - 65536)

andMask Mask to be applied as a logic AND to the register

orMask Mask to be applied as a logic OR to the register

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

readRef Start register for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read

readCnt Number of registers to be read (Range: 1-125)

writeRef Start register for writing (Range: 1 - 65536)

writeArr Buffer with data to be sent

writeCnt Number of registers to be written (Range: 1-121)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integers to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of float values to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer which will be filled with the data read

refCnt Number of floats to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

float32Arr Buffer with the data to be sent

refCnt Number of float values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 65536)

int32Arr Buffer with the data to be sent

refCnt Number of long integer values to be sent (Range: 1-61)

Returns:

FTALK_SUCCESS on success or error code. See **Error Management** (p. 22) for a list of error codes.

Note:

Broadcast supported for serial protocols

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

statusBytePtr 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

queryArr Buffer with data to be sent

echoArr Buffer which will contain the data read

len 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** (p. 22) for a list of error codes.

Note:

No broadcast supported

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

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 1 - 255)

clearEventLog Flag when set to one clears in addition the slave's communication even log.

Returns:

FTALK_SUCCESS on success. See **Error Management** (p. 22) for a list of error codes.

Note:

No broadcast supported

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:

msTime Timeout value in ms (Range: 1 - 100000)

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

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

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getPollDelay ( ) [inline, inherited]

Returns the poll delay time.

Returns:

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

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:

retries Retry count (Range: 0 - 10), 0 disables retries

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getRetryCnt ( ) [inline, inherited]

Returns the automatic retry count.

Returns:

Retry count

long getTotalCounter ( ) [inline, inherited]

Returns how often a message transfer has been executed.

Returns:

Counter value

long getSuccessCounter ( ) [inline, inherited]

Returns how often a message transfer was successful.

Returns:

Counter value

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.

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:

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

void configureLittleEndianInts ( ) [inherited]

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

This is the default.

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:

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

void configureIeeeFloats ( ) [inherited]

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

This is the default.

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:

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

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.

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:

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

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.

Remarks:

This is the default mode

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:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Remarks:

This is the default mode

Note:

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

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.

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:

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

Return values:

FTALK_SUCCESS Success
FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

Note:

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

void configureCountFromOne ( ) [inherited]

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

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

Remarks:

This is the default mode

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 0x10000 and register #0 is an illegal register.

Parameters:

slaveAddr 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

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 0xFFFF.

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

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 0xFFFF.

Parameters:

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

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: `http://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.

Index