FieldTalk Modbus Master Library, Java Edition

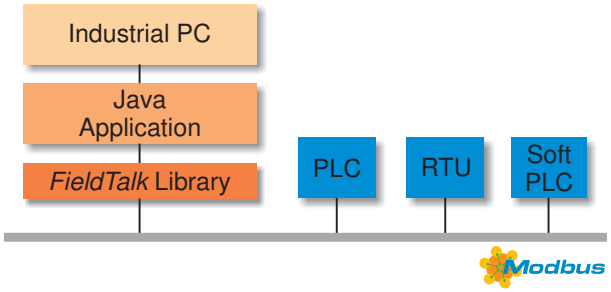
Software manual

Library version 2.6.2

Contents

1 Introduction

The *FieldTalk*™ Modbus Master® Library, Java™ 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). It also helps publishing plant floor data on the web using Java applets and Java servlets.

Features:

- Robust design suitable for 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 and float data types, including Daniel/Enron protocol extensions
- Support for the Extended Lufkin Automation (ELAM) Modbus protocol
- 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 with Exception classes
- Thread-safe implementation to support concurrent programming

1.1 Prerequisites

The package is designed for the Java 2 Platform Standard Edition. It is also compatible with the J2ME Foundation and TINI SDK 1.11 platforms.

The standard Java API does not provide any support for serial ports. However serial port support is available in the form of a Java extension, which must be installed and

licensed separately to the JDK and JRE. The Communications API is not part of the supply of *FieldTalk*.

The serial versions of this package are based on the Java Communications API 2.0. This API is a Java extension, implemented in the package javax.comm.

The Java Communication API is available from different sources:

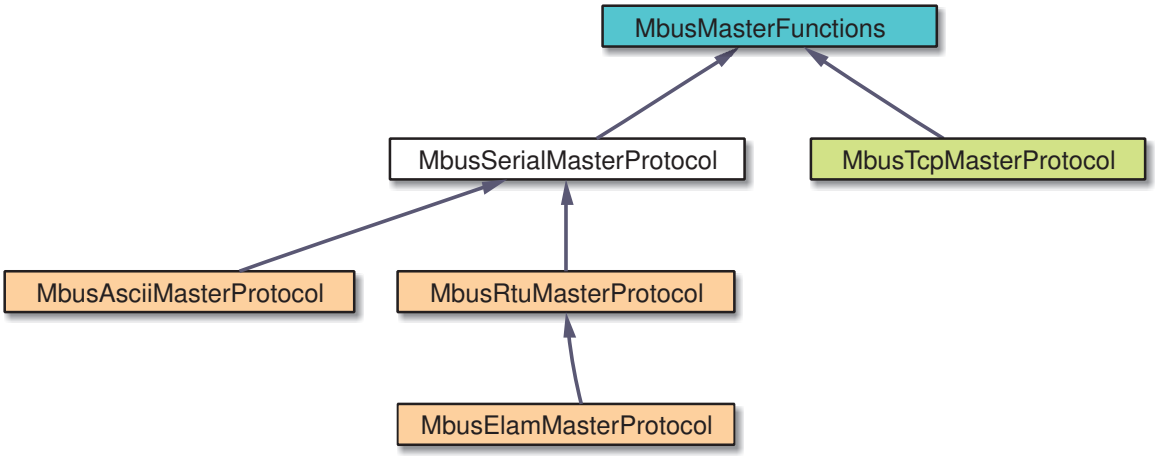
1. SUN's Java Communications API 2.0 for Windows can be found in the lib sub-directory of this package.
2. Sun Microsystems is publishing versions for the Solaris and Linux platform at `http://java.sun.com/products/javacomm/`.
3. Keane Jarvi publishes a LGPL licensed Linux and cross-platform COMM API at `http://www.rxtx.org/`.
4. An excellently performing package for a large variety of platforms is the SerialPort package from Solutions Consulting at `http://www.serialio.com`. Using this package, poll cycles of less than 10 ms can be achieved!

Note:

Without proper installation of a Java Communications API, the serial versions of the FieldTalk package cannot be used!

1.2 Library Structure

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

The class MbusAsciiMasterProtocol implements the Modbus ASCII protocol, the class MbusRtuMasterProtocol implements the Modbus RTU protocol. The class MbusTcpMasterProtocol implements the MODBUS/TCP protocol. Support for the Extended Lufkin Automation (ELAM) Modbus protocol is realized by the class MbusElamMasterProtocol.

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

```
MbusRtuMasterProtocol mbusProtocol = new MbusRtuMasterProtocol();
```

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

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

2 What You should know about Modbus

2.1 Some Background

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

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

2.2 Technical Information

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

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

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

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

Note:

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

2.2.1 The Protocol Functions

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

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

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

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

Function Code	Current Terminology	Classic Terminology
Bit Access		
1	Read Coils	Read Coil Status
2	Read Discrete Inputs	Read Input Status
5	Write Single Coil	Force Single Coil
15 (0F hex)	Write Multiple Coils	Force Multiple Coils
16 Bits Access		
3	Read Multiple Registers	Read Holding Registers
4	Read Input Registers	Read Input Registers
6	Write Single Register	Preset Single Register
16 (10 Hex)	Write Multiple Registers	Preset Multiple Registers
22 (16 hex)	Mask Write Register	Mask Write 4X Register
23 (17 hex)	Read/Write Multiple Registers	Read/Write 4X Registers
Diagnostics		
7	Read Exception Status	Read Exception Status
Vendor Specific		
100 (64 hex)	Read Log History	
Advantech	Send/Receive ADAM 5000/6000 ASCII commands	

2.2.2 How Slave Devices are identified

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

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

2.2.3 The Register Model and Data Tables

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

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

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

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

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

2.2.4 Data Encoding

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

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

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

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

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

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

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

2.2.5 Register and Discrete Numbering Scheme

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

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

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

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

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

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

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

When exchanging register number and discrete number parameters with FieldTalk functions and methdos you have to use the Modbus register and discrete numbering scheme. (Internally the functions will deduct 1 from the start register value before transmitting the value to the slave device.)

2.2.6 The ASCII Protocol

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

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

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

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

2.2.7 The RTU Protocol

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

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

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

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

2.2.8 The MODBUS/TCP Protocol

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

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

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

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

3 How to integrate the Protocol in your Application

3.1 Serial Protocols

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

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

1. Import the library package

```
import com.focus_sw.fieldtalk.*;
```

2. Device data profile definition

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

```
short[] readRegSet = new short[20]
short[] writeRegSet = new short[20]
boolean[] readBitSet = new boolean[20]
boolean[] writeBitSet = new boolean[20]
```

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

```
float[] readRegSet = new float[10]
float[] writeRegSet = new float[10]
```

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

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

```
int[] readRegSet = new int[10]
int[] writeRegSet = new int[10]
```

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

3. Declare and instantiate a protocol object

```
MbusRtuMasterProtocol mbus = new MbusRtuMasterProtocol ();
```

4. Open the protocol port

```
mbus.openPort("COM1", 19200, MbusRtuMasterProtocol.DATABITS_8,
                             MbusRtuMasterProtocol.STOPBITS_1,
                             MbusRtuMasterProtocol.PARITY_EVEN);
```

5. Perform the data transfer functions

- To read register values:

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

- To write a single register value:

```
mbus.writeSingleRegister(1, 200, writeRegSet);
```

- To write a single float value:

```
mbus.writeSingleRegister(1, 210, 3.141F);
```

- To write mutliple register values:

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

- To read boolean (discrete) values:

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

- To write a single boolean (discrete) value:

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

- To write multiple boolean (discrete) values:

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

6. Close the protocol port if not needed any more

```
mbus.closePort();
```

7. Error Handling

Serial protocol errors like slave device failures, transmission failures, checksum errors and time-outs throw appropriate exceptions, which are all derived from the IOException and BusProtocolException class. The following code snippet can handle these errors:

```
try
{
   mbus.readMultipleRegisters(1, 100, dataSetArray);
}
catch (BusProtocolException e)
{
   Has to be caught before IOException!
   Protocol error management, something is wrong
   with the slave device or the bus.
}
catch (IOException e)
{
   If this occurrs after we checked for a BusProtocolException
   then it signals that the local I/O subsytem failed, which is
   fatal.

}
```

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

3.2 TCP/IP Protocols

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

```
import com.focus_sw.fieldtalk.*;
```

2. Device data profile definition

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

```
short[] readRegSet = new short[20]
short[] writeRegSet = new short[20]
boolean[] readBitSet = new boolean[20]
boolean[] writeBitSet = new boolean[20]
```

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

```
float[] readRegSet = new float[10]
float[] writeRegSet = new float[10]
```

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

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

```
int[] readRegSet = new int[10]
int[] writeRegSet = new int[10]
```

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

3. Declare and instantiate a protocol object

```
MbusTcpMasterProtocol mbus = new MbusTcpMasterProtocol ();
```

4. Open the protocol port

```
mbus.openConnection("10.0.0.11);
```

5. Perform the data transfer functions

- To read register values:

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

- To write a single register value:

```
mbus.writeSingleRegister(1, 200, writeRegSet);
```

- To write a single float value:

```
mbus.writeSingleRegister(1, 210, 3.141F);
```

- To write mutliple register values:

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

- To read boolean (discrete) values:

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

- To write a single boolean (discrete) value:

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

- To write multiple boolean (discrete) values:

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

6. Close the connection if not needed any more

```
mbus.closeConnection();
```

7. Error Handling

TCP/IP protocol errors like slave failures, TCP/IP connection failures and time-outs throw appropriate exceptions, which are all derived from the IOException and BusProtocolException class. The following code snippet can handle these errors:

```
try
{
   mbus.readMultipleRegisters(1, 100, dataSetArray);
}
catch (BusProtocolException e)
{
   Protocol error management, something is wrong
   with the slave device but the connection is still alive.
}
catch (IOException e)
{
   If this occurrs after we checked for a BusProtocolException
   then 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.
}
```

3.3 Examples

- **Serial Example** (p. **??**)
- **Multithreaded Example** (p. **??**)
- **MODBUS/TCP Example** (p. **??**)
- **Applet Example with Cyclic Polling** (p. **??**)

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.

The Java edition is using the Java 2 Platform Standard Edition API and the Java Communications API. This enables compatibility with most VM implementations.

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

5 Module Documentation

5.1 Data and Control Functions for all Modbus Protocol Flavours

This Modbus protocol library implements the most commonly used data functions as well as some control functions.

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

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

The following table lists the supported Modbus function codes:

Function Code	Current Terminology	Classic Terminology
Bit Access		
1	Read Coils	Read Coil Status
2	Read Discrete Inputs	Read Input Status
5	Write Single Coil	Force Single Coil
15 (0F hex)	Write Multiple Coils	Force Multiple Coils
16 Bits Access		
3	Read Multiple Registers	Read Holding Registers
4	Read Input Registers	Read Input Registers
6	Write Single Register	Preset Single Register
16 (10 Hex)	Write Multiple Registers	Preset Multiple Registers
22 (16 hex)	Mask Write Register	Mask Write 4X Register
23 (17 hex)	Read/Write Multiple Registers	Read/Write 4X Registers
Diagnostics		
7	Read Exception Status	Read Exception Status
Vendor Specific		
100 (64 hex)	Read Log History	
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. 7). (Internally the functions will deduct 1 from the start register value before transmitting the value to the slave device.)

All master data and control functions are designed thread-safe. All methods of a particular

MbusMaster... object can only be accessed by one thread at the same time. In addition, access to the register array objects is synchronized to allow concurrent access. This enables one thread to update a register array while a second thread is sending the data to a Modbus slave. In this case the update thread must also synchronize his access to the register array by embedding the access into a synchronized(regArr) clause.

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).

5.2 Serial Protocols

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

Classes

- class **MbusAsciiMasterProtocol**

  Modbus ASCII Master Protocol class.

- class **MbusRtuMasterProtocol**

  Modbus RTU Master Protocol class.

- class **MbusElamMasterProtocol**

  Extended Lufkin Automation Modbus Master Protocol.

5.2.1 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. 8) and **The ASCII Protocol** (p. 8) for some background information about the two serial Modbus protocols.

See section **Serial Protocols** (p. 9) for an example how to use the MbusRtuMasterProtocol class.

5.3 TCP/IP Protocols

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

Classes

- class **MbusTcpMasterProtocol**

  MODBUS/TCP Master Protocol class.

- class **MbusRtuOverTcpMasterProtocol**

  Encapsulated Modbus RTU Master Protocol class.

5.3.1 Detailed Description

The library provides two flavours of TCP/IP based Modbus protocols. The MODBUS/TCP master protocol is implemented in the class MbusTcpMasterProtocol. The MbusTcpMasterProtocol 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. 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. 8) for some background information about MODBUS/TCP.

See section **TCP/IP Protocols** (p. 11) for an example how to use the MbusTcpMasterProtocol class.

5.4 Device and Vendor Specific Modbus Functions

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

Custom Function Codes

- synchronized void **readHistoryLog** (int slaveAddr, int channelNo, int resolution, short readArr[ ]) throws IOException, BusProtocolException

  Vendor Specific Modbus function 100 (64 hex), Read History Log.

Advantec ADAM 5000/6000 Series Commands

- String **adamSendReceiveAsciiCmd** (String commandStr) throws IOException, BusProtocolException

Send/Receive ADAM 5000/6000 ASCII command.

5.4.1 Detailed Description

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

5.4.2 Function Documentation

synchronized void readHistoryLog (int *slaveAddr*, int *channelNo*, int *resolution*, short *readArr[]*) throws IOException, BusProtocolException `[inherited]`

Vendor Specific Modbus function 100 (64 hex), Read History Log.

Parameters:

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

channelNo A/D channel Number (Range: 1 - 10)

resolution Resolution in minutes (Range: 1 - 60)

readArr Buffer with log data received. The length of the array determines how many data points are read (Range: 1 - 125).

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(p. 20)** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

String adamSendReceiveAsciiCmd (String *commandStr*) throws IOException, BusProtocolException `[inherited]`

Send/Receive ADAM 5000/6000 ASCII command.

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

Parameters:

commandStr Buffer which holds command string. Must not be longer than 255 characters. A possible trailing CR is removed.

Returns:

Response string.

Note:

No broadcast supported

5.5 Error Management

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

Classes

- class **BusProtocolException**
  Communication Error.
- class **ChecksumException**
  Checksum error.
- class **InvalidFrameException**
  Invalid frame error.
- class **InvalidReplyException**
  Invalid reply error.
- class **EvaluationExpiredException**
  Evaluation expired error.
- class **MbusIllegalAddressException**
  Illegal Data Address exception response.
- class **MbusIllegalFunctionException**
  Illegal Function exception response.
- class **MbusIllegalValueException**
  Illegal Data Value exception response.
- class **MbusResponseException**
  Modbus exception response.
- class **MbusSlaveFailureException**
  Slave Device Failure exception response.
- class **ReplyTimeoutException**
  Reply time-out.

5.5.1 Detailed Description

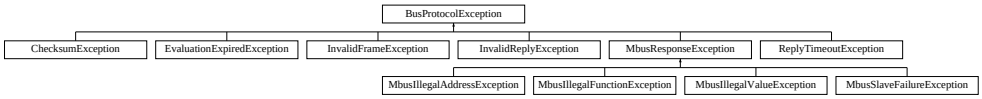
This module documents all the exception classes, error and return codes reported by the various library functions.

6 Java Class Documentation

6.1 BusProtocolException Class Reference

Communication Error.

Inheritance diagram for BusProtocolException:



Public Member Functions

- **BusProtocolException** (String text)

  *Constructs a **BusProtocolException** (p. 20) with the specified detail message.*

- **BusProtocolException** ()

  *Constructs a **BusProtocolException** (p. 20) without a detail message.*

Static Public Member Functions

- static long **getCounter** ()

  Returns how often this exception has been triggered.

- static void **resetCounter** ()

  Resets exception counter.

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:

IOException

6.1.2 Constructor & Destructor Documentation

BusProtocolException (String *text*)

Constructs a **BusProtocolException** (p. 20) with the specified detail message.

Parameters:

text The detail message.

6.1.3 Member Function Documentation

static long getCounter ( ) [static]

Returns how often this exception has been triggered.

Returns:

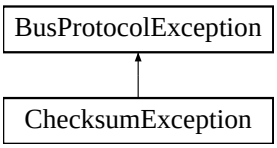
Counter value

Reimplemented in **ChecksumException** (p. 22), **InvalidFrameException** (p. 26), **InvalidReplyException** (p. 27), **EvaluationExpiredException** (p. 25), **MbusIllegalAddressException** (p. 79), **MbusIllegalFunctionException** (p. 80), **MbusIllegalValueException** (p. 81), **MbusResponseException** (p. 104), **MbusSlaveFailureException** (p. 152), and **ReplyTimeoutException** (p. 177).

6.2 ChecksumException Class Reference

Checksum error.

Inheritance diagram for ChecksumException:



Public Member Functions

- **ChecksumException** ()

  *Constructs a **ChecksumException** (p. 21).*

Static Public Member Functions

- static long **getCounter** ()

  Returns how often this exception has been triggered.

- static void **resetCounter** ()

  Resets exception counter.

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:

BusProtocolException (p. 20)

6.2.2 Member Function Documentation

static long getCounter ( ) `[static]`

Returns how often this exception has been triggered.

Returns:

Counter value

Reimplemented from **BusProtocolException** (p. 21).

6.3 Converter Class Reference

Conversion Routines.

Public Member Functions

- **Converter** (boolean swapMode)

  *Creates a **Converter** (p. 22) object.*

- void **floatsToShorts** (float[ ] floatArr, short[ ] shortArr)

  Binary conversion from a float array to a short array.

- void **intsToShorts** (int[ ] intArr, short[ ] shortArr)

  Binary conversion from a int array to a short array.

- void **shortsToInts** (short[ ] shortArr, int[ ] intArr)

  Binary conversion from a short array to a int array.

- void **shortsToFloats** (short[ ] shortArr, float[ ] floatArr)

  Binary conversion from a short array to a float array.

Static Public Attributes

- static final boolean **BIG_ENDIAN** = false

  Big endian conversion.

- static final boolean **LITTLE_ENDIAN** = true

  Little endian conv.

6.3.1 Detailed Description

Conversion Routines. Utility class which provides functions to convert between different binary data repesentations.

6.3.2 Constructor & Destructor Documentation

Converter (boolean ***swapMode*** **)**

Creates a **Converter** (p. 22) object.

Parameters:

swapMode BIG_ENDIAN: most significant word is the 0th element, LITTLE_-ENDIAN: least significant word is the 0th element.

6.3.3 Member Function Documentation

void floatsToShorts (float[ ] ***floatArr,*** **short[ ]** ***shortArr*** **)**

Binary conversion from a float array to a short array.

Note:

The short array must be twice as large as the float array!

Exceptions:

ArrayStoreException Array size mismatch

void intsToShorts (int[ ] ***intArr,*** **short[ ]** ***shortArr*** **)**

Binary conversion from a int array to a short array.

Note:

The short array must be twice as large as the int array!

Exceptions:

ArrayStoreException Array size mismatch

void shortsToInts (short[ ] *shortArr,* int[ ] *intArr*)

Binary conversion from a short array to a int array.

Note:

The short array must be twice as large as the int array!

Exceptions:

ArrayStoreException Array size mismatch

void shortsToFloats (short[ ] *shortArr,* float[ ] *floatArr*)

Binary conversion from a short array to a float array.

Note:

The short array must be twice as large as the float array!

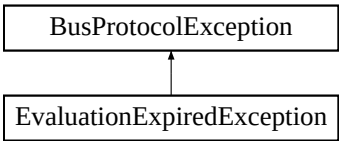
Exceptions:

ArrayStoreException Array size mismatch

6.4 EvaluationExpiredException Class Reference

Evaluation expired error.

Inheritance diagram for EvaluationExpiredException:



Public Member Functions

- **EvaluationExpiredException** ()

  Creates a new ***EvaluationExpiredException*** *(p. 24) instance.*

Static Public Member Functions

- static long **getCounter** ()

  Returns how often this exception has been triggered.

- static void **resetCounter** ()

  Resets exception counter.

6.4.1 Detailed Description

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

See also:

BusProtocolException (p. 20)

6.4.2 Member Function Documentation

static long getCounter ( ) `[static]`

Returns how often this exception has been triggered.

Returns:

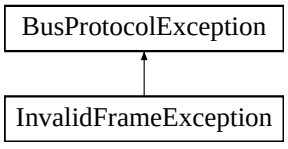
Counter value

Reimplemented from **BusProtocolException** (p. 21).

6.5 InvalidFrameException Class Reference

Invalid frame error.

Inheritance diagram for InvalidFrameException:



Public Member Functions

- **InvalidFrameException** ()

  *Creates a new **InvalidFrameException** (p. 25) instance.*

Static Public Member Functions

- static long **getCounter** ()

  Returns how often this exception has been triggered.

- static void **resetCounter** ()

  Resets exception counter.

6.5.1 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:

BusProtocolException (p. 20)

6.5.2 Member Function Documentation

static long getCounter ( ) `[static]`

Returns how often this exception has been triggered.

Returns:

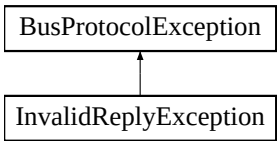
Counter value

Reimplemented from **BusProtocolException** (p. 21).

6.6 InvalidReplyException Class Reference

Invalid reply error.

Inheritance diagram for InvalidReplyException:



Public Member Functions

- **InvalidReplyException** ()

  Creates a new ***InvalidReplyException*** *(p. 26) instance.*

Static Public Member Functions

- static long **getCounter** ()

  Returns how often this exception has been triggered.

- static void **resetCounter** ()

  Resets exception counter.

6.6.1 Detailed Description

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

See also:

BusProtocolException (p. 20)

6.6.2 Member Function Documentation

static long getCounter ( ) `[static]`

Returns how often this exception has been triggered.

Returns:

Counter value

Reimplemented from **BusProtocolException** (p. 21).

6.7 Logger Class Reference

Logger (p. 27) class.

Public Member Functions

- **Logger** (String categoryName)
  Creates a new **`Logger`** (p. 27) *instance.*

Static Package Functions

- **[static initializer]**
  Create static context and loads logger.proprties configuration file.

Print and Output Routines

- synchronized void **println** (String text)
  Prints a log message.
- synchronized void **println** (String text, int val1)
  Prints a log message including 1 integer parameter.
- synchronized void **println** (String text, int val1, int val2)
  Prints a log message including 2 integer parameters.
- synchronized void **println** (String text, int val1, int val2, int val3)
  Prints a log message including 3 integer parameters.
- synchronized void **println** (String text, int val1, int val2, int val3, int val4)
  Prints a log message including 4 integer parameters.
- synchronized void **printHexDump** (byte[ ] dataArr, int ofs, int len)
  Prints an array of bytes in hexadecimal notation.
- synchronized void **printAsciiDump** (byte[ ] dataArr, int ofs, int len)
  Interprets the data bytes as ASCII characters and outStream.prints them.

Logger Configuration

- static synchronized void **loggerEnable** ()
  Enables logging to stdout.
- static synchronized void **loggerDisable** ()
  Disbles logging.
- static synchronized void **categoryEnable** (String categoryName)
  Enabled log outputs for this category.

- static synchronized void **categoryDisable** (String categoryName)

  Disables log outputs for this category.

6.7.1 Detailed Description

Logger (p. 27) class. Used for debugging and performance analysis.

Logging can be enabled and configured via an optional logger.properties file and programmatically during run-time. The logger.properties is searched in the current directory.

Sample logger.properties `file:`

```
#
# Logger configuration file
#

# Set to true to enable logging in principle
logger.enable=true

# Exception category
logger.category.EXCP=true

# Function category
logger.category.FUNC=true

# Send category
logger.category.SEND=true

# Receive category
logger.category.RECV=true
```

See also:

PrintStream

6.7.2 Constructor & Destructor Documentation

Logger (String *categoryName* **)**

Creates a new **`Logger`** (p. 27) instance.

Parameters:

categoryName A log category name (typically 4 letters)

6.7.3 Member Function Documentation

static synchronized void categoryEnable (String *categoryName* **)** **`[static]`**

Enabled log outputs for this category.

Parameters:

categoryName The log category name

static synchronized void categoryDisable (String *categoryName*) [static]

Disables log outputs for this category.

Parameters:

categoryName The log category name

synchronized void println (String *text*)

Prints a log message.

Parameters:

text Log text

synchronized void println (String *text,* int *val1*)

Prints a log message including 1 integer parameter.

Parameters:

text Log text
val1 Integer value

synchronized void println (String *text,* int *val1,* int *val2*)

Prints a log message including 2 integer parameters.

Parameters:

text Log text
val1 Integer value
val2 Integer value

synchronized void println (String *text*, int *val1*, int *val2*, int *val3*)

Prints a log message including 3 integer parameters.

Parameters:

text Log text
val1 Integer value
val2 Integer value
val3 Integer value

synchronized void println (String *text*, int *val1*, int *val2*, int *val3*, int *val4*)

Prints a log message including 4 integer parameters.

Parameters:

text Log text
val1 Integer value
val2 Integer value
val3 Integer value
val4 Integer value

synchronized void printHexDump (byte[ ] *dataArr*, int *ofs*, int *len*)

Prints an array of bytes in hexadecimal notation.

Parameters:

dataArr Data bytes to outStream.print as chars
ofs Subarray offset
len Number of bytes to outStream.print

synchronized void printAsciiDump (byte[ ] *dataArr*, int *ofs*, int *len*)

Interprets the data bytes as ASCII characters and outStream.prints them.

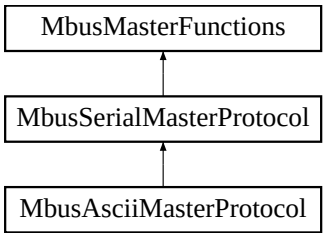
Parameters:

dataArr Data bytes to outStream.print as chars
ofs Subarray offset
len Number of bytes to outStream.print

6.8 MbusAsciiMasterProtocol Class Reference

Modbus ASCII Master Protocol class.

Inheritance diagram for MbusAsciiMasterProtocol:



Public Member Functions

- synchronized void **openProtocol** (String portName, int baudRate, int dataBits, int stopBits, int parity) throws IOException, PortInUseException, UnsupportedCommOperationException

  Opens a Modbus ASCII serial port with specific port parameters.

- synchronized void **closeProtocol** () throws IOException

  Closes the serial port and releases any system resources associated with the port.

- boolean **isOpen** ()

  Returns whether the port is open or not.

Static Public Attributes

- static final int **DATABITS_7** = SerialPort.DATABITS_7

  7 data bits

- static final int **DATABITS_8** = SerialPort.DATABITS_8

  8 data bits

- static final int **STOPBITS_1** = SerialPort.STOPBITS_1

  1 stop bit

- static final int **STOPBITS_1_5** = SerialPort.STOPBITS_1_5

  1.5 stop bits

- static final int **STOPBITS_2** = SerialPort.STOPBITS_2

  2 stop bits

- static final int **PARITY_NONE** = SerialPort.PARITY_NONE

  no parity

- static final int **PARITY_ODD** = SerialPort.PARITY_ODD

  odd parity

- static final int **PARITY_EVEN** = SerialPort.PARITY_EVEN

  even parity

- static final int **PARITY_MARK** = SerialPort.PARITY_MARK

  mark parity

- static final int **PARITY_SPACE** = SerialPort.PARITY_SPACE

  space parity

Bit Access

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

- synchronized void **readCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 1 (01 hex), Read Coil Status/Read Coils.

- synchronized void **readInputDiscretes** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

- synchronized void **writeCoil** (int slaveAddr, int bitAddr, boolean bitVal) throws IOException, BusProtocolException

  Modbus function 5 (05 hex), Force Single Coil/Write Coil.

- synchronized void **forceMultipleCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

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

Register Access (16-bit, 32-bit and floating point)

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

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex), Read Input Registers.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

- synchronized void **writeSingleRegister** (int slaveAddr, int regAddr, short regVal) throws IOException, BusProtocolException

  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

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

- synchronized void **maskWriteRegister** (int slaveAddr, int regAddr, short andMask, short orMask) throws IOException, BusProtocolException

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

- synchronized void **readWriteRegisters** (int slaveAddr, int readRef, short[ ] readArr, int writeRef, short[ ] writeArr) throws IOException, BusProtocolException

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

Diagnostics

- synchronized byte **readExceptionStatus** (int slaveAddr) throws IOException, BusProtocolException

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

Custom Function Codes

- synchronized void **readHistoryLog** (int slaveAddr, int channelNo, int resolution, short readArr[ ]) throws IOException, BusProtocolException

  Vendor Specific Modbus function 100 (64 hex), Read History Log.

Protocol Configuration

- synchronized void **setTimeout** (int timeOut)

  Configures time-out.

- int **getTimeout** ()

  Returns the operation time-out value.

- synchronized void **setPollDelay** (int pollDelay)

  Configures poll delay.

- int **getPollDelay** ()

  Returns the poll delay time.

- synchronized void **setRetryCnt** (int retryCnt)

  Configures the automatic retry setting.

- int **getRetryCnt** ()

  Returns the automatic retry count.

Transmission Statistic Functions

- synchronized long **getTotalCounter** ()

  Returns how often a transmit/receive cycle has been executed.

- synchronized void **resetTotalCounter** ()

  Resets total transmit/receive cycle counter.

- synchronized long **getSuccessCounter** ()

  Returns how often a transmit/receive cycle was successful.

- synchronized void **resetSuccessCounter** ()

  Resets successful transmit/receive counter.

Slave Configuration

- void **configureStandard32BitMode** ()
  Configures all slaves for Standard 32-bit Mode.
- synchronized void **configureStandard32BitMode** (int slaveAddr)
  Configures a slave for Standard 32-bit Register Mode.
- void **configureSingleReg32BitMode** ()
  Configures all slaves for Single Register 32-bit Mode.
- synchronized void **configureSingleReg32BitMode** (int slaveAddr)
  Configures all slaves for Single Register 32-bit Mode.
- void **configureCountFromOne** ()
  Configures the reference counting scheme to start with one for all slaves.
- synchronized void **configureCountFromOne** (int slaveAddr)
  Configures a slave's reference counting scheme to start with one.
- void **configureCountFromZero** ()
  Configures the reference counting scheme to start with zero for all slaves.
- synchronized void **configureCountFromZero** (int slaveAddr)
  Configures a slave's reference counting scheme to start with zero.
- void **configureLittleEndianInts** ()
  Disables word swapping for int data type functions for all slaves.
- synchronized void **configureLittleEndianInts** (int slaveAddr)
  Disables word swapping for int data type functions on a per slave basis.
- void **configureBigEndianInts** ()
  Configures int data type functions to do a word swap for all slaves.
- synchronized void **configureBigEndianInts** (int slaveAddr)
  Enables int data type functions to do a word swap on a per slave basis.
- void **configureLittleEndianFloats** ()
  Disables float data type functions to do a word swap for all slaves.
- synchronized void **configureLittleEndianFloats** (int slaveAddr)
  Disables float data type functions to do a word swap on a per slave basis.
- void **configureSwappedFloats** ()
  Configures float data type functions to do a word swap for all slaves.
- synchronized void **configureSwappedFloats** (int slaveAddr)

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

6.8.1 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).

All functions have been implemented thread-safe. It is also possible to instantiate multiple instances for establishing multiple connections on different serial ports.

See also:

Data and Control Functions for all Modbus Protocol Flavours (p. 14), **Serial Protocols** (p. 15)
MbusRtuMasterProtocol (p. 105), **MbusElamMasterProtocol** (p. 55), **MbusTcpMasterProtocol** (p. 153)

6.8.2 Member Function Documentation

synchronized void openProtocol (String *portName*, int *baudRate*, int *dataBits*, int *stopBits*, int *parity*) throws IOException, PortInUseException, UnsupportedCommOperationException

Opens a Modbus ASCII serial port with specific port parameters.

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

Note:

The default time-out for the data transfer is 1000 ms.
The default poll delay is 0 ms.
Automatic retries are switched off (retry count is 0).
The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.

Parameters:

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

baudRate The port baudRate in bps (typically 1200 - 19200)

dataBits DATABITS_7: 7 data bits, DATABITS_8: data bits

stopBits STOPBITS_1: 1 stop bit, STOPBITS_1_5: 1.5 stop bits, STOPBITS_2: 2 stop bits

parity PARITY_NONE: no parity, PARITY_ODD: odd parity, PARITY_EVEN: even parity, PARITY_MARK: mark parity, PARITY_SPACE: space parity

Exceptions:

IOException An I/O error occured

UnsupportedCommOperationException A communication parameter is not supported

PortInUseException Port is already used by somebody else

InvalidParameterException A parameter is invalid

Reimplemented from **MbusSerialMasterProtocol** (p. 134).

synchronized void closeProtocol ( ) throws IOException `[virtual, inherited]`

Closes the serial port and releases any system resources associated with the port.

Exceptions:

IOException An I/O error occured

Implements **MbusMasterFunctions** (p. 82).

boolean isOpen ( ) `[virtual, inherited]`

Returns whether the port is open or not.

Return values:

true = open

false = closed

Implements **MbusMasterFunctions** (p. 103).

synchronized void readCoils (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException `[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: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputDiscretes (int *slaveAddr,* int *startRef,* boolean[ ] *bitArr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeCoil (int *slaveAddr*, int *bitAddr*, boolean *bitVal*) throws IOException, BusProtocolException `[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 - 247)

bitAddr Coil address (Range: depends on configuration setting)

bitVal true sets, false clears discrete output variable

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void forceMultipleCoils (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException `[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: depends on configuration setting)

bitArr Buffer which contains the data to be sent. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException `[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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int *slaveAddr,* int *startRef,* float[] *float32Arr*) throws IOException, BusProtocolException `[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 two consecutive output registers (holding registers, 4:00000 table) into float values.

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* short[] *regArr*) throws IOException, BusProtocolException `[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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

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

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr*, int *startRef*, float[ ] *float32Arr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeSingleRegister (int *slaveAddr*, int *regAddr*, short *regVal*) throws IOException, BusProtocolException `[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 - 247)

regAddr Register address (Range: depends on configuration setting)

regVal Data to be sent

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

*BusProtocolException (*p. *20)* A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException `[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 - 247)

startRef Start reference (Range depends on configuration setting)

regArr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

*BusProtocolException (*p. *20)* A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

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

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

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int ***slaveAddr,*** **int** ***startRef,*** **float[]** ***float32Arr*** **) throws IOException, BusProtocolException** `[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 a pairs of output registers (holding registers, 4:00000 table).

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void maskWriteRegister (int *slaveAddr*, int *regAddr*, short *andMask*, short *orMask*) throws IOException, BusProtocolException [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: depends on configuration setting)

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

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readWriteRegisters (int *slaveAddr*, int *readRef*, short[ ] *readArr*, int *writeRef*, short[ ] *writeArr*) throws IOException, BusProtocolException [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 reference for reading (Range: depends on configuration setting)

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

writeRef Start reference for writing (Range: depends on configuration setting)

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20***) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized byte readExceptionStatus (int *slaveAddr*) throws IOException, BusProtocolException `[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)

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20***) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void setTimeout (int *timeOut*) `[inherited]`

Configures time-out.

This function sets the time-out to the specified value. A value of 0 disables the time-out, which causes all subsequent calls to data and control functions to block.

Note:

A port or connection must be closed in order to configure it.

Parameters:

timeOut Timeout value in ms (Range: 0 - 100000), 0 disables time-out

Exceptions:

IllegalStateException Port is open

IllegalArgumentException Parameter is out of range

int getTimeout ( ) [inherited]

Returns the operation time-out value.

Returns:

Timeout value in ms

synchronized void setPollDelay (int *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.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open

IllegalArgumentException Parameter is out of range

int getPollDelay ( ) [inherited]

Returns the poll delay time.

Returns:

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

synchronized void setRetryCnt (int *retryCnt*) [inherited]

Configures the automatic retry setting.
A value of 0 disables any automatic retries.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open
IllegalArgumentException Parameter is out of range

int getRetryCnt ( ) [inherited]

Returns the automatic retry count.

Returns:

Retry count

synchronized long getTotalCounter ( ) [inherited]

Returns how often a transmit/receive cycle has been executed.

Returns:

Counter value

synchronized long getSuccessCounter ( ) [inherited]

Returns how often a transmit/receive cycle was successful.

Returns:

Counter value

void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

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

Remarks:

This is the default mode

synchronized void configureStandard32BitMode (int *slaveAddr*) [inherited]

Configures a slave for Standard 32-bit Register Mode.

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

Parameters:

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

Remarks:

This is the default mode

void configureSingleReg32BitMode ( ) [inherited]

Configures all slaves for Single Register 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.

synchronized void configureSingleReg32BitMode (int *slaveAddr*) [inherited]

Configures all slaves for Single Register 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.

void configureCountFromOne ( ) [inherited]

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

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

Remarks:

This is the default mode

synchronized void configureCountFromOne (int *slaveAddr*) [inherited]

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

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

Parameters:

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

Remarks:

This is the default mode

void configureCountFromZero ( ) [inherited]

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

This is also known as PDU addressing.

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

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

synchronized void configureCountFromZero (int *slaveAddr*) [inherited]

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

This is also known as PDU addressing.

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

Parameters:

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

void configureLittleEndianInts ( ) [inherited]

Disables word swapping for int data type functions for all slaves.

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

Remarks:

This is the default mode

synchronized void configureLittleEndianInts (int *slaveAddr*) [inherited]

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

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

Remarks:

This is the default mode

Parameters:

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

void configureBigEndianInts ( ) [inherited]

Configures int data type functions to do a word swap for all slaves.

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.

synchronized void configureBigEndianInts (int *slaveAddr*) [inherited]

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

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

Parameters:

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

void configureLittleEndianFloats ( ) [inherited]

Disables float data type functions to do a word swap for all slaves.

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

Remarks:

This is the default mode

synchronized void configureLittleEndianFloats (int *slaveAddr*) [inherited]

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

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

Remarks:

This is the default mode

Parameters:

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

void configureSwappedFloats ( ) [inherited]

Configures float data type functions to do a word swap for all slaves.

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.

synchronized void configureSwappedFloats (int *slaveAddr*) [inherited]

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

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

Note:

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

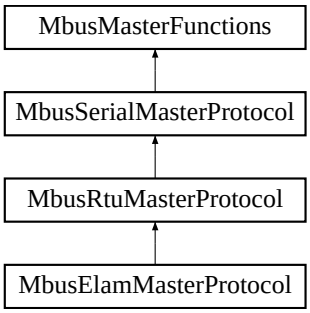
Parameters:

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

6.9 MbusElamMasterProtocol Class Reference

Extended Lufkin Automation Modbus Master Protocol.

Inheritance diagram for MbusElamMasterProtocol:



Public Member Functions

- **MbusElamMasterProtocol** ()

  Creates new instance.

- synchronized void **openProtocol** (String portName, int baudRate, int dataBits, int stopBits, int parity) throws IOException, PortInUseException, UnsupportedCommOperationException

  Opens a Modbus serial port with specific port parameters.

- synchronized void **closeProtocol** () throws IOException

  Closes the serial port and releases any system resources associated with the port.

- boolean **isOpen** ()

  Returns whether the port is open or not.

Static Public Attributes

- static final int **DATABITS_8** = SerialPort.DATABITS_8

  8 data bits

- static final int **STOPBITS_1** = SerialPort.STOPBITS_1

  1 stop bit

- static final int **STOPBITS_1_5** = SerialPort.STOPBITS_1_5

  1.5 stop bits

- static final int **STOPBITS_2** = SerialPort.STOPBITS_2

  2 stop bits

- static final int **PARITY_NONE** = SerialPort.PARITY_NONE

  no parity

- static final int **PARITY_ODD** = SerialPort.PARITY_ODD

  odd parity

- static final int **PARITY_EVEN** = SerialPort.PARITY_EVEN

  even parity

- static final int **PARITY_MARK** = SerialPort.PARITY_MARK

  mark parity

- static final int **PARITY_SPACE** = SerialPort.PARITY_SPACE

  space parity

Bit Access

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

- synchronized void **readCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 1 (01 hex), Read Coil Status/Read Coils.

- synchronized void **readInputDiscretes** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

- synchronized void **writeCoil** (int slaveAddr, int bitAddr, boolean bitVal) throws IOException, BusProtocolException

  Modbus function 5 (05 hex), Force Single Coil/Write Coil.

- synchronized void **forceMultipleCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

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

Register Access (16-bit, 32-bit and floating point)

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

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex), Read Input Registers.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

- synchronized void **writeSingleRegister** (int slaveAddr, int regAddr, short regVal) throws IOException, BusProtocolException

  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

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

- synchronized void **maskWriteRegister** (int slaveAddr, int regAddr, short andMask, short orMask) throws IOException, BusProtocolException

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

- synchronized void **readWriteRegisters** (int slaveAddr, int readRef, short[ ] readArr, int writeRef, short[ ] writeArr) throws IOException, BusProtocolException

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

Diagnostics

- synchronized byte **readExceptionStatus** (int slaveAddr) throws IOException, BusProtocolException

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

Custom Function Codes

- synchronized void **readHistoryLog** (int slaveAddr, int channelNo, int resolution, short readArr[ ]) throws IOException, BusProtocolException

  Vendor Specific Modbus function 100 (64 hex), Read History Log.

Protocol Configuration

- synchronized void **setTimeout** (int timeOut)

  Configures time-out.

- int **getTimeout** ()

  Returns the operation time-out value.

- synchronized void **setPollDelay** (int pollDelay)

  Configures poll delay.

- int **getPollDelay** ()

  Returns the poll delay time.

- synchronized void **setRetryCnt** (int retryCnt)

  Configures the automatic retry setting.

- int **getRetryCnt** ()

  Returns the automatic retry count.

Transmission Statistic Functions

- synchronized long **getTotalCounter** ()
  Returns how often a transmit/receive cycle has been executed.
- synchronized void **resetTotalCounter** ()
  Resets total transmit/receive cycle counter.
- synchronized long **getSuccessCounter** ()
  Returns how often a transmit/receive cycle was successful.
- synchronized void **resetSuccessCounter** ()
  Resets successful transmit/receive counter.

Slave Configuration

- void **configureStandard32BitMode** ()
  Configures all slaves for Standard 32-bit Mode.
- synchronized void **configureStandard32BitMode** (int slaveAddr)
  Configures a slave for Standard 32-bit Register Mode.
- void **configureSingleReg32BitMode** ()
  Configures all slaves for Single Register 32-bit Mode.
- synchronized void **configureSingleReg32BitMode** (int slaveAddr)
  Configures all slaves for Single Register 32-bit Mode.
- void **configureCountFromOne** ()
  Configures the reference counting scheme to start with one for all slaves.
- synchronized void **configureCountFromOne** (int slaveAddr)
  Configures a slave's reference counting scheme to start with one.
- void **configureCountFromZero** ()
  Configures the reference counting scheme to start with zero for all slaves.
- synchronized void **configureCountFromZero** (int slaveAddr)
  Configures a slave's reference counting scheme to start with zero.
- void **configureLittleEndianInts** ()
  Disables word swapping for int data type functions for all slaves.
- synchronized void **configureLittleEndianInts** (int slaveAddr)
  Disables word swapping for int data type functions on a per slave basis.

- void **configureBigEndianInts** ()

  Configures int data type functions to do a word swap for all slaves.

- synchronized void **configureBigEndianInts** (int slaveAddr)

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

- void **configureLittleEndianFloats** ()

  Disables float data type functions to do a word swap for all slaves.

- synchronized void **configureLittleEndianFloats** (int slaveAddr)

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

- void **configureSwappedFloats** ()

  Configures float data type functions to do a word swap for all slaves.

- synchronized void **configureSwappedFloats** (int slaveAddr)

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

6.9.1 Detailed Description

Extended Lufkin Automation Modbus Master Protocol. This class realizes the Extended Lufkin Automation (ELAM) Modbus protocol. This proprietary Modbus extension allows adressing of up to 2295 slave units and the retrieval of up to 2500 registers for Modbus functions 3 and 4.

It's implementation is based on the specification "ELAM Extended Lufkin Automation Modbus Version 1.01" published by LUFKIN Automation. The ELAM multiple instruction requests extensions are not implemented.

Tests showed the following size limits with a LUFKIN SAM Well Manager device:

Coils: 1992 for read Registers: 2500 to read, 60 for write

See also:

Data and Control Functions for all Modbus Protocol Flavours (p. 14), **Serial Protocols** (p. 15)
MbusRtuMasterProtocol (p. 105), **MbusAsciiMasterProtocol** (p. 32), **MbusTcpMasterProtocol** (p. 153)

6.9.2 Member Function Documentation

synchronized void openProtocol (String *portName*, **int** *baudRate*, **int** *dataBits*, **int** *stopBits*, **int** *parity* **) throws IOException, PortInUseException, UnsupportedCommOperationException** `[inherited]`

Opens a Modbus serial port with specific port parameters.

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

Note:

The default time-out for the data transfer is 1000 ms.
The default poll delay is 0 ms.
Automatic retries are switched off (retry count is 0).
The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.

Parameters:

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

baudRate The port baudRate in bps (typically 1200 - 9600).

dataBits Must be DATABITS_8 for RTU

stopBits STOPBITS_1: 1 stop bit, STOPBITS_1_5: 1.5 stop bits, STOPBITS_2: 2 stop bits

parity PARITY_NONE: no parity, PARITY_ODD: odd parity, PARITY_EVEN: even parity, PARITY_MARK: mark parity, PARITY_SPACE: space parity

Exceptions:

IOException An I/O error occured

UnsupportedCommOperationException A communication parameter is not supported

PortInUseException Port is already used by somebody else

IllegalArgumentException A parameter is invalid

Reimplemented from **MbusSerialMasterProtocol** (p. 134).

synchronized void closeProtocol ( ) throws IOException `[virtual, inherited]`

Closes the serial port and releases any system resources associated with the port.

Exceptions:

IOException An I/O error occured

Implements **MbusMasterFunctions** (p. 82).

boolean isOpen ( ) `[virtual, inherited]`

Returns whether the port is open or not.

Return values:

true = open

false = closed

Implements **MbusMasterFunctions** (p. 103).

**synchronized void readCoils (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException `[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: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. *20)* A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

**synchronized void readInputDiscretes (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeCoil (int *slaveAddr*, int *bitAddr*, boolean *bitVal*) throws IOException, BusProtocolException `[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 - 247)

bitAddr Coil address (Range: depends on configuration setting)

bitVal true sets, false clears discrete output variable

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void forceMultipleCoils (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException `[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: depends on configuration setting)

bitArr Buffer which contains the data to be sent. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException `[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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int ***slaveAddr,*** **int** ***startRef,*** **float[]** ***float32Arr*** **) throws IOException, BusProtocolException** `[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 two consecutive output registers (holding registers, 4:00000 table) into float values.

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException* (p. *20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* short[ ] *regArr*) throws IOException, BusProtocolException `[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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException* (p. *20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

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

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr*, int *startRef*, float[ ] *float32Arr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeSingleRegister (int *slaveAddr*, int *regAddr*, short *regVal*) throws IOException, BusProtocolException `[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 - 247)
regAddr Register address (Range: depends on configuration setting)
regVal Data to be sent

Exceptions:

IllegalStateException Port or connection is closed
IOException An I/O error occured
IllegalArgumentException A parameter is out of range
BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException `[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 - 247)
startRef Start reference (Range depends on configuration setting)
regArr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IllegalStateException Port or connection is closed
IOException An I/O error occured
IllegalArgumentException A parameter is out of range
BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, int[] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

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

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

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, float[] *float32Arr*) throws IOException, BusProtocolException `[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 a pairs of output registers (holding registers, 4:00000 table).

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. *20)* A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void maskWriteRegister (int *slaveAddr,* **int** *regAddr,* **short** *andMask,* **short** *orMask* **) throws IOException, BusProtocolException** `[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: depends on configuration setting)

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

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. *20)* A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readWriteRegisters (int *slaveAddr*, int *readRef*, short[ ] *readArr*, int *writeRef*, short[ ] *writeArr*) throws IOException, BusProtocolException [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 reference for reading (Range: depends on configuration setting)

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

writeRef Start reference for writing (Range: depends on configuration setting)

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized byte readExceptionStatus (int *slaveAddr*) throws IOException, BusProtocolException [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)

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException* (p. *20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void setTimeout (int *timeOut*) [inherited]

Configures time-out.

This function sets the time-out to the specified value. A value of 0 disables the time-out, which causes all subsequent calls to data and control functions to block.

Note:

A port or connection must be closed in order to configure it.

Parameters:

timeOut Timeout value in ms (Range: 0 - 100000), 0 disables time-out

Exceptions:

IllegalStateException Port is open

IllegalArgumentException Parameter is out of range

int getTimeout ( ) [inherited]

Returns the operation time-out value.

Returns:

Timeout value in ms

synchronized void setPollDelay (int *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.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open
IllegalArgumentException Parameter is out of range

int getPollDelay ( ) [inherited]

Returns the poll delay time.

Returns:

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

synchronized void setRetryCnt (int *retryCnt*) [inherited]

Configures the automatic retry setting.
A value of 0 disables any automatic retries.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open
IllegalArgumentException Parameter is out of range

int getRetryCnt ( ) [inherited]

Returns the automatic retry count.

Returns:

Retry count

synchronized long getTotalCounter ( ) [inherited]

Returns how often a transmit/receive cycle has been executed.

Returns:

Counter value

synchronized long getSuccessCounter ( ) [inherited]

Returns how often a transmit/receive cycle was successful.

Returns:

Counter value

void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

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

Remarks:

This is the default mode

synchronized void configureStandard32BitMode (int *slaveAddr*) [inherited]

Configures a slave for Standard 32-bit Register Mode.

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

Parameters:

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

Remarks:

This is the default mode

void configureSingleReg32BitMode ( ) [inherited]

Configures all slaves for Single Register 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.

synchronized void configureSingleReg32BitMode (int *slaveAddr*) [inherited]

Configures all slaves for Single Register 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.

void configureCountFromOne ( ) [inherited]

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

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

Remarks:

This is the default mode

synchronized void configureCountFromOne (int *slaveAddr*) [inherited]

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

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

Parameters:

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

Remarks:

This is the default mode

void configureCountFromZero ( ) [inherited]

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

This is also known as PDU addressing.

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

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

synchronized void configureCountFromZero (int *slaveAddr*) [inherited]

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

This is also known as PDU addressing.

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

Parameters:

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

void configureLittleEndianInts ( ) [inherited]

Disables word swapping for int data type functions for all slaves.

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

Remarks:

This is the default mode

synchronized void configureLittleEndianInts (int *slaveAddr*) [inherited]

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

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

Remarks:

This is the default mode

Parameters:

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

void configureBigEndianInts ( ) [inherited]

Configures int data type functions to do a word swap for all slaves.

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.

synchronized void configureBigEndianInts (int *slaveAddr*) [inherited]

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

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

Parameters:

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

void configureLittleEndianFloats ( ) [inherited]

Disables float data type functions to do a word swap for all slaves.

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

Remarks:

This is the default mode

synchronized void configureLittleEndianFloats (int *slaveAddr*) [inherited]

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

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

Remarks:

This is the default mode

Parameters:

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

void configureSwappedFloats ( ) [inherited]

Configures float data type functions to do a word swap for all slaves.

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.

synchronized void configureSwappedFloats (int *slaveAddr*) [inherited]

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

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

Note:

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

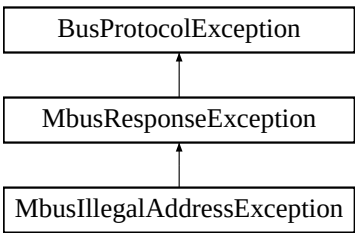
Parameters:

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

6.10 MbusIllegalAddressException Class Reference

Illegal Data Address exception response.

Inheritance diagram for MbusIllegalAddressException:



Public Member Functions

- **MbusIllegalAddressException** ()

  *Creates a new **MbusIllegalAddressException** (p. 78) instance.*

Static Public Member Functions

- static long **getCounter** ()

  Returns how often this exception has been triggered.

- static void **resetCounter** ()

  Resets exception counter.

6.10.1 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:

MbusResponseException (p. 103)

6.10.2 Member Function Documentation

static long getCounter ( ) `[static]`

Returns how often this exception has been triggered.

Returns:

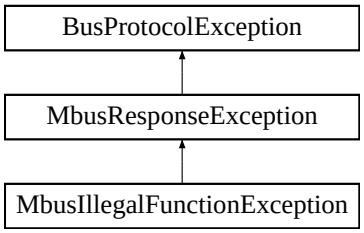
Counter value

Reimplemented from **MbusResponseException** (p. 104).

6.11 MbusIllegalFunctionException Class Reference

Illegal Function exception response.

Inheritance diagram for MbusIllegalFunctionException:



Public Member Functions

- **MbusIllegalFunctionException** ()

  *Creates a new **MbusIllegalFunctionException** (p. 79) instance.*

Static Public Member Functions

- static long **getCounter** ()

  Returns how often this exception has been triggered.

- static void **resetCounter** ()

  Resets exception counter.

6.11.1 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:

MbusResponseException (p. 103)

6.11.2 Member Function Documentation

static long getCounter ( ) `[static]`

Returns how often this exception has been triggered.

Returns:

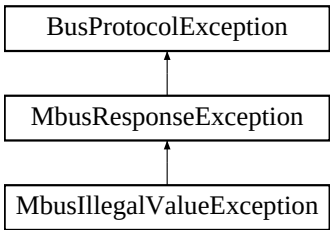
Counter value

Reimplemented from **MbusResponseException** (p. 104).

6.12 MbusIllegalValueException Class Reference

Illegal Data Value exception response.

Inheritance diagram for MbusIllegalValueException:



Public Member Functions

- **MbusIllegalValueException** ()

  *Creates a new **MbusIllegalValueException** (p. 81) instance.*

Static Public Member Functions

- static long **getCounter** ()

  Returns how often this exception has been triggered.

- static void **resetCounter** ()

  Resets exception counter.

6.12.1 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:

MbusResponseException (p. 103)

6.12.2 Member Function Documentation

static long getCounter ( ) `[static]`

Returns how often this exception has been triggered.

Returns:

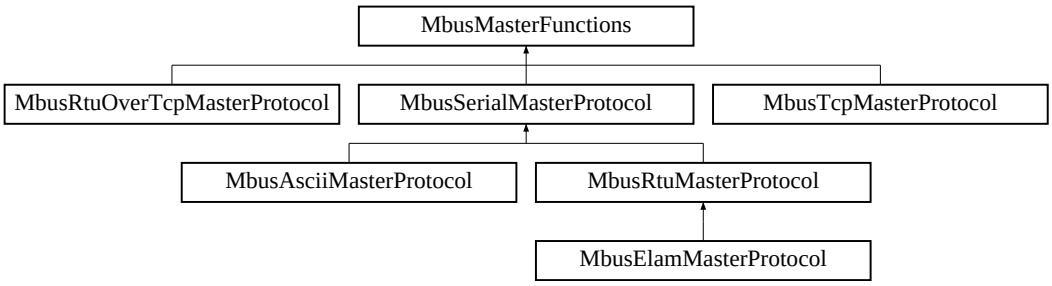
Counter value

Reimplemented from **MbusResponseException** (p. 104).

6.13 MbusMasterFunctions Class Reference

Base class which implements Modbus data and control functions.

Inheritance diagram for MbusMasterFunctions:



Public Member Functions

- **MbusMasterFunctions** ()

  Creates new instance.

- abstract boolean **isOpen** ()

  Returns whether the protocol has been opened or not (e.g.

- abstract void **closeProtocol** () throws IOException

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

Bit Access

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

- synchronized void **readCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 1 (01 hex), Read Coil Status/Read Coils.

- synchronized void **readInputDiscretes** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

- synchronized void **writeCoil** (int slaveAddr, int bitAddr, boolean bitVal) throws IOException, BusProtocolException

Modbus function 5 (05 hex), Force Single Coil/Write Coil.

- synchronized void **forceMultipleCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

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

Register Access (16-bit, 32-bit and floating point)

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

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex), Read Input Registers.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

- synchronized void **writeSingleRegister** (int slaveAddr, int regAddr, short regVal) throws IOException, BusProtocolException

  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

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

- synchronized void **maskWriteRegister** (int slaveAddr, int regAddr, short andMask, short orMask) throws IOException, BusProtocolException

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

- synchronized void **readWriteRegisters** (int slaveAddr, int readRef, short[ ] readArr, int writeRef, short[ ] writeArr) throws IOException, BusProtocolException

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

Diagnostics

- synchronized byte **readExceptionStatus** (int slaveAddr) throws IOException, BusProtocolException

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

Custom Function Codes

- synchronized void **readHistoryLog** (int slaveAddr, int channelNo, int resolution, short readArr[ ]) throws IOException, BusProtocolException

  Vendor Specific Modbus function 100 (64 hex), Read History Log.

Protocol Configuration

- synchronized void **setTimeout** (int timeOut)

  Configures time-out.

- int **getTimeout** ()

  Returns the operation time-out value.

- synchronized void **setPollDelay** (int pollDelay)

  Configures poll delay.

- int **getPollDelay** ()

  Returns the poll delay time.

- synchronized void **setRetryCnt** (int retryCnt)

Configures the automatic retry setting.

- int **getRetryCnt** ()
  Returns the automatic retry count.

Transmission Statistic Functions

- synchronized long **getTotalCounter** ()
  Returns how often a transmit/receive cycle has been executed.
- synchronized void **resetTotalCounter** ()
  Resets total transmit/receive cycle counter.
- synchronized long **getSuccessCounter** ()
  Returns how often a transmit/receive cycle was successful.
- synchronized void **resetSuccessCounter** ()
  Resets successful transmit/receive counter.

Slave Configuration

- void **configureStandard32BitMode** ()
  Configures all slaves for Standard 32-bit Mode.
- void **configureSingleReg32BitMode** ()
  Configures all slaves for Single Register 32-bit Mode.
- void **configureCountFromOne** ()
  Configures the reference counting scheme to start with one for all slaves.
- void **configureCountFromZero** ()
  Configures the reference counting scheme to start with zero for all slaves.
- void **configureLittleEndianInts** ()
  Disables word swapping for int data type functions for all slaves.
- void **configureBigEndianInts** ()
  Configures int data type functions to do a word swap for all slaves.
- void **configureLittleEndianFloats** ()
  Disables float data type functions to do a word swap for all slaves.
- void **configureSwappedFloats** ()
  Configures float data type functions to do a word swap for all slaves.

- synchronized void **configureStandard32BitMode** (int slaveAddr)
  Configures a slave for Standard 32-bit Register Mode.
- synchronized void **configureSingleReg32BitMode** (int slaveAddr)
  Configures all slaves for Single Register 32-bit Mode.
- synchronized void **configureCountFromOne** (int slaveAddr)
  Configures a slave's reference counting scheme to start with one.
- synchronized void **configureCountFromZero** (int slaveAddr)
  Configures a slave's reference counting scheme to start with zero.
- synchronized void **configureLittleEndianInts** (int slaveAddr)
  Disables word swapping for int data type functions on a per slave basis.
- synchronized void **configureBigEndianInts** (int slaveAddr)
  Enables int data type functions to do a word swap on a per slave basis.
- synchronized void **configureLittleEndianFloats** (int slaveAddr)
  Disables float data type functions to do a word swap on a per slave basis.
- synchronized void **configureSwappedFloats** (int slaveAddr)
  Enables float data type functions to do a word swap on a per slave basis.

6.13.1 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).

All functions have been implemented thread-safe.

See also:

Data and Control Functions for all Modbus Protocol Flavours (p. 14)
MbusRtuMasterProtocol (p. 105), **MbusElamMasterProtocol** (p. 55), **MbusAsciiMasterProtocol** (p. 32), **MbusTcpMasterProtocol** (p. 153)

6.13.2 Member Function Documentation

synchronized void readCoils (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException

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: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputDiscretes (int *slaveAddr,* int *startRef,* boolean[ ] *bitArr*) throws IOException, BusProtocolException

Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeCoil (int *slaveAddr,* int *bitAddr,* boolean *bitVal*) throws IOException, BusProtocolException

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 - 247)

bitAddr Coil address (Range: depends on configuration setting)

bitVal true sets, false clears discrete output variable

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void forceMultipleCoils (int *slaveAddr,* int *startRef,* boolean[ ] *bitArr*) throws IOException, BusProtocolException

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: depends on configuration setting)

bitArr Buffer which contains the data to be sent. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException

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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, int[ ] *int32Arr*) throws IOException, BusProtocolException

Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. 20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int *slaveAddr,* int *startRef,* float[ ] *float32Arr*) throws IOException, BusProtocolException

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 two consecutive output registers (holding registers, 4:00000 table) into float values.

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. 20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* short[ ] *regArr*) throws IOException, BusProtocolException

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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* int[ ] *int32Arr*) throws IOException, BusProtocolException

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

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr*, int *startRef*, float[ ] *float32Arr*) throws IOException, BusProtocolException

Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeSingleRegister (int *slaveAddr*, int *regAddr*, short *regVal*) throws IOException, BusProtocolException

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 - 247)

regAddr Register address (Range: depends on configuration setting)

regVal Data to be sent

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException

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 - 247)

startRef Start reference (Range depends on configuration setting)

regArr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, int[ ] *int32Arr*) throws IOException, BusProtocolException

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

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

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr,* int *startRef,* float[ ] *float32Arr*) throws IOException, BusProtocolException

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

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

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void maskWriteRegister (int *slaveAddr*, int *regAddr*, short *andMask*, short *orMask*) throws IOException, BusProtocolException

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: depends on configuration setting)

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

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readWriteRegisters (int *slaveAddr*, int *readRef*, short[ ] *readArr*, int *writeRef*, short[ ] *writeArr*) throws IOException, BusProtocolException

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 reference for reading (Range: depends on configuration setting)

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

writeRef Start reference for writing (Range: depends on configuration setting)

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20***) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized byte readExceptionStatus (int *slaveAddr*) throws IOException, BusProtocolException

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)

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20***) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void setTimeout (int *timeOut*)

Configures time-out.

This function sets the time-out to the specified value. A value of 0 disables the time-out, which causes all subsequent calls to data and control functions to block.

Note:

A port or connection must be closed in order to configure it.

Parameters:

timeOut Timeout value in ms (Range: 0 - 100000), 0 disables time-out

Exceptions:

IllegalStateException Port is open
IllegalArgumentException Parameter is out of range

int getTimeout ( )

Returns the operation time-out value.

Returns:

Timeout value in ms

synchronized void setPollDelay (int *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.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open
IllegalArgumentException Parameter is out of range

int getPollDelay ( )

Returns the poll delay time.

Returns:

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

synchronized void setRetryCnt (int *retryCnt*)

Configures the automatic retry setting.
A value of 0 disables any automatic retries.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open
IllegalArgumentException Parameter is out of range

int getRetryCnt ( )

Returns the automatic retry count.

Returns:

Retry count

synchronized long getTotalCounter ( )

Returns how often a transmit/receive cycle has been executed.

Returns:

Counter value

synchronized long getSuccessCounter ( )

Returns how often a transmit/receive cycle was successful.

Returns:

Counter value

void configureStandard32BitMode ( )

Configures all slaves for Standard 32-bit Mode.

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

Remarks:

This is the default mode

void configureSingleReg32BitMode ( )

Configures all slaves for Single Register 32-bit Mode.

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

void configureCountFromOne ( )

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

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

Remarks:

This is the default mode

void configureCountFromZero ( )

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

This is also known as PDU addressing.

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

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

void configureLittleEndianInts ( )

Disables word swapping for int data type functions for all slaves.

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

Remarks:

This is the default mode

void configureBigEndianInts ( )

Configures int data type functions to do a word swap for all slaves.

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.

void configureLittleEndianFloats ( )

Disables float data type functions to do a word swap for all slaves.

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

Remarks:

This is the default mode

void configureSwappedFloats ( )

Configures float data type functions to do a word swap for all slaves.

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.

synchronized void configureStandard32BitMode (int *slaveAddr*)

Configures a slave for Standard 32-bit Register Mode.

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

Parameters:

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

Remarks:

This is the default mode

synchronized void configureSingleReg32BitMode (int *slaveAddr*)

Configures all slaves for Single Register 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.

synchronized void configureCountFromOne (int *slaveAddr*)

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

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

Parameters:

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

Remarks:

This is the default mode

synchronized void configureCountFromZero (int *slaveAddr*)

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

This is also known as PDU addressing.

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

Parameters:

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

synchronized void configureLittleEndianInts (int *slaveAddr*)

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

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

Remarks:

This is the default mode

Parameters:

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

synchronized void configureBigEndianInts (int *slaveAddr*)

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

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

Parameters:

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

synchronized void configureLittleEndianFloats (int *slaveAddr*)

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

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

Remarks:

This is the default mode

Parameters:

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

synchronized void configureSwappedFloats (int *slaveAddr*)

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

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

Note:

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

Parameters:

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

abstract boolean isOpen ( ) `[pure virtual]`

Returns whether the protocol has been opened or not (e.g.

the port is opened or connected)

Return values:

true = connected

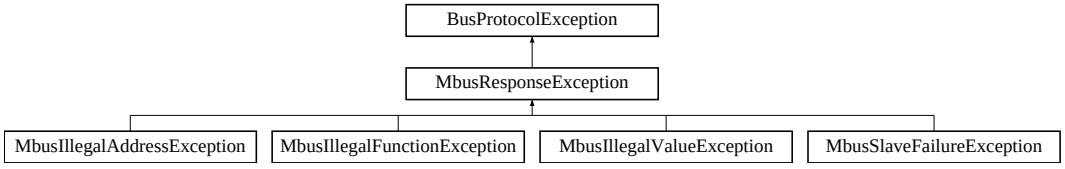
false = not connected

Implemented in **MbusSerialMasterProtocol** (p. 135), **MbusTcpMasterProtocol** (p. 159), and **MbusRtuOverTcpMasterProtocol** (p. **??**).

6.14 MbusResponseException Class Reference

Modbus exception response.

Inheritance diagram for MbusResponseException:



Public Member Functions

- **MbusResponseException** (byte id)

*Creates a new **MbusResponseException** (p. 103) instance.*

- **MbusResponseException** ()

  Creates a new ResponseException instance.

Static Public Member Functions

- static long **getCounter** ()

  Returns how often this exception has been triggered.

- static void **resetCounter** ()

  Resets exception counter.

6.14.1 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:

BusProtocolException (p. 20)

6.14.2 Constructor & Destructor Documentation

MbusResponseException (byte *id*)

Creates a new **MbusResponseException** (p. 103) instance.

Parameters:

id Exception code as reported by slave device.

6.14.3 Member Function Documentation

static long getCounter ( ) `[static]`

Returns how often this exception has been triggered.

Returns:

Counter value

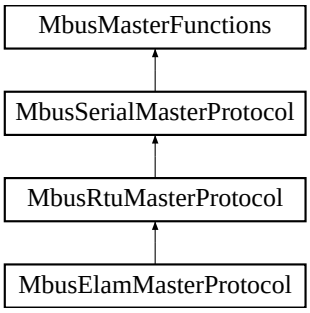
Reimplemented from **BusProtocolException** (p. 21).

Reimplemented in **MbusIllegalAddressException** (p. 79), **MbusIllegalFunctionException** (p. 80), **MbusIllegalValueException** (p. 81), and **MbusSlaveFailureException** (p. 152).

6.15 MbusRtuMasterProtocol Class Reference

Modbus RTU Master Protocol class.

Inheritance diagram for MbusRtuMasterProtocol:



Public Member Functions

- **MbusRtuMasterProtocol** ()

  Creates new instance.

- synchronized void **openProtocol** (String portName, int baudRate, int dataBits, int stopBits, int parity) throws IOException, PortInUseException, UnsupportedCommOperationException

  Opens a Modbus serial port with specific port parameters.

- synchronized void **closeProtocol** () throws IOException

  Closes the serial port and releases any system resources associated with the port.

- boolean **isOpen** ()

  Returns whether the port is open or not.

Static Public Attributes

- static final int **DATABITS_8** = SerialPort.DATABITS_8

  8 data bits

- static final int **STOPBITS_1** = SerialPort.STOPBITS_1

  1 stop bit

- static final int **STOPBITS_1_5** = SerialPort.STOPBITS_1_5

  1.5 stop bits

- static final int **STOPBITS_2** = SerialPort.STOPBITS_2

  2 stop bits

- static final int **PARITY_NONE** = SerialPort.PARITY_NONE

  no parity

- static final int **PARITY_ODD** = SerialPort.PARITY_ODD

  odd parity

- static final int **PARITY_EVEN** = SerialPort.PARITY_EVEN

  even parity

- static final int **PARITY_MARK** = SerialPort.PARITY_MARK

  mark parity

- static final int **PARITY_SPACE** = SerialPort.PARITY_SPACE

  space parity

Bit Access

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

- synchronized void **readCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 1 (01 hex), Read Coil Status/Read Coils.

- synchronized void **readInputDiscretes** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

- synchronized void **writeCoil** (int slaveAddr, int bitAddr, boolean bitVal) throws IOException, BusProtocolException

  Modbus function 5 (05 hex), Force Single Coil/Write Coil.

- synchronized void **forceMultipleCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

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

Register Access (16-bit, 32-bit and floating point)

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

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex), Read Input Registers.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

- synchronized void **writeSingleRegister** (int slaveAddr, int regAddr, short regVal) throws IOException, BusProtocolException

  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

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

- synchronized void **maskWriteRegister** (int slaveAddr, int regAddr, short andMask, short orMask) throws IOException, BusProtocolException

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

- synchronized void **readWriteRegisters** (int slaveAddr, int readRef, short[ ] readArr, int writeRef, short[ ] writeArr) throws IOException, BusProtocolException

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

Diagnostics

- synchronized byte **readExceptionStatus** (int slaveAddr) throws IOException, BusProtocolException

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

Custom Function Codes

- synchronized void **readHistoryLog** (int slaveAddr, int channelNo, int resolution, short readArr[ ]) throws IOException, BusProtocolException

  Vendor Specific Modbus function 100 (64 hex), Read History Log.

Protocol Configuration

- synchronized void **setTimeout** (int timeOut)

  Configures time-out.

- int **getTimeout** ()

  Returns the operation time-out value.

- synchronized void **setPollDelay** (int pollDelay)

  Configures poll delay.

- int **getPollDelay** ()

  Returns the poll delay time.

- synchronized void **setRetryCnt** (int retryCnt)

  Configures the automatic retry setting.

- int **getRetryCnt** ()

  Returns the automatic retry count.

Transmission Statistic Functions

- synchronized long **getTotalCounter** ()

  Returns how often a transmit/receive cycle has been executed.

- synchronized void **resetTotalCounter** ()
  Resets total transmit/receive cycle counter.
- synchronized long **getSuccessCounter** ()
  Returns how often a transmit/receive cycle was successful.
- synchronized void **resetSuccessCounter** ()
  Resets successful transmit/receive counter.

Slave Configuration

- void **configureStandard32BitMode** ()
  Configures all slaves for Standard 32-bit Mode.
- synchronized void **configureStandard32BitMode** (int slaveAddr)
  Configures a slave for Standard 32-bit Register Mode.
- void **configureSingleReg32BitMode** ()
  Configures all slaves for Single Register 32-bit Mode.
- synchronized void **configureSingleReg32BitMode** (int slaveAddr)
  Configures all slaves for Single Register 32-bit Mode.
- void **configureCountFromOne** ()
  Configures the reference counting scheme to start with one for all slaves.
- synchronized void **configureCountFromOne** (int slaveAddr)
  Configures a slave's reference counting scheme to start with one.
- void **configureCountFromZero** ()
  Configures the reference counting scheme to start with zero for all slaves.
- synchronized void **configureCountFromZero** (int slaveAddr)
  Configures a slave's reference counting scheme to start with zero.
- void **configureLittleEndianInts** ()
  Disables word swapping for int data type functions for all slaves.
- synchronized void **configureLittleEndianInts** (int slaveAddr)
  Disables word swapping for int data type functions on a per slave basis.
- void **configureBigEndianInts** ()
  Configures int data type functions to do a word swap for all slaves.
- synchronized void **configureBigEndianInts** (int slaveAddr)
  Enables int data type functions to do a word swap on a per slave basis.

- void **configureLittleEndianFloats** ()

  Disables float data type functions to do a word swap for all slaves.

- synchronized void **configureLittleEndianFloats** (int slaveAddr)

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

- void **configureSwappedFloats** ()

  Configures float data type functions to do a word swap for all slaves.

- synchronized void **configureSwappedFloats** (int slaveAddr)

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

6.15.1 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).

All functions have been implemented thread-safe. It is also possible to instantiate multiple instances for establishing multiple connections on different serial ports.

Remarks:

The frame delimition of the RTU protocol leads to some practical limitations with Java: The RTU protocol is using silence periods in the data stream as start and end of frame detection. The specification determines a silence period of 3.5 character transmission times as an end of frame signal. If the combination of Java thread scheduler, the Communications API and the underlaying operating system are not able to transmit a message without a silence period, a sensitive slave will interpret the received message as garbage and not reply. The silence period for a baudrate of 115200 bps is 330 us, which is very difficult to obey for a non-real-time platform. In practical situations the RTU protocol should not be used at all in conjunction with high bit rates (e.g. above 9600 bps) and large message sizes (e.g. $>$ 20 registers). Tests have shown that RTU is still usable with 115200 bps and message sizes below 20 registers as well as 9600 bps and message sizes of 125 registers on a Win32 platform. The behaviour depends very much on computer speed, operating system, Java virtual machine and Communication API. Therefore the usability of the RTU protocol has to be decided on a case to case basis. The ASCII protocol does not have this limitation and is therefore recommended to be used instead.

See also:

Data and Control Functions for all Modbus Protocol Flavours (p. 14), **Serial Protocols** (p. 15)

MbusElamMasterProtocol (p. 55), **MbusAsciiMasterProtocol** (p. 32), **MbusTcpMasterProtocol** (p. 153)

6.15.2 Member Function Documentation

synchronized void openProtocol (String *portName*, int *baudRate*, int *dataBits*, int *stopBits*, int *parity*) throws IOException, PortInUseException, UnsupportedCommOperationException

Opens a Modbus serial port with specific port parameters.

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

Note:

The default time-out for the data transfer is 1000 ms.
The default poll delay is 0 ms.
Automatic retries are switched off (retry count is 0).
The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.

Parameters:

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

baudRate The port baudRate in bps (typically 1200 - 9600).

dataBits Must be DATABITS_8 for RTU

stopBits STOPBITS_1: 1 stop bit, STOPBITS_1_5: 1.5 stop bits, STOPBITS_2: 2 stop bits

parity PARITY_NONE: no parity, PARITY_ODD: odd parity, PARITY_EVEN: even parity, PARITY_MARK: mark parity, PARITY_SPACE: space parity

Exceptions:

IOException An I/O error occured

UnsupportedCommOperationException A communication parameter is not supported

PortInUseException Port is already used by somebody else

IllegalArgumentException A parameter is invalid

Reimplemented from **MbusSerialMasterProtocol** (p. 134).

synchronized void closeProtocol ( ) throws IOException `[virtual, inherited]`

Closes the serial port and releases any system resources associated with the port.

Exceptions:

IOException An I/O error occured

Implements **MbusMasterFunctions** (p. 82).

boolean isOpen ( ) [virtual, inherited]

Returns whether the port is open or not.

Return values:

true = open

false = closed

Implements **MbusMasterFunctions** (p. 103).

synchronized void readCoils (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException [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: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20***) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputDiscretes (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

***BusProtocolException* (p. *20*)** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeCoil (int *slaveAddr*, int *bitAddr*, boolean *bitVal*) throws IOException, BusProtocolException `[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 - 247)

bitAddr Coil address (Range: depends on configuration setting)

bitVal true sets, false clears discrete output variable

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

***BusProtocolException* (p. *20*)** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void forceMultipleCoils (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException `[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: depends on configuration setting)

bitArr Buffer which contains the data to be sent. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

***BusProtocolException* (p. *20*)** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException `[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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

***BusProtocolException* (p. *20*)** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, float[ ] *float32Arr*) throws IOException, BusProtocolException `[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 two consecutive output registers (holding registers, 4:00000 table) into float values.

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. **20)** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* short[ ] *regArr*) throws IOException, BusProtocolException `[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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. **20)** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

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

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr*, int *startRef*, float[ ] *float32Arr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeSingleRegister (int *slaveAddr*, int *regAddr*, short *regVal*) throws IOException, BusProtocolException `[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 - 247)
regAddr Register address (Range: depends on configuration setting)
regVal Data to be sent

Exceptions:

IllegalStateException Port or connection is closed
IOException An I/O error occured
IllegalArgumentException A parameter is out of range
BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException `[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 - 247)
startRef Start reference (Range depends on configuration setting)
regArr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IllegalStateException Port or connection is closed
IOException An I/O error occured
IllegalArgumentException A parameter is out of range
BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, int[] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

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

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

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, float[] *float32Arr*) throws IOException, BusProtocolException `[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 a pairs of output registers (holding registers, 4:00000 table).

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. *20)* A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void maskWriteRegister (int *slaveAddr,* **int** *regAddr,* **short** *andMask,* **short** *orMask* **) throws IOException, BusProtocolException** `[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: depends on configuration setting)

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

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. *20)* A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readWriteRegisters (int *slaveAddr*, int *readRef*, short[ ] *readArr*, int *writeRef*, short[ ] *writeArr*) throws IOException, BusProtocolException [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 reference for reading (Range: depends on configuration setting)

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

writeRef Start reference for writing (Range: depends on configuration setting)

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized byte readExceptionStatus (int *slaveAddr*) throws IOException, BusProtocolException [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)

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void setTimeout (int *timeOut*) [inherited]

Configures time-out.

This function sets the time-out to the specified value. A value of 0 disables the time-out, which causes all subsequent calls to data and control functions to block.

Note:

A port or connection must be closed in order to configure it.

Parameters:

timeOut Timeout value in ms (Range: 0 - 100000), 0 disables time-out

Exceptions:

IllegalStateException Port is open

IllegalArgumentException Parameter is out of range

int getTimeout ( ) [inherited]

Returns the operation time-out value.

Returns:

Timeout value in ms

synchronized void setPollDelay (int *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.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open
IllegalArgumentException Parameter is out of range

int getPollDelay ( ) [inherited]

Returns the poll delay time.

Returns:

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

synchronized void setRetryCnt (int *retryCnt*) [inherited]

Configures the automatic retry setting.
A value of 0 disables any automatic retries.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open
IllegalArgumentException Parameter is out of range

int getRetryCnt ( ) [inherited]

Returns the automatic retry count.

Returns:

Retry count

synchronized long getTotalCounter ( ) [inherited]

Returns how often a transmit/receive cycle has been executed.

Returns:

Counter value

synchronized long getSuccessCounter ( ) [inherited]

Returns how often a transmit/receive cycle was successful.

Returns:

Counter value

void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

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

Remarks:

This is the default mode

synchronized void configureStandard32BitMode (int *slaveAddr*) [inherited]

Configures a slave for Standard 32-bit Register Mode.

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

Parameters:

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

Remarks:

This is the default mode

void configureSingleReg32BitMode ( ) [inherited]

Configures all slaves for Single Register 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.

synchronized void configureSingleReg32BitMode (int *slaveAddr*) [inherited]

Configures all slaves for Single Register 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.

void configureCountFromOne ( ) [inherited]

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

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

Remarks:

This is the default mode

synchronized void configureCountFromOne (int *slaveAddr*) [inherited]

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

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

Parameters:

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

Remarks:

This is the default mode

void configureCountFromZero ( ) [inherited]

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

This is also known as PDU addressing.

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

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

synchronized void configureCountFromZero (int *slaveAddr*) [inherited]

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

This is also known as PDU addressing.

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

Parameters:

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

void configureLittleEndianInts ( ) [inherited]

Disables word swapping for int data type functions for all slaves.

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

Remarks:

This is the default mode

synchronized void configureLittleEndianInts (int *slaveAddr*) [inherited]

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

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

Remarks:

This is the default mode

Parameters:

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

void configureBigEndianInts ( ) [inherited]

Configures int data type functions to do a word swap for all slaves.

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.

synchronized void configureBigEndianInts (int *slaveAddr*) [inherited]

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

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

Parameters:

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

void configureLittleEndianFloats ( ) [inherited]

Disables float data type functions to do a word swap for all slaves.

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

Remarks:

This is the default mode

synchronized void configureLittleEndianFloats (int *slaveAddr*) [inherited]

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

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

Remarks:

This is the default mode

Parameters:

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

void configureSwappedFloats ( ) [inherited]

Configures float data type functions to do a word swap for all slaves.

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.

synchronized void configureSwappedFloats (int *slaveAddr*) [inherited]

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

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

Note:

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

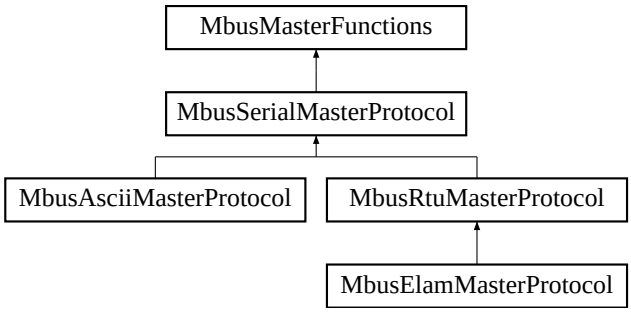
Parameters:

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

6.16 MbusSerialMasterProtocol Class Reference

Base class for serial serial master protocols.

Inheritance diagram for MbusSerialMasterProtocol:



Public Member Functions

- **MbusSerialMasterProtocol** ()

  Creates new instance.

- synchronized void **openProtocol** (String portName, int baudRate, int dataBits, int stopBits, int parity) throws IOException, UnsupportedCommOperationException, PortInUseException

  Opens a Modbus serial port with specific port parameters.

- synchronized void **closeProtocol** () throws IOException

  Closes the serial port and releases any system resources associated with the port.

- boolean **isOpen** ()

  Returns whether the port is open or not.

Static Public Attributes

- static final int **DATABITS_7** = SerialPort.DATABITS_7

  7 data bits

- static final int **DATABITS_8** = SerialPort.DATABITS_8

  8 data bits

- static final int **STOPBITS_1** = SerialPort.STOPBITS_1

  1 stop bit

- static final int **STOPBITS_1_5** = SerialPort.STOPBITS_1_5

  1.5 stop bits

- static final int **STOPBITS_2** = SerialPort.STOPBITS_2

  2 stop bits

- static final int **PARITY_NONE** = SerialPort.PARITY_NONE

  no parity

- static final int **PARITY_ODD** = SerialPort.PARITY_ODD

  odd parity

- static final int **PARITY_EVEN** = SerialPort.PARITY_EVEN

  even parity

- static final int **PARITY_MARK** = SerialPort.PARITY_MARK

  mark parity

- static final int **PARITY_SPACE** = SerialPort.PARITY_SPACE

  space parity

Bit Access

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

- synchronized void **readCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 1 (01 hex), Read Coil Status/Read Coils.

- synchronized void **readInputDiscretes** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

- synchronized void **writeCoil** (int slaveAddr, int bitAddr, boolean bitVal) throws IOException, BusProtocolException

  Modbus function 5 (05 hex), Force Single Coil/Write Coil.

- synchronized void **forceMultipleCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

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

Register Access (16-bit, 32-bit and floating point)

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

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex), Read Input Registers.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

- synchronized void **writeSingleRegister** (int slaveAddr, int regAddr, short regVal) throws IOException, BusProtocolException

  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

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

- synchronized void **maskWriteRegister** (int slaveAddr, int regAddr, short andMask, short orMask) throws IOException, BusProtocolException

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

- synchronized void **readWriteRegisters** (int slaveAddr, int readRef, short[ ] readArr, int writeRef, short[ ] writeArr) throws IOException, BusProtocolException

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

Diagnostics

- synchronized byte **readExceptionStatus** (int slaveAddr) throws IOException, BusProtocolException

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

Custom Function Codes

- synchronized void **readHistoryLog** (int slaveAddr, int channelNo, int resolution, short readArr[ ]) throws IOException, BusProtocolException

  Vendor Specific Modbus function 100 (64 hex), Read History Log.

Protocol Configuration

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

Transmission Statistic Functions

- synchronized long **getTotalCounter** ()
  Returns how often a transmit/receive cycle has been executed.
- synchronized void **resetTotalCounter** ()
  Resets total transmit/receive cycle counter.
- synchronized long **getSuccessCounter** ()
  Returns how often a transmit/receive cycle was successful.
- synchronized void **resetSuccessCounter** ()
  Resets successful transmit/receive counter.

Slave Configuration

- void **configureStandard32BitMode** ()
  Configures all slaves for Standard 32-bit Mode.
- synchronized void **configureStandard32BitMode** (int slaveAddr)
  Configures a slave for Standard 32-bit Register Mode.
- void **configureSingleReg32BitMode** ()
  Configures all slaves for Single Register 32-bit Mode.

- synchronized void **configureSingleReg32BitMode** (int slaveAddr)
  Configures all slaves for Single Register 32-bit Mode.
- void **configureCountFromOne** ()
  Configures the reference counting scheme to start with one for all slaves.
- synchronized void **configureCountFromOne** (int slaveAddr)
  Configures a slave's reference counting scheme to start with one.
- void **configureCountFromZero** ()
  Configures the reference counting scheme to start with zero for all slaves.
- synchronized void **configureCountFromZero** (int slaveAddr)
  Configures a slave's reference counting scheme to start with zero.
- void **configureLittleEndianInts** ()
  Disables word swapping for int data type functions for all slaves.
- synchronized void **configureLittleEndianInts** (int slaveAddr)
  Disables word swapping for int data type functions on a per slave basis.
- void **configureBigEndianInts** ()
  Configures int data type functions to do a word swap for all slaves.
- synchronized void **configureBigEndianInts** (int slaveAddr)
  Enables int data type functions to do a word swap on a per slave basis.
- void **configureLittleEndianFloats** ()
  Disables float data type functions to do a word swap for all slaves.
- synchronized void **configureLittleEndianFloats** (int slaveAddr)
  Disables float data type functions to do a word swap on a per slave basis.
- void **configureSwappedFloats** ()
  Configures float data type functions to do a word swap for all slaves.
- synchronized void **configureSwappedFloats** (int slaveAddr)
  Enables float data type functions to do a word swap on a per slave basis.

6.16.1 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).

All functions have been implemented thread-safe. It is also possible to instantiate multiple instances for establishing multiple connections on different serial ports.

See also:

Data and Control Functions for all Modbus Protocol Flavours (p. 14), **Serial Protocols** (p. 15)
MbusRtuMasterProtocol (p. 105), **MbusElamMasterProtocol** (p. 55), **MbusAsciiMasterProtocol** (p. 32), **MbusTcpMasterProtocol** (p. 153)

6.16.2 Member Function Documentation

synchronized void openProtocol (String *portName*, int *baudRate*, int *dataBits*, int *stopBits*, int *parity*) throws IOException, UnsupportedCommOperationException, PortInUseException

Opens a Modbus serial port with specific port parameters.

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

Note:

The default time-out for the data transfer is 1000 ms.
The default poll delay is 0 ms.
Automatic retries are switched off (retry count is 0).
The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.

Parameters:

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

baudRate The port baudRate in bps (typically 1200 - 19200)

dataBits DATABITS_7: 7 data bits (ASCII protocol only), DATABITS_8: data bits

stopBits STOPBITS_1: 1 stop bit, STOPBITS_1_5: 1.5 stop bits, STOPBITS_2: 2 stop bits

parity PARITY_NONE: no parity, PARITY_ODD: odd parity, PARITY_EVEN: even parity, PARITY_MARK: mark parity, PARITY_SPACE: space parity

Exceptions:

IOException An I/O error occured

UnsupportedCommOperationException A communication parameter is not supported

PortInUseException Port is already used by somedbody else

IllegalArgumentException A parameter is invalid

Reimplemented in **MbusAsciiMasterProtocol** (p. 37), and **MbusRtuMasterProtocol** (p. 111).

synchronized void closeProtocol ( ) throws IOException `[virtual]`

Closes the serial port and releases any system resources associated with the port.

Exceptions:

IOException An I/O error occured

Implements **MbusMasterFunctions** (p. 82).

boolean isOpen ( ) `[virtual]`

Returns whether the port is open or not.

Return values:

true = open

false = closed

Implements **MbusMasterFunctions** (p. 103).

synchronized void readCoils (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException `[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: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. 20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputDiscretes (int *slaveAddr*, int *startRef*, boolean[] *bitArr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. 20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeCoil (int *slaveAddr*, int *bitAddr*, boolean *bitVal*) throws IOException, BusProtocolException `[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 - 247)

bitAddr Coil address (Range: depends on configuration setting)

bitVal true sets, false clears discrete output variable

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. 20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void forceMultipleCoils (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException `[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: depends on configuration setting)

bitArr Buffer which contains the data to be sent. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

BusProtocolException (p. 20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException `[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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int *slaveAddr,* int *startRef,* int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, float[ ] *float32Arr*) throws IOException, BusProtocolException [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 two consecutive output registers (holding registers, 4:00000 table) into float values.

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException [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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

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

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* float[ ] *float32Arr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeSingleRegister (int *slaveAddr,* int *regAddr,* short *regVal*) throws IOException, BusProtocolException `[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 - 247)

regAddr Register address (Range: depends on configuration setting)

regVal Data to be sent

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException `[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 - 247)

startRef Start reference (Range depends on configuration setting)

regArr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

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

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

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20***) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, float[] *float32Arr*) throws IOException, BusProtocolException `[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 a pairs of output registers (holding registers, 4:00000 table).

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20***) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void maskWriteRegister (int *slaveAddr*, int *regAddr*, short *andMask*, short *orMask*) throws IOException, BusProtocolException `[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: depends on configuration setting)

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

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readWriteRegisters (int *slaveAddr,* int *readRef,* short[ ] *readArr,* int *writeRef,* short[ ] *writeArr*) throws IOException, BusProtocolException `[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 reference for reading (Range: depends on configuration setting)

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

writeRef Start reference for writing (Range: depends on configuration setting)

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized byte readExceptionStatus (int *slaveAddr*) throws IOException, BusProtocolException [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)

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void setTimeout (int *timeOut*) [inherited]

Configures time-out.

This function sets the time-out to the specified value. A value of 0 disables the time-out, which causes all subsequent calls to data and control functions to block.

Note:

A port or connection must be closed in order to configure it.

Parameters:

timeOut Timeout value in ms (Range: 0 - 100000), 0 disables time-out

Exceptions:

IllegalStateException Port is open

IllegalArgumentException Parameter is out of range

int getTimeout ( ) [inherited]

Returns the operation time-out value.

Returns:

Timeout value in ms

synchronized void setPollDelay (int *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.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open

IllegalArgumentException Parameter is out of range

int getPollDelay ( ) [inherited]

Returns the poll delay time.

Returns:

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

synchronized void setRetryCnt (int *retryCnt*) [inherited]

Configures the automatic retry setting.

A value of 0 disables any automatic retries.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open
IllegalArgumentException Parameter is out of range

int getRetryCnt ( ) [inherited]

Returns the automatic retry count.

Returns:

Retry count

synchronized long getTotalCounter ( ) [inherited]

Returns how often a transmit/receive cycle has been executed.

Returns:

Counter value

synchronized long getSuccessCounter ( ) [inherited]

Returns how often a transmit/receive cycle was successful.

Returns:

Counter value

void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

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

Remarks:

This is the default mode

synchronized void configureStandard32BitMode (int *slaveAddr*) [inherited]

Configures a slave for Standard 32-bit Register Mode.

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

Parameters:

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

Remarks:

This is the default mode

void configureSingleReg32BitMode ( ) [inherited]

Configures all slaves for Single Register 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.

synchronized void configureSingleReg32BitMode (int *slaveAddr*) [inherited]

Configures all slaves for Single Register 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.

void configureCountFromOne ( ) [inherited]

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

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

Remarks:

This is the default mode

synchronized void configureCountFromOne (int *slaveAddr*) [inherited]

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

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

Parameters:

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

Remarks:

This is the default mode

void configureCountFromZero ( ) [inherited]

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

This is also known as PDU addressing.

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

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

synchronized void configureCountFromZero (int *slaveAddr*) [inherited]

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

This is also known as PDU addressing.

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

Parameters:

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

void configureLittleEndianInts ( ) [inherited]

Disables word swapping for int data type functions for all slaves.

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

Remarks:

This is the default mode

synchronized void configureLittleEndianInts (int *slaveAddr*) [inherited]

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

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

Remarks:

This is the default mode

Parameters:

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

void configureBigEndianInts ( ) [inherited]

Configures int data type functions to do a word swap for all slaves.

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.

synchronized void configureBigEndianInts (int *slaveAddr*) [inherited]

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

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

Parameters:

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

void configureLittleEndianFloats ( ) [inherited]

Disables float data type functions to do a word swap for all slaves.

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

Remarks:

This is the default mode

synchronized void configureLittleEndianFloats (int *slaveAddr*) [inherited]

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

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

Remarks:

This is the default mode

Parameters:

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

void configureSwappedFloats ( ) [inherited]

Configures float data type functions to do a word swap for all slaves.

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.

synchronized void configureSwappedFloats (int *slaveAddr*) [inherited]

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

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

Note:

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

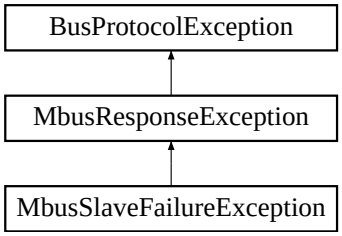
Parameters:

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

6.17 MbusSlaveFailureException Class Reference

Slave Device Failure exception response.

Inheritance diagram for MbusSlaveFailureException:



Public Member Functions

- **MbusSlaveFailureException** ()

  *Constructs a **MbusSlaveFailureException** (p. 152).*

Static Public Member Functions

- static long **getCounter** ()

  Returns how often this exception has been triggered.

- static void **resetCounter** ()

  Resets exception counter.

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

See also:

MbusResponseException (p. 103)

6.17.2 Member Function Documentation

static long getCounter ( ) `[static]`

Returns how often this exception has been triggered.

Returns:

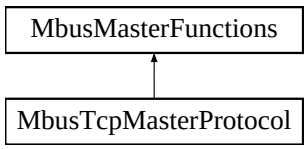
Counter value

Reimplemented from **MbusResponseException** (p. 104).

6.18 MbusTcpMasterProtocol Class Reference

MODBUS/TCP Master Protocol class.

Inheritance diagram for MbusTcpMasterProtocol:



Public Member Functions

- synchronized void **openProtocol** (String hostName) throws IOException

  Connects to a MODBUS/TCP slave.

- synchronized void **closeProtocol** () throws IOException

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

- synchronized void **setPort** (int **portNo**)

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

- int **getPort** ()

  Returns the TCP port number used by the protocol.

- boolean **isOpen** ()

  Returns whether currently connected or not.

Protected Attributes

- int **portNo** = 502

  TCP port number, defaults to 502.

Advantec ADAM 5000/6000 Series Commands

- String **adamSendReceiveAsciiCmd** (String commandStr) throws IOException, BusProtocolException

Send/Receive ADAM 5000/6000 ASCII command.

Bit Access

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

- synchronized void **readCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 1 (01 hex), Read Coil Status/Read Coils.

- synchronized void **readInputDiscretes** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

- synchronized void **writeCoil** (int slaveAddr, int bitAddr, boolean bitVal) throws IOException, BusProtocolException

  Modbus function 5 (05 hex), Force Single Coil/Write Coil.

- synchronized void **forceMultipleCoils** (int slaveAddr, int startRef, boolean[ ] bitArr) throws IOException, BusProtocolException

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

Register Access (16-bit, 32-bit and floating point)

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

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

- synchronized void **readMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex), Read Input Registers.

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **readInputRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

- synchronized void **writeSingleRegister** (int slaveAddr, int regAddr, short regVal) throws IOException, BusProtocolException

  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, short[ ] regArr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, int[ ] int32Arr) throws IOException, BusProtocolException

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

- synchronized void **writeMultipleRegisters** (int slaveAddr, int startRef, float[ ] float32Arr) throws IOException, BusProtocolException

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

- synchronized void **maskWriteRegister** (int slaveAddr, int regAddr, short andMask, short orMask) throws IOException, BusProtocolException

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

- synchronized void **readWriteRegisters** (int slaveAddr, int readRef, short[ ] readArr, int writeRef, short[ ] writeArr) throws IOException, BusProtocolException

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

Diagnostics

- synchronized byte **readExceptionStatus** (int slaveAddr) throws IOException, BusProtocolException

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

Custom Function Codes

- synchronized void **readHistoryLog** (int slaveAddr, int channelNo, int resolution, short readArr[ ]) throws IOException, BusProtocolException

  Vendor Specific Modbus function 100 (64 hex), Read History Log.

Protocol Configuration

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

Transmission Statistic Functions

- synchronized long **getTotalCounter** ()
  Returns how often a transmit/receive cycle has been executed.
- synchronized void **resetTotalCounter** ()
  Resets total transmit/receive cycle counter.
- synchronized long **getSuccessCounter** ()
  Returns how often a transmit/receive cycle was successful.
- synchronized void **resetSuccessCounter** ()
  Resets successful transmit/receive counter.

Slave Configuration

- void **configureStandard32BitMode** ()
  Configures all slaves for Standard 32-bit Mode.
- synchronized void **configureStandard32BitMode** (int slaveAddr)
  Configures a slave for Standard 32-bit Register Mode.
- void **configureSingleReg32BitMode** ()
  Configures all slaves for Single Register 32-bit Mode.

- synchronized void **configureSingleReg32BitMode** (int slaveAddr)
  Configures all slaves for Single Register 32-bit Mode.
- void **configureCountFromOne** ()
  Configures the reference counting scheme to start with one for all slaves.
- synchronized void **configureCountFromOne** (int slaveAddr)
  Configures a slave's reference counting scheme to start with one.
- void **configureCountFromZero** ()
  Configures the reference counting scheme to start with zero for all slaves.
- synchronized void **configureCountFromZero** (int slaveAddr)
  Configures a slave's reference counting scheme to start with zero.
- void **configureLittleEndianInts** ()
  Disables word swapping for int data type functions for all slaves.
- synchronized void **configureLittleEndianInts** (int slaveAddr)
  Disables word swapping for int data type functions on a per slave basis.
- void **configureBigEndianInts** ()
  Configures int data type functions to do a word swap for all slaves.
- synchronized void **configureBigEndianInts** (int slaveAddr)
  Enables int data type functions to do a word swap on a per slave basis.
- void **configureLittleEndianFloats** ()
  Disables float data type functions to do a word swap for all slaves.
- synchronized void **configureLittleEndianFloats** (int slaveAddr)
  Disables float data type functions to do a word swap on a per slave basis.
- void **configureSwappedFloats** ()
  Configures float data type functions to do a word swap for all slaves.
- synchronized void **configureSwappedFloats** (int slaveAddr)
  Enables float data type functions to do a word swap on a per slave basis.

6.18.1 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.

All functions have been implemented thread-safe. It is also possible to instantiate multiple instances 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. 16)
MbusRtuMasterProtocol (p. 105), **MbusElamMasterProtocol** (p. 55), **MbusAsciiMasterProtocol** (p. 32)

6.18.2 Member Function Documentation

synchronized void openProtocol (String *hostName*) throws IOException

Connects to a MODBUS/TCP slave.

This function establishes a logical network connection between master and slave. After a connection has been established data and control functions can be used. A TCP/IP connection should be closed if it is no longer needed.

Note:

The default time-out for the connection is 1000 ms.
The default TCP port number is 502.

Parameters:

hostName String with IP address or host name

Exceptions:

IOException An I/O error occured

ConnectException Connection refused by remote, no service listening

BindException Remote port in use

NoRouteToHostException No route to host

UnknownHostException Unknown host, e.g. wrong IP address or name

SecurityException Security violation

Note:

Java 1.4 or higher is needed for connection time-out to work otherwise the system's default time-out value will be used for connections.

synchronized void closeProtocol ( ) throws IOException [virtual]

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

Exceptions:

IOException An I/O error occured

Implements **MbusMasterFunctions** (p. 82).

synchronized void setPort (int *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 openConnection().

Parameters:

portNo Port number to be used when opening the connection (Range: 0 - 0xFFFF)

Exceptions:

IllegalStateException Conection is open

IllegalArgumentException Parameter is out of range

int getPort ( )

Returns the TCP port number used by the protocol.

Returns:

Port number used by the protocol

boolean isOpen ( ) [virtual]

Returns whether currently connected or not.

Return values:

true = connected

false = not connected

Implements **MbusMasterFunctions** (p. 103).

synchronized void readCoils (int *slaveAddr,* int *startRef,* boolean[ ] *bitArr*) throws IOException, BusProtocolException `[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: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputDiscretes (int *slaveAddr,* int *startRef,* boolean[ ] *bitArr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

bitArr Buffer which will contain the data read. The length of the array determines how many coils are read (Range: 1-2000).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeCoil (int *slaveAddr*, int *bitAddr*, boolean *bitVal*) throws IOException, BusProtocolException `[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 - 247)

bitAddr Coil address (Range: depends on configuration setting)

bitVal true sets, false clears discrete output variable

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void forceMultipleCoils (int *slaveAddr*, int *startRef*, boolean[ ] *bitArr*) throws IOException, BusProtocolException `[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: depends on configuration setting)

bitArr Buffer which contains the data to be sent. The length of the array determines how many coils are written (Range: 1-1968).

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException `[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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int *slaveAddr*, int *startRef*, int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 3 (03 hex) for 32-bit int data types, Read Holding Registers/Read Multiple Registers as int data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readMultipleRegisters (int ***slaveAddr,*** **int** ***startRef,*** **float[]** ***float32Arr*** **) throws IOException, BusProtocolException** `[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 two consecutive output registers (holding registers, 4:00000 table) into float values.

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException* (p. *20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* short[ ] *regArr*) throws IOException, BusProtocolException `[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 reference (Range: depends on configuration setting)

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:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException* (p. *20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr,* int *startRef,* int[ ] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

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

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. 20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readInputRegisters (int *slaveAddr*, int *startRef*, float[ ] *float32Arr*) throws IOException, BusProtocolException `[inherited]`

Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer which will be filled with the data read. The length of the array determines how many registers are read (Range: 1-62).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. 20) A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void writeSingleRegister (int *slaveAddr*, int *regAddr*, short *regVal*) throws IOException, BusProtocolException `[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 - 247)
regAddr Register address (Range: depends on configuration setting)
regVal Data to be sent

Exceptions:

IllegalStateException Port or connection is closed
IOException An I/O error occured
IllegalArgumentException A parameter is out of range
BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, short[ ] *regArr*) throws IOException, BusProtocolException `[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 - 247)
startRef Start reference (Range depends on configuration setting)
regArr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IllegalStateException Port or connection is closed
IOException An I/O error occured
IllegalArgumentException A parameter is out of range
BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, int[] *int32Arr*) throws IOException, BusProtocolException `[inherited]`

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

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

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

int32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void writeMultipleRegisters (int *slaveAddr*, int *startRef*, float[] *float32Arr*) throws IOException, BusProtocolException `[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 a pairs of output registers (holding registers, 4:00000 table).

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 247)

startRef Start reference (Range: depends on configuration setting)

float32Arr Buffer with the data to be sent. The length of the array determines how many registers are sent (Range: 1-61).

Exceptions:

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. *20)* A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

Broadcast supported for serial protocols

synchronized void maskWriteRegister (int *slaveAddr,* **int** *regAddr,* **short** *andMask,* **short** *orMask* **) throws IOException, BusProtocolException** `[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: depends on configuration setting)

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

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. *20)* A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void readWriteRegisters (int *slaveAddr*, int *readRef*, short[ ] *readArr*, int *writeRef*, short[ ] *writeArr*) throws IOException, BusProtocolException [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 reference for reading (Range: depends on configuration setting)

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

writeRef Start reference for writing (Range: depends on configuration setting)

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

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException (p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized byte readExceptionStatus (int *slaveAddr*) throws IOException, BusProtocolException [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)

Exceptions:

IllegalStateException Port or connection is closed

IOException An I/O error occured

IllegalArgumentException A parameter is out of range

BusProtocolException **(**p. ***20)*** A protocol failure occured. See descendants of **BusProtocolException** (p. 20) for a more detailed failure list.

Note:

No broadcast supported

synchronized void setTimeout (int *timeOut*) [inherited]

Configures time-out.

This function sets the time-out to the specified value. A value of 0 disables the time-out, which causes all subsequent calls to data and control functions to block.

Note:

A port or connection must be closed in order to configure it.

Parameters:

timeOut Timeout value in ms (Range: 0 - 100000), 0 disables time-out

Exceptions:

IllegalStateException Port is open

IllegalArgumentException Parameter is out of range

int getTimeout ( ) [inherited]

Returns the operation time-out value.

Returns:

Timeout value in ms

synchronized void setPollDelay (int *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.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open
IllegalArgumentException Parameter is out of range

int getPollDelay ( ) [inherited]

Returns the poll delay time.

Returns:

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

synchronized void setRetryCnt (int *retryCnt*) [inherited]

Configures the automatic retry setting.
A value of 0 disables any automatic retries.

Note:

A port or connection must be closed in order to configure it.

Parameters:

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

Exceptions:

IllegalStateException Port is open
IllegalArgumentException Parameter is out of range

int getRetryCnt ( ) [inherited]

Returns the automatic retry count.

Returns:

Retry count

synchronized long getTotalCounter ( ) [inherited]

Returns how often a transmit/receive cycle has been executed.

Returns:

Counter value

synchronized long getSuccessCounter ( ) [inherited]

Returns how often a transmit/receive cycle was successful.

Returns:

Counter value

void configureStandard32BitMode ( ) [inherited]

Configures all slaves for Standard 32-bit Mode.

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

Remarks:

This is the default mode

synchronized void configureStandard32BitMode (int *slaveAddr*) [inherited]

Configures a slave for Standard 32-bit Register Mode.

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

Parameters:

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

Remarks:

This is the default mode

void configureSingleReg32BitMode ( ) [inherited]

Configures all slaves for Single Register 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.

synchronized void configureSingleReg32BitMode (int *slaveAddr*) [inherited]

Configures all slaves for Single Register 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.

void configureCountFromOne ( ) [inherited]

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

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

Remarks:

This is the default mode

synchronized void configureCountFromOne (int *slaveAddr*) [inherited]

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

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

Parameters:

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

Remarks:

This is the default mode

void configureCountFromZero ( ) [inherited]

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

This is also known as PDU addressing.

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

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

synchronized void configureCountFromZero (int *slaveAddr*) [inherited]

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

This is also known as PDU addressing.

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

Parameters:

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

void configureLittleEndianInts ( ) [inherited]

Disables word swapping for int data type functions for all slaves.

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

Remarks:

This is the default mode

synchronized void configureLittleEndianInts (int *slaveAddr*) [inherited]

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

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

Remarks:

This is the default mode

Parameters:

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

void configureBigEndianInts ( ) [inherited]

Configures int data type functions to do a word swap for all slaves.

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.

synchronized void configureBigEndianInts (int *slaveAddr*) [inherited]

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

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

Parameters:

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

void configureLittleEndianFloats ( ) [inherited]

Disables float data type functions to do a word swap for all slaves.

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

Remarks:

This is the default mode

synchronized void configureLittleEndianFloats (int *slaveAddr*) [inherited]

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

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

Remarks:

This is the default mode

Parameters:

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

void configureSwappedFloats ( ) [inherited]

Configures float data type functions to do a word swap for all slaves.

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.

synchronized void configureSwappedFloats (int *slaveAddr*) [inherited]

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

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

Note:

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

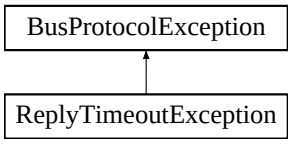
Parameters:

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

6.19 ReplyTimeoutException Class Reference

Reply time-out.

Inheritance diagram for ReplyTimeoutException:



Public Member Functions

- **ReplyTimeoutException** ()

  *Constructs a **ReplyTimeoutException** (p. 176).*

Static Public Member Functions

- static long **getCounter** ()

  Returns how often this exception has been triggered.

- static void **resetCounter** ()

  Resets exception counter.

6.19.1 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 adress will also cause this error. In some occassions this expection is also produced if the characters received don't constitute a complete frame.

See also:

BusProtocolException (p. 20)

6.19.2 Member Function Documentation

static long getCounter ( ) `[static]`

Returns how often this exception has been triggered.

Returns:

Counter value

Reimplemented from **BusProtocolException** (p. 21).

6.20 Version Class Reference

Version (p. 177) class.

Static Public Member Functions

- static String **getPackageVersion** ()

  Returns the package version number.

Static Package Functions

- **[static initializer]**

  Prints banner to stdout.

Static Package Attributes

- static final String **packageVersion** = "EXPERIMENTAL"

  Package version.

6.20.1 Detailed Description

Version (p. 177) class. Provides functions to identify the product version.

6.20.2 Member Function Documentation

static String getPackageVersion ( ) `[static]`

Returns the package version number.

Returns:

Package version string

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.

Index