Professional Fieldbus Connections

FieldTalk Modbus Slave Library for .NET

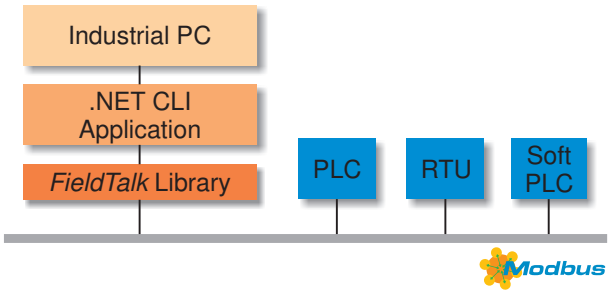
Software manual

Library version 2.7.1

Contents

1 Introduction

This *FieldTalk*™ Modbus® Slave Library for .NET allows you to incorporate Modbus slave functionality into your your VB.net and C# programs.



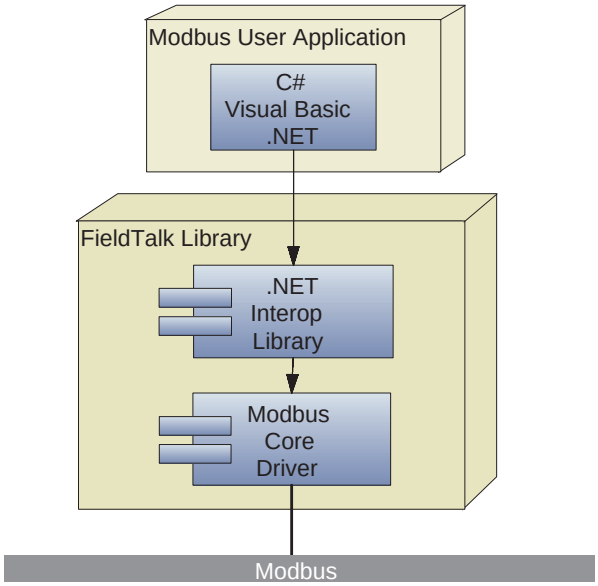
Typical applications are Modbus based Supervisory Control and Data Acquisition Systems (SCADA), Modbus data concentrators, Modbus gateways, User Interfaces and Factory Information Systems (FIS).

Features:

- Robust design suitable for real-time and industrial applications
- Full implementation of Bit Access and 16 Bits Access Function Codes as well as a subset of the most commonly used Diagnostics Function Codes
- Standard Modbus bit and 16-bit integer data types (coils, discretes & registers)
- Daniel/Enron single register 32-bit transfers
- Support of Broadcasting
- Master time-out supervision
- Failure and transmission counters
- Supports single or multiple slave addresses

1.1 Library Structure

The FieldTalk .NET library consists of two components: a *.NET Interop Library* with MS↩ IL code (managed code) and a *Modbus Core Driver* written in native code (unmanaged code). This architecture has significant performance benefits for .NET applications because the time critical communication code is executed as native code.



The two components are contained in separate files: `FieldTalk.Modbus.Slave.dll` contains the *.NET Interop Library* and `libmbusslave.dll` the native *Modbus Core Driver*. These two library files have to be deployed with your application.

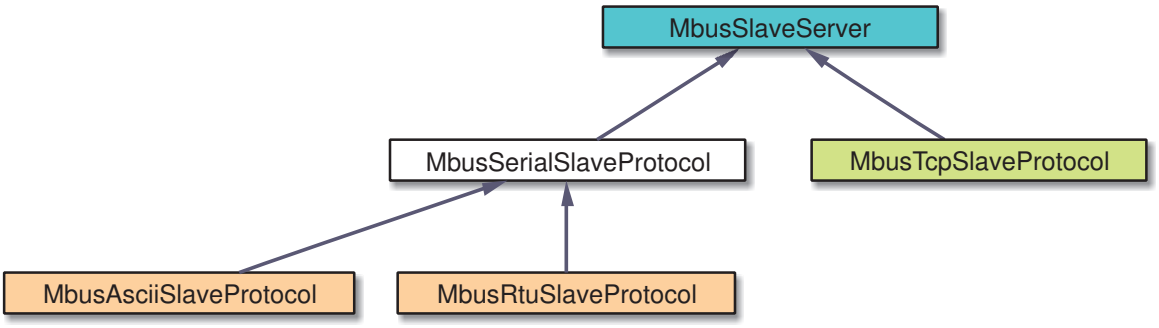
For .NET Core, the deployment and loading of the native code library is taken care of by .NET core's package management. This works for Windows as well as for Linux platforms.

For classic .NET Framework 4.0 and 2.0, the deployment of the native code library is taken care of by the `FieldTalk.Modbus.Slave.prop` file which the NutGet package manager will automatically add to your project. With that file, when compiling, a `bin/x86` and a `bin/x64` subfolder is created in the bin directory and the native libraries copied into these subfolders. When the .NET assembly is loaded, the correct native library will be loaded from one of these two sub-folders.

For the .NET Compact Framework (Windows Embedded Compact, Windows CE and Windows Mobile) the deployment of the native libray must be done manually.

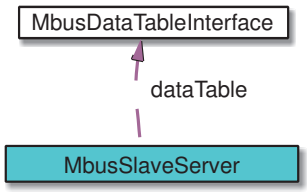
The library's API The library is organised in two categories of classes.

One category implements the Server Engines for each Modbus slave protocol flavour. There is one Server Engine class for each protocol flavour and a common Server Engine base class, which applies to all protocol flavours. Because the two serial protocols ASCII and RTU share some common code, an intermediate base class implements the functions specific to serial protocols.



The second category of classes is Data Providers classes. Data Provider classes represent

the interface between the Server Engine and your application.



The base class MbusSlaveServer contains a protocol unspecific contains a protocol unspecific Server Engine and the protocol state machine. All protocol flavours inherit from this base class.

The class MbusAsciiSlaveProtocol implements the Modbus ASCII protocol, the class MbusRtuSlaveProtocol implements the Modbus RTU protocol and the class MbusTcp↩ SlaveProtocol implements the MODBUS/TCP protocol.

Before a server can be used, a Data Provider has to be declared. A Data Provider is created by declaring a new class derived from MbusDataTableInterface. The class Mbus↩ DataTableInterface is the base class for a Data Provider and implements a set of default methods. An application specific Data Provider simply overrides selected default methods and the Modbus slave is ready.

C#

```
class MyDatatable: MbusDataTableInterface
{
    ... // Application specific data interface
};

MyDatatable dataTable = new MyDatatable();
```

VB.net

```
Class MyDatatable
    Inherits MbusDataTableInterface
        ... ' Application specific data interface
End Class

Dim dataTable As MyDatatable = New MyDatatable
```

In order to use one of the three slave protocols, the desired protocol flavour class has to be instantiated and associated with the Data Provider. The following example creates an RTU protocol and links a data table to slave address 20:

C#

```
MbusRtuSlaveProtocol mbusServer = new MbusRtuSlaveProtocol();
mbusProtocol.addDataTable(20, dataTable);
```

VB.net

```
Dim mbusServer As MbusRtuSlaveProtocol = New MbusRtuSlaveProtocol
mbusProtocol.addDataTable(20, dataTable);
```

After a protocol object has been declared and started up the server loop has to be executed cyclically. The Modbus slave is ready to accept connections and to reply to master queries.

C#

```
do
{
    result = mbusServer.serverLoop();
} while (!(result != BusProtocolErrors.FTALK_SUCCESS));
```

VB.net

```
Do
    result = mbusServer.serverLoop()
Loop Until result <> FTALK_SUCCESS
```

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

Function Code	Current Terminology	Classic Terminology
2	Read Discrete Inputs	Read Input Status
5	Write Single Coil	Force Single Coil
15 (0F hex)	Write Multiple Coils	Force Multiple Coils
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
Record Access		
20 (14 hex)	Read File Record	Read General Reference
21 (15 hex)	Write File Record	Write General Reference
Diagnostics		
7	Read Exception Status	Read Exception Status
17 (11 hex)	Report Slave ID	Report Slave ID
43 (2B hex) subcode 14 (0E hex)	Read Device Identification	

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

Table	Classic Terminology	Modicon Register Table	Characteristics
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 Slave Library defines services to:

- Read and Write bit values
- Read and Write 16-bit integers
- Reading and Writing of 32-bit values using Daniel/Enron Modbus extension

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 Design Background

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

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

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

4 Module Documentation

4.1 Server Functions common to all Modbus Protocol Flavours

The *FieldTalk* Modbus Slave Protocol Library's server engine implements the most commonly used Modbus 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.

The following table lists the functions supported by the slave:

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
Record Access		
20 (14 hex)	Read File Record	Read General Reference
21 (15 hex)	Write File Record	Write General Reference
Diagnostics		
7	Read Exception Status	Read Exception Status
17 (11 hex)	Report Slave ID	Report Slave ID
43 (2B hex) subcode 14 (0E hex)	Read Device Identification	

4.2 Serial Protocols

Classes

- class MbusRtuSlaveProtocol

  This class realises the server side of the Modbus RTU slave protocol.
- class MbusAsciiSlaveProtocol

  This class realises the server side of the Modbus ASCII slave protocol.

4.2.1 Detailed Description

The Server Engines of the two serial protocol flavours are implemented in the classes MbusRtuSlaveProtocol and MbusAsciiSlaveProtocol. These classes provide functions to start-up and to execute the server engine which includes opening and closing of the serial port. Upon receipt of a valid master query the server engine calls Data Provider methods to exchange data with the user application. For a more detailed description which Modbus data and control functions have been implemented in the server engine see section Server Functions common to all Modbus Protocol Flavours.

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

4.3 TCP/IP Protocols

Classes

- class MbusTcpSlaveProtocol

  This class realises the server side of the MODBUS/TCP slave protocol.

4.3.1 Detailed Description

The Server Engine of the MODBUS/TCP slave protocol is implemented in the class MbusTcpSlaveProtocol. It provides functions to start-up and to execute the server engine. This server engine can handle multiple master connections and is implemented as a single threaded TCP server. Upon receipt of a valid master query the server engine calls Data Provider methods to exchange data with the user application. For a more detailed description which Modbus data and control functions have been implemented in the server engine see section Server Functions common to all Modbus Protocol Flavours.

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

4.4 Data Provider

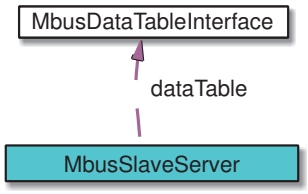
Classes

- class MbusDataTableInterface

  This class defines the interface between a Modbus slave Server Engine and your application.

4.4.1 Detailed Description

A Data Provider acts as an agent between your Application and the Server Engine.



After instantiating a Server Engine class of any protocol flavour, you have to associate it with a Data Provider by calling addDataTable and passing a reference of the Data Provider object.

C#

```
MbusRtuSlaveProtocol mbusServer = new MbusRtuSlaveProtocol();
MyDatatable dataTable = new MyDatatable();
mbusServer.addDataTable(1, dataTable);
```

VB.net

```
Dim mbusServer As New MbusRtuSlaveProtocol()
Dim dataTable As New MyDatatable()
mbusServer.addDataTable(1, dataTable)
```

To create an application specific Data Provider derive a new class from MbusDataTable↩ Interface and override the required data access methods.

A minimal Data Provider which realises a Modbus slave with read access to holding registers would be:

C#

```
class MyDatatable: MbusDataTableInterface
{
    // Override readHoldingRegistersTable method:
    protected override bool readHoldingRegistersTable(Int32 startRef, Int16[] regArr)
    {
        ... your application specific implementation
    }
}
```

VB.net

```
Class MyDatatable
    Inherits MbusDataTableInterface
    ' Override readHoldingRegistersTable method:
    Protected Overrides Function readHoldingRegistersTable(ByVal startRef As Int32, ByVal regArr() As
       Int16) As Boolean
        ... your application specific implementation
    End Function
End Class
```

4.5 Error Management

Classes

- class BusProtocolErrors

  Protocol Errors and Modbus exceptions codes

4.5.1 Detailed Description

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

5 Class Documentation

5.1 MbusRtuSlaveProtocol Class Reference

This class realises the server side of the Modbus RTU slave protocol.

Public Member Functions

- MbusRtuSlaveProtocol ()

  Instantiates a Modbus RTU protocol server object.
- System.Int32 startupServer ()

  Opens a serial Modbus protocol and the associated serial port with the port parameters configured via properties
- Int32 startupServer (string portName, Int32 baudRate, Int32 dataBits, Int32 stopBits, Int32 parity)

  Opens a serial Modbus protocol and the associated serial port with the port parameters configured via properties
- Int32 enableRs485Mode (Int32 rtsDelay)

  Enables RS485 mode.
- void Dispose ()

  Implement IDisposable.
- int addDataTable (int slaveAddr, MbusDataTableInterface dataTable)

  Associates a protocol object with a Data Provider and a Modbus slave ID.
- int serverLoop ()

  Modbus slave server loop
- bool isStarted ()

  Returns whether server has been started up.
- void shutdownServer ()

  Shuts down the Modbus Server.
- bool getConnectionStatus ()

  Checks if a Modbus master is polling periodically.
- Int32 setTimeout (Int32 timeOut)

  Configures master transmit time-out supervision.
- Int32 getTimeout ()

  Returns the currently set master time-out supervision value.
- Int32 getTotalCounter ()

  Returns how often a message transfer has been executed.
- void resetTotalCounter ()

  Resets total message transfer counter.
- Int32 getSuccessCounter ()

  Returns how often a message transfer was successful.
- void resetSuccessCounter ()

  Resets successful message transfer counter.

Static Public Member Functions

- static string getPackageVersion ()
  Returns the library version number.

Public Attributes

- const Int32 SER_DATABITS_7 = 7
  7 data bits
- const Int32 SER_DATABITS_8 = 8
  8 data bits
- const Int32 SER_STOPBITS_1 = 1
  1 stop bit
- const Int32 SER_STOPBITS_2 = 2
  2 stop bits
- const Int32 SER_PARITY_NONE = 0
  No parity
- const Int32 SER_PARITY_ODD = 1
  Odd parity
- const Int32 SER_PARITY_EVEN = 2
  Even parity

Properties

- string portName `[get, set]`
  Serial port identifier property
- Int32 baudRate `[get, set]`
  Baud rate property in bps
- Int32 dataBits `[get, set]`
  Data bits property
- Int32 stopBits `[get, set]`
  Stop bits property
- Int32 parity `[get, set]`
  Parity property
- Int32 timeout `[get, set]`
  Time-out port property

5.1.1 Detailed Description

This class provides functions to start-up and to execute the server engine which includes opening and closing of the serial port. Upon receipt of a valid master query the server engine calls Data Provider methods to exchange data with the user application.

For a more detailed description which Modbus data and control functions have been implemented in the server engine see section `Server Functions common to all Protocol Flavours`.

It is possible to instantiate multiple instances for establishing multiple connections on different serial ports, however they must be executed in separate threads.

See also

MbusDataTableInterface

5.1.2 Constructor & Destructor Documentation

MbusRtuSlaveProtocol() `MbusRtuSlaveProtocol ( ) [inline]`

The association with a Data Provider is done after construction using the addDataTable method.

Exceptions

OutOfMemoryException	Creation of class failed

5.1.3 Member Function Documentation

startupServer() `[1/2] System.Int32 startupServer ( ) [inline], [inherited]`

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

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

startupServer() `[2/2]`

```
Int32 startupServer (
    string portName,
    Int32 baudRate,
    Int32 dataBits,
    Int32 stopBits,
    Int32 parity ) [inline], [inherited]
```

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

Note

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

Parameters

portName	Serial port identifier (eg "COM1")
baudRate	The port baud rate in bps (1200 - 115200, higher on some platforms)
dataBits	SER_DATABITS_7: 7 data bits (ASCII protocol only), SER_DATABITS_8: 8 data bits
stopBits	SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits
parity	SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_EVEN: even parity

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

enableRs485Mode() `Int32 enableRs485Mode ( Int32 rtsDelay ) [inline], [inherited]`

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

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

Note

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.
This mode must be set before starting the server in order to come into effect.

Parameters

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

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

Dispose() `void Dispose ( ) [inline], [inherited]`

Shuts down server, closes any associated communication resources (serial ports or sockets) and releases any resources.

addDataTable()
```
int addDataTable (
        int slaveAddr,
        MbusDataTableInterface dataTable ) [inline], [inherited]
```

Parameters

slaveAddr	Modbus slave address for server to listen on (-1 - 255). 0 is regarded as a valid value for a MODBUS/TCP server address. A value of -1 means the server disregards the slave address and listens to all slave addresses. 0 or -1 is only valid for MODBUS/TCP!
dataTable	Reference to a Modbus data table. Must be a Data Provider object derived from the MbusDataTableInterface class. The Data Provider is the interface between your application data and the Modbus network.

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

serverLoop() `int serverLoop ( ) [inline], [inherited]`

This server loop must be called continuously. It must not be blocked. The server has to be started before calling the serverLoop() method.

In most cases the server loop is executed in an infinite loop in its own thread:

```
while (notTerminated)
{
    result = mbusProtocol.serverLoop();
}
```

```
Do While notTerminated
    result = mbusProtocol.serverLoop()
Loop
```

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

isStarted() `bool isStarted ( ) [inline], [inherited]`

Returns

true = started
false = shutdown

shutdownServer() `void shutdownServer ( ) [inline], [inherited]`

This function also closes any associated serial ports or sockets.

getConnectionStatus() `bool getConnectionStatus ( ) [inline], [inherited]`

Note

The master transmit time-out value must be set $>$ 0 in order for this function to work.

Returns

true = A master is polling at a frequency higher than the master transmit time-out value
false = No master is polling within the time-out period

setTimeout() `Int32 setTimeout ( Int32 timeOut ) [inline], [inherited]`

Configures master transmit time-out supervision. The slave can monitor whether a master is actually transmitting characters or not. This function sets the transmit time-out to the specified value. A value of 0 disables the time-out, which stops time-out notifications being sent to the Data Provider.

Note

The time-out value is a minimum value only and the actual time may be longer. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.
The time-out does not check whether a master is sending valid frames. The transmission of characters is sufficient to avoid the time-out.

Parameters

timeOut	Timeout value in ms (Range: 1 - 100000), 0 disables time-out

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

getTimeout() `Int32 getTimeout ( ) [inline], [inherited]`

Note

The time-out value is a minimum value only and the actual time may be longer. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

Returns

Timeout value in ms

getTotalCounter() `Int32 getTotalCounter ( ) [inline], [inherited]`

Returns

Counter value

resetTotalCounter() `void resetTotalCounter ( ) [inline], [inherited]`

getSuccessCounter() `Int32 getSuccessCounter ( ) [inline], [inherited]`

Returns

Counter value

resetSuccessCounter() `void resetSuccessCounter ( )  [inline], [inherited]`

getPackageVersion() `static string getPackageVersion ( )  [inline], [static], [inherited]`

Returns

Version string

5.1.4 Member Data Documentation

SER_DATABITS_7 `const Int32 SER_DATABITS_7 = 7  [inherited]`

SER_DATABITS_8 `const Int32 SER_DATABITS_8 = 8  [inherited]`

SER_STOPBITS_1 `const Int32 SER_STOPBITS_1 = 1  [inherited]`

SER_STOPBITS_2 `const Int32 SER_STOPBITS_2 = 2  [inherited]`

SER_PARITY_NONE `const Int32 SER_PARITY_NONE = 0  [inherited]`

SER_PARITY_ODD `const Int32 SER_PARITY_ODD = 1  [inherited]`

SER_PARITY_EVEN `const Int32 SER_PARITY_EVEN = 2  [inherited]`

5.1.5 Property Documentation

portName `string portName [get], [set], [inherited]`

Note

This property must be set before starting the server in order to come into effect.

Serial port identifier (eg "COM1")

baudRate `Int32 baudRate [get], [set], [inherited]`

Note

This property must be set before starting the server in order to come into effect.

Typically 1200 - 115200, maximum value depends on UART hardware

dataBits `Int32 dataBits [get], [set], [inherited]`

Note

This property must be set before starting the server in order to come into effect.

SER_DATABITS_7 as 7 data bits (ASCII protocol only), SER_DATABITS_8 as data bits

stopBits `Int32 stopBits [get], [set], [inherited]`

Note

The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.
This property must be set before starting the server in order to come into effect.

SER_STOPBITS_1 as 1 stop bit, SER_STOPBITS_2 as 2 stop bits

parity `Int32 parity [get], [set], [inherited]`

Note

This property must be set before starting the server in order to come into effect.

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

timeout `Int32 timeout [get], [set], [inherited]`

Configures operation or socket time-out.

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

Note

This property must be set before starting the server in order to come into effect.

Timeout value in ms (Range: 1 - 100000)

5.2 MbusAsciiSlaveProtocol Class Reference

This class realises the server side of the Modbus ASCII slave protocol.

Public Member Functions

- MbusAsciiSlaveProtocol ()

  Instantiates a Modbus ASCII protocol server object.
- System.Int32 startupServer ()

  Opens a serial Modbus protocol and the associated serial port with the port parameters configured via properties
- Int32 startupServer (string portName, Int32 baudRate, Int32 dataBits, Int32 stopBits, Int32 parity)

  Opens a serial Modbus protocol and the associated serial port with the port parameters configured via properties
- Int32 enableRs485Mode (Int32 rtsDelay)

  Enables RS485 mode.
- void Dispose ()

  Implement IDisposable.
- int addDataTable (int slaveAddr, MbusDataTableInterface dataTable)

  Associates a protocol object with a Data Provider and a Modbus slave ID.
- int serverLoop ()

  Modbus slave server loop
- bool isStarted ()

  Returns whether server has been started up.
- void shutdownServer ()

  Shuts down the Modbus Server.
- bool getConnectionStatus ()

  Checks if a Modbus master is polling periodically.
- Int32 setTimeout (Int32 timeOut)

  Configures master transmit time-out supervision.
- Int32 getTimeout ()

  Returns the currently set master time-out supervision value.
- Int32 getTotalCounter ()

  Returns how often a message transfer has been executed.
- void resetTotalCounter ()

  Resets total message transfer counter.
- Int32 getSuccessCounter ()

Returns how often a message transfer was successful.

- void resetSuccessCounter ()

  Resets successful message transfer counter.

Static Public Member Functions

- static string getPackageVersion ()

  Returns the library version number.

Public Attributes

- const Int32 SER_DATABITS_7 = 7

  7 data bits
- const Int32 SER_DATABITS_8 = 8

  8 data bits
- const Int32 SER_STOPBITS_1 = 1

  1 stop bit
- const Int32 SER_STOPBITS_2 = 2

  2 stop bits
- const Int32 SER_PARITY_NONE = 0

  No parity
- const Int32 SER_PARITY_ODD = 1

  Odd parity
- const Int32 SER_PARITY_EVEN = 2

  Even parity

Properties

- string portName `[get, set]`

  Serial port identifier property
- Int32 baudRate `[get, set]`

  Baud rate property in bps
- Int32 dataBits `[get, set]`

  Data bits property
- Int32 stopBits `[get, set]`

  Stop bits property
- Int32 parity `[get, set]`

  Parity property
- Int32 timeout `[get, set]`

  Time-out port property

5.2.1 Detailed Description

This class provides functions to start-up and to execute the Modbus ASCII server engine which includes opening and closing of the serial port. Upon receipt of a valid master query the server engine calls Data Provider methods to exchange data with the user application.

For a more detailed description which Modbus data and control functions have been implemented in the server engine see section `Server Functions common to all Protocol Flavours`.

It is possible to instantiate multiple instances for establishing multiple connections on different serial ports, however they must be executed in separate threads.

See also

MbusDataTableInterface

5.2.2 Constructor & Destructor Documentation

MbusAsciiSlaveProtocol() `MbusAsciiSlaveProtocol ( ) [inline]`

The association with a Data Provider is done after construction using the MbusSlaveServer.addDataTab method.

Exceptions

OutOfMemoryException	Creation of class failed

5.2.3 Member Function Documentation

startupServer() `[1/2] System.Int32 startupServer ( ) [inline], [inherited]`

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

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

startupServer() `[2/2] Int32 startupServer (`
`string` *`portName`*`,`

```
Int32 baudRate,
Int32 dataBits,
Int32 stopBits,
Int32 parity )  [inline], [inherited]
```

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

Note

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

Parameters

portName	Serial port identifier (eg "COM1")
baudRate	The port baud rate in bps (1200 - 115200, higher on some platforms)
dataBits	SER_DATABITS_7: 7 data bits (ASCII protocol only), SER_DATABITS_8: 8 data bits
stopBits	SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits
parity	SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_EVEN: even parity

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

enableRs485Mode() `Int32 enableRs485Mode (`

```
Int32 rtsDelay )  [inline], [inherited]
```

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

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

Note

The delay value is indicative only and not guaranteed to be maintained. How precise it is followed depends on the operating system used, it's scheduling priority and it's system timer resolution.
This mode must be set before starting the server in order to come into effect.

Parameters

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

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

Dispose()

```
void Dispose ( )  [inline], [inherited]
```

Shuts down server, closes any associated communication resources (serial ports or sockets) and releases any resources.

addDataTable()

```
int addDataTable (
        int slaveAddr,
        MbusDataTableInterface dataTable )  [inline], [inherited]
```

Parameters

slaveAddr	Modbus slave address for server to listen on (-1 - 255). 0 is regarded as a valid value for a MODBUS/TCP server address. A value of -1 means the server disregards the slave address and listens to all slave addresses. 0 or -1 is only valid for MODBUS/TCP!
dataTable	Reference to a Modbus data table. Must be a Data Provider object derived from the MbusDataTableInterface class. The Data Provider is the interface between your application data and the Modbus network.

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

serverLoop()

```
int serverLoop ( )  [inline], [inherited]
```

This server loop must be called continuously. It must not be blocked. The server has to be started before calling the serverLoop() method.

In most cases the server loop is executed in an infinite loop in its own thread:

```
while (notTerminated)
{
    result = mbusProtocol.serverLoop();
}
```

```
Do While notTerminated
    result = mbusProtocol.serverLoop()
Loop
```

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

isStarted() `bool isStarted ( ) [inline], [inherited]`

Returns

true = started
false = shutdown

shutdownServer() `void shutdownServer ( ) [inline], [inherited]`

This function also closes any associated serial ports or sockets.

getConnectionStatus() `bool getConnectionStatus ( ) [inline], [inherited]`

Note

The master transmit time-out value must be set $>$ 0 in order for this function to work.

Returns

true = A master is polling at a frequency higher than the master transmit time-out value
false = No master is polling within the time-out period

setTimeout() `Int32 setTimeout (`
`Int32 *timeOut* ) [inline], [inherited]`

Configures master transmit time-out supervision. The slave can monitor whether a master is actually transmitting characters or not. This function sets the transmit time-out to the specified value. A value of 0 disables the time-out, which stops time-out notifications being sent to the Data Provider.

Note

The time-out value is a minimum value only and the actual time may be longer. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.
The time-out does not check whether a master is sending valid frames. The transmission of characters is sufficient to avoid the time-out.

Parameters

timeOut	Timeout value in ms (Range: 1 - 100000), 0 disables time-out

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

getTimeout() `Int32 getTimeout ( ) [inline], [inherited]`

Note

The time-out value is a minimum value only and the actual time may be longer. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

Returns

Timeout value in ms

getTotalCounter() `Int32 getTotalCounter ( ) [inline], [inherited]`

Returns

Counter value

resetTotalCounter() `void resetTotalCounter ( )  [inline], [inherited]`

getSuccessCounter() `Int32 getSuccessCounter ( )  [inline], [inherited]`

Returns

Counter value

resetSuccessCounter() `void resetSuccessCounter ( )  [inline], [inherited]`

getPackageVersion() `static string getPackageVersion ( )  [inline], [static], [inherited]`

Returns

Version string

5.2.4 Member Data Documentation

SER_DATABITS_7 `const Int32 SER_DATABITS_7 = 7  [inherited]`

SER_DATABITS_8 `const Int32 SER_DATABITS_8 = 8  [inherited]`

SER_STOPBITS_1 `const Int32 SER_STOPBITS_1 = 1  [inherited]`

SER_STOPBITS_2 `const Int32 SER_STOPBITS_2 = 2  [inherited]`

SER_PARITY_NONE `const Int32 SER_PARITY_NONE = 0  [inherited]`

SER_PARITY_ODD `const Int32 SER_PARITY_ODD = 1  [inherited]`

SER_PARITY_EVEN `const Int32 SER_PARITY_EVEN = 2 [inherited]`

5.2.5 Property Documentation

portName `string portName [get], [set], [inherited]`

Note

This property must be set before starting the server in order to come into effect.

Serial port identifier (eg "COM1")

baudRate `Int32 baudRate [get], [set], [inherited]`

Note

This property must be set before starting the server in order to come into effect.

Typically 1200 - 115200, maximum value depends on UART hardware

dataBits `Int32 dataBits [get], [set], [inherited]`

Note

This property must be set before starting the server in order to come into effect.

SER_DATABITS_7 as 7 data bits (ASCII protocol only), SER_DATABITS_8 as data bits

stopBits `Int32 stopBits [get], [set], [inherited]`

Note

The Modbus standard requires two stop bits if no parity is chosen. This library is not enforcing this but it is a recommended configuration.
This property must be set before starting the server in order to come into effect.

SER_STOPBITS_1 as 1 stop bit, SER_STOPBITS_2 as 2 stop bits

parity `Int32 parity [get], [set], [inherited]`

Note

This property must be set before starting the server in order to come into effect.

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

timeout `Int32 timeout [get], [set], [inherited]`

Configures operation or socket time-out.

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

Note

This property must be set before starting the server in order to come into effect.

Timeout value in ms (Range: 1 - 100000)

5.3 MbusTcpSlaveProtocol Class Reference

This class realises the server side of the MODBUS/TCP slave protocol.

Public Member Functions

- MbusTcpSlaveProtocol ()
  Instantiates a MODBUS/TCP protocol server object.
- System.Int32 startupServer ()
  Puts the Modbus server into operation.
- System.Int32 startupServer (string hostName)
  Puts the Modbus server into operation.
- System.Int32 setPort (Int32 portNo)
  Sets the TCP port number to be used by the protocol.
- Int32 getPort ()
  Returns the TCP port number used by the protocol.
- void Dispose ()
  Implement IDisposable.
- int addDataTable (int slaveAddr, MbusDataTableInterface dataTable)
  Associates a protocol object with a Data Provider and a Modbus slave ID.
- int serverLoop ()
  Modbus slave server loop
- bool isStarted ()
  Returns whether server has been started up.
- void shutdownServer ()
  Shuts down the Modbus Server.
- bool getConnectionStatus ()
  Checks if a Modbus master is polling periodically.
- Int32 setTimeout (Int32 timeOut)
  Configures master transmit time-out supervision.
- Int32 getTimeout ()
  Returns the currently set master time-out supervision value.

- Int32 getTotalCounter ()

  Returns how often a message transfer has been executed.
- void resetTotalCounter ()

  Resets total message transfer counter.
- Int32 getSuccessCounter ()

  Returns how often a message transfer was successful.
- void resetSuccessCounter ()

  Resets successful message transfer counter.

Static Public Member Functions

- static string getPackageVersion ()

  Returns the library version number.

Properties

- string hostName `[get, set]`

  Host name of server interface
- Int32 port `[get, set]`

  TCP port number to be used by the protocol
- Int32 timeout `[get, set]`

  Time-out port property

5.3.1 Detailed Description

It provides functions to start-up and to execute the server engine. This server engine can handle multiple master connections and is implemented as a single threaded TCP server. Upon receipt of a valid master query the server engine calls Data Provider methods to exchange data with the user application.

For a more detailed description which Modbus data and control functions have been implemented in the server engine see section `Server Functions common to all Protocol Flavours`.

See also

MbusDataTableInterface

5.3.2 Constructor & Destructor Documentation

MbusTcpSlaveProtocol() `MbusTcpSlaveProtocol ( ) [inline]`

The association with a Data Provider is done after construction using the addDataTable method.

Exceptions

OutOfMemoryException	Creation of class failed

5.3.3 Member Function Documentation

startupServer() `[1/2]` `System.Int32 startupServer ( ) [inline]`

This function opens a TCP/IP socket, binds the configured TCP port to the Modbus/TCP protocol and initialises the server engine.

If the hostName property is set, the server accepts connections only on the interfaces which match it. This allows to run different servers on multiple interfaces (so called multihomed servers).

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

startupServer() `[2/2]`

```
System.Int32 startupServer (
    string hostName ) [inline]
```

This function opens a TCP/IP socket, binds the configured TCP port to the Modbus/TCP protocol and initialises the server engine.

The server accepts connections only on the interfaces which match the supplied hostname or IP address parameter. This method allows to run different servers on multiple interfaces (so called multihomed servers).

Parameters

hostName	String with IP address for a specific host interface or empty string if connections are accepted on any interface

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

setPort()

```
System.Int32 setPort (
    Int32 portNo ) [inline]
```

Note

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* starting the server with startupServer(). This parameter must be set before starting the server in order to come into effect.

Parameters

portNo	Port number the server shall listen on

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

getPort() `Int32 getPort ( ) [inline]`

Returns

Port number used by the protocol

Dispose() `void Dispose ( ) [inline], [inherited]`

Shuts down server, closes any associated communication resources (serial ports or sockets) and releases any resources.

addDataTable()
```
int addDataTable (
        int slaveAddr,
        MbusDataTableInterface dataTable ) [inline], [inherited]
```

Parameters

slaveAddr	Modbus slave address for server to listen on (-1 - 255). 0 is regarded as a valid value for a MODBUS/TCP server address. A value of -1 means the server disregards the slave address and listens to all slave addresses. 0 or -1 is only valid for MODBUS/TCP!
dataTable	Reference to a Modbus data table. Must be a Data Provider object derived from the MbusDataTableInterface class. The Data Provider is the interface between your application data and the Modbus network.

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

serverLoop() `int serverLoop ( ) [inline], [inherited]`

This server loop must be called continuously. It must not be blocked. The server has to be started before calling the serverLoop() method.

In most cases the server loop is executed in an infinite loop in its own thread:

```
while (notTerminated)
{
    result = mbusProtocol.serverLoop();
}
```

```
Do While notTerminated
    result = mbusProtocol.serverLoop()
Loop
```

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

isStarted() `bool isStarted ( ) [inline], [inherited]`

Returns

true = started
false = shutdown

shutdownServer() `void shutdownServer ( ) [inline], [inherited]`

This function also closes any associated serial ports or sockets.

getConnectionStatus() `bool getConnectionStatus ( ) [inline], [inherited]`

Note

The master transmit time-out value must be set $>$ 0 in order for this function to work.

Returns

true = A master is polling at a frequency higher than the master transmit time-out value
false = No master is polling within the time-out period

setTimeout() `Int32 setTimeout ( Int32 timeOut ) [inline], [inherited]`

Configures master transmit time-out supervision. The slave can monitor whether a master is actually transmitting characters or not. This function sets the transmit time-out to the specified value. A value of 0 disables the time-out, which stops time-out notifications being sent to the Data Provider.

Note

The time-out value is a minimum value only and the actual time may be longer. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.
The time-out does not check whether a master is sending valid frames. The transmission of characters is sufficient to avoid the time-out.

Parameters

timeOut	Timeout value in ms (Range: 1 - 100000), 0 disables time-out

Returns

BusProtocolErrors.FTALK_SUCCESS on success or error code. See BusProtocolErrors for possible error codes.

getTimeout() `Int32 getTimeout ( ) [inline], [inherited]`

Note

The time-out value is a minimum value only and the actual time may be longer. How precise it is followed depends on the operating system used, its scheduling priority and its system timer resolution.

Returns

Timeout value in ms

getTotalCounter() `Int32 getTotalCounter ( )  [inline], [inherited]`

Returns

Counter value

resetTotalCounter() `void resetTotalCounter ( )  [inline], [inherited]`

getSuccessCounter() `Int32 getSuccessCounter ( )  [inline], [inherited]`

Returns

Counter value

resetSuccessCounter() `void resetSuccessCounter ( )  [inline], [inherited]`

getPackageVersion() `static string getPackageVersion ( )  [inline], [static], [inherited]`

Returns

Version string

5.3.4 Property Documentation

hostName `string hostName  [get], [set]`

Setting this property is conly required for multihomed servers. If set, the server accepts connections only on the interfaces which match it, instead on any interface. This allows to run different servers on multiple interfaces.

Note

This parameter must be set before starting the server in order to come into effect.

String with IP address or host name (eg "127.0.0.1")

port `Int32 port [get], [set]`

Usually the port number remains unchanged and defaults to 502.

Note

This parameter must be set before starting the server in order to come into effect.

TCP Port number of slave device. Default value is 502.

timeout `Int32 timeout [get], [set], [inherited]`

Configures operation or socket time-out.

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

Note

This property must be set before starting the server in order to come into effect.

Timeout value in ms (Range: 1 - 100000)

5.4 MbusDataTableInterface Class Reference

This class defines the interface between a Modbus slave Server Engine and your application.

Data Access Methods for Table 4:00000 (Holding Registers)

Data Access Methods to support read and write of output registers (holding registers) in table 4:00000.

This table is accessed by the following Modbus functions:

- Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers
- Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers
- Modbus function 6 (06 hex), Preset Single Register/Write Single Register.
- Modbus function 22 (16 hex), Mask Write Register.
- Modbus function 23 (17 hex), Read/Write Registers.

- virtual bool readHoldingRegistersTable (int startRef, [In, Out] short[ ] regArr)

  Override this method to implement a Data Provider function to read Holding Registers.
- virtual bool writeHoldingRegistersTable (int startRef, short[ ] regArr)

  Override this method to implement a Data Provider function to write Holding Registers.

- virtual bool readEnronRegistersTable (int startRef, [In, Out] int[ ] regArr)

  Implement this function only if your slave device has to process register ranges as Daniel/↩ ENRON 32-bit registers.
- virtual bool writeEnronRegistersTable (int startRef, int[ ] regArr)

  Implement this function only if your slave device has to process register ranges as Daniel/↩ ENRON 32-bit registers.

Data Access Methods for Table 3:00000 (Input Registers)

Data Access Methods to support read of input registers in table 3:00000.

This table is accessed by the following Modbus functions:

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

Note

Input registers cannot be written

- virtual bool readInputRegistersTable (int startRef, [In, Out] short[ ] regArr)

  Override this method to implement a Data Provider function to read Input Registers.

Data Access Methods for Table 0:00000 (Coils)

Data Access Methods to support read and write of discrete outputs (coils) in table 0↩ :00000.

This table is accessed by the following Modbus functions:

- Modbus function 1 (01 hex), Read Coil Status/Read Coils.
- Modbus function 5 (05 hex), Force Single Coil/Write Coil.
- Modbus function 15 (0F hex), Force Multiple Coils.

- virtual bool readCoilsTable (int startRef, [In, Out] bool[ ] bitArr)

  Override this method to implement a Data Provider function to read Coils.
- virtual bool writeCoilsTable (int startRef, bool[ ] bitArr)

  Override this method to implement a Data Provider function to write Coils.

Data Access Methods for Table 1:00000 (Input Discretes)

Data Access Methods to support read discrete inputs (input status) in table 1:00000.

This table is accessed by the following Modbus functions:

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

Note

Input Discretes cannot be written

- virtual bool readInputDiscretesTable (int startRef, [In, Out] bool[ ] bitArr)

  Override this method to implement a Data Provider function to read Coils.

Data Access Methods for File Records

File record data is accessed using the following Modbus functions:

- Modbus function 20 (13 hex), Read File Records.
- Modbus function 21 (15 hex), Write File Records.

- virtual bool readFileRecord (int refType, int fileNo, int startRef, [In, Out] short[ ] regArr)

  Override this method to implement a Data Provider function to read File Records which is Modbus function code 20 (14 hex).
- virtual bool writeFileRecord (int refType, int fileNo, int startRef, short[ ] regArr)

  Override this method to implement a Data Provider function to write File Records which is Modbus function code 21 (15 hex).

Auxiliary Functions

- virtual byte readExceptionStatus ()

  Override this method to implement a function with reports the eight exception status coils (bits) within the slave device.
- virtual string reportSlaveId ()

  Override this method to implement a function which reports the Slave ID.
- virtual bool reportRunIndicatorStatus ()

  Override this method to implement a function which reports the Run Indicator of a device.
- virtual string readDeviceIdentification (int objId)

  Override this method to implement Read Device Identification objects to support Modbus function 43 (hex 2B) subfunction 14 (hex 0E).
- virtual void timeOutHandler ()

  Override this method to implement a function to handle master poll time-outs.

5.4.1 Detailed Description

Descendants of this class are referred to as Data Providers.

To create an application specific Data Provider derive a new class from MbusDataTableInterface and override the required data access methods.

See also

MbusTcpSlaveProtocol, MbusRtuSlaveProtocol, MbusAsciiSlaveProtocol

5.4.2 Member Function Documentation

readHoldingRegistersTable() `virtual bool readHoldingRegistersTable (`
`int` *`startRef,`*
`[In, Out] short [] regArr )  [inline], [protected], [virtual]`

When a slave receives a poll request for the 4:00000 data table it calls this method to retrieve the data.

Required: Yes

Default Implementation: Returns false which indicates to Server Engine that this address range is unsupported.

A simple implementation which holds the application data in an array of Int16 called localRegisters could be:

```
protected override int readHoldingRegistersTable(Int32 startRef, Int16[] regArr)
{
    // Adjust Modbus reference counting from 1-based to 0-based
    startRef&ndash;;
    // Validate range
    if (startRef + regArr.Length > localRegisters.Length)
        return false;
    // Copy registers from local data array to Modbus
    for (int i = 0; i < regArr.Length; i++)
        regArr[i] = localRegisters[startRef + i];
    return true;
}
```

```
Protected Overrides Function readHoldingRegistersTable(ByVal startRef As Int32,
      ByVal regArr() As Int16) As Integer
    Dim i As Integer

    ' Adjust Modbus reference counting from 1-based to 0-based
    startRef = startRef - 1
    ' Validate range
    If startRef + regArr.Length > localRegisters.Length Then
        Return False
    End If
    ' Copy registers from local data array to Modbus
    For i = 0 To regArr.Length - 1
        regArr(i) = localRegisters(startRef + i)
    Next
    Return True
End Function
```

Parameters

startRef	Start register (Range: 1 - 65536)
regArr	Buffer which has to be filled with the reply data (Length: 0 - 125)

Returns

true indicates a successful access and that valid reply data is contained in regArr. The Server Engine will reply the data passed in regArr to the master.
false indicates that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

writeHoldingRegistersTable() `virtual bool writeHoldingRegistersTable (`
`int` *`startRef,`*
`short [] regArr )` `[inline], [protected], [virtual]`

When a slave receives a write request for the 4:00000 data table it calls this method to pass the data to the application.

Required: Yes

Default Implementation: Returns false which indicates to Server Engine that this address range is unsupported.

A simple implementation which holds the application data in an array of Int16 called localRegisters could be:

```
protected override int writeHoldingRegistersTable(Int32 startRef, Int16[] regArr)
{
    // Adjust Modbus reference counting from 1-based to 0-based
    startRef&ndash;;
    // Validate range
    if (startRef + regArr.Length > localRegisters.Length)
        return false;
    // Copy registers from Modbus to local data block
    for (int i = 0; i < regArr.Length; i++)
        localRegisters[startRef + i] = regArr[i];
    return true;
}
```

```
Protected Overrides Function writeHoldingRegistersTable(ByVal startRef As Int32,
      ByVal regArr() As Int16) As Integer
    Dim i As Integer
    ' Adjust Modbus reference counting from 1-based to 0-based
    startRef = startRef - 1
    ' Validate range
    If startRef + regArr.Length > localRegisters.Length Then
        Return False
    End If
    ' Copy registers from Modbus to local data block
    For i = 0 To regArr.Length - 1
        localRegisters(startRef + i) = regArr(i)
    Next
    Return True
End Function
```

Parameters

startRef	Start register (Range: 1 - 65536)
regArr	Buffer which contains the received data (Length: 0 - 125)

Returns

true indicates a successful access. The Server Engine will send a positive reply to the master.
false indicates that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

readEnronRegistersTable() `virtual bool readEnronRegistersTable (`
`int` *startRef*`,`
`[In, Out] int [] regArr ) [inline], [protected], [virtual]`

If a register range is processed as Daniel/ENRON register then this range is not available as normal Holding Register range.

Required: Yes

Default Implementation: Returns 0 which indicates that the requested register range is processed as standard Modbus registers by a subsequent call to readHoldingRegistersTable().

Note: Daniel/ENRON is a prorietary 32-bit format which uses a non-standard Modbus frame and not understood by most master devices.

Parameters

startRef	Start register (Range: 1 - 65536)
regArr	Buffer which has to be filled with the reply data (Length: 0 - 62)

Returns

true indicates a successful access and that valid reply data is contained in regArr. The Server Engine will reply the data passed in regArr to the master using the Daniel/ENRON frame format.
false indicates that the requested range is not to be processed using the Daniel/ENRON frame format. A subsequent call to readHoldingRegistersTable() will be made to process the range in standard Modbus frame format.

writeEnronRegistersTable() `virtual bool writeEnronRegistersTable (`
`int` *startRef*`,`
`int [] regArr ) [inline], [protected], [virtual]`

If a register range is processed as Daniel/ENRON register then this range is not available as normal Holding Register range.

Required: No

Default Implementation: Returns 0 which indicates that the requested register range is processed as standard Modbus registers by a subsequent call to writeHoldingRegistersTable().

Note: Daniel/ENRON is a prorietary 32-bit format which uses a non-standard Modbus frame and not understood by most master devices.

Parameters

startRef	Start register (Range: 1 - 65536)
regArr	Buffer which contains the received data (Length: 0 - 62)

Returns

true indicates a successful access. The Server Engine will send a positive reply to the master.
false indicates that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

readInputRegistersTable() `virtual bool readInputRegistersTable (`
`int` *`startRef,`*
`[In, Out] short [] regArr ) [inline], [protected], [virtual]`

When a slave receives a poll request for the 3:00000 data table it calls this method to retrieve the data.

Required: Yes

Default Implementation: Returns false which indicates to Server Engine that this address range is unsupported.

A simple and very common implementation is to map the Input Registers to the same address space than the Holding Registers table:

```
protected override int readInputRegistersTable(int startRef, short[] regArr)
{
   return readHoldingRegistersTable(startRef, regArr);
}
```

```
Protected Overrides Function readInputRegistersTable(ByVal startRef As Integer,
      ByVal regArr() As Short) As Integer
    Return readHoldingRegistersTable(startRef, regArr)
End Function
```

Parameters

startRef	Start register (Range: 1 - 65536)
regArr	Buffer which has to be filled with the reply data (Length: 0 - 125)

Returns

true indicates a successful access and that valid reply data is contained in regArr. The Server Engine will reply the data passed in regArr to the master.
false indicates that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

readCoilsTable() `virtual bool readCoilsTable ( int startRef, [In, Out] bool [] bitArr ) [inline], [protected], [virtual]`

When a slave receives a poll request for the 0:00000 data table it calls this method to retrieve the data.

Required: No

Default Implementation: Returns false which indicates to Server Engine that this address range is unsupported.

A simple implementation which holds the boolean application data in an array of bool called bitData could be:

```
protected override int readCoilsTable(Int32 startRef, bool[] bitArr)
{
    // Adjust Modbus reference counting from 1-based to 0-based
    startRef&ndash;;
    // Validate range
    if (startRef + bitArr.Length > localCoils.Length)
        return false;
    // Copy registers from local data array to Modbus
    for (int i = 0; i < bitArr.Length; i++)
        bitArr[i] = localCoils[startRef + i];
    return true;
}
```

```
Protected Overrides Function readCoilsTable(ByVal startRef As Int32, ByVal bitArr() As
      Boolean) As Integer
    ' Adjust Modbus reference counting from 1-based to 0-based
    startRef -= 1
    ' Validate range
    If startRef + bitArr.Length > localCoils.Length Then
     Return False
    End If
    ' Copy registers from local data array to Modbus
    For i As Integer = 0 To bitArr.Length - 1
     bitArr(i) = localCoils(startRef + i)
    Next i
    Return True
End Function
```

Parameters

startRef	Start register (Range: 1 - 65536)
bitArr	Buffer which has to be filled with the reply data (Length: 0 - 2000). Each char represents one coil!

Returns

true indicates a successful access and that valid reply data is contained in regArr. The Server Engine will reply the data passed in regArr to the master.
false indicates that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

writeCoilsTable() `virtual bool writeCoilsTable ( int startRef, bool [] bitArr ) [inline], [protected], [virtual]`

When a slave receives a write request for the 0:00000 data table it calls this method to pass the data to the application.

Required: No

Default Implementation: Returns false which indicates to Server Engine that this address range is unsupported.

A simple implementation which holds the boolean application data in an array of bool call bitData could be:

```
protected override int writeCoilsTable(Int32 startRef, bool[] bitArr)
{
    // Adjust Modbus reference counting from 1-based to 0-based
    startRef&ndash;;
    // Validate range
    if (startRef + bitArr.Length > localCoils.Length)
        return false;
    // Copy registers from Modbus to local data block
    for (int i = 0; i < bitArr.Length; i++)
        localCoils[startRef + i] = bitArr[i];
    return true;
}
```

```
Protected Overrides Function writeCoilsTable(ByVal startRef As Int32, ByVal bitArr() As
      Boolean) As Integer
    ' Adjust Modbus reference counting from 1-based to 0-based
    startRef -= 1
    ' Validate range
    If startRef + bitArr.Length > localCoils.Length Then
        Return False
    End If
    ' Copy registers from Modbus to local data block
    For i As Integer = 0 To bitArr.Length - 1
        localCoils(startRef + i) = bitArr(i)
    Next i
    Return True
End Function
```

Parameters

startRef	Start register (Range: 1 - 65536)
bitArr	Buffer which contains the received data (Length: 0 - 2000). Each char repesents one coil!

Returns

true indicates a successful access. The Server Engine will send a positive reply to the master.
false indicates that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

readInputDiscretesTable() `virtual bool readInputDiscretesTable ( int startRef, [In, Out] bool [] bitArr ) [inline], [protected], [virtual]`

When a slave receives a poll request for the 0:00000 data table it calls this method to retrieve the data.

Required: No

Default Implementation: Returns false which indicates to Server Engine that this address range is unsupported.

A simple and very common implementation is to map the Input Discretes to the same address space than the Coils table:

```
protected override int readInputDiscretesTable(int startRef, bool[] bitArr)
{
   return readCoilsTable(startRef, bitArr);
}
```

```
Protected Overrides Function readInputDiscretesTable(ByVal startRef As Integer,
      ByVal bitArr() As Boolean) As Integer
    Return readCoilsTable(startRef, bitArr)
End Function
```

Parameters

startRef	Start register (Range: 1 - 65536)
bitArr	Buffer which has to be filled with the reply data (Length: 0 - 2000). Each char repesents one discrete!

Returns

true indicates a successful access and that valid reply data is contained in regArr. The Server Engine will reply the data passed in regArr to the master.
false indicates that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

readFileRecord() `virtual bool readFileRecord ( int refType, int fileNo,`

```
        int startRef,
        [In, Out] short [] regArr )  [inline], [protected], [virtual]
```

When a slave receives a poll request for function code 20 it calls this method to retrieve the data.

Required: No

Default Implementation: Returns false which indicates to Server Engine that this address range is unsupported.

Parameters

refType	Reference type (typically this is 6)
fileNo	File number (typically 0, 1, 3 or 4)
startRef	Record Number (equivalent to the start register)
regArr	Buffer which has to be filled with the reply data (Length: 0 - 125)

Returns

true indicates a successful access and that valid reply data is contained in regArr. The Server Engine will reply the data passed in regArr to the master.
false indicates that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

writeFileRecord()

```
virtual bool writeFileRecord (
        int refType,
        int fileNo,
        int startRef,
        short [] regArr )  [inline], [protected], [virtual]
```

When a slave receives a write request for function code 21 it calls this method to pass the data to the application.

Required: No

Default Implementation: Returns false which indicates to Server Engine that this address range is unsupported.

Parameters

refType	Reference type (typically this is 6)
fileNo	File number (typically 0, 1, 3 or 4)
startRef	Record Number (equivalent to the start register)
regArr	Buffer which contains the received data

Returns

true indicates a successful access. The Server Engine will send a positive reply to the master.
false indicates that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

readExceptionStatus() `virtual byte readExceptionStatus ( ) [inline], [protected], [virtual]`

The exception status coils are device specific and usually used to report a device' principal status or a device' major failure codes as a 8-bit word.

Required: No

Default Implementation: Returns 0 as exception status byte.

Returns

Exception status byte

reportSlaveId() `virtual string reportSlaveId ( ) [inline], [protected], [virtual]`

This is called when function code 17 (11 hex) Report Slave ID is sent by a master.

The Slave ID is a device-specific ASCII string.

Required: No

Default Implementation: Returns null which sends a slave failure exception reply

Note: The Slave ID is not to be mistaken for the Modbus Slave Address. The Slave ID is a string vs the Slave Address is a numeric value and the latter is used by a master device to address a specific device.

Returns

A string with the slave ID

reportRunIndicatorStatus() `virtual bool reportRunIndicatorStatus ( ) [inline], [protected], [virtual]`

This is called when function code 17 (11 hex) Report Slave ID is sent by a master.

Required: No

Default Implementation: Returns false (OFF) as run indicator status

Returns

Run Indicator status, true for ON, false for OFF

readDeviceIdentification() `virtual string readDeviceIdentification ( int objId ) [inline], [protected], [virtual]`

This function allows a master to retrieve various objects with meta information about a slave device. The objects are returned as ASCII string.

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

Required: No

Default Implementation: Returns null which causes an unsupported ID exception reply

Parameters

objId	objId ID number (0x00 - 0xFF)

Returns

Requested Device ID string object

timeOutHandler() `virtual void timeOutHandler ( ) [inline], [protected], [virtual]`

A master should poll a slave cyclically. If no master is polling within the time-out period this method is called. A slave can take certain actions if the master has lost connection, e.g. go into a fail-safe state.

Required: No

Default Implementation: Empty

6 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.

---

7 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.

8 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.