



FieldTalk Modbus Master Library, Delphi Edition Software manual

Library version 2.6.2

Contents

1	Introduction	1
1.1	Library Structure	1
2	What You should know about Modbus	3
2.1	Some Background	3
2.2	Technical Information	3
2.2.1	The Protocol Functions	3
2.2.2	How Slave Devices are identified	4
2.2.3	The Register Model and Data Tables	4
2.2.4	Data Encoding	5
2.2.5	Register and Discrete Numbering Scheme	6
2.2.6	The ASCII Protocol	7
2.2.7	The RTU Protocol	7
2.2.8	The MODBUS/TCP Protocol	7
3	How to integrate the Protocol in your Application	8
3.1	Using Serial Protocols	8
3.2	Using MODBUS/TCP Protocol	10
4	Design Background	12
5	Module Documentation	13
5.1	TCP/IP Protocols	13
5.1.1	Detailed Description	13
5.2	Encapsulated Modbus RTU Protocol	13
5.2.1	Detailed Description	14
5.3	Data and Control Functions for all Modbus Protocol Flavours	14
5.4	Serial Protocols	15
5.4.1	Detailed Description	16
5.5	Error Management	16
5.5.1	Detailed Description	18
6	Delphi Class Documentation	19
6.1	EBusProtocolException Class Reference	19
6.1.1	Detailed Description	19
6.2	EChecksumException Class Reference	19

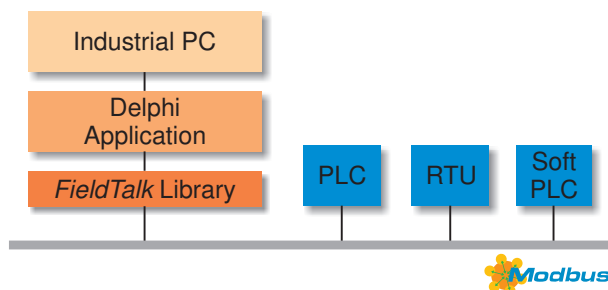
6.2.1	Detailed Description	19
6.3	EConnectionWasClosed Class Reference	19
6.3.1	Detailed Description	20
6.4	EEvaluationExpired Class Reference	20
6.4.1	Detailed Description	20
6.5	EFiledesExceeded Class Reference	20
6.5.1	Detailed Description	21
6.6	EIllegalArgumentError Class Reference	21
6.6.1	Detailed Description	21
6.7	EIllegalSlaveAddress Class Reference	21
6.7.1	Detailed Description	21
6.8	EIllegalStateError Class Reference	22
6.8.1	Detailed Description	22
6.9	EInvalidFrameException Class Reference	22
6.9.1	Detailed Description	22
6.10	EInvalidMbapIdException Class Reference	22
6.10.1	Detailed Description	23
6.11	EInvalidReplyException Class Reference	23
6.11.1	Detailed Description	23
6.12	EListenFailed Class Reference	23
6.12.1	Detailed Description	24
6.13	EMbusGatewayPathUnavailableException Class Reference	24
6.13.1	Detailed Description	24
6.14	EMbusGatewayTargetFailureException Class Reference	24
6.14.1	Detailed Description	25
6.15	EMbusIllegalAddressException Class Reference	25
6.15.1	Detailed Description	25
6.16	EMbusIllegalFunctionException Class Reference	26
6.16.1	Detailed Description	26
6.17	EMbusIllegalValueException Class Reference	26
6.17.1	Detailed Description	26
6.18	EMbusResponseException Class Reference	27
6.18.1	Detailed Description	27
6.19	EMbusSlaveFailureException Class Reference	27
6.19.1	Detailed Description	27
6.20	EOpenErr Class Reference	28

6.20.1 Detailed Description	28
6.21 EPortAlreadyBound Class Reference	28
6.21.1 Detailed Description	28
6.22 EPortAlreadyOpen Class Reference	29
6.22.1 Detailed Description	29
6.23 EPortNoAccess Class Reference	29
6.23.1 Detailed Description	29
6.24 EPortNotAvail Class Reference	29
6.24.1 Detailed Description	30
6.25 EReplyTimeoutException Class Reference	30
6.25.1 Detailed Description	30
6.26 ESendTimeoutException Class Reference	30
6.26.1 Detailed Description	31
6.27 ESocketLibError Class Reference	31
6.27.1 Detailed Description	31
6.28 ETcpipConnectErr Class Reference	31
6.28.1 Detailed Description	32
6.29 TMBusAsciiMasterProtocol Class Reference	32
6.29.1 Detailed Description	37
6.29.2 Constructor & Destructor Documentation	38
6.29.3 Member Function Documentation	38
6.29.4 Member Data Documentation	60
6.30 TMBusMasterFunctions Class Reference	63
6.30.1 Detailed Description	68
6.30.2 Constructor & Destructor Documentation	68
6.30.3 Member Function Documentation	68
6.30.4 Member Data Documentation	89
6.31 TMBusRtuMasterProtocol Class Reference	90
6.31.1 Detailed Description	95
6.31.2 Constructor & Destructor Documentation	95
6.31.3 Member Function Documentation	96
6.31.4 Member Data Documentation	118
6.32 TMBusRtuOverTcpMasterProtocol Class Reference	121
6.32.1 Detailed Description	126
6.32.2 Constructor & Destructor Documentation	127
6.32.3 Member Function Documentation	127

6.32.4	Member Data Documentation	149
6.33	TMbusSerialMasterProtocol Class Reference	151
6.33.1	Detailed Description	156
6.33.2	Member Function Documentation	156
6.33.3	Member Data Documentation	179
6.34	TMbusTcpMasterProtocol Class Reference	181
6.34.1	Detailed Description	187
6.34.2	Constructor & Destructor Documentation	187
6.34.3	Member Function Documentation	187
6.34.4	Member Data Documentation	209
7	License	212
8	Support	215
9	Notices	216

1 Introduction

The *FieldTalk*[™] Modbus[®] Master Library, Delphi Edition 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).

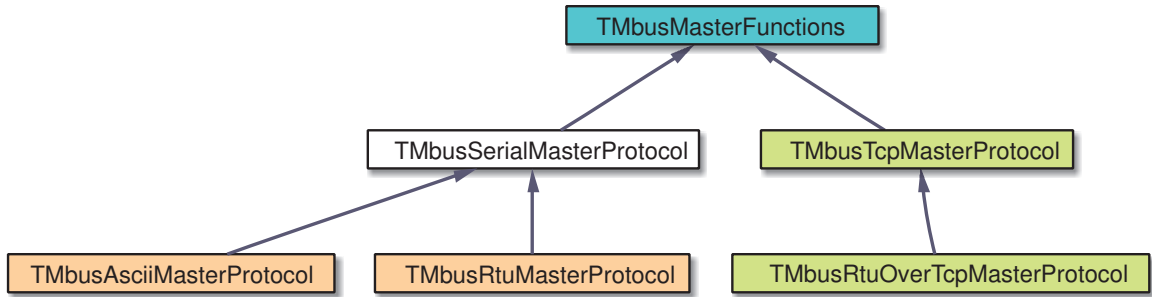
Features:

- 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
- 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 is implemented as a VCL class library. The VCL classes wrap around a libmbus-master DLL which performs the core protocol functions. The libmbusmaster DLL is based on the FieldTalk Modbus Master C++ Library, a proven and industrial-strength Modbus driver implementation.

The library 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 **TMbusMasterFunctions** (p. 63) contains all protocol unspecific functions, in particular the data and control functions defined by Modbus. All protocol flavours inherit from this base class.

The class **TMbusAsciiMasterProtocol** (p. 32) implements the Modbus ASCII protocol, the class **TMbusRtuMasterProtocol** (p. 90) implements the Modbus RTU protocol and the class **TMbusTcpMasterProtocol** (p. 181) implements the MODBUS/TCP protocol and the class **TMbusRtuOverTcpMasterProtocol** (p. 121) 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:

```
TMbusRtuMasterProtocol mbusProtocol;
```

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

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


2 What You should know about Modbus

2.1 Some Background

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

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

2.2 Technical Information

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

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

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

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

Note:

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

2.2.1 The Protocol Functions

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

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

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

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

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

2.2.5 Register and Discrete Numbering Scheme

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

2.2.6 The ASCII Protocol

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

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

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

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

2.2.7 The RTU Protocol

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

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

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

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

2.2.8 The MODBUS/TCP Protocol

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

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

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

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

3 How to integrate the Protocol in your Application

3.1 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 discrettes for reading are in the reference range 0:00010 to 0:00019 and the discrettes for writing are in the range 0:00020 to 0:00029.

1. Import the library packages

```
uses
    MbusRtuMasterProtocol,
    BusProtocolExceptions;
```

2. Device data profile definition

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

```
var
    readRegSet: array[1..20] of word;
    writeRegSet: array[1..10] of word;
    readBitSet: array[1..10] of boolean;
    writeBitSet: array[1..10] of boolean;
```

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

```
var
    readFloatSet: array[1..10] of single;
    writeFloatSet: array[1..10] of single;
```

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

```
var
    readLongSet: array[1..10] of integer;
    writeLongSet: array[1..10] of integer;
```

3. Declare and instantiate a protocol object

```
var
    TmbusRtuMasterProtocol mbusProtocol;
```

4. Instantiate and open the protocol

```
try
    mbusProtocol := TmbusRtuMasterProtocol.Create(nil);
    mbusProtocol.portName := 'COM1';
    mbusProtocol.baudRate := 19200;
    mbusProtocol.dataBits := 8;
    mbusProtocol.stopBits := 1;
    mbusProtocol.parity := 2;
    mbusProtocol.openProtocol;
```

```
except
  on e: Exception do
  begin
    writeln('Error opening protocol: ', e.message, '!');
    halt(1);
  end;
end;
```

5. Perform the data transfer functions

- To read register values:

```
mbusProtocol.readMultipleRegisters(1, 100, readRegSet);
```

- To write a single register value:

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

- To write multiple register values:

```
mbusProtocol.writeMultipleRegisters(1, 200, writeRegSet);
```

- To read discrete values:

```
mbusProtocol.readCoils(1, 10, readBitSet);
```

- To write a single discrete value:

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

- To write multiple discrete values:

```
mbusProtocol.forceMultipleCoils(1, 20, writeBitSet);
```

- To read float values:

```
mbusProtocol.readMultipleFloats(1, 100, readFloatSet);
```

- To read long integer values:

```
mbusProtocol.readMultipleLongInts(1, 100, readLongSet);
```

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 throw an exception. The following code snippet can handle and report these errors:

```
try
  mbusProtocol.readMultipleRegisters(1, 100, dataSetArray);
except
  on e: EBusProtocolException do
    writeln(e.message, '!');
  on e: Exception do
  begin
    writeln('Fatal error: ', e.message, '!');
    halt(1);
  end;
end;
```

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

3.2 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. Import the library package

```
uses
  MbusTcpMasterProtocol,
  BusProtocolExceptions;
```

2. Device data profile definition

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

```
var
  readRegSet: array[1..20] of word;
  writeRegSet: array[1..10] of word;
  readBitSet: array[1..10] of boolean;
  writeBitSet: array[1..10] of boolean;
```

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

```
var
  readFloatSet: array[1..10] of single;
  writeFloatSet: array[1..10] of single;
```

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

```
var
  readLongSet: array[1..10] of integer;
  writeLongSet: array[1..10] of integer;
```

3. Declare and instantiate a protocol object

```
var
  TmbusTcpMasterProtocol mbusProtocol;
```

4. Open the protocol

```
mbusProtocol.hostName := ' 10.0.0.11';
mbusProtocol.openProtocol;
```

5. Perform the data transfer functions

- To read register values:

```
mbusProtocol.readMultipleRegisters(1, 100, readRegSet);
```

- To write a single register value:

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


- To write mutiple register values:

```
mbusProtocol.writeMultipleRegisters(1, 200, writeRegSet);
```

- To read discrete values:

```
mbusProtocol.readCoils(1, 10, readBitSet);
```

- To write a single discrete value:

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

- To write multiple discrete values:

```
mbusProtocol.forceMultipleCoils(1, 20, writeBitSet);
```

- To read float values:

```
mbusProtocol.readMultipleFloats(1, 100, readFloatSet);
```

- To read long integer values:

```
mbusProtocol.readMultipleLongInts(1, 100, readLongSet);
```

6. Close the protocol port 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:

```
try
    mbusProtocol.readMultipleRegisters(1, 100, dataSetArray);
except
    on e: EBusProtocolException do
        writeln(e.message, '!');
    on e: Exception do
        begin
            writeln('Fatal error: ', e.message, '!');
            halt(1);
        end;
end;
```

If the method throws **EConnectionWasClosed** (p. 19), 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.

4 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.

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

5 Module Documentation

5.1 TCP/IP Protocols

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

Classes

- class **TMbusTcpMasterProtocol**
MODBUS/TCP Master Protocol class.
- class **TMbusRtuOverTcpMasterProtocol**
Encapsulated Modbus RTU Master Protocol class.

5.1.1 Detailed Description

The library provides two flavours of TCP/IP based Modbus protocols. The MODBUS/TCP master protocol is implemented in the class `MbusTcpMasterProtocol`. The Encapsulated Modbus RTU master protocol is implemented in the class `MbusRtuOverTcpMasterProtocol`.

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. 14).

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

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

See section **Using MODBUS/TCP Protocol** (p. 10) for an example how to use the `MbusTcpMasterProtocol` class.

5.2 Encapsulated Modbus RTU Protocol

The Encapsulated Modbus RTU master protocol is implemented in the class `MbusRtuOverTcpMasterProtocol`.

Classes

- class **TMbusRtuOverTcpMasterProtocol**
Encapsulated Modbus RTU Master Protocol class.

5.2.1 Detailed Description

The Encapsulated Modbus RTU master protocol is implemented in the class `MbusRtuOverTcpMasterProtocol`. 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 Protocol Flavours** (p. 14).

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

5.3 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).

5.4 Serial Protocols

The two serial protocol flavours are implemented in the MbusRtuMasterProtocol and MbusAsciiMasterProtocol class.

Classes

- class **TMbusRtuMasterProtocol**
Modbus RTU Master Protocol class.

- class **TModbusAsciiMasterProtocol**
Modbus ASCII Master Protocol class.

5.4.1 Detailed Description

The two serial protocol flavours are implemented in the `MbusRtuMasterProtocol` and `MbusAsciiMasterProtocol` 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. 14).

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

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

See section **Using Serial Protocols** (p. 8) for an example how to use the `MbusRtuMasterProtocol` class.

5.5 Error Management

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

Classes

- class **EIllegalArgumentError**
Illegal argument error.
- class **EIllegalStateError**
Illegal state error.
- class **EEvaluationExpired**
Evaluation expired.
- class **EIllegalSlaveAddress**
Slave address 0 illegal for serial protocols.
- class **EOpenErr**
Port or socket open error.
- class **EPortAlreadyOpen**
Serial port already open.

- class **ETcpipConnectErr**
TCP/IP connection error.
- class **EConnectionWasClosed**
Remote peer closed TCP/IP connection.
- class **ESocketLibError**
Socket library error.
- class **EPortAlreadyBound**
TCP port already bound.
- class **EListenFailed**
Listen failed.
- class **EFiledesExceeded**
File descriptors exceeded.
- class **EPortNoAccess**
No permission to access serial port or TCP port.
- class **EPortNotAvail**
TCP port not available.
- class **EBusProtocolException**
Communication Error.
- class **EChecksumException**
Checksum error.
- class **EInvalidFrameException**
Invalid frame error.
- class **EInvalidReplyException**
Invalid reply error.
- class **EReplyTimeoutException**
Reply time-out.
- class **ESendTimeoutException**
Send time-out.
- class **EInvalidMbapIdException**
Invalid MPAB identifier.
- class **EMbusResponseException**

Modbus[®] exception response.

- class **EMbusIllegalFunctionException**
Illegal Function exception response.
- class **EMbusIllegalAddressException**
Illegal Data Address exception response.
- class **EMbusIllegalValueException**
Illegal Data Value exception response.
- class **EMbusSlaveFailureException**
Slave Device Failure exception response.
- class **EMbusGatewayPathUnavailableException**
Gateway Path Unavailable exception response.
- class **EMbusGatewayTargetFailureException**
Gateway Target Device Failed exception response.

5.5.1 Detailed Description

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

6 Delphi Class Documentation

6.1 EBusProtocolException Class Reference

Communication Error.

Inheritance diagram for EBusProtocolException:



6.1.1 Detailed Description

Communication Error. Errors derived from class indicate either communication faults or Modbus exceptions reported by the slave device.

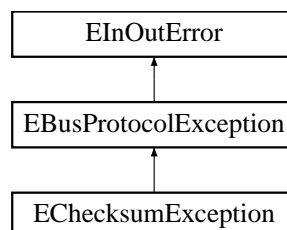
See also:

EInOutError (p. ??)

6.2 EChecksumException Class Reference

Checksum error.

Inheritance diagram for EChecksumException:



6.2.1 Detailed Description

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

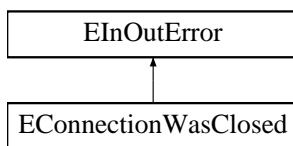
See also:

EBusProtocolException (p. 19)

6.3 EConnectionWasClosed Class Reference

Remote peer closed TCP/IP connection.

Inheritance diagram for EConnectionWasClosed:



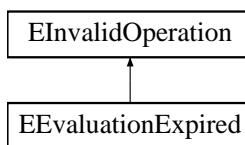
6.3.1 Detailed Description

Remote peer closed TCP/IP connection. Signals that the TCP/IP connection was closed by the remote peer or is broken.

6.4 EEvaluationExpired Class Reference

Evaluation expired.

Inheritance diagram for EEvaluationExpired:



6.4.1 Detailed Description

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

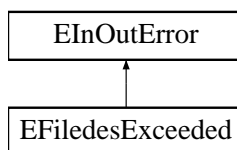
See also:

EInvalidOperation (p. ??)

6.5 EFiledesExceeded Class Reference

File descriptors exceeded.

Inheritance diagram for EFiledesExceeded:



6.5.1 Detailed Description

File descriptors exceeded. Maximum number of usable file descriptors exceeded.

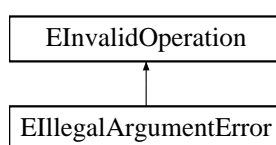
See also:

EInOutError (p. ??)

6.6 EIllegalArgumentError Class Reference

Illegal argument error.

Inheritance diagram for EIllegalArgumentError:



6.6.1 Detailed Description

Illegal argument error. A parameter passed to the method is invalid or out of range.

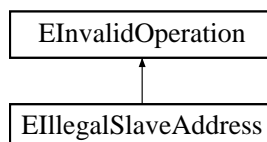
See also:

EInvalidOperation (p. ??)

6.7 EIllegalSlaveAddress Class Reference

Slave address 0 illegal for serial protocols.

Inheritance diagram for EIllegalSlaveAddress:



6.7.1 Detailed Description

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 therefore illegal.

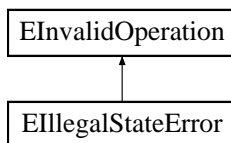
See also:

EInvalidOperation (p. ??)

6.8 EIllegalStateError Class Reference

Illegal state error.

Inheritance diagram for EIllegalStateError:



6.8.1 Detailed Description

Illegal state error. The object is called in a wrong state. This exception is thrown by all methods if the protocol has not been opened yet successfully.

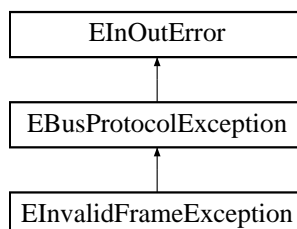
See also:

EInvalidOperation (p. ??)

6.9 EInvalidFrameException Class Reference

Invalid frame error.

Inheritance diagram for EInvalidFrameException:



6.9.1 Detailed Description

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.

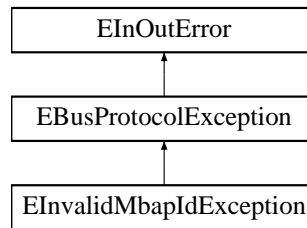
See also:

EBusProtocolException (p. 19)

6.10 EInvalidMbapIdException Class Reference

Invalid MPAB indentifer.

Inheritance diagram for EInvalidMbapIdException:



6.10.1 Detailed Description

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

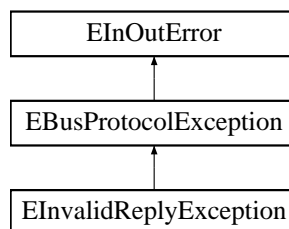
See also:

EBusProtocolException (p. 19)

6.11 EInvalidReplyException Class Reference

Invalid reply error.

Inheritance diagram for EInvalidReplyException:



6.11.1 Detailed Description

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

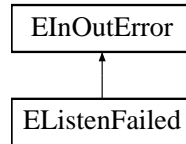
See also:

BusProtocolException

6.12 EListenFailed Class Reference

Listen failed.

Inheritance diagram for EListenFailed:



6.12.1 Detailed Description

Listen failed. The listen operation on the specified TCP port failed..

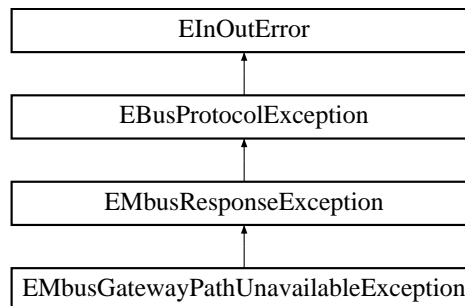
See also:

EInOutError (p. ??)

6.13 EBusGatewayPathUnavailableException Class Reference

Gateway Path Unavailable exception response.

Inheritance diagram for EBusGatewayPathUnavailableException:



6.13.1 Detailed Description

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.

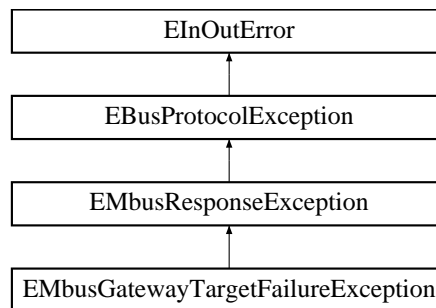
See also:

EMbusResponseException (p. 27)

6.14 EBusGatewayTargetFailureException Class Reference

Gateway Target Device Failed exception response.

Inheritance diagram for `EMbusGatewayTargetFailureException`:



6.14.1 Detailed Description

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.

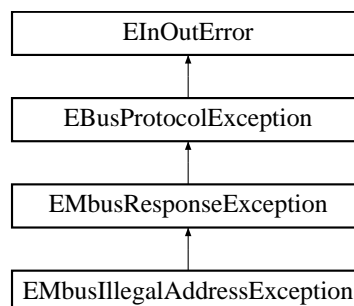
See also:

[EMbusResponseException](#) (p. 27)

6.15 EMbusIllegalAddressException Class Reference

Illegal Data Address exception response.

Inheritance diagram for `EMbusIllegalAddressException`:



6.15.1 Detailed Description

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.

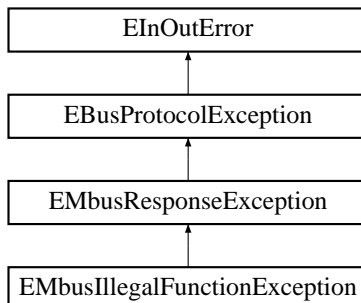
See also:

[EMbusResponseException](#) (p. 27)

6.16 EMBusIllegalFunctionException Class Reference

Illegal Function exception response.

Inheritance diagram for EMBusIllegalFunctionException:



6.16.1 Detailed Description

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.

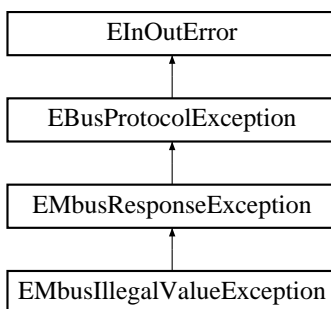
See also:

EMbusResponseException (p. 27)

6.17 EMBusIllegalValueException Class Reference

Illegal Data Value exception response.

Inheritance diagram for EMBusIllegalValueException:



6.17.1 Detailed Description

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.

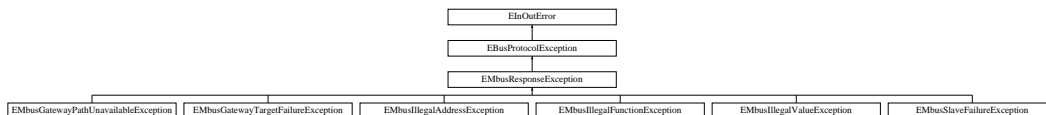
See also:

[EMbusResponseException](#) (p. 27)

6.18 EMbusResponseException Class Reference

Modbus[®] exception response.

Inheritance diagram for EMbusResponseException:



6.18.1 Detailed Description

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.

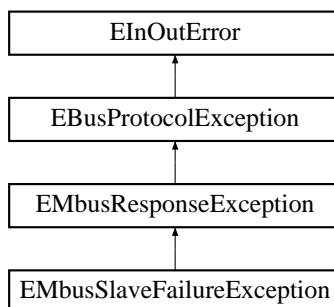
See also:

[EBusProtocolException](#) (p. 19)

6.19 EMbusSlaveFailureException Class Reference

Slave Device Failure exception response.

Inheritance diagram for EMbusSlaveFailureException:



6.19.1 Detailed Description

Slave Device Failure exception response. Signals that a Slave Device Failure exception response (code 04) was received. This exception response is sent by a slave device instead

of a normal response message if an unrecoverable error occurred while processing the requested action. This response is also sent if the request would generate a response whose size exceeds the allowable data size.

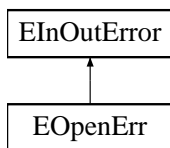
See also:

EMbusResponseException (p. 27)

6.20 EOpenErr Class Reference

Port or socket open error.

Inheritance diagram for EOpenErr:



6.20.1 Detailed Description

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.

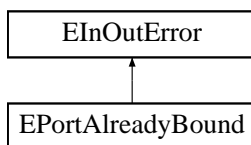
See also:

EInOutError (p. ??)

6.21 EPortAlreadyBound Class Reference

TCP port already bound.

Inheritance diagram for EPortAlreadyBound:



6.21.1 Detailed Description

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.

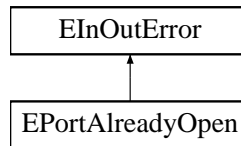
See also:

EInOutError (p. ??)

6.22 EPortAlreadyOpen Class Reference

Serial port already open.

Inheritance diagram for EPortAlreadyOpen:



6.22.1 Detailed Description

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

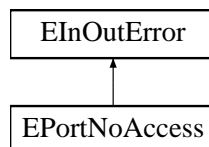
See also:

EInOutError (p. ??)

6.23 EPortNoAccess Class Reference

No permission to access serial port or TCP port.

Inheritance diagram for EPortNoAccess:



6.23.1 Detailed Description

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.

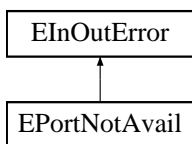
See also:

EInOutError (p. ??)

6.24 EPortNotAvail Class Reference

TCP port not available.

Inheritance diagram for EPortNotAvail:



6.24.1 Detailed Description

TCP port not available. The specified TCP port is not available on this machine.

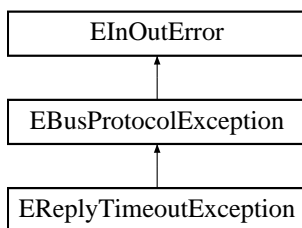
See also:

EInOutError (p. ??)

6.25 EReplyTimeoutException Class Reference

Reply time-out.

Inheritance diagram for EReplyTimeoutException:



6.25.1 Detailed Description

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

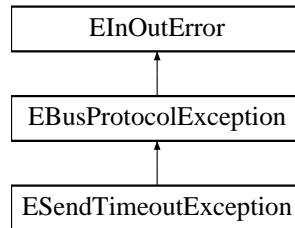
See also:

EBusProtocolException (p. 19)

6.26 ESendTimeoutException Class Reference

Send time-out.

Inheritance diagram for ESendTimeoutException:



6.26.1 Detailed Description

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

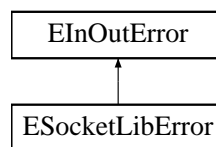
See also:

EBusProtocolException (p. 19)

6.27 ESocketLibError Class Reference

Socket library error.

Inheritance diagram for ESocketLibError:



6.27.1 Detailed Description

Socket library error. The TCP/IP socket library eg WINSOCK; could not be loaded or the DLL is missing or not installed.

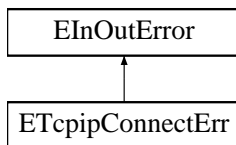
See also:

EInOutError (p. ??)

6.28 ETcpipConnectErr Class Reference

TCP/IP connection error.

Inheritance diagram for ETcpipConnectErr:



6.28.1 Detailed Description

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.

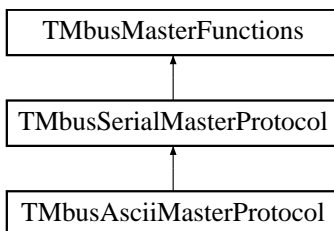
See also:

EInOutError (p. ??)

6.29 TModbusAsciiMasterProtocol Class Reference

Modbus ASCII Master Protocol class.

Inheritance diagram for TModbusAsciiMasterProtocol:



Public Member Functions

- **TModbusAsciiMasterProtocol** (TComponent aOwner)
Constructs a TModbusAsciiMasterProtocol (p. 32) object and initialises its data.
- **openProtocol** ()
Opens a serial Modbus protocol and the associated serial port with the port parameters configured via properties.
- **openUseExistingConnection** (integer cnxnHandle)
Opens a serial Modbus protocol using an existing and open handle.
- **enableRs485Mode** (integer rtsDelay)
Enables RS485 mode.
- **boolean isOpen** ()
Returns whether the protocol is open or not.

- **closeProtocol ()**
Closes an open protocol including any associated communication resources (COM ports or sockets).
- **string getPackageVersion ()**
Returns the package version number.

Public Attributes

- **string portName**
Serial port identifier property (eg 'COM1').
- **longint baudRate**
Baud rate property in bps (typically 1200 - 115200, maximum value depends on UART hardware).
- **integer dataBits**
Data bits property.
- **integer stopBits**
Stop bits property.
- **integer parity**
Parity property.

Bit Access

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

- **readCoils** (integer slaveAddr, integer startRef, boolean[] bitArr)
Modbus function 1 (01 hex), Read Coil Status/Read Coils.
- **readInputDiscretes** (integer slaveAddr, integer startRef, boolean[] bitArr)
Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.
- **writeCoil** (integer slaveAddr, integer bitAddr, boolean bitVal)
Modbus function 5 (05 hex), Force Single Coil/Write Coil.
- **forceMultipleCoils** (integer slaveAddr, integer startRef, boolean[] bitArr)
Modbus function 15 (0F hex), Force Multiple Coils.

16-bit Access

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

- **readMultipleRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.
- **readInputRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 4 (04 hex), Read Input Registers.
- **writeSingleRegister** (integer slaveAddr, integer regAddr, word regVal)
Modbus function 6 (06 hex), Preset Single Register/Write Single Register.
- **writeMultipleRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.
- **maskWriteRegister** (integer slaveAddr, integer regAddr, word andMask, word orMask)
Modbus function 22 (16 hex), Mask Write Register.
- **readWriteRegisters** (integer slaveAddr, integer readRef, word[]readArr, integer writeRef, word[]writeArr)
Modbus function 23 (17 hex), Read/Write Registers.

32-bit Access

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

- **readMultipleLongInts** (integer slaveAddr, integer startRef, integer[]jint32Arr)
Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.
- **readInputLongInts** (integer slaveAddr, integer startRef, integer[]jint32Arr)
Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.
- **writeMultipleLongInts** (integer slaveAddr, integer startRef, integer[]jint32Arr)
Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.
- **readMultipleFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.
- **readInputFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.
- **writeMultipleFloats** (integer slaveAddr, integer startRef, single[]float32Arr)

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

- **readMultipleMod10000** (integer slaveAddr, integer startRef, integer[] jint32Arr)
Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.
- **readInputMod10000** (integer slaveAddr, integer startRef, integer[] jint32Arr)
Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.
- **writeMultipleMod10000** (integer slaveAddr, integer startRef, integer[] jint32Arr)
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

- **readExceptionStatus** (integer slaveAddr, byte &statusByte)
Modbus function 7 (07 hex), Read Exception Status.
- **returnQueryData** (integer slaveAddr, byte[] queryArr, byte[] echoArr)
Modbus function code 8, sub-function 00, Return Query Data.
- **restartCommunicationsOption** (integer slaveAddr, boolean clearEventLog)
Modbus function code 8, sub-function 01, Restart Communications Option.

Custom Function Codes

- **customFunction** (integer slaveAddr, integer functionCode, byte[] requestArr, byte[] responseArr, integer &responseLen)
*User Defined Function Code
This method can be used to implement User Defined Function Codes.*

Protocol Configuration

- **setTimeout** (const integer timeOut)
Configures time-out.
- integer **getTimeout** ()
Returns the time-out value.
- **setPollDelay** (const integer pollDelay)
Configures poll delay.

- integer **getPollDelay** ()
Returns the poll delay time.
- **setRetryCnt** (const integer **retryCnt**)
Configures the automatic retry setting.
- integer **getRetryCnt** ()
Returns the automatic retry count.
- integer **timeout**
Time-out port property.
- integer **pollDelay**
Poll delay property.
- integer **retryCnt**
Retry count property.

Transmission Statistic Functions

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

Slave Configuration

- **configureBigEndianInts** ()
Configures int data type functions to do a word swap.
- **configureBigEndianInts** (integer **slaveAddr**)
Enables int data type functions to do a word swap on a per slave basis.
- **configureSwappedFloats** ()
Configures float data type functions to do a word swap.
- **configureSwappedFloats** (integer **slaveAddr**)

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

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

6.29.1 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. 14).

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. 14), **Serial Protocols** (p. 15)
TMbusMasterFunctions (p. 63), **TMbusSerialMasterProtocol** (p. 151), **TMbusRtuMasterProtocol** (p. 90), **TMbusTcpMasterProtocol** (p. 181), **TMbusRtuOverTcpMasterProtocol** (p. 121)

6.29.2 Constructor & Destructor Documentation

TMbusAsciiMasterProtocol (**TComponent** *aOwner*)

Constructs a **TMbusAsciiMasterProtocol** (p. 32) object and initialises its data.

Exceptions:

EOutOfResources Creation of class failed

6.29.3 Member Function Documentation

openProtocol () [**inherited**]

Opens a serial Modbus protocol and the associated serial port with the port parameters configured via properties.

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

Exceptions:

EInOutError (p. ??) An I/O error occurred

EOpenErr (p. 28) The serial port does not exist

EPortAlreadyOpen (p. 29) Port is already used by somebody else

EPortNoAccess (p. 29) No permission to access serial

EIllegalArgumentError (p. 21) A parameter is invalid

openUseExistingConnection (**integer** *cnxnHandle*) [**inherited**]

Opens a serial Modbus protocol using an existing and open handle.

Useful for using the protocol over a modem link.

Parameters:

cnxnHandle Win32 API handle pointing to the existing and open connection.

Exceptions:

ElIllegalArgumentError (p. 21) A parameter is invalid

enableRs485Mode (integer *rtsDelay*) [inherited]

Enables RS485 mode.

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

Warning:

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

Remarks:

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

Note:

A protocol must be closed in order to configure it.

Parameters:

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

Exceptions:

ElIllegalStateError (p. 22) Protocol is already open

ElIllegalArgumentError (p. 21) A parameter is out of range

readCoils (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*) [inherited]

Modbus function 1 (01 hex), 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. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputDiscretes (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*) [inherited]

Modbus function 2 (02 hex), 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. The length of the array determines how many inputs are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeCoil (integer *slaveAddr*, integer *bitAddr*, boolean *bitVal*) [inherited]

Modbus function 5 (05 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

forceMultipleCoils (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*) [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. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[inherited]

Modbus function 3 (03 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[inherited]

Modbus function 4 (04 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeSingleRegister (integer *slaveAddr*, integer *regAddr*, word *regVal*)
[inherited]

Modbus function 6 (06 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

writeMultipleRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[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. The length of the array determines how many registers are written (Range: 1-123).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

maskWriteRegister (integer *slaveAddr*, integer *regAddr*, word *andMask*, word *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

readWriteRegisters (integer *slaveAddr*, integer *readRef*, word[] *readArr*, integer *writeRef*, word[] *writeArr*) [**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 registers for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read. The length of the array determines how many registers are read (Range: 1-125).

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

writeArr Buffer with data to be sent. The length of the array determines how many registers are written (Range: 1-121).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readMultipleLongInts (*integer slaveAddr*, *integer startRef*, *integer[] int32Arr*)
[inherited]

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[inherited]

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discretes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[inherited]

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discretues and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

**readInputMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]**

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discretues and 16-bit registers. Because an modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

```
writeMultipleMod10000 ( integer slaveAddr, integer startRef, integer[] int32Arr )  
[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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

```
readExceptionStatus ( integer slaveAddr, byte & statusByte ) [inherited]
```

Modbus function 7 (07 hex), 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)

statusByte Slave status byte. The meaning of this status byte is slave specific and varies from device to device.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

returnQueryData (integer *slaveAddr*, byte[] *queryArr*, byte[] *echoArr*)
[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 Data to be sent

echoArr Buffer which will contain the data read. Array must be of the same size as *queryArr*.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

restartCommunicationsOption (integer *slaveAddr*, boolean *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.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

setTimeout (const integer *timeOut*) [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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getTimeout () [inherited]

Returns the time-out value.

Remarks:

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

Returns:

Timeout value in ms

setPollDelay (const integer *pollDelay*) [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:

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

Exceptions:

EIllegalStateException (p. 22) Protocol is already open

EIllegalArgumentException (p. 21) A parameter is out of range

integer getPollDelay () [inherited]

Returns the poll delay time.

Returns:

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

setRetryCnt (const integer *retryCnt*) [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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getRetryCnt () [inherited]

Returns the automatic retry count.

Returns:

Retry count

cardinal getTotalCounter () [inherited]

Returns how often a message transfer has been executed.

Returns:

Counter value

cardinal getSuccessCounter () [inherited]

Returns how often a message transfer was successful.

Returns:

Counter value

configureBigEndianInts () [inherited]

Configures 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 machine.

configureBigEndianInts (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureSwappedFloats () [inherited]

Configures float data type functions to do a word swap.

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

Note:

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

configureSwappedFloats (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureLittleEndianInts () [inherited]

Configures int data type functions *not* to do a word swap.

This is the default.

configureLittleEndianInts (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureIeeeFloats () [inherited]

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

This is the default.

configureIeeeFloats (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

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

configureStandard32BitMode (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

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.

configureEnron32BitMode (integer *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.

Exceptions:

ElIllegalArgumentError (p. 21) A parameter is out of range

configureCountFromOne () [inherited]

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

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

Remarks:

This is the default mode

configureCountFromOne (integer *slaveAddr*) [inherited]

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

This renders the reference range to be 1 to 65536 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

Exceptions:

ElIllegalArgumentError (p. 21) A parameter is out of range

configureCountFromZero () [inherited]

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

This renders the valid reference range to be 0 to 65535.

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

configureCountFromZero (integer *slaveAddr*) [inherited]

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

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535.

Parameters:

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

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

boolean isOpen () [inherited]

Returns whether the protocol is open or not.

Return values:

true = open

false = closed

class string getPackageVersion () [inherited]

Returns the package version number.

Returns:

Package version string

6.29.4 Member Data Documentation

string portName [inherited]

Serial port identifier property (eg 'COM1').

Note:

A protocol must be closed in order to configure it.

See also:

fPortName For reading
fPortName For writing

longint baudRate [inherited]

Baud rate property in bps (typically 1200 - 115200, maximum value depends on UART hardware).

Note:

A protocol must be closed in order to configure it.

See also:

fBaudRate For reading
fBaudRate For writing

integer dataBits [inherited]

Data bits property.

SER_DATABITS_7: 7 data bits (ASCII protocol only), SER_DATABITS_8: 8 data bits

Note:

A protocol must be closed in order to configure it.

See also:

fDataBits For reading
fDataBits For writing

integer stopBits [inherited]

Stop bits property.

SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits

Note:

The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.
A protocol must be closed in order to configure it.

See also:

fStopBits For reading
fStopBits For writing

integer parity [inherited]

Parity property.

SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_EVEN: even parity

Note:

The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.
A protocol must be closed in order to configure it.

See also:

fParity For reading
fParity For writing

integer timeout [inherited]

Time-out port property.

Note:

A protocol must be closed in order to configure it.

See also:

getTimeout (p. 202) For reading
setTimeout (p. 202) For writing

integer pollDelay [inherited]

Poll delay property.

Delay between two Modbus read/writes in ms

Note:

A protocol must be closed in order to configure it.

See also:

getPollDelay (p. 203) For reading
setPollDelay (p. 203) For writing

integer retryCnt [**inherited**]

Retry count property.

Note:

A protocol must be closed in order to configure it.

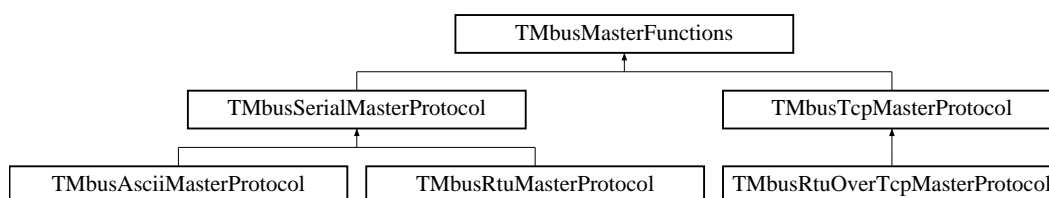
See also:

getRetryCnt (p. 204) For reading
setRetryCnt (p. 203) For writing

6.30 TModbusMasterFunctions Class Reference

Base class which implements Modbus data and control functions.

Inheritance diagram for TModbusMasterFunctions:



Public Member Functions

- **~TModbusMasterFunctions ()**
Destructor.
- **boolean isOpen ()**
Returns whether the protocol is open or not.
- **closeProtocol ()**
Closes an open protocol including any associated communication resources (COM ports or sockets).
- **string getPackageVersion ()**
Returns the package version number.

Protocol Configuration

- integer **timeout**
Time-out port property.
- integer **pollDelay**
Poll delay property.
- integer **retryCnt**
Retry count property.
- **setTimeout** (const integer **timeOut**)
Configures time-out.
- integer **getTimeout** ()
Returns the time-out value.
- **setPollDelay** (const integer **pollDelay**)
Configures poll delay.
- integer **getPollDelay** ()
Returns the poll delay time.
- **setRetryCnt** (const integer **retryCnt**)
Configures the automatic retry setting.
- integer **getRetryCnt** ()
Returns the automatic retry count.

Bit Access

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

- **readCoils** (integer **slaveAddr**, integer **startRef**, boolean[] **bitArr**)
Modbus function 1 (01 hex), Read Coil Status/Read Coils.
- **readInputDiscretes** (integer **slaveAddr**, integer **startRef**, boolean[] **bitArr**)
Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.
- **writeCoil** (integer **slaveAddr**, integer **bitAddr**, boolean **bitVal**)
Modbus function 5 (05 hex), Force Single Coil/Write Coil.
- **forceMultipleCoils** (integer **slaveAddr**, integer **startRef**, boolean[] **bitArr**)
Modbus function 15 (0F hex), Force Multiple Coils.

16-bit Access

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

- **readMultipleRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.
- **readInputRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 4 (04 hex), Read Input Registers.
- **writeSingleRegister** (integer slaveAddr, integer regAddr, word regVal)
Modbus function 6 (06 hex), Preset Single Register/Write Single Register.
- **writeMultipleRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.
- **maskWriteRegister** (integer slaveAddr, integer regAddr, word andMask, word orMask)
Modbus function 22 (16 hex), Mask Write Register.
- **readWriteRegisters** (integer slaveAddr, integer readRef, word[]readArr, integer writeRef, word[]writeArr)
Modbus function 23 (17 hex), Read/Write Registers.

32-bit Access

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

- **readMultipleLongInts** (integer slaveAddr, integer startRef, integer[]jint32Arr)
Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.
- **readInputLongInts** (integer slaveAddr, integer startRef, integer[]jint32Arr)
Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.
- **writeMultipleLongInts** (integer slaveAddr, integer startRef, integer[]jint32Arr)
Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.
- **readMultipleFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.
- **readInputFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.
- **writeMultipleFloats** (integer slaveAddr, integer startRef, single[]float32Arr)

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

- **readMultipleMod10000** (integer slaveAddr, integer startRef, integer[]int32Arr)
Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.
- **readInputMod10000** (integer slaveAddr, integer startRef, integer[]int32Arr)
Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.
- **writeMultipleMod10000** (integer slaveAddr, integer startRef, integer[]int32Arr)
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

- **readExceptionStatus** (integer slaveAddr, byte &statusByte)
Modbus function 7 (07 hex), Read Exception Status.
- **returnQueryData** (integer slaveAddr, byte[]queryArr, byte[]echoArr)
Modbus function code 8, sub-function 00, Return Query Data.
- **restartCommunicationsOption** (integer slaveAddr, boolean clearEventLog)
Modbus function code 8, sub-function 01, Restart Communications Option.

Custom Function Codes

- **customFunction** (integer slaveAddr, integer functionCode, byte[]requestArr, byte[]responseArr, integer &responseLen)
*User Defined Function Code
This method can be used to implement User Defined Function Codes.*

Transmission Statistic Functions

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

- **resetSuccessCounter ()**
Resets successful message transfer counter.

Slave Configuration

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

- **configureCountFromOne** (integer slaveAddr)
Configures a slave's reference counting scheme to start with one.
- **configureCountFromZero** (integer slaveAddr)
Configures a slave's reference counting scheme to start with zero.

6.30.1 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. 14).

See also:

Data and Control Functions for all Modbus Protocol Flavours (p. 14)
TMbusSerialMasterProtocol (p. 151), **TMbusRtuMasterProtocol** (p. 90), **TMbusAsciiMasterProtocol** (p. 32), **TMbusTcpMasterProtocol** (p. 181), **TMbusRtuOverTcpMasterProtocol** (p. 121)

6.30.2 Constructor & Destructor Documentation

`~TMbusMasterFunctions ()`

Destructor.

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

6.30.3 Member Function Documentation

`readCoils (integer slaveAddr, integer startRef, boolean[] bitArr)`

Modbus function 1 (01 hex), 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. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputDiscretes (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*)

Modbus function 2 (02 hex), 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. The length of the array determines how many inputs are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeCoil (integer *slaveAddr*, integer *bitAddr*, boolean *bitVal*)

Modbus function 5 (05 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

forceMultipleCoils (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*)

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. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)

Modbus function 3 (03 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)

Modbus function 4 (04 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeSingleRegister (integer *slaveAddr*, integer *regAddr*, word *regVal*)

Modbus function 6 (06 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

writeMultipleRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)

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. The length of the array determines how many registers are written (Range: 1-123).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

maskWriteRegister (integer *slaveAddr*, integer *regAddr*, word *andMask*, word *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

readWriteRegisters (integer *slaveAddr*, integer *readRef*, word[] *readArr*, integer *writeRef*, word[] *writeArr*)

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 registers for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read. The length of the array determines how many registers are read (Range: 1-125).

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

writeArr Buffer with data to be sent. The length of the array determines how many registers are written (Range: 1-121).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readMultipleLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)

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:

Modbus does not know about any other data type than discretues and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discretes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discretes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)

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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discretely and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discretely and 16-bit registers. Because an modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)

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:

Modbus does not know about any other data type than discretes and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readExceptionStatus (integer *slaveAddr*, byte & *statusByte*)

Modbus function 7 (07 hex), 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)

statusByte Slave status byte. The meaning of this status byte is slave specific and varies from device to device.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

returnQueryData (integer *slaveAddr*, byte[] *queryArr*, byte[] *echoArr*)

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

Parameters:

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

queryArr Data to be sent

echoArr Buffer which will contain the data read. Array must be of the same size as *queryArr*.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

restartCommunicationsOption (integer *slaveAddr*, boolean *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.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

setTimeout (const integer *timeOut*)

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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getTimeout ()

Returns the time-out value.

Remarks:

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

Returns:

Timeout value in ms

setPollDelay (const integer *pollDelay*)

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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getPollDelay ()

Returns the poll delay time.

Returns:

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

setRetryCnt (const integer retryCnt)

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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getRetryCnt ()

Returns the automatic retry count.

Returns:

Retry count

cardinal getTotalCounter ()

Returns how often a message transfer has been executed.

Returns:

Counter value

cardinal getSuccessCounter ()

Returns how often a message transfer was successful.

Returns:

Counter value

configureBigEndianInts ()

Configures 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 machine.

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.

configureLittleEndianInts ()

Configures int data type functions *not* to do a word swap.

This is the default.

configureIeeeFloats ()

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

This is the default.

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

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.

configureCountFromOne ()

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

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

Remarks:

This is the default mode

configureCountFromZero ()

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

This renders the valid reference range to be 0 to 65535.

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

configureBigEndianInts (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureLittleEndianInts (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureIeeeFloats (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureSwappedFloats (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureStandard32BitMode (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureEnron32BitMode (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureCountFromOne (integer *slaveAddr*)

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

This renders the reference range to be 1 to 65536 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

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureCountFromZero (integer *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 65535.

Parameters:

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

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

boolean isOpen ()

Returns whether the protocol is open or not.

Return values:

true = open

false = closed

class string getPackageVersion ()

Returns the package version number.

Returns:

Package version string

6.30.4 Member Data Documentation

integer timeout

Time-out port property.

Note:

A protocol must be closed in order to configure it.

See also:

getTimeout (p. 202) For reading

setTimeout (p. 202) For writing

integer pollDelay

Poll delay property.

Delay between two Modbus read/writes in ms

Note:

A protocol must be closed in order to configure it.

See also:

getPollDelay (p. 203) For reading

setPollDelay (p. 203) For writing

integer retryCnt

Retry count property.

Note:

A protocol must be closed in order to configure it.

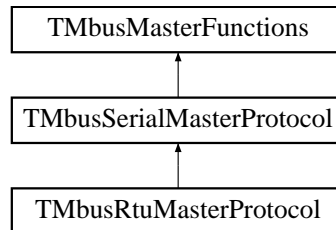
See also:

- getRetryCnt** (p. 204) For reading
- setRetryCnt** (p. 203) For writing

6.31 TModbusRtuMasterProtocol Class Reference

Modbus RTU Master Protocol class.

Inheritance diagram for TModbusRtuMasterProtocol:



Public Member Functions

- **TModbusRtuMasterProtocol** (TComponent aOwner)
Constructs a TModbusRtuMasterProtocol (p. 90) object and initialises its data.
- **openProtocol** ()
Opens a serial Modbus protocol and the associated serial port with the port parameters configured via properties.
- **openUseExistingConnection** (integer cnxnHandle)
Opens a serial Modbus protocol using an existing and open handle.
- **enableRs485Mode** (integer rtsDelay)
Enables RS485 mode.
- boolean **isOpen** ()
Returns whether the protocol is open or not.
- **closeProtocol** ()
Closes an open protocol including any associated communication resources (COM ports or sockets).
- string **getPackageVersion** ()
Returns the package version number.

Public Attributes

- string **portName**

Serial port identifier property (eg 'COM1').

- **longint baudRate**
Baud rate property in bps (typically 1200 - 115200, maximum value depends on UART hardware).
- **integer dataBits**
Data bits property.
- **integer stopBits**
Stop bits property.
- **integer parity**
Parity property.

Bit Access

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

- **readCoils** (integer slaveAddr, integer startRef, boolean[]bitArr)
Modbus function 1 (01 hex), Read Coil Status/Read Coils.
- **readInputDiscretes** (integer slaveAddr, integer startRef, boolean[]bitArr)
Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.
- **writeCoil** (integer slaveAddr, integer bitAddr, boolean bitVal)
Modbus function 5 (05 hex), Force Single Coil/Write Coil.
- **forceMultipleCoils** (integer slaveAddr, integer startRef, boolean[]bitArr)
Modbus function 15 (0F hex), Force Multiple Coils.

16-bit Access

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

- **readMultipleRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.
- **readInputRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 4 (04 hex), Read Input Registers.
- **writeSingleRegister** (integer slaveAddr, integer regAddr, word regVal)
Modbus function 6 (06 hex), Preset Single Register/Write Single Register.
- **writeMultipleRegisters** (integer slaveAddr, integer startRef, word[]regArr)

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

- **maskWriteRegister** (integer slaveAddr, integer regAddr, word andMask, word orMask)

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

- **readWriteRegisters** (integer slaveAddr, integer readRef, word[] readArr, integer writeRef, word[] writeArr)

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

32-bit Access

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

- **readMultipleLongInts** (integer slaveAddr, integer startRef, integer[] jint32Arr)
Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.
- **readInputLongInts** (integer slaveAddr, integer startRef, integer[] jint32Arr)
Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.
- **writeMultipleLongInts** (integer slaveAddr, integer startRef, integer[] jint32Arr)
Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.
- **readMultipleFloats** (integer slaveAddr, integer startRef, single[] jfloat32Arr)
Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.
- **readInputFloats** (integer slaveAddr, integer startRef, single[] jfloat32Arr)
Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.
- **writeMultipleFloats** (integer slaveAddr, integer startRef, single[] jfloat32Arr)
Modbus function 16 (10 hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.
- **readMultipleMod10000** (integer slaveAddr, integer startRef, integer[] jint32Arr)
Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.
- **readInputMod10000** (integer slaveAddr, integer startRef, integer[] jint32Arr)
Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.
- **writeMultipleMod10000** (integer slaveAddr, integer startRef, integer[] jint32Arr)
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

- **readExceptionStatus** (integer slaveAddr, byte &statusByte)
Modbus function 7 (07 hex), Read Exception Status.
- **returnQueryData** (integer slaveAddr, byte[] queryArr, byte[] echoArr)
Modbus function code 8, sub-function 00, Return Query Data.
- **restartCommunicationsOption** (integer slaveAddr, boolean clearEventLog)
Modbus function code 8, sub-function 01, Restart Communications Option.

Custom Function Codes

- **customFunction** (integer slaveAddr, integer functionCode, byte[] requestArr, byte[] responseArr, integer &responseLen)
*User Defined Function Code
This method can be used to implement User Defined Function Codes.*

Protocol Configuration

- **setTimeout** (const integer timeOut)
Configures time-out.
- integer **getTimeout** ()
Returns the time-out value.
- **setPollDelay** (const integer pollDelay)
Configures poll delay.
- integer **getPollDelay** ()
Returns the poll delay time.
- **setRetryCnt** (const integer retryCnt)
Configures the automatic retry setting.
- integer **getRetryCnt** ()
Returns the automatic retry count.
- integer **timeout**
Time-out port property.
- integer **pollDelay**
Poll delay property.
- integer **retryCnt**

Retry count property.

Transmission Statistic Functions

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

Slave Configuration

- **configureBigEndianInts** ()
Configures int data type functions to do a word swap.
- **configureBigEndianInts** (integer slaveAddr)
Enables int data type functions to do a word swap on a per slave basis.
- **configureSwappedFloats** ()
Configures float data type functions to do a word swap.
- **configureSwappedFloats** (integer slaveAddr)
Enables float data type functions to do a word swap on a per slave basis.
- **configureLittleEndianInts** ()
Configures int data type functions not to do a word swap.
- **configureLittleEndianInts** (integer slaveAddr)
Disables word swapping for int data type functions on a per slave basis.
- **configureIeeeFloats** ()
Configures float data type functions not to do a word swap.
- **configureIeeeFloats** (integer slaveAddr)
Disables float data type functions to do a word swap on a per slave basis.
- **configureStandard32BitMode** ()
Configures all slaves for Standard 32-bit Mode.

- **configureStandard32BitMode** (integer slaveAddr)
Configures a slave for Standard 32-bit Register Mode.
- **configureEnron32BitMode** ()
Configures all slaves for Daniel/ENRON 32-bit Mode.
- **configureEnron32BitMode** (integer slaveAddr)
Configures all slaves for Daniel/ENRON 32-bit Mode.
- **configureCountFromOne** ()
Configures the reference counting scheme to start with one for all slaves.
- **configureCountFromOne** (integer slaveAddr)
Configures a slave's reference counting scheme to start with one.
- **configureCountFromZero** ()
Configures the reference counting scheme to start with zero for all slaves.
- **configureCountFromZero** (integer slaveAddr)
Configures a slave's reference counting scheme to start with zero.

6.31.1 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. 14).

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. 14), **Serial Protocols** (p. 15)
TModbusMasterFunctions (p. 63), **TModbusSerialMasterProtocol** (p. 151), **TModbusAsciiMasterProtocol** (p. 32), **TModbusTcpMasterProtocol** (p. 181), **TModbusRtuOverTcpMasterProtocol** (p. 121)

6.31.2 Constructor & Destructor Documentation

TModbusRtuMasterProtocol (TComponent aOwner)

Constructs a **TModbusRtuMasterProtocol** (p. 90) object and initialises its data.

Exceptions:

EOutOfResources Creation of class failed

6.31.3 Member Function Documentation

openProtocol () [inherited]

Opens a serial Modbus protocol and the associated serial port with the port parameters configured via properties.

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

Exceptions:

EInOutError (p. ??) An I/O error occurred

EOpenErr (p. 28) The serial port does not exist

EPortAlreadyOpen (p. 29) Port is already used by somebody else

EPortNoAccess (p. 29) No permission to access serial

EIllegalArgumentError (p. 21) A parameter is invalid

openUseExistingConnection (integer *cnxnHandle*) [inherited]

Opens a serial Modbus protocol using an existing and open handle.

Useful for using the protocol over a modem link.

Parameters:

cnxnHandle Win32 API handle pointing to the existing and open connection.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is invalid

enableRs485Mode (integer *rtsDelay*) [inherited]

Enables RS485 mode.

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

Warning:

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

Remarks:

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

Note:

A protocol must be closed in order to configure it.

Parameters:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

readCoils (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*) [inherited]

Modbus function 1 (01 hex), 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. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

**readInputDiscretes (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*)
[inherited]**

Modbus function 2 (02 hex), 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. The length of the array determines how many inputs are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeCoil (integer *slaveAddr*, integer *bitAddr*, boolean *bitVal*) [inherited]

Modbus function 5 (05 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

ElIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

forceMultipleCoils (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*)
[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. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

ElIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

ElIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[inherited]

Modbus function 3 (03 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[inherited]

Modbus function 4 (04 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeSingleRegister (integer *slaveAddr*, integer *regAddr*, word *regVal*)
[inherited]

Modbus function 6 (06 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

writeMultipleRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[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. The length of the array determines how many registers are written (Range: 1-123).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

maskWriteRegister (integer *slaveAddr*, integer *regAddr*, word *andMask*, word *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

readWriteRegisters (integer *slaveAddr*, integer *readRef*, word[] *readArr*, integer *writeRef*, word[] *writeArr*) [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 registers for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read. The length of the array determines how many registers are read (Range: 1-125).

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

writeArr Buffer with data to be sent. The length of the array determines how many registers are written (Range: 1-121).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readMultipleLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

**writeMultipleLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[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:

Modbus does not know about any other data type than discretes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[inherited]

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[inherited]

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed
EInOutError (p. ??) An I/O error occurred
EIllegalArgumentError (p. 21) A parameter is out of range
EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because an modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readExceptionStatus (integer *slaveAddr*, byte & *statusByte*) [inherited]

Modbus function 7 (07 hex), 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)

statusByte Slave status byte. The meaning of this status byte is slave specific and varies from device to device.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

returnQueryData (integer *slaveAddr*, byte[] *queryArr*, byte[] *echoArr*)
[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 Data to be sent

echoArr Buffer which will contain the data read. Array must be of the same size as *queryArr*.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

restartCommunicationsOption (integer *slaveAddr*, boolean *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.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

setTimeout (const integer *timeOut*) [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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getTimeout () [inherited]

Returns the time-out value.

Remarks:

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

Returns:

Timeout value in ms

setPollDelay (const integer *pollDelay*) [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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getPollDelay () [inherited]

Returns the poll delay time.

Returns:

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

setRetryCnt (const integer *retryCnt*) [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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getRetryCnt () [inherited]

Returns the automatic retry count.

Returns:

Retry count

cardinal getTotalCounter () [inherited]

Returns how often a message transfer has been executed.

Returns:

Counter value

cardinal getSuccessCounter () [inherited]

Returns how often a message transfer was successful.

Returns:

Counter value

configureBigEndianInts () [inherited]

Configures 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 machine.

configureBigEndianInts (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureSwappedFloats () [inherited]

Configures float data type functions to do a word swap.

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

Note:

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

configureSwappedFloats (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureLittleEndianInts () [inherited]

Configures int data type functions *not* to do a word swap.

This is the default.

configureLittleEndianInts (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureIeeeFloats () [inherited]

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

This is the default.

configureIeeeFloats (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

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

configureStandard32BitMode (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

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.

configureEnron32BitMode (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureCountFromOne () [inherited]

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

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

Remarks:

This is the default mode

configureCountFromOne (integer *slaveAddr*) [inherited]

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

This renders the reference range to be 1 to 65536 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

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureCountFromZero () [inherited]

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

This renders the valid reference range to be 0 to 65535.

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

configureCountFromZero (integer *slaveAddr*) [inherited]

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

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535.

Parameters:

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

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

boolean isOpen () [inherited]

Returns whether the protocol is open or not.

Return values:

true = open

false = closed

class string getPackageVersion () [inherited]

Returns the package version number.

Returns:

Package version string

6.31.4 Member Data Documentation

string portName [inherited]

Serial port identifier property (eg 'COM1').

Note:

A protocol must be closed in order to configure it.

See also:

fPortName For reading

fPortName For writing

longint baudRate [inherited]

Baud rate property in bps (typically 1200 - 115200, maximum value depends on UART hardware).

Note:

A protocol must be closed in order to configure it.

See also:

fBaudRate For reading
fBaudRate For writing

integer dataBits [inherited]

Data bits property.

SER_DATABITS_7: 7 data bits (ASCII protocol only), SER_DATABITS_8: 8 data bits

Note:

A protocol must be closed in order to configure it.

See also:

fDataBits For reading
fDataBits For writing

integer stopBits [inherited]

Stop bits property.

SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits

Note:

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

A protocol must be closed in order to configure it.

See also:

fStopBits For reading
fStopBits For writing

integer parity [inherited]

Parity property.

SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_EVEN: even parity

Note:

The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.
A protocol must be closed in order to configure it.

See also:

fParity For reading
fParity For writing

integer timeout [inherited]

Time-out port property.

Note:

A protocol must be closed in order to configure it.

See also:

getTimeout (p. 202) For reading
setTimeout (p. 202) For writing

integer pollDelay [inherited]

Poll delay property.

Delay between two Modbus read/writes in ms

Note:

A protocol must be closed in order to configure it.

See also:

getPollDelay (p. 203) For reading
setPollDelay (p. 203) For writing

integer `retryCnt` [**inherited**]

Retry count property.

Note:

A protocol must be closed in order to configure it.

See also:

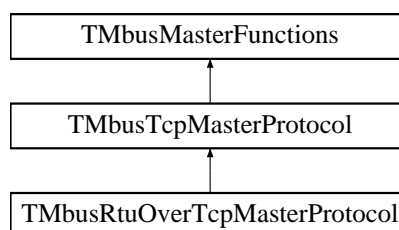
`getRetryCnt` (p. 204) For reading

`setRetryCnt` (p. 203) For writing

6.32 TModbusRtuOverTcpMasterProtocol Class Reference

Encapsulated Modbus RTU Master Protocol class.

Inheritance diagram for TModbusRtuOverTcpMasterProtocol:



Public Member Functions

- **TModbusRtuOverTcpMasterProtocol** (TComponent aOwner)
Constructs a TModbusRtuOverTcpMasterProtocol (p. 121) object and initialises its data.
- **openProtocol** ()
Connects to a MODBUS/TCP slave.
- **setPort** (word portNo)
Sets the TCP port number to be used by the protocol.
- word **getPort** ()
Returns the TCP port number used by the protocol.
- boolean **isOpen** ()
Returns whether the protocol is open or not.
- **closeProtocol** ()
Closes an open protocol including any associated communication resources (COM ports or sockets).

- string **getPackageVersion** ()
Returns the package version number.

Public Attributes

- string **hostName**
Host name property (eg '127.0.0.1').
- word **port**
TCP port property (eg 502).

Advantec ADAM 5000/6000 Series Commands

- **adamSendReceiveAsciiCmd** (string command, string &response)
Send/Receive ADAM 5000/6000 ASCII command.

Bit Access

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

- **readCoils** (integer slaveAddr, integer startRef, boolean[] bitArr)
Modbus function 1 (01 hex), Read Coil Status/Read Coils.
- **readInputDiscretes** (integer slaveAddr, integer startRef, boolean[] bitArr)
Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.
- **writeCoil** (integer slaveAddr, integer bitAddr, boolean bitVal)
Modbus function 5 (05 hex), Force Single Coil/Write Coil.
- **forceMultipleCoils** (integer slaveAddr, integer startRef, boolean[] bitArr)
Modbus function 15 (0F hex), Force Multiple Coils.

16-bit Access

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

- **readMultipleRegisters** (integer slaveAddr, integer startRef, word[] regArr)
Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.
- **readInputRegisters** (integer slaveAddr, integer startRef, word[] regArr)
Modbus function 4 (04 hex), Read Input Registers.

- **writeSingleRegister** (integer slaveAddr, integer regAddr, word regVal)
Modbus function 6 (06 hex), Preset Single Register/Write Single Register.
- **writeMultipleRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.
- **maskWriteRegister** (integer slaveAddr, integer regAddr, word andMask, word orMask)
Modbus function 22 (16 hex), Mask Write Register.
- **readWriteRegisters** (integer slaveAddr, integer readRef, word[]readArr, integer writeRef, word[]writeArr)
Modbus function 23 (17 hex), Read/Write Registers.

32-bit Access

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

- **readMultipleLongInts** (integer slaveAddr, integer startRef, integer[]int32Arr)
Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.
- **readInputLongInts** (integer slaveAddr, integer startRef, integer[]int32Arr)
Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.
- **writeMultipleLongInts** (integer slaveAddr, integer startRef, integer[]int32Arr)
Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.
- **readMultipleFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.
- **readInputFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.
- **writeMultipleFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 16 (10 hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.
- **readMultipleMod10000** (integer slaveAddr, integer startRef, integer[]int32Arr)
Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.
- **readInputMod10000** (integer slaveAddr, integer startRef, integer[]int32Arr)

Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

- **writeMultipleMod10000** (integer slaveAddr, integer startRef, integer[] jint32Arr)

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

- **readExceptionStatus** (integer slaveAddr, byte &statusByte)

Modbus function 7 (07 hex), Read Exception Status.

- **returnQueryData** (integer slaveAddr, byte[] queryArr, byte[] echoArr)

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

- **restartCommunicationsOption** (integer slaveAddr, boolean clearEventLog)

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

Custom Function Codes

- **customFunction** (integer slaveAddr, integer functionCode, byte[] requestArr, byte[] responseArr, integer &responseLen)

User Defined Function Code

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

Protocol Configuration

- **setTimeout** (const integer timeOut)

Configures time-out.

- integer **getTimeout** ()

Returns the time-out value.

- **setPollDelay** (const integer pollDelay)

Configures poll delay.

- integer **getPollDelay** ()

Returns the poll delay time.

- **setRetryCnt** (const integer retryCnt)

Configures the automatic retry setting.

- integer **getRetryCnt** ()

Returns the automatic retry count.

- integer **timeout**
Time-out port property.
- integer **pollDelay**
Poll delay property.
- integer **retryCnt**
Retry count property.

Transmission Statistic Functions

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

Slave Configuration

- **configureBigEndianInts ()**
Configures int data type functions to do a word swap.
- **configureBigEndianInts (integer slaveAddr)**
Enables int data type functions to do a word swap on a per slave basis.
- **configureSwappedFloats ()**
Configures float data type functions to do a word swap.
- **configureSwappedFloats (integer slaveAddr)**
Enables float data type functions to do a word swap on a per slave basis.
- **configureLittleEndianInts ()**
Configures int data type functions not to do a word swap.
- **configureLittleEndianInts (integer slaveAddr)**
Disables word swapping for int data type functions on a per slave basis.

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

6.32.1 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. 14).

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. 14), **TCP/IP Protocols** (p. 13)

TMbusMasterFunctions (p. 63), **TMbusSerialMasterProtocol** (p. 151), **TMbusRtuMasterProtocol** (p. 90), **TMbusAsciiMasterProtocol** (p. 32), **TMbusTcpMasterProtocol** (p. 181)

6.32.2 Constructor & Destructor Documentation

TMbusRtuOverTcpMasterProtocol (*TComponent* *aOwner*)

Constructs a **TMbusRtuOverTcpMasterProtocol** (p. 121) object and initialises its data.

Exceptions:

EOutOfResources Creation of class failed

6.32.3 Member Function Documentation

openProtocol () [**inherited**]

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 TCP port number is 502.

Exceptions:

EInOutError (p. ??) An I/O error occurred

EOpenErr (p. 28) The port could not be opened

EPortNoAccess (p. 29) No permission to access port

ETcpipConnectErr (p. 31) TCP/IP connection error, host not reachable

EConnectionWasClosed (p. 19) Remote peer closed TCP/IP connection

EIllegalArgumentError (p. 21) A parameter is invalid

setPort (*word portNo*) [**inherited**]

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()** (p. 187).

Parameters:

portNo Port number to be used when opening the connection

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

word getPort () [inherited]

Returns the TCP port number used by the protocol.

Returns:

Port number used by the protocol

readCoils (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*) [inherited]

Modbus function 1 (01 hex), 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. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

**readInputDiscretes (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*)
[inherited]**

Modbus function 2 (02 hex), 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. The length of the array determines how many inputs are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeCoil (integer *slaveAddr*, integer *bitAddr*, boolean *bitVal*) [inherited]

Modbus function 5 (05 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

**forceMultipleCoils (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*)
[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. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

EIllegalStateException (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

**readMultipleRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[inherited]**

Modbus function 3 (03 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateException (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[inherited]

Modbus function 4 (04 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeSingleRegister (integer *slaveAddr*, integer *regAddr*, word *regVal*)
[inherited]

Modbus function 6 (06 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

writeMultipleRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[**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. The length of the array determines how many registers are written (Range: 1-123).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

maskWriteRegister (integer *slaveAddr*, integer *regAddr*, word *andMask*, word *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

readWriteRegisters (integer *slaveAddr*, integer *readRef*, word[] *readArr*, integer *writeRef*, word[] *writeArr*) [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 registers for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read. The length of the array determines how many registers are read (Range: 1-125).

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

writeArr Buffer with data to be sent. The length of the array determines how many registers are written (Range: 1-121).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readMultipleLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*) [inherited]

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[inherited]

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[inherited]

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[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:

Modbus does not know about any other data type than discretes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

**readMultipleMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]**

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discretely and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

**readInputMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]**

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discretely and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

```
writeMultipleMod10000 ( integer slaveAddr, integer startRef, integer[] int32Arr )  
[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:

Modbus does not know about any other data type than discretely and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readExceptionStatus (integer *slaveAddr*, byte & *statusByte*) [inherited]

Modbus function 7 (07 hex), 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)

statusByte Slave status byte. The meaning of this status byte is slave specific and varies from device to device.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

returnQueryData (integer *slaveAddr*, byte[] *queryArr*, byte[] *echoArr*) [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 Data to be sent

echoArr Buffer which will contain the data read. Array must be of the same size as *queryArr*.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

restartCommunicationsOption (integer *slaveAddr*, boolean *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.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

setTimeout (const integer *timeOut*) **[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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getTimeout () [inherited]

Returns the time-out value.

Remarks:

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

Returns:

Timeout value in ms

setPollDelay (const integer pollDelay) [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:

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

Exceptions:

EIllegalStateException (p. 22) Protocol is already open

EIllegalArgumentException (p. 21) A parameter is out of range

integer getPollDelay () [inherited]

Returns the poll delay time.

Returns:

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

setRetryCnt (const integer retryCnt) [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:

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

Exceptions:

EIllegalStateException (p. 22) Protocol is already open

EIllegalArgumentException (p. 21) A parameter is out of range

integer getRetryCnt () [inherited]

Returns the automatic retry count.

Returns:

Retry count

cardinal getTotalCounter () [inherited]

Returns how often a message transfer has been executed.

Returns:

Counter value

cardinal getSuccessCounter () [inherited]

Returns how often a message transfer was successful.

Returns:

Counter value

configureBigEndianInts () [inherited]

Configures 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 machine.

configureBigEndianInts (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureSwappedFloats () [inherited]

Configures float data type functions to do a word swap.

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

Note:

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

configureSwappedFloats (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureLittleEndianInts () [inherited]

Configures int data type functions *not* to do a word swap.

This is the default.

configureLittleEndianInts (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureIeeeFloats () [inherited]

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

This is the default.

configureIeeeFloats (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

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

configureStandard32BitMode (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

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.

configureEnron32BitMode (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureCountFromOne () [inherited]

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

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

Remarks:

This is the default mode

configureCountFromOne (integer *slaveAddr*) [inherited]

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

This renders the reference range to be 1 to 65536 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

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureCountFromZero () [inherited]

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

This renders the valid reference range to be 0 to 65535.

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

configureCountFromZero (integer *slaveAddr*) [inherited]

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

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535.

Parameters:

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

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

boolean isOpen () [inherited]

Returns whether the protocol is open or not.

Return values:

true = open

false = closed

class string getPackageVersion () [inherited]

Returns the package version number.

Returns:

Package version string

6.32.4 Member Data Documentation

string hostName [inherited]

Host name property (eg '127.0.0.1').

Note:

A protocol must be closed in order to configure it.

See also:

fHostName For reading

fHostName For writing

word port [inherited]

TCP port property (eg 502).

Note:

A protocol must be closed in order to configure it.

Remarks:

Usually the port number remains unchanged and defaults to 502. However if the port number has to be different from 502 this property must be called *before* opening the connection with **openProtocol()** (p. 187).

See also:

getPort (p. 188) For reading
setPort (p. 188) For writing

integer timeout [inherited]

Time-out port property.

Note:

A protocol must be closed in order to configure it.

See also:

getTimeout (p. 202) For reading
setTimeout (p. 202) For writing

integer pollDelay [inherited]

Poll delay property.

Delay between two Modbus read/writes in ms

Note:

A protocol must be closed in order to configure it.

See also:

getPollDelay (p. 203) For reading
setPollDelay (p. 203) For writing

integer retryCnt [inherited]

Retry count property.

Note:

A protocol must be closed in order to configure it.

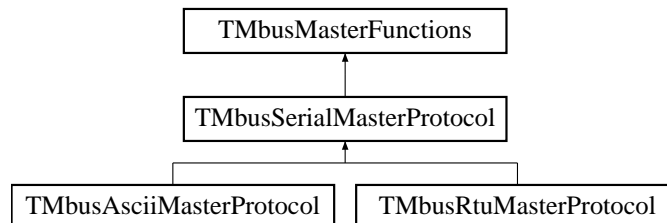
See also:

getRetryCnt (p. 204) For reading
setRetryCnt (p. 203) For writing

6.33 TModbusSerialMasterProtocol Class Reference

Base class for serial master protocols.

Inheritance diagram for TModbusSerialMasterProtocol:



Public Member Functions

- **TModbusSerialMasterProtocol** (TComponent aOwner)
Constructs a TModbusSerialMasterProtocol (p. 151) object and initialises its data.
- **openProtocol** ()
Opens a serial Modbus protocol and the associated serial port with the port parameters configured via properties.
- **openUseExistingConnection** (integer cnxnHandle)
Opens a serial Modbus protocol using an existing and open handle.
- **enableRs485Mode** (integer rtsDelay)
Enables RS485 mode.
- boolean **isOpen** ()
Returns whether the protocol is open or not.
- **closeProtocol** ()
Closes an open protocol including any associated communication resources (COM ports or sockets).
- string **getPackageVersion** ()
Returns the package version number.

Public Attributes

- string **portName**
Serial port identifier property (eg 'COM1').
- longint **baudRate**
Baud rate property in bps (typically 1200 - 115200, maximum value depends on UART hardware).

- integer **dataBits**
Data bits property.
- integer **stopBits**
Stop bits property.
- integer **parity**
Parity property.

Protocol Configuration

- integer **timeout**
Time-out port property.
- integer **pollDelay**
Poll delay property.
- integer **retryCnt**
Retry count property.
- **setTimeout** (const integer **timeOut**)
Configures time-out.
- integer **getTimeout** ()
Returns the time-out value.
- **setPollDelay** (const integer **pollDelay**)
Configures poll delay.
- integer **getPollDelay** ()
Returns the poll delay time.
- **setRetryCnt** (const integer **retryCnt**)
Configures the automatic retry setting.
- integer **getRetryCnt** ()
Returns the automatic retry count.

Bit Access

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

- **readCoils** (integer **slaveAddr**, integer **startRef**, boolean[]**bitArr**)
Modbus function 1 (01 hex), Read Coil Status/Read Coils.

- **readInputDiscretes** (integer slaveAddr, integer startRef, boolean[] bitArr)
Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.
- **writeCoil** (integer slaveAddr, integer bitAddr, boolean bitVal)
Modbus function 5 (05 hex), Force Single Coil/Write Coil.
- **forceMultipleCoils** (integer slaveAddr, integer startRef, boolean[] bitArr)
Modbus function 15 (0F hex), Force Multiple Coils.

16-bit Access

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

- **readMultipleRegisters** (integer slaveAddr, integer startRef, word[] regArr)
Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.
- **readInputRegisters** (integer slaveAddr, integer startRef, word[] regArr)
Modbus function 4 (04 hex), Read Input Registers.
- **writeSingleRegister** (integer slaveAddr, integer regAddr, word regVal)
Modbus function 6 (06 hex), Preset Single Register/Write Single Register.
- **writeMultipleRegisters** (integer slaveAddr, integer startRef, word[] regArr)
Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.
- **maskWriteRegister** (integer slaveAddr, integer regAddr, word andMask, word orMask)
Modbus function 22 (16 hex), Mask Write Register.
- **readWriteRegisters** (integer slaveAddr, integer readRef, word[] readArr, integer writeRef, word[] writeArr)
Modbus function 23 (17 hex), Read/Write Registers.

32-bit Access

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

- **readMultipleLongInts** (integer slaveAddr, integer startRef, integer[] jint32Arr)
Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.
- **readInputLongInts** (integer slaveAddr, integer startRef, integer[] jint32Arr)
Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.

- **writeMultipleLongInts** (integer slaveAddr, integer startRef, integer[]int32Arr)
Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.
- **readMultipleFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.
- **readInputFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.
- **writeMultipleFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 16 (10 hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.
- **readMultipleMod10000** (integer slaveAddr, integer startRef, integer[]int32Arr)
Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.
- **readInputMod10000** (integer slaveAddr, integer startRef, integer[]int32Arr)
Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.
- **writeMultipleMod10000** (integer slaveAddr, integer startRef, integer[]int32Arr)
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

- **readExceptionStatus** (integer slaveAddr, byte &statusByte)
Modbus function 7 (07 hex), Read Exception Status.
- **returnQueryData** (integer slaveAddr, byte[]queryArr, byte[]echoArr)
Modbus function code 8, sub-function 00, Return Query Data.
- **restartCommunicationsOption** (integer slaveAddr, boolean clearEventLog)
Modbus function code 8, sub-function 01, Restart Communications Option.

Custom Function Codes

- **customFunction** (integer slaveAddr, integer functionCode, byte[]requestArr, byte[]responseArr, integer &responseLen)
*User Defined Function Code
This method can be used to implement User Defined Function Codes.*

Transmission Statistic Functions

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

Slave Configuration

- **configureBigEndianInts** ()
Configures int data type functions to do a word swap.
- **configureBigEndianInts** (integer slaveAddr)
Enables int data type functions to do a word swap on a per slave basis.
- **configureSwappedFloats** ()
Configures float data type functions to do a word swap.
- **configureSwappedFloats** (integer slaveAddr)
Enables float data type functions to do a word swap on a per slave basis.
- **configureLittleEndianInts** ()
Configures int data type functions not to do a word swap.
- **configureLittleEndianInts** (integer slaveAddr)
Disables word swapping for int data type functions on a per slave basis.
- **configureIeeeFloats** ()
Configures float data type functions not to do a word swap.
- **configureIeeeFloats** (integer slaveAddr)
Disables float data type functions to do a word swap on a per slave basis.
- **configureStandard32BitMode** ()
Configures all slaves for Standard 32-bit Mode.
- **configureStandard32BitMode** (integer slaveAddr)
Configures a slave for Standard 32-bit Register Mode.

- **configureEnron32BitMode ()**
Configures all slaves for Daniel/ENRON 32-bit Mode.
- **configureEnron32BitMode (integer slaveAddr)**
Configures all slaves for Daniel/ENRON 32-bit Mode.
- **configureCountFromOne ()**
Configures the reference counting scheme to start with one for all slaves.
- **configureCountFromOne (integer slaveAddr)**
Configures a slave's reference counting scheme to start with one.
- **configureCountFromZero ()**
Configures the reference counting scheme to start with zero for all slaves.
- **configureCountFromZero (integer slaveAddr)**
Configures a slave's reference counting scheme to start with zero.

6.33.1 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. 14).

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. 14), **Serial Protocols** (p. 15)

TMbusMasterFunctions (p. 63), **TMbusRtuMasterProtocol** (p. 90), **TMbusAsciiMasterProtocol** (p. 32), **TMbusTcpMasterProtocol** (p. 181), **TMbusRtuOverTcpMasterProtocol** (p. 121)

6.33.2 Member Function Documentation

openProtocol ()

Opens a serial Modbus protocol and the associated serial port with the port parameters configured via properties.

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

Exceptions:

EInOutError (p. ??) An I/O error occurred

EOpenErr (p. 28) The serial port does not exist

EPortAlreadyOpen (p. 29) Port is already used by somebody else

EPortNoAccess (p. 29) No permission to access serial

EIllegalArgumentError (p. 21) A parameter is invalid

openUseExistingConnection (integer *cnxnHandle*)

Opens a serial Modbus protocol using an existing and open handle.

Useful for using the protocol over a modem link.

Parameters:

cnxnHandle Win32 API handle pointing to the existing and open connection.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is invalid

enableRs485Mode (integer *rtsDelay*)

Enables RS485 mode.

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

Warning:

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

Remarks:

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

Note:

A protocol must be closed in order to configure it.

Parameters:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

readCoils (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*) [inherited]

Modbus function 1 (01 hex), 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. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputDiscretes (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*) [inherited]

Modbus function 2 (02 hex), 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. The length of the array determines how many inputs are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeCoil (integer *slaveAddr*, integer *bitAddr*, boolean *bitVal*) [inherited]

Modbus function 5 (05 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

forceMultipleCoils (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*)
[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. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[inherited]

Modbus function 3 (03 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[inherited]

Modbus function 4 (04 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeSingleRegister (integer *slaveAddr*, integer *regAddr*, word *regVal*)
[inherited]

Modbus function 6 (06 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

writeMultipleRegisters (*integer slaveAddr*, *integer startRef*, *word[] regArr*)
[**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. The length of the array determines how many registers are written (Range: 1-123).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

maskWriteRegister (*integer slaveAddr*, *integer regAddr*, *word andMask*, *word 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

readWriteRegisters (integer *slaveAddr*, integer *readRef*, word[] *readArr*, integer *writeRef*, word[] *writeArr*) [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 registers for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read. The length of the array determines how many registers are read (Range: 1-125).

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

writeArr Buffer with data to be sent. The length of the array determines how many registers are written (Range: 1-121).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readMultipleLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*) [inherited]

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[inherited]

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[inherited]

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[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:

Modbus does not know about any other data type than discretes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

**readMultipleMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]**

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

**readInputMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]**

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discretely and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

```
writeMultipleMod10000 ( integer slaveAddr, integer startRef, integer[] int32Arr )  
[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:

Modbus does not know about any other data type than discretely and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readExceptionStatus (integer *slaveAddr*, byte & *statusByte*) [inherited]

Modbus function 7 (07 hex), 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)

statusByte Slave status byte. The meaning of this status byte is slave specific and varies from device to device.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

returnQueryData (integer *slaveAddr*, byte[] *queryArr*, byte[] *echoArr*) [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 Data to be sent

echoArr Buffer which will contain the data read. Array must be of the same size as *queryArr*.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

restartCommunicationsOption (integer *slaveAddr*, boolean *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.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

setTimeout (const integer *timeOut*) **[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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getTimeout () [inherited]

Returns the time-out value.

Remarks:

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

Returns:

Timeout value in ms

setPollDelay (const integer pollDelay) [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:

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

Exceptions:

EIllegalStateException (p. 22) Protocol is already open

EIllegalArgumentException (p. 21) A parameter is out of range

integer getPollDelay () [inherited]

Returns the poll delay time.

Returns:

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

setRetryCnt (const integer *retryCnt*) [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:

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

Exceptions:

EIllegalStateException (p. 22) Protocol is already open

EIllegalArgumentException (p. 21) A parameter is out of range

integer getRetryCnt () [inherited]

Returns the automatic retry count.

Returns:

Retry count

cardinal getTotalCounter () [inherited]

Returns how often a message transfer has been executed.

Returns:

Counter value

cardinal getSuccessCounter () [inherited]

Returns how often a message transfer was successful.

Returns:

Counter value

configureBigEndianInts () [inherited]

Configures 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 machine.

configureBigEndianInts (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureSwappedFloats () [inherited]

Configures float data type functions to do a word swap.

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

Note:

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

configureSwappedFloats (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureLittleEndianInts () [inherited]

Configures int data type functions *not* to do a word swap.

This is the default.

configureLittleEndianInts (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureIeeeFloats () [inherited]

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

This is the default.

configureIeeeFloats (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

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

configureStandard32BitMode (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

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.

configureEnron32BitMode (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

configureCountFromOne () [inherited]

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

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

Remarks:

This is the default mode

configureCountFromOne (integer *slaveAddr*) [inherited]

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

This renders the reference range to be 1 to 65536 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

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureCountFromZero () [inherited]

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

This renders the valid reference range to be 0 to 65535.

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

configureCountFromZero (integer *slaveAddr*) [inherited]

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

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535.

Parameters:

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

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

boolean isOpen () [inherited]

Returns whether the protocol is open or not.

Return values:

true = open

false = closed

class string getPackageVersion () [inherited]

Returns the package version number.

Returns:

Package version string

6.33.3 Member Data Documentation

string portName

Serial port identifier property (eg 'COM1').

Note:

A protocol must be closed in order to configure it.

See also:

fPortName For reading

fPortName For writing

longint baudRate

Baud rate property in bps (typically 1200 - 115200, maximum value depends on UART hardware).

Note:

A protocol must be closed in order to configure it.

See also:

fBaudRate For reading

fBaudRate For writing

integer dataBits

Data bits property.

SER_DATABITS_7: 7 data bits (ASCII protocol only), SER_DATABITS_8: 8 data bits

Note:

A protocol must be closed in order to configure it.

See also:

fDataBits For reading
fDataBits For writing

integer stopBits

Stop bits property.

SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits

Note:

The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.
A protocol must be closed in order to configure it.

See also:

fStopBits For reading
fStopBits For writing

integer parity

Parity property.

SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_EVEN: even parity

Note:

The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.
A protocol must be closed in order to configure it.

See also:

fParity For reading
fParity For writing

integer timeout [inherited]

Time-out port property.

Note:

A protocol must be closed in order to configure it.

See also:

getTimeout (p. 202) For reading

setTimeout (p. 202) For writing

integer pollDelay [inherited]

Poll delay property.

Delay between two Modbus read/writes in ms

Note:

A protocol must be closed in order to configure it.

See also:

getPollDelay (p. 203) For reading

setPollDelay (p. 203) For writing

integer retryCnt [inherited]

Retry count property.

Note:

A protocol must be closed in order to configure it.

See also:

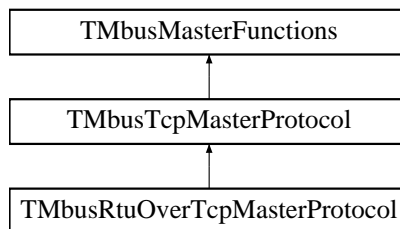
getRetryCnt (p. 204) For reading

setRetryCnt (p. 203) For writing

6.34 TModbusTcpMasterProtocol Class Reference

MODBUS/TCP Master Protocol class.

Inheritance diagram for TModbusTcpMasterProtocol:



Public Member Functions

- **TmbusTcpMasterProtocol** (TComponent aOwner)
Constructs a TmbusTcpMasterProtocol (p. 181) object and initialises its data.
- **openProtocol ()**
Connects to a MODBUS/TCP slave.
- **setPort** (word portNo)
Sets the TCP port number to be used by the protocol.
- word **getPort ()**
Returns the TCP port number used by the protocol.
- boolean **isOpen ()**
Returns whether the protocol is open or not.
- **closeProtocol ()**
Closes an open protocol including any associated communication resources (COM ports or sockets).
- string **getPackageVersion ()**
Returns the package version number.

Public Attributes

- string **hostName**
Host name property (eg '127.0.0.1').
- word **port**
TCP port property (eg 502).

Protocol Configuration

- integer **timeout**
Time-out port property.

- integer **pollDelay**
Poll delay property.
- integer **retryCnt**
Retry count property.
- **setTimeout** (const integer timeOut)
Configures time-out.
- integer **getTimeout** ()
Returns the time-out value.
- **setPollDelay** (const integer pollDelay)
Configures poll delay.
- integer **getPollDelay** ()
Returns the poll delay time.
- **setRetryCnt** (const integer retryCnt)
Configures the automatic retry setting.
- integer **getRetryCnt** ()
Returns the automatic retry count.

Advantec ADAM 5000/6000 Series Commands

- **adamSendReceiveAsciiCmd** (string command, string &response)
Send/Receive ADAM 5000/6000 ASCII command.

Bit Access

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

- **readCoils** (integer slaveAddr, integer startRef, boolean[] bitArr)
Modbus function 1 (01 hex), Read Coil Status/Read Coils.
- **readInputDiscretes** (integer slaveAddr, integer startRef, boolean[] bitArr)
Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.
- **writeCoil** (integer slaveAddr, integer bitAddr, boolean bitVal)
Modbus function 5 (05 hex), Force Single Coil/Write Coil.
- **forceMultipleCoils** (integer slaveAddr, integer startRef, boolean[] bitArr)
Modbus function 15 (0F hex), Force Multiple Coils.

16-bit Access

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

- **readMultipleRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.
- **readInputRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 4 (04 hex), Read Input Registers.
- **writeSingleRegister** (integer slaveAddr, integer regAddr, word regVal)
Modbus function 6 (06 hex), Preset Single Register/Write Single Register.
- **writeMultipleRegisters** (integer slaveAddr, integer startRef, word[]regArr)
Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.
- **maskWriteRegister** (integer slaveAddr, integer regAddr, word andMask, word orMask)
Modbus function 22 (16 hex), Mask Write Register.
- **readWriteRegisters** (integer slaveAddr, integer readRef, word[]readArr, integer writeRef, word[]writeArr)
Modbus function 23 (17 hex), Read/Write Registers.

32-bit Access

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

- **readMultipleLongInts** (integer slaveAddr, integer startRef, integer[]jint32Arr)
Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.
- **readInputLongInts** (integer slaveAddr, integer startRef, integer[]jint32Arr)
Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.
- **writeMultipleLongInts** (integer slaveAddr, integer startRef, integer[]jint32Arr)
Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.
- **readMultipleFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.
- **readInputFloats** (integer slaveAddr, integer startRef, single[]float32Arr)
Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.
- **writeMultipleFloats** (integer slaveAddr, integer startRef, single[]float32Arr)

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

- **readMultipleMod10000** (integer slaveAddr, integer startRef, integer[]int32Arr)
Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.
- **readInputMod10000** (integer slaveAddr, integer startRef, integer[]int32Arr)
Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.
- **writeMultipleMod10000** (integer slaveAddr, integer startRef, integer[]int32Arr)
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

- **readExceptionStatus** (integer slaveAddr, byte &statusByte)
Modbus function 7 (07 hex), Read Exception Status.
- **returnQueryData** (integer slaveAddr, byte[]queryArr, byte[]echoArr)
Modbus function code 8, sub-function 00, Return Query Data.
- **restartCommunicationsOption** (integer slaveAddr, boolean clearEventLog)
Modbus function code 8, sub-function 01, Restart Communications Option.

Custom Function Codes

- **customFunction** (integer slaveAddr, integer functionCode, byte[]requestArr, byte[]responseArr, integer &responseLen)
*User Defined Function Code
This method can be used to implement User Defined Function Codes.*

Transmission Statistic Functions

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

- **resetSuccessCounter ()**
Resets successful message transfer counter.

Slave Configuration

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

- **configureCountFromZero** ()
Configures the reference counting scheme to start with zero for all slaves.
- **configureCountFromZero** (integer slaveAddr)
Configures a slave's reference counting scheme to start with zero.

6.34.1 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. 14).

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. 14), **TCP/IP Protocols** (p. 13)
TMbusMasterFunctions (p. 63), **TMbusSerialMasterProtocol** (p. 151), **TMbusRtuMasterProtocol** (p. 90), **TMbusAsciiMasterProtocol** (p. 32), **TMbusRtuOverTcpMasterProtocol** (p. 121)

6.34.2 Constructor & Destructor Documentation

TMbusTcpMasterProtocol (TComponent aOwner)

Constructs a **TMbusTcpMasterProtocol** (p. 181) object and initialises its data.

Exceptions:

EOutOfResources Creation of class failed

6.34.3 Member Function Documentation

openProtocol ()

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 TCP port number is 502.

Exceptions:

EInOutError (p. ??) An I/O error occurred

EOpenErr (p. 28) The port could not be opened

EPortNoAccess (p. 29) No permission to access port

ETcpipConnectErr (p. 31) TCP/IP connection error, host not reachable

EConnectionWasClosed (p. 19) Remote peer closed TCP/IP connection

EIllegalArgumentError (p. 21) A parameter is invalid

setPort (word *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()** (p. 187).

Parameters:

portNo Port number to be used when opening the connection

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

word getPort ()

Returns the TCP port number used by the protocol.

Returns:

Port number used by the protocol

readCoils (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*) [inherited]

Modbus function 1 (01 hex), 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. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputDiscretes (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*) [inherited]

Modbus function 2 (02 hex), 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. The length of the array determines how many inputs are read (Range: 1-2000).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeCoil (integer *slaveAddr*, integer *bitAddr*, boolean *bitVal*) [inherited]

Modbus function 5 (05 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

forceMultipleCoils (integer *slaveAddr*, integer *startRef*, boolean[] *bitArr*) [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. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readMultipleRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[inherited]

Modbus function 3 (03 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[inherited]

Modbus function 4 (04 hex), 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. The length of the array determines how many registers are read (Range: 1-125).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeSingleRegister (integer *slaveAddr*, integer *regAddr*, word *regVal*)
[inherited]

Modbus function 6 (06 hex), 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

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

writeMultipleRegisters (integer *slaveAddr*, integer *startRef*, word[] *regArr*)
[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. The length of the array determines how many registers are written (Range: 1-123).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

maskWriteRegister (integer *slaveAddr*, integer *regAddr*, word *andMask*, word *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

readWriteRegisters (integer *slaveAddr*, integer *readRef*, word[] *readArr*, integer *writeRef*, word[] *writeArr*) [**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 registers for reading (Range: 1 - 65536)

readArr Buffer which will contain the data read. The length of the array determines how many registers are read (Range: 1-125).

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

writeArr Buffer with data to be sent. The length of the array determines how many registers are written (Range: 1-121).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readMultipleLongInts (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

```
readInputLongInts ( integer slaveAddr, integer startRef, integer[] int32Arr )  
[inherited]
```

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discretely and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

```
writeMultipleLongInts ( integer slaveAddr, integer startRef, integer[] int32Arr )  
[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:

Modbus does not know about any other data type than discretely and 16-bit registers. Because a long int value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-61).

Exceptions:

IllegalStateError (p. 22) Port or connection is closed

InOutError (p. ??) An I/O error occurred

IllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

**readMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[inherited]**

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discretes and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

IllegalStateError (p. 22) Port or connection is closed

InOutError (p. ??) An I/O error occurred

IllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of ***EBusProtocolException*** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[inherited]

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discretues and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of ***EBusProtocolException*** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

writeMultipleFloats (integer *slaveAddr*, integer *startRef*, single[] *float32Arr*)
[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:

Modbus does not know about any other data type than discretely and 16-bit registers. Because a float value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of float values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

IllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

IllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

**readMultipleMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]**

Modbus function 3 (03 hex) 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:

Modbus does not know about any other data type than discretely and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

readInputMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[inherited]

Modbus function 4 (04 hex) 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:

Modbus does not know about any other data type than discrettes and 16-bit registers. Because an modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are read (Range: 1-62).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

**writeMultipleMod10000 (integer *slaveAddr*, integer *startRef*, integer[] *int32Arr*)
[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:

Modbus does not know about any other data type than discretes and 16-bit registers. Because a modulo-10000 value is of 32-bit length, it will be transferred as two consecutive 16-bit registers. This means that the amount of registers transferred with this function is twice the amount of int values passed to this function.

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. The length of the array determines how many values are written (Range: 1-61).

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBusProtocolException** (p. 19) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

readExceptionStatus (integer *slaveAddr*, byte & *statusByte*) [inherited]

Modbus function 7 (07 hex), 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)

statusByte Slave status byte. The meaning of this status byte is slave specific and varies from device to device.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

```
returnQueryData ( integer slaveAddr, byte[] queryArr, byte[] echoArr )  
[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 Data to be sent

echoArr Buffer which will contain the data read. Array must be of the same size as queryArr.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

```
restartCommunicationsOption ( integer slaveAddr, boolean 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.

Exceptions:

EIllegalStateError (p. 22) Port or connection is closed

EInOutError (p. ??) An I/O error occurred

EIllegalArgumentError (p. 21) A parameter is out of range

EBusProtocolException (p. 19) A protocol failure occurred. See descendants of **EBus-ProtocolException** (p. 19) for a more detailed failure list.

Note:

No broadcast supported

setTimeout (const integer *timeOut*) [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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getTimeout () [inherited]

Returns the time-out value.

Remarks:

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

Returns:

Timeout value in ms

setPollDelay (const integer *pollDelay*) [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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getPollDelay () [inherited]

Returns the poll delay time.

Returns:

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

setRetryCnt (const integer *retryCnt*) [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:

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

Exceptions:

EIllegalStateError (p. 22) Protocol is already open

EIllegalArgumentError (p. 21) A parameter is out of range

integer getRetryCnt () [inherited]

Returns the automatic retry count.

Returns:

Retry count

cardinal getTotalCounter () [inherited]

Returns how often a message transfer has been executed.

Returns:

Counter value

cardinal getSuccessCounter () [inherited]

Returns how often a message transfer was successful.

Returns:

Counter value

configureBigEndianInts () [inherited]

Configures 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 machine.

configureBigEndianInts (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureSwappedFloats () [inherited]

Configures float data type functions to do a word swap.

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

Note:

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

configureSwappedFloats (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureLittleEndianInts () [inherited]

Configures int data type functions *not* to do a word swap.

This is the default.

configureLittleEndianInts (integer *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.

Exceptions:

EIllegalArgumentError (p. 21) A parameter is out of range

configureIeeeFloats () [inherited]

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

This is the default.

configureIeeeFloats (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

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

configureStandard32BitMode (integer *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.

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

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.

configureEnron32BitMode (integer *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.

Exceptions:

ElIlegalArgumentError (p. 21) A parameter is out of range

configureCountFromOne () [inherited]

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

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

Remarks:

This is the default mode

configureCountFromOne (integer *slaveAddr*) [inherited]

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

This renders the reference range to be 1 to 65536 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

Exceptions:

ElIlegalArgumentError (p. 21) A parameter is out of range

configureCountFromZero () [inherited]

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

This renders the valid reference range to be 0 to 65535.

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

configureCountFromZero (integer *slaveAddr*) [inherited]

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

This is also known as PDU addressing.

This renders the valid reference range to be 0 to 65535.

Parameters:

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

Exceptions:

IllegalArgumentError (p. 21) A parameter is out of range

boolean isOpen () [inherited]

Returns whether the protocol is open or not.

Return values:

true = open

false = closed

class string getPackageVersion () [inherited]

Returns the package version number.

Returns:

Package version string

6.34.4 Member Data Documentation

string hostName

Host name property (eg '127.0.0.1').

Note:

A protocol must be closed in order to configure it.

See also:

fHostName For reading
fHostName For writing

word port

TCP port property (eg 502).

Note:

A protocol must be closed in order to configure it.

Remarks:

Usually the port number remains unchanged and defaults to 502. However if the port number has to be different from 502 this property must be called *before* opening the connection with **openProtocol()** (p. 187).

See also:

getPort (p. 188) For reading
setPort (p. 188) For writing

integer timeout [inherited]

Time-out port property.

Note:

A protocol must be closed in order to configure it.

See also:

getTimeout (p. 202) For reading
setTimeout (p. 202) For writing

integer pollDelay [inherited]

Poll delay property.

Delay between two Modbus read/writes in ms

Note:

A protocol must be closed in order to configure it.

See also:

getPollDelay (p. 203) For reading
setPollDelay (p. 203) For writing

integer retryCnt [inherited]

Retry count property.

Note:

A protocol must be closed in order to configure it.

See also:

getRetryCnt (p. 204) For reading
setRetryCnt (p. 203) For writing

7 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.
-

8 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.

9 Notices

Disclaimer:

proconX Pty Ltd makes no warranty for the use of its products, other than those expressly contained in the Company's standard warranty which is detailed in the Terms and Conditions located on the Company's Website. The Company assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information contained herein. No licenses to patents or other intellectual property of proconX are granted by the Company in connection with the sale of proconX products, expressly or by implication. proconX products are not authorized for use as critical components in life support devices or systems.

This FieldTalk™ library was developed by:

proconX Pty Ltd, Australia.

Copyright © 2003-2011. All rights reserved.

proconX and FieldTalk are trademarks of proconX Pty Ltd. Modbus is a registered trademark of Schneider Automation Inc. Delphi is a trademark or registered trademark of Embarcadero Technologies Inc. All other product and brand names mentioned in this document may be trademarks or registered trademarks of their respective owners.

Index

- ~TMbusMasterFunctions
 - TMbusMasterFunctions, 68
- baudRate
 - TMbusAsciiMasterProtocol, 61
 - TMbusRtuMasterProtocol, 118
 - TMbusSerialMasterProtocol, 179
- configureBigEndianInts
 - TMbusAsciiMasterProtocol, 55
 - TMbusMasterFunctions, 83, 85
 - TMbusRtuMasterProtocol, 113
 - TMbusRtuOverTcpMasterProtocol, 144
 - TMbusSerialMasterProtocol, 174
 - TMbusTcpMasterProtocol, 204
- configureCountFromOne
 - TMbusAsciiMasterProtocol, 59
 - TMbusMasterFunctions, 85, 87
 - TMbusRtuMasterProtocol, 117
 - TMbusRtuOverTcpMasterProtocol, 147, 148
 - TMbusSerialMasterProtocol, 177, 178
 - TMbusTcpMasterProtocol, 208
- configureCountFromZero
 - TMbusAsciiMasterProtocol, 59
 - TMbusMasterFunctions, 85, 88
 - TMbusRtuMasterProtocol, 117
 - TMbusRtuOverTcpMasterProtocol, 148
 - TMbusSerialMasterProtocol, 178
 - TMbusTcpMasterProtocol, 208
- configureEnron32BitMode
 - TMbusAsciiMasterProtocol, 58
 - TMbusMasterFunctions, 84, 87
 - TMbusRtuMasterProtocol, 116
 - TMbusRtuOverTcpMasterProtocol, 147
 - TMbusSerialMasterProtocol, 177
 - TMbusTcpMasterProtocol, 207
- configureIeeeFloats
 - TMbusAsciiMasterProtocol, 57
 - TMbusMasterFunctions, 84, 86
 - TMbusRtuMasterProtocol, 115
 - TMbusRtuOverTcpMasterProtocol, 146
 - TMbusSerialMasterProtocol, 176
 - TMbusTcpMasterProtocol, 206
- configureLittleEndianInts
 - TMbusAsciiMasterProtocol, 56, 57
 - TMbusMasterFunctions, 84, 85
 - TMbusRtuMasterProtocol, 114
 - TMbusRtuOverTcpMasterProtocol, 145
 - TMbusSerialMasterProtocol, 175
 - TMbusTcpMasterProtocol, 205, 206
- configureStandard32BitMode
 - TMbusAsciiMasterProtocol, 58
 - TMbusMasterFunctions, 84, 87
 - TMbusRtuMasterProtocol, 115, 116
 - TMbusRtuOverTcpMasterProtocol, 146
 - TMbusSerialMasterProtocol, 176
 - TMbusTcpMasterProtocol, 207
- configureSwappedFloats
 - TMbusAsciiMasterProtocol, 56
 - TMbusMasterFunctions, 84, 86
 - TMbusRtuMasterProtocol, 114
 - TMbusRtuOverTcpMasterProtocol, 144, 145
 - TMbusSerialMasterProtocol, 174, 175
 - TMbusTcpMasterProtocol, 205
- Data and Control Functions for all Modbus Protocol Flavours, 14
- dataBits
 - TMbusAsciiMasterProtocol, 61
 - TMbusRtuMasterProtocol, 119
 - TMbusSerialMasterProtocol, 179
- EBusProtocolException, 19
- EChecksumException, 19
- EConnectionWasClosed, 19
- EEvaluationExpired, 20
- EFiledesExceeded, 20
- EIllegalArgumentError, 21
- EIllegalSlaveAddress, 21
- EIllegalStateError, 22
- EInvalidFrameException, 22
- EInvalidMbapIdException, 22
- EInvalidReplyException, 23
- EListenFailed, 23
- EMbusGatewayPathUnavailableException, 24
- EMbusGatewayTargetFailureException, 24
- EMbusIllegalAddressException, 25
- EMbusIllegalFunctionException, 26
- EMbusIllegalValueException, 26
- EMbusResponseException, 27
- EMbusSlaveFailureException, 27

- enableRs485Mode
 - TMbusAsciiMasterProtocol, 39
 - TMbusRtuMasterProtocol, 96
 - TMbusSerialMasterProtocol, 157
- Encapsulated Modbus RTU Protocol, 13
- EOpenErr, 28
- EPortAlreadyBound, 28
- EPortAlreadyOpen, 29
- EPortNoAccess, 29
- EPortNotAvail, 29
- EReplyTimeoutException, 30
- Error Management, 16
- ESendTimeoutException, 30
- ESocketLibError, 31
- ETcpipConnectErr, 31
- forceMultipleCoils
 - TMbusAsciiMasterProtocol, 41
 - TMbusMasterFunctions, 70
 - TMbusRtuMasterProtocol, 99
 - TMbusRtuOverTcpMasterProtocol, 129
 - TMbusSerialMasterProtocol, 159
 - TMbusTcpMasterProtocol, 190
- getPackageVersion
 - TMbusAsciiMasterProtocol, 60
 - TMbusMasterFunctions, 88
 - TMbusRtuMasterProtocol, 118
 - TMbusRtuOverTcpMasterProtocol, 149
 - TMbusSerialMasterProtocol, 179
 - TMbusTcpMasterProtocol, 209
- getPollDelay
 - TMbusAsciiMasterProtocol, 54
 - TMbusMasterFunctions, 82
 - TMbusRtuMasterProtocol, 112
 - TMbusRtuOverTcpMasterProtocol, 143
 - TMbusSerialMasterProtocol, 173
 - TMbusTcpMasterProtocol, 203
- getPort
 - TMbusRtuOverTcpMasterProtocol, 128
 - TMbusTcpMasterProtocol, 188
- getRetryCnt
 - TMbusAsciiMasterProtocol, 55
 - TMbusMasterFunctions, 83
 - TMbusRtuMasterProtocol, 112
 - TMbusRtuOverTcpMasterProtocol, 143
 - TMbusSerialMasterProtocol, 173
 - TMbusTcpMasterProtocol, 204
- getSuccessCounter
 - TMbusAsciiMasterProtocol, 55
 - TMbusMasterFunctions, 83
- TMbusRtuMasterProtocol, 113
- TMbusRtuOverTcpMasterProtocol, 144
- TMbusSerialMasterProtocol, 174
- TMbusTcpMasterProtocol, 204
- getTimeout
 - TMbusAsciiMasterProtocol, 53
 - TMbusMasterFunctions, 81
 - TMbusRtuMasterProtocol, 111
 - TMbusRtuOverTcpMasterProtocol, 142
 - TMbusSerialMasterProtocol, 172
 - TMbusTcpMasterProtocol, 202
- getTotalCounter
 - TMbusAsciiMasterProtocol, 55
 - TMbusMasterFunctions, 83
 - TMbusRtuMasterProtocol, 113
 - TMbusRtuOverTcpMasterProtocol, 143
 - TMbusSerialMasterProtocol, 173
 - TMbusTcpMasterProtocol, 204
- hostName
 - TMbusRtuOverTcpMasterProtocol, 149
 - TMbusTcpMasterProtocol, 209
- isOpen
 - TMbusAsciiMasterProtocol, 60
 - TMbusMasterFunctions, 88
 - TMbusRtuMasterProtocol, 118
 - TMbusRtuOverTcpMasterProtocol, 148
 - TMbusSerialMasterProtocol, 178
 - TMbusTcpMasterProtocol, 209
- maskWriteRegister
 - TMbusAsciiMasterProtocol, 44
 - TMbusMasterFunctions, 72
 - TMbusRtuMasterProtocol, 101
 - TMbusRtuOverTcpMasterProtocol, 132
 - TMbusSerialMasterProtocol, 162
 - TMbusTcpMasterProtocol, 193
- openProtocol
 - TMbusAsciiMasterProtocol, 38
 - TMbusRtuMasterProtocol, 96
 - TMbusRtuOverTcpMasterProtocol, 127
 - TMbusSerialMasterProtocol, 156
 - TMbusTcpMasterProtocol, 187
- openUseExistingConnection
 - TMbusAsciiMasterProtocol, 38
 - TMbusRtuMasterProtocol, 96
 - TMbusSerialMasterProtocol, 157
- parity
 - TMbusAsciiMasterProtocol, 62

- TMbusRtuMasterProtocol, 119
- TMbusSerialMasterProtocol, 180
- pollDelay
 - TMbusAsciiMasterProtocol, 62
 - TMbusMasterFunctions, 89
 - TMbusRtuMasterProtocol, 120
 - TMbusRtuOverTcpMasterProtocol, 150
 - TMbusSerialMasterProtocol, 181
 - TMbusTcpMasterProtocol, 210
- port
 - TMbusRtuOverTcpMasterProtocol, 149
 - TMbusTcpMasterProtocol, 210
- portName
 - TMbusAsciiMasterProtocol, 60
 - TMbusRtuMasterProtocol, 118
 - TMbusSerialMasterProtocol, 179
- readCoils
 - TMbusAsciiMasterProtocol, 39
 - TMbusMasterFunctions, 68
 - TMbusRtuMasterProtocol, 97
 - TMbusRtuOverTcpMasterProtocol, 128
 - TMbusSerialMasterProtocol, 158
 - TMbusTcpMasterProtocol, 188
- readExceptionStatus
 - TMbusAsciiMasterProtocol, 51
 - TMbusMasterFunctions, 80
 - TMbusRtuMasterProtocol, 109
 - TMbusRtuOverTcpMasterProtocol, 140
 - TMbusSerialMasterProtocol, 170
 - TMbusTcpMasterProtocol, 200
- readInputDiscretes
 - TMbusAsciiMasterProtocol, 40
 - TMbusMasterFunctions, 69
 - TMbusRtuMasterProtocol, 98
 - TMbusRtuOverTcpMasterProtocol, 128
 - TMbusSerialMasterProtocol, 158
 - TMbusTcpMasterProtocol, 189
- readInputFloats
 - TMbusAsciiMasterProtocol, 48
 - TMbusMasterFunctions, 76
 - TMbusRtuMasterProtocol, 105
 - TMbusRtuOverTcpMasterProtocol, 136
 - TMbusSerialMasterProtocol, 166
 - TMbusTcpMasterProtocol, 197
- readInputLongInts
 - TMbusAsciiMasterProtocol, 45
 - TMbusMasterFunctions, 74
 - TMbusRtuMasterProtocol, 103
 - TMbusRtuOverTcpMasterProtocol, 134
 - TMbusSerialMasterProtocol, 164
 - TMbusTcpMasterProtocol, 194
- readInputMod10000
 - TMbusAsciiMasterProtocol, 50
 - TMbusMasterFunctions, 78
 - TMbusRtuMasterProtocol, 107
 - TMbusRtuOverTcpMasterProtocol, 138
 - TMbusSerialMasterProtocol, 168
 - TMbusTcpMasterProtocol, 199
- readInputRegisters
 - TMbusAsciiMasterProtocol, 42
 - TMbusMasterFunctions, 71
 - TMbusRtuMasterProtocol, 100
 - TMbusRtuOverTcpMasterProtocol, 131
 - TMbusSerialMasterProtocol, 161
 - TMbusTcpMasterProtocol, 191
- readMultipleFloats
 - TMbusAsciiMasterProtocol, 47
 - TMbusMasterFunctions, 75
 - TMbusRtuMasterProtocol, 105
 - TMbusRtuOverTcpMasterProtocol, 135
 - TMbusSerialMasterProtocol, 165
 - TMbusTcpMasterProtocol, 196
- readMultipleLongInts
 - TMbusAsciiMasterProtocol, 45
 - TMbusMasterFunctions, 73
 - TMbusRtuMasterProtocol, 102
 - TMbusRtuOverTcpMasterProtocol, 133
 - TMbusSerialMasterProtocol, 163
 - TMbusTcpMasterProtocol, 194
- readMultipleMod10000
 - TMbusAsciiMasterProtocol, 49
 - TMbusMasterFunctions, 77
 - TMbusRtuMasterProtocol, 107
 - TMbusRtuOverTcpMasterProtocol, 138
 - TMbusSerialMasterProtocol, 168
 - TMbusTcpMasterProtocol, 198
- readMultipleRegisters
 - TMbusAsciiMasterProtocol, 41
 - TMbusMasterFunctions, 70
 - TMbusRtuMasterProtocol, 99
 - TMbusRtuOverTcpMasterProtocol, 130
 - TMbusSerialMasterProtocol, 160
 - TMbusTcpMasterProtocol, 190
- readWriteRegisters
 - TMbusAsciiMasterProtocol, 44
 - TMbusMasterFunctions, 73
 - TMbusRtuMasterProtocol, 102
 - TMbusRtuOverTcpMasterProtocol, 133
 - TMbusSerialMasterProtocol, 163
 - TMbusTcpMasterProtocol, 193

- restartCommunicationsOption
 - TMbusAsciiMasterProtocol, 52
 - TMbusMasterFunctions, 81
 - TMbusRtuMasterProtocol, 110
 - TMbusRtuOverTcpMasterProtocol, 141
 - TMbusSerialMasterProtocol, 171
 - TMbusTcpMasterProtocol, 201
- retryCnt
 - TMbusAsciiMasterProtocol, 63
 - TMbusMasterFunctions, 89
 - TMbusRtuMasterProtocol, 120
 - TMbusRtuOverTcpMasterProtocol, 150
 - TMbusSerialMasterProtocol, 181
 - TMbusTcpMasterProtocol, 211
- returnQueryData
 - TMbusAsciiMasterProtocol, 52
 - TMbusMasterFunctions, 80
 - TMbusRtuMasterProtocol, 109
 - TMbusRtuOverTcpMasterProtocol, 140
 - TMbusSerialMasterProtocol, 170
 - TMbusTcpMasterProtocol, 201
- Serial Protocols, 15
- setPollDelay
 - TMbusAsciiMasterProtocol, 54
 - TMbusMasterFunctions, 82
 - TMbusRtuMasterProtocol, 111
 - TMbusRtuOverTcpMasterProtocol, 142
 - TMbusSerialMasterProtocol, 172
 - TMbusTcpMasterProtocol, 203
- setPort
 - TMbusRtuOverTcpMasterProtocol, 127
 - TMbusTcpMasterProtocol, 188
- setRetryCnt
 - TMbusAsciiMasterProtocol, 54
 - TMbusMasterFunctions, 82
 - TMbusRtuMasterProtocol, 112
 - TMbusRtuOverTcpMasterProtocol, 143
 - TMbusSerialMasterProtocol, 173
 - TMbusTcpMasterProtocol, 203
- setTimeout
 - TMbusAsciiMasterProtocol, 53
 - TMbusMasterFunctions, 81
 - TMbusRtuMasterProtocol, 110
 - TMbusRtuOverTcpMasterProtocol, 141
 - TMbusSerialMasterProtocol, 171
 - TMbusTcpMasterProtocol, 202
- stopBits
 - TMbusAsciiMasterProtocol, 61
 - TMbusRtuMasterProtocol, 119
 - TMbusSerialMasterProtocol, 180
- TCP/IP Protocols, 13
- timeout
 - TMbusAsciiMasterProtocol, 62
 - TMbusMasterFunctions, 89
 - TMbusRtuMasterProtocol, 120
 - TMbusRtuOverTcpMasterProtocol, 150
 - TMbusSerialMasterProtocol, 180
 - TMbusTcpMasterProtocol, 210
- TMbusAsciiMasterProtocol, 32
 - baudRate, 61
 - configureBigEndianInts, 55
 - configureCountFromOne, 59
 - configureCountFromZero, 59
 - configureEnron32BitMode, 58
 - configureIeeeFloats, 57
 - configureLittleEndianInts, 56, 57
 - configureStandard32BitMode, 58
 - configureSwappedFloats, 56
 - dataBits, 61
 - enableRs485Mode, 39
 - forceMultipleCoils, 41
 - getPackageVersion, 60
 - getPollDelay, 54
 - getRetryCnt, 55
 - getSuccessCounter, 55
 - getTimeout, 53
 - getTotalCounter, 55
 - isOpen, 60
 - maskWriteRegister, 44
 - openProtocol, 38
 - openUseExistingConnection, 38
 - parity, 62
 - pollDelay, 62
 - portName, 60
 - readCoils, 39
 - readExceptionStatus, 51
 - readInputDiscretes, 40
 - readInputFloats, 48
 - readInputLongInts, 45
 - readInputMod10000, 50
 - readInputRegisters, 42
 - readMultipleFloats, 47
 - readMultipleLongInts, 45
 - readMultipleMod10000, 49
 - readMultipleRegisters, 41
 - readWriteRegisters, 44
 - restartCommunicationsOption, 52
 - retryCnt, 63
 - returnQueryData, 52
 - setPollDelay, 54

- setRetryCnt, 54
- setTimeout, 53
- stopBits, 61
- timeout, 62
- TMbusAsciiMasterProtocol, 38
- writeCoil, 40
- writeMultipleFloats, 48
- writeMultipleLongInts, 46
- writeMultipleMod10000, 50
- writeMultipleRegisters, 43
- writeSingleRegister, 43
- TMbusMasterFunctions, 63
 - ~TMbusMasterFunctions, 68
 - configureBigEndianInts, 83, 85
 - configureCountFromOne, 85, 87
 - configureCountFromZero, 85, 88
 - configureEnron32BitMode, 84, 87
 - configureIeeeFloats, 84, 86
 - configureLittleEndianInts, 84, 85
 - configureStandard32BitMode, 84, 87
 - configureSwappedFloats, 84, 86
 - forceMultipleCoils, 70
 - getPackageVersion, 88
 - getPollDelay, 82
 - getRetryCnt, 83
 - getSuccessCounter, 83
 - getTimeout, 81
 - getTotalCounter, 83
 - isOpen, 88
 - maskWriteRegister, 72
 - pollDelay, 89
 - readCoils, 68
 - readExceptionStatus, 80
 - readInputDiscretes, 69
 - readInputFloats, 76
 - readInputLongInts, 74
 - readInputMod10000, 78
 - readInputRegisters, 71
 - readMultipleFloats, 75
 - readMultipleLongInts, 73
 - readMultipleMod10000, 77
 - readMultipleRegisters, 70
 - readWriteRegisters, 73
 - restartCommunicationsOption, 81
 - retryCnt, 89
 - returnQueryData, 80
 - setPollDelay, 82
 - setRetryCnt, 82
 - setTimeout, 81
 - timeout, 89
 - writeCoil, 69
 - writeMultipleFloats, 77
 - writeMultipleLongInts, 75
 - writeMultipleMod10000, 79
 - writeMultipleRegisters, 72
 - writeSingleRegister, 71
- TMbusRtuMasterProtocol, 90
 - baudRate, 118
 - configureBigEndianInts, 113
 - configureCountFromOne, 117
 - configureCountFromZero, 117
 - configureEnron32BitMode, 116
 - configureIeeeFloats, 115
 - configureLittleEndianInts, 114
 - configureStandard32BitMode, 115, 116
 - configureSwappedFloats, 114
 - dataBits, 119
 - enableRs485Mode, 96
 - forceMultipleCoils, 99
 - getPackageVersion, 118
 - getPollDelay, 112
 - getRetryCnt, 112
 - getSuccessCounter, 113
 - getTimeout, 111
 - getTotalCounter, 113
 - isOpen, 118
 - maskWriteRegister, 101
 - openProtocol, 96
 - openUseExistingConnection, 96
 - parity, 119
 - pollDelay, 120
 - portName, 118
 - readCoils, 97
 - readExceptionStatus, 109
 - readInputDiscretes, 98
 - readInputFloats, 105
 - readInputLongInts, 103
 - readInputMod10000, 107
 - readInputRegisters, 100
 - readMultipleFloats, 105
 - readMultipleLongInts, 102
 - readMultipleMod10000, 107
 - readMultipleRegisters, 99
 - readWriteRegisters, 102
 - restartCommunicationsOption, 110
 - retryCnt, 120
 - returnQueryData, 109
 - setPollDelay, 111
 - setRetryCnt, 112
 - setTimeout, 110

- stopBits, 119
- timeout, 120
- TModbusRtuMasterProtocol, 95
- writeCoil, 98
- writeMultipleFloats, 106
- writeMultipleLongInts, 104
- writeMultipleMod10000, 108
- writeMultipleRegisters, 101
- writeSingleRegister, 100
- TModbusRtuOverTcpMasterProtocol, 121
 - configureBigEndianInts, 144
 - configureCountFromOne, 147, 148
 - configureCountFromZero, 148
 - configureEnron32BitMode, 147
 - configureIeeeFloats, 146
 - configureLittleEndianInts, 145
 - configureStandard32BitMode, 146
 - configureSwappedFloats, 144, 145
 - forceMultipleCoils, 129
 - getPackageVersion, 149
 - getPollDelay, 143
 - getPort, 128
 - getRetryCnt, 143
 - getSuccessCounter, 144
 - getTimeout, 142
 - getTotalCounter, 143
 - hostName, 149
 - isOpen, 148
 - maskWriteRegister, 132
 - openProtocol, 127
 - pollDelay, 150
 - port, 149
 - readCoils, 128
 - readExceptionStatus, 140
 - readInputDiscretes, 128
 - readInputFloats, 136
 - readInputLongInts, 134
 - readInputMod10000, 138
 - readInputRegisters, 131
 - readMultipleFloats, 135
 - readMultipleLongInts, 133
 - readMultipleMod10000, 138
 - readMultipleRegisters, 130
 - readWriteRegisters, 133
 - restartCommunicationsOption, 141
 - retryCnt, 150
 - returnQueryData, 140
 - setPollDelay, 142
 - setPort, 127
 - setRetryCnt, 143
 - setTimeout, 141
 - timeout, 150
 - TModbusRtuOverTcpMasterProtocol, 127
 - writeCoil, 129
 - writeMultipleFloats, 137
 - writeMultipleLongInts, 135
 - writeMultipleMod10000, 139
 - writeMultipleRegisters, 132
 - writeSingleRegister, 131
- TModbusSerialMasterProtocol, 151
 - baudRate, 179
 - configureBigEndianInts, 174
 - configureCountFromOne, 177, 178
 - configureCountFromZero, 178
 - configureEnron32BitMode, 177
 - configureIeeeFloats, 176
 - configureLittleEndianInts, 175
 - configureStandard32BitMode, 176
 - configureSwappedFloats, 174, 175
 - dataBits, 179
 - enableRs485Mode, 157
 - forceMultipleCoils, 159
 - getPackageVersion, 179
 - getPollDelay, 173
 - getRetryCnt, 173
 - getSuccessCounter, 174
 - getTimeout, 172
 - getTotalCounter, 173
 - isOpen, 178
 - maskWriteRegister, 162
 - openProtocol, 156
 - openUseExistingConnection, 157
 - parity, 180
 - pollDelay, 181
 - portName, 179
 - readCoils, 158
 - readExceptionStatus, 170
 - readInputDiscretes, 158
 - readInputFloats, 166
 - readInputLongInts, 164
 - readInputMod10000, 168
 - readInputRegisters, 161
 - readMultipleFloats, 165
 - readMultipleLongInts, 163
 - readMultipleMod10000, 168
 - readMultipleRegisters, 160
 - readWriteRegisters, 163
 - restartCommunicationsOption, 171
 - retryCnt, 181
 - returnQueryData, 170

- setPollDelay, 172
- setRetryCnt, 173
- setTimeout, 171
- stopBits, 180
- timeout, 180
- writeCoil, 159
- writeMultipleFloats, 167
- writeMultipleLongInts, 165
- writeMultipleMod10000, 169
- writeMultipleRegisters, 162
- writeSingleRegister, 161
- TMbusTcpMasterProtocol, 181
 - configureBigEndianInts, 204
 - configureCountFromOne, 208
 - configureCountFromZero, 208
 - configureEnron32BitMode, 207
 - configureIeeeFloats, 206
 - configureLittleEndianInts, 205, 206
 - configureStandard32BitMode, 207
 - configureSwappedFloats, 205
 - forceMultipleCoils, 190
 - getPackageVersion, 209
 - getPollDelay, 203
 - getPort, 188
 - getRetryCnt, 204
 - getSuccessCounter, 204
 - getTimeout, 202
 - getTotalCounter, 204
 - hostName, 209
 - isOpen, 209
 - maskWriteRegister, 193
 - openProtocol, 187
 - pollDelay, 210
 - port, 210
 - readCoils, 188
 - readExceptionStatus, 200
 - readInputDiscretEs, 189
 - readInputFloats, 197
 - readInputLongInts, 194
 - readInputMod10000, 199
 - readInputRegisters, 191
 - readMultipleFloats, 196
 - readMultipleLongInts, 194
 - readMultipleMod10000, 198
 - readMultipleRegisters, 190
 - readWriteRegisters, 193
 - restartCommunicationsOption, 201
 - retryCnt, 211
 - returnQueryData, 201
 - setPollDelay, 203
 - setPort, 188
 - setRetryCnt, 203
 - setTimeout, 202
 - timeout, 210
 - TMbusTcpMasterProtocol, 187
 - writeCoil, 189
 - writeMultipleFloats, 197
 - writeMultipleLongInts, 195
 - writeMultipleMod10000, 199
 - writeMultipleRegisters, 192
 - writeSingleRegister, 192
- writeCoil
 - TMbusAsciiMasterProtocol, 40
 - TMbusMasterFunctions, 69
 - TMbusRtuMasterProtocol, 98
 - TMbusRtuOverTcpMasterProtocol, 129
 - TMbusSerialMasterProtocol, 159
 - TMbusTcpMasterProtocol, 189
- writeMultipleFloats
 - TMbusAsciiMasterProtocol, 48
 - TMbusMasterFunctions, 77
 - TMbusRtuMasterProtocol, 106
 - TMbusRtuOverTcpMasterProtocol, 137
 - TMbusSerialMasterProtocol, 167
 - TMbusTcpMasterProtocol, 197
- writeMultipleLongInts
 - TMbusAsciiMasterProtocol, 46
 - TMbusMasterFunctions, 75
 - TMbusRtuMasterProtocol, 104
 - TMbusRtuOverTcpMasterProtocol, 135
 - TMbusSerialMasterProtocol, 165
 - TMbusTcpMasterProtocol, 195
- writeMultipleMod10000
 - TMbusAsciiMasterProtocol, 50
 - TMbusMasterFunctions, 79
 - TMbusRtuMasterProtocol, 108
 - TMbusRtuOverTcpMasterProtocol, 139
 - TMbusSerialMasterProtocol, 169
 - TMbusTcpMasterProtocol, 199
- writeMultipleRegisters
 - TMbusAsciiMasterProtocol, 43
 - TMbusMasterFunctions, 72
 - TMbusRtuMasterProtocol, 101
 - TMbusRtuOverTcpMasterProtocol, 132
 - TMbusSerialMasterProtocol, 162
 - TMbusTcpMasterProtocol, 192
- writeSingleRegister
 - TMbusAsciiMasterProtocol, 43
 - TMbusMasterFunctions, 71
 - TMbusRtuMasterProtocol, 100

TMbusRtuOverTcpMasterProtocol, 131
TMbusSerialMasterProtocol, 161
TMbusTcpMasterProtocol, 192