FieldTalk Modbus Slave C++ Library
Software manual

Library version 2.10.0

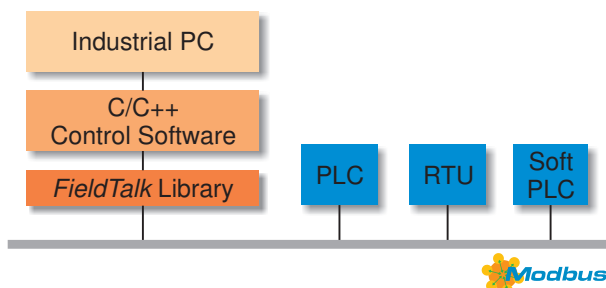# Contents

1   Introduction

This *FieldTalk*™ Modbus® Slave C++ Library allows you to incorporate Modbus slave functionality into your own 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

- File Record read & write functions

- Support of Broadcasting

- Master time-out supervision

- Failure and transmission counters

- Supports single or multiple slave addresses

1.1   Library Structure

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.
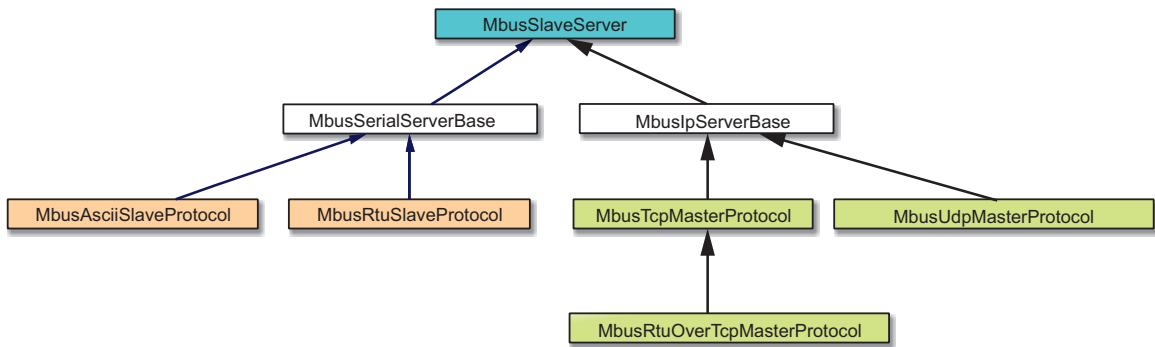
The second category of classes is Data Providers classes. Data Provider classes represent the interface between the Server Engine and your application.
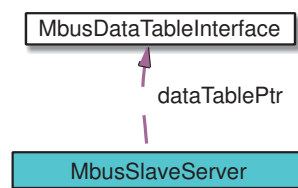
The base class MbusSlaveServer contains a protocol unspecific contains a protocol unspecific Server Engine and the protocol state machine. All protocol flavours inherit from this base class.

The class MbusAsciiSlaveProtocol implements the Modbus ASCII protocol, the class MbusRtuSlaveProtocol implements the Modbus RTU protocol and the class MbusTcpSlaveProtocol 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 MbusDataTableInterface 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.

```
class MyMbusDataTable: public MbusDataTableInterface
{
    ... // Application specific data interface
} dataTable;
```

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:

```
MbusRtuSlaveProtocol mbusProtocol;
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.

```
while (1)
{
    mbusProtocol.serverLoop();
}
```

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. Field↩ Talk 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 Field↩Talk functions. The register table is selected by choosing the corresponding function call as the following table illustrates.

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

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

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

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

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

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

2.2.6 The ASCII Protocol

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

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

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

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

2.2.7 The RTU Protocol

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

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

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

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

2.2.8 The MODBUS/TCP Protocol

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

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

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

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

3 Installation and Source Code Compilation

3.1 Windows Platforms: Unpacking and Preparation

1. Download and save the zip archive into a project directory.

2. Uncompress the archive using unzip or another zip tool of your choice:

```
# unzip FT-MBSV-WIN-ALL.2.8.0.zip
```

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

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

3. The library is ready to be used.

4. Optionally re-compile from source:

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

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

Platform	Library Directory
Windows 64-bit, Visual Studio 2019	lib\win\x64\Release
Windows 32-bit, Visual Studio 2019	lib\win\Win32\Release
Windows CE, Visual Studio 2015 or 2013	lib\wce\$(Platform)\Release

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

1. Download and save the zipped tarball into your project directory.

2. Uncompress the zipped tarball using gzip:

```
# gunzip FT-MBSV-??-ALL.2.8.0.tar.gz
```

3. Untar the tarball

```
# tar xf FT-MBSV-??-ALL.2.8.0.tar
```

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

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

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

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

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

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

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

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

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

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

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

Your directory structure looks now like:

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

6. The library is ready to be used.

3.3 Specific Platform Notes

3.3.1 VxWorks

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

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

4 Linking your Applications against the Library

4.1 Windows Platforms: Compiling and Linking Applications

Let's assume the following project directory structure:

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

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

Visual Studio Example:
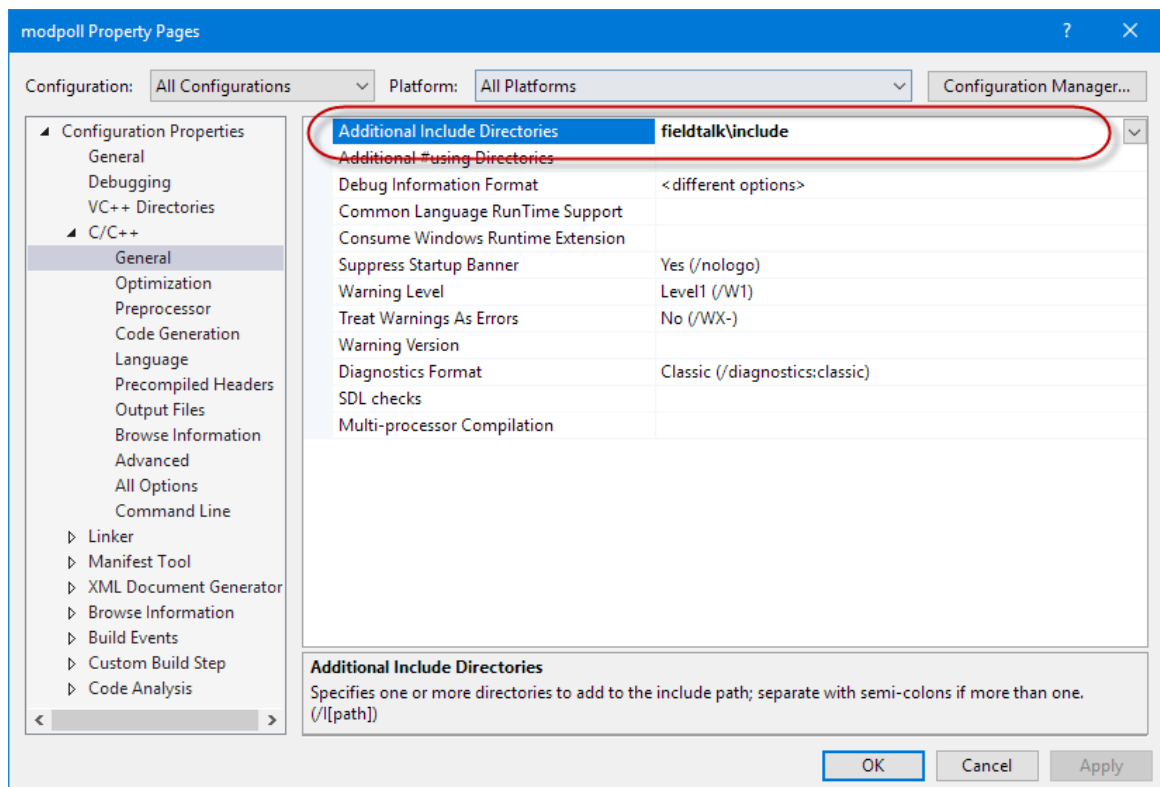
Visual C++ command line Example:

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

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

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

Visual Studio Example:
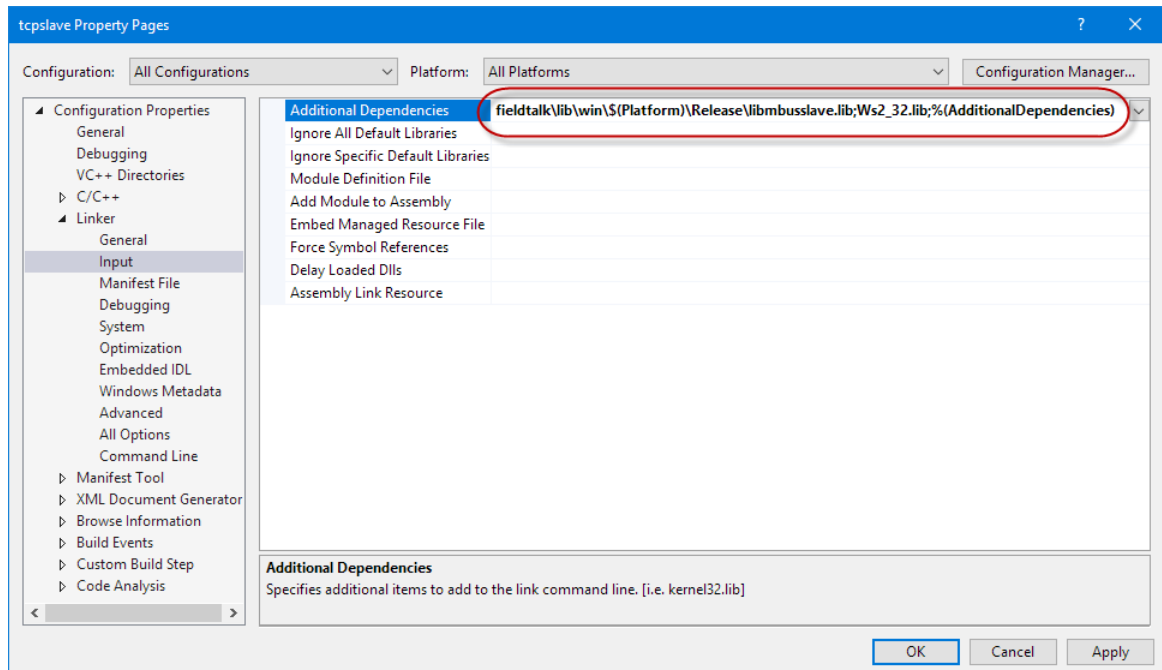
Visual C++ command line Example:

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

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

Let's assume the following project directory structure:

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

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

Example:

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

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

Example:

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

5 How to integrate the Protocol in your Application

5.1 Using Serial Protocols

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

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

1. Include the package header files

```
#include "MbusRtuSlaveProtocol.hpp"
```

2. Device data profile definition

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

```
short readRegSet[20];
short writeRegSet[20];
char readBitSet[10];
char writeBitSet[10];
```

3. Declare a Data Provider

```
class MyDataProvider: public MbusDataTableInterface
{

  public:

    int readHoldingRegistersTable(int startRef, short regArr[], int refCnt)
    {
      // Adjust Modbus reference counting
      startRef--;

      // Our start address for reading is at 100, so deduct offset
      startRef -= 100;

      // Validate range
      if (startRef + refCnt > (int) sizeof(readRegSet) / sizeof(short))
        return (0);

      // Copy data
      memcpy(regArr, &readRegSet[startRef], refCnt * sizeof(short));
      return (1);
    }


    int writeHoldingRegistersTable(int startRef,
                                   const short regArr[],
                                   int refCnt)
    {
      // Adjust Modbus reference counting
      startRef--;

      // Our start address for writing is at 200, so deduct offset
      startRef -= 200;
```

```c
        // Validate range
        if (startRef + refCnt > (int) sizeof(writeRegSet) / sizeof(short))
            return (0);

        // Copy data
        memcpy(&writeRegSet[startRef], regArr, refCnt * sizeof(short));
        return (1);
    }


    int readCoilsTable(int startRef,
                       char bitArr[],
                       int refCnt)
    {
        // Adjust Modbus reference counting
        startRef--;

        // Our start address for reading is at 10, so deduct offset
        startRef -= 10;

        // Validate range
        if (startRef + refCnt > (int) sizeof(readBitSet) / sizeof(char))
            return (0);

        // Copy data
        memcpy(bitArr, &readBitSet[startRef], refCnt * sizeof(char));
        return (1);
    }


    int writeCoilsTable(int startRef,
                        const char bitArr[],
                        int refCnt)
    {
        // Adjust Modbus reference counting
        startRef--;

        // Our start address for writing is at 20, so deduct offset
        startRef -= 20;

        // Validate range
        if (startRef + refCnt > (int) sizeof(writeBitSet) / sizeof(char))
            return (0);

        // Copy data
        memcpy(&writeBitSet[startRef], bitArr, refCnt * sizeof(char));
        return (1);
    }

} dataProvider;
```

4. Declare and instantiate a server object and associate it with the Data Provider

```c
MbusRtuSlaveProtocol mbusServer;
mbusServer.addDataTable(1, &dataProvider);
```

5. Start-up the server

```c
int result;

result = mbusServer.startupServer(portName,
                                  19200L, // Baudrate
                                  8,      // Databits
                                  1,      // Stopbits
                                  2);     // Even parity
if (result != FTALK_SUCCESS)
{
    fprintf(stderr, "Error starting server: %s!\n",
```

```
                    getBusProtocolErrorText(result));
        exit(EXIT_FAILURE);
}
```

6. Execute cyclically the server loop

```
int result = FTALK_SUCCESS;

while (result == FTALK_SUCCESS)
{
   result = mbusServer.serverLoop();
   if (result != FTALK_SUCCESS)
      fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));
}
```

7. Shutdown the server if not needed any more

```
mbusServer.shutdownServer();
```

5.2   Using MODBUS/TCP Protocol

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

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

1. Include the package header files

```
#include "MbusTcpSlaveProtocol.hpp"
```

2. Device data profile definition

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

```
short readRegSet[20];
short writeRegSet[20];
char readBitSet[10];
char writeBitSet[10];
```

3. Declare a Data Provider

```
class MyDataProvider: public MbusDataTableInterface
{

  public:

    int readHoldingRegistersTable(int startRef, short regArr[], int refCnt)
    {
       // Adjust Modbus reference counting
       startRef--;

       // Our start address for reading is at 100, so deduct offset
       startRef -= 100;

       // Validate range
       if (startRef + refCnt > (int) sizeof(readRegSet) / sizeof(short))
```

```
        return (0);

    // Copy data
    memcpy(regArr, &readRegSet[startRef], refCnt * sizeof(short));
    return (1);
}


int writeHoldingRegistersTable(int startRef,
                               const short regArr[],
                               int refCnt)
{
    // Adjust Modbus reference counting
    startRef--;

    // Our start address for writing is at 200, so deduct offset
    startRef -= 200;

    // Validate range
    if (startRef + refCnt > (int) sizeof(writeRegSet) / sizeof(short))
        return (0);

    // Copy data
    memcpy(&writeRegSet[startRef], regArr, refCnt * sizeof(short));
    return (1);
}


int readCoilsTable(int startRef,
                   char bitArr[],
                   int refCnt)
{
    // Adjust Modbus reference counting
    startRef--;

    // Our start address for reading is at 10, so deduct offset
    startRef -= 10;

    // Validate range
    if (startRef + refCnt > (int) sizeof(readBitSet) / sizeof(char))
        return (0);

    // Copy data
    memcpy(bitArr, &readBitSet[startRef], refCnt * sizeof(char));
    return (1);
}


int writeCoilsTable(int startRef,
                    const char bitArr[],
                    int refCnt)
{
    // Adjust Modbus reference counting
    startRef--;

    // Our start address for writing is at 20, so deduct offset
    startRef -= 20;

    // Validate range
    if (startRef + refCnt > (int) sizeof(writeBitSet) / sizeof(char))
        return (0);

    // Copy data
    memcpy(&writeBitSet[startRef], bitArr, refCnt * sizeof(char));
    return (1);
}

} dataProvider;
```

4. Declare and instantiate a server object and associate it with the Data Provider and the slave address.

```
MbusTcpSlaveProtocol mbusServer();
mbusServer.addDataTable(1, &dataProvider);
```

5. Change the default port from 502 to something else if server shall not run as root. This step is not necessary when the server can run with root privilege.

```
mbusServer.setPort(5000);
```

6. Start-up the server

```
int result;

result = mbusServer.startupServer();
if (result != FTALK_SUCCESS)
{
    fprintf(stderr, "Error starting server: %s!\n",
            getBusProtocolErrorText(result));
    exit(EXIT_FAILURE);
}
```

7. Execute cyclically the server loop

```
int result = FTALK_SUCCESS;

while (result == FTALK_SUCCESS)
{
    result = mbusServer.serverLoop();
    if (result != FTALK_SUCCESS)
        fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));
}
```

8. Shutdown the server if not needed any more

```
mbusServer.shutdownServer();
```

6 Design Background

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

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

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

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

7 Module Documentation

7.1 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 Bit Access	Current Terminology	Classic Terminology
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	

7.2 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.

7.2.1 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.

See section Using Serial Protocols for an example how to use the MbusRtuSlaveProtocol and MbusAsciiSlaveProtocol class.

7.3 IP based Protocols

Classes

- class MbusTcpSlaveProtocol

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

  This class realises the Encapsulated Modbus RTU slave protocol.
- class MbusUdpSlaveProtocol

  This class realises a Modbus server using MODBUS over UDP protocol variant.

Macros

- #define MAX_CONNECTIONS 32

  Maximum concurrent TCP/IP connections handled by server engine.

7.3.1 Detailed Description

The library provides several flavours of IP based Modbus protocols.

The Server Engine of the MODBUS/TCP slave protocol is implemented in the class MbusTcpSlaveProtocol and is the only IP based protocol officially specified by the Modbus organisation.

In addition to MODBUS/TCP, the library offers implementations of the serial protocol RTU transported over TCP streams. It is implemented in the class MbusRtuOverTcpSlaveProtocol.

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

All classes provide functions to start-up and to execute the respective server engines. The server engines can handle multiple master connections and are implemented as single threaded TCP servers. 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.

Note

> If the configured TCP port is below IPPORT_RESERVED (usually 1024), the process has to run with root privilege! This applies if you are using the default MODBUS/TCP port 502.

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

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

7.3.2   Macro Definition Documentation

MAX_CONNECTIONS   `#define MAX_CONNECTIONS 32`

Maximum concurrent TCP/IP connections handled by server engine.

This can be increased by re-defining this preprocessor macro (from the comannd line compiler using `-DMAX_CONNECTIONS=64`).

7.4   Data Provider

Classes

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

7.4.1   Detailed Description

A Data Provider acts as an agent between your Application and the Server Engine.
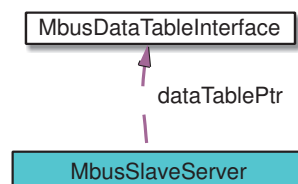
After instantiating a Server Engine class of any protocol flavour, you have to associate it with a Data Provider by calling addDataTable and passing a pointer to the Data Provider object.

```
MbusRtuSlaveProtocol mbusProtocol;
mbusProtocol.addDataTable(1, &dataTable);
```

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

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

```
class MyDataProvider: public MbusDataTableInterface
{
  public:

  MyDataProvider() {}

  // Override readHoldingRegistersTable method:
  int readHoldingRegistersTable(int startRef, short regArr[], int refCnt)
  {
      ... your application specific implementation
  }
};
```

7.5   Error Management

Macros

- #define FTALK_SUCCESS 0

  Operation was successful.

- #define FTALK_ILLEGAL_ARGUMENT_ERROR 1

  Illegal argument error.

- #define FTALK_ILLEGAL_STATE_ERROR 2

  Illegal state error.

- #define FTALK_EVALUATION_EXPIRED 3

  Evaluation expired.

- #define FTALK_NO_DATA_TABLE_ERROR 4

  No data table configured.

- #define FTALK_ILLEGAL_SLAVE_ADDRESS 5

  Slave address 0 illegal for serial protocols.

- #define FTALK_INSUFFICIENT_BUFFER 6

  Size of response buffer insufficient.

Functions

- const TCHAR * getBusProtocolErrorText (int errCode)

  Translates a numeric error code into a description string.

Fatal I/O Errors

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

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

- #define FTALK_IO_ERROR_CLASS 64

  I/O error class.
- #define FTALK_IO_ERROR 65

  I/O error.
- #define FTALK_OPEN_ERR 66

  Port or socket open error.
- #define FTALK_PORT_ALREADY_OPEN 67

  Serial port already open.
- #define FTALK_TCPIP_CONNECT_ERR 68

  TCP/IP connection error.
- #define FTALK_CONNECTION_WAS_CLOSED 69

  Remote peer closed TCP/IP connection.
- #define FTALK_SOCKET_LIB_ERROR 70

  Socket library error.
- #define FTALK_PORT_ALREADY_BOUND 71

  TCP port already bound.
- #define FTALK_LISTEN_FAILED 72

  Listen failed.
- #define FTALK_FILEDES_EXCEEDED 73

  File descriptors exceeded.
- #define FTALK_PORT_NO_ACCESS 74

  No permission to access serial port or TCP port.
- #define FTALK_PORT_NOT_AVAIL 75

  TCP port not available.
- #define FTALK_LINE_BUSY_ERROR 76

  Serial line busy/noisy.

Communication Errors

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

- #define FTALK_BUS_PROTOCOL_ERROR_CLASS 128

  Fieldbus protocol error class.
- #define FTALK_CHECKSUM_ERROR 129

  Checksum error.
- #define FTALK_INVALID_FRAME_ERROR 130

  Invalid frame error.
- #define FTALK_INVALID_REPLY_ERROR 131

Invalid reply error.

- #define FTALK_REPLY_TIMEOUT_ERROR 132

    Reply time-out.

- #define FTALK_SEND_TIMEOUT_ERROR 133

    Send time-out.

- #define FTALK_INVALID_MBAP_ID 134

    Invalid MPAB indentifer.

- #define FTALK_LINE_ERROR 135

    Serial line error.

- #define FTALK_OVERRUN_ERROR 136

    Serial buffer overrun.

- #define FTALK_MBUS_EXCEPTION_RESPONSE 160

    Modbus exception response.

- #define FTALK_MBUS_ILLEGAL_FUNCTION_RESPONSE 161

    Illegal Function exception response.

- #define FTALK_MBUS_ILLEGAL_ADDRESS_RESPONSE 162

    Illegal Data Address exception response.

- #define FTALK_MBUS_ILLEGAL_VALUE_RESPONSE 163

    Illegal Data Value exception response.

- #define FTALK_MBUS_SLAVE_FAILURE_RESPONSE 164

    Slave Device Failure exception response.

- #define FTALK_MBUS_GW_PATH_UNAVAIL_RESPONSE 170

    Gateway Path Unavailable exception response.

- #define FTALK_MBUS_GW_TARGET_FAIL_RESPONSE 171

    Gateway Target Device Failed exception response.

7.5.1   Detailed Description

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

7.5.2   Macro Definition Documentation

FTALK_SUCCESS   `#define FTALK_SUCCESS 0`

Operation was successful.

This return codes indicates no error.

FTALK_ILLEGAL_ARGUMENT_ERROR   `#define FTALK_ILLEGAL_ARGUMENT_ERROR 1`

Illegal argument error.

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

FTALK_ILLEGAL_STATE_ERROR `#define FTALK_ILLEGAL_STATE_ERROR 2`

Illegal state error.

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

FTALK_EVALUATION_EXPIRED `#define FTALK_EVALUATION_EXPIRED 3`

Evaluation expired.

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

FTALK_NO_DATA_TABLE_ERROR `#define FTALK_NO_DATA_TABLE_ERROR 4`

No data table configured.

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

FTALK_ILLEGAL_SLAVE_ADDRESS `#define FTALK_ILLEGAL_SLAVE_ADDRESS 5`

Slave address 0 illegal for serial protocols.

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

FTALK_INSUFFICIENT_BUFFER `#define FTALK_INSUFFICIENT_BUFFER 6`

Size of response buffer insufficient.

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

FTALK_IO_ERROR_CLASS `#define FTALK_IO_ERROR_CLASS 64`

I/O error class.

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

FTALK_IO_ERROR `#define FTALK_IO_ERROR 65`

I/O error.

The underlaying I/O system reported an error.

FTALK_OPEN_ERR `#define FTALK_OPEN_ERR 66`

Port or socket open error.

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

FTALK_PORT_ALREADY_OPEN `#define FTALK_PORT_ALREADY_OPEN 67`

Serial port already open.

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

FTALK_TCPIP_CONNECT_ERR `#define FTALK_TCPIP_CONNECT_ERR 68`

TCP/IP connection error.

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

FTALK_CONNECTION_WAS_CLOSED `#define FTALK_CONNECTION_WAS_CLOSED 69`

Remote peer closed TCP/IP connection.

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

FTALK_SOCKET_LIB_ERROR `#define FTALK_SOCKET_LIB_ERROR 70`

Socket library error.

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

FTALK_PORT_ALREADY_BOUND `#define FTALK_PORT_ALREADY_BOUND 71`

TCP port already bound.

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

FTALK_LISTEN_FAILED `#define FTALK_LISTEN_FAILED 72`

Listen failed.

The listen operation on the specified TCP port failed..

FTALK_FILEDES_EXCEEDED `#define FTALK_FILEDES_EXCEEDED 73`

File descriptors exceeded.

Maximum number of usable file descriptors exceeded.

FTALK_PORT_NO_ACCESS `#define FTALK_PORT_NO_ACCESS 74`

No permission to access serial port or TCP port.

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

FTALK_PORT_NOT_AVAIL `#define FTALK_PORT_NOT_AVAIL 75`

TCP port not available.

The specified TCP port is not available on this machine.

FTALK_LINE_BUSY_ERROR `#define FTALK_LINE_BUSY_ERROR 76`

Serial line busy/noisy.

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

FTALK_BUS_PROTOCOL_ERROR_CLASS `#define FTALK_BUS_PROTOCOL_ERROR_CLASS 128`

Fieldbus protocol error class.

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

FTALK_CHECKSUM_ERROR `#define FTALK_CHECKSUM_ERROR 129`

Checksum error.

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

FTALK_INVALID_FRAME_ERROR `#define FTALK_INVALID_FRAME_ERROR 130`

Invalid frame error.

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

FTALK_INVALID_REPLY_ERROR `#define FTALK_INVALID_REPLY_ERROR 131`

Invalid reply error.

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

FTALK_REPLY_TIMEOUT_ERROR `#define FTALK_REPLY_TIMEOUT_ERROR 132`

Reply time-out.

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

FTALK_SEND_TIMEOUT_ERROR `#define FTALK_SEND_TIMEOUT_ERROR 133`

Send time-out.

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

FTALK_INVALID_MBAP_ID `#define FTALK_INVALID_MBAP_ID 134`

Invalid MPAB indentifer.

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

FTALK_LINE_ERROR `#define FTALK_LINE_ERROR 135`

Serial line error.

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

FTALK_OVERRUN_ERROR `#define FTALK_OVERRUN_ERROR 136`

Serial buffer overrun.

More characters have been received then expected.

FTALK_MBUS_EXCEPTION_RESPONSE `#define FTALK_MBUS_EXCEPTION_RESPONSE 160`

Modbus exception response.

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

FTALK_MBUS_ILLEGAL_FUNCTION_RESPONSE `#define FTALK_MBUS_ILLEGAL_FUNCTION_RESPONSE 161`

Illegal Function exception response.

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

FTALK_MBUS_ILLEGAL_ADDRESS_RESPONSE `#define FTALK_MBUS_ILLEGAL_ADDRESS_RESPONSE 162`

Illegal Data Address exception response.

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

FTALK_MBUS_ILLEGAL_VALUE_RESPONSE `#define FTALK_MBUS_ILLEGAL_VALUE_RESPONSE 163`

Illegal Data Value exception response.

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

FTALK_MBUS_SLAVE_FAILURE_RESPONSE `#define FTALK_MBUS_SLAVE_FAILURE_RESPONSE 164`

Slave Device Failure exception response.

Signals that a Slave Device Failure exception response (code 04) was received. This exception response is sent by a slave device instead of a normal response message if an unrecoverable error 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.

FTALK_MBUS_GW_PATH_UNAVAIL_RESPONSE `#define FTALK_MBUS_GW_PATH_UNAVAIL_RESPONSE 170`

Gateway Path Unavailable exception response.

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

FTALK_MBUS_GW_TARGET_FAIL_RESPONSE `#define FTALK_MBUS_GW_TARGET_FAIL_RESPONSE 171`

Gateway Target Device Failed exception response.

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

7.5.3   Function Documentation

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

Translates a numeric error code into a description string.

Parameters

errCode	FieldTalk error code

Returns

Error text string

8   C++ Class Documentation

8.1   MbusRtuSlaveProtocol Class Reference

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

Public Types

- enum { SER_DATABITS_7 = 7, SER_DATABITS_8 = 8 }
- enum { SER_STOPBITS_1 = 1, SER_STOPBITS_2 = 2 }
- enum { SER_PARITY_NONE = 0, SER_PARITY_EVEN = 2, SER_PARITY_ODD = 1 }

Public Member Functions

- MbusRtuSlaveProtocol ()

    Instantiates a Modbus RTU protocol server object.

- int startupServer (const TCHAR ∗const portName, long baudRate, int dataBits, int stop↩
  Bits, int parity)

    Puts the Modbus RTU server into operation and opens the associated serial port with specific port parameters.

- int serverLoop ()

    Modbus slave server loop.

- int setFrameTolerance (long frameToleranceMs)

    Configures the tolerance the RTU protocol engine should apply for the detection of inter-frame gaps.

- void shutdownServer ()

    Shuts down the Modbus server.

- int isStarted ()

    Returns whether server has been started up.

- virtual int enableRs485Mode (int rtsDelay)

    Enables RS485 mode.

- int addDataTable (int slaveAddr, MbusDataTableInterface ∗dataTablePtr)

    Associates a protocol object with a Data Provider and a Modbus slave ID.

- int getConnectionStatus ()

    Checks if a Modbus master is polling periodically.

Static Public Member Functions

- static const TCHAR ∗ getPackageVersion ()

    Returns the library version number.

Protocol Configuration

- int setTimeout (long timeOut)

  Configures master activity time-out supervision.
- long getTimeout ()

  Returns the currently set master activity time-out value.
- void disableExceptionReplies ()

  Supress exception replies to be sent.
- void enableExceptionReplies ()

  Enables exception replies after they have been turned off.

Transmission Statistic Functions

- unsigned long getTotalCounter ()

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

  Resets total message transfer counter.
- unsigned long getSuccessCounter ()

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

  Resets successful message transfer counter.

8.1.1   Detailed Description

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

It 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 Modbus 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

Server Functions common to all Modbus Protocol Flavours, Serial Protocols MbusDataTableInterface, MbusAsciiSlaveProtocol, MbusTcpSlaveProtocol

8.1.2   Member Enumeration Documentation

anonymous enum `anonymous enum  [inherited]`

Enumerator

SER_DATABITS↩_7	7 data bits
SER_DATABITS↩_8	8 data bits

anonymous enum `anonymous enum` `[inherited]`

Enumerator

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

anonymous enum `anonymous enum` `[inherited]`

Enumerator

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

8.1.3 Constructor & Destructor Documentation

MbusRtuSlaveProtocol() `MbusRtuSlaveProtocol ( )`

Instantiates a Modbus RTU protocol server object.

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

References MasterInfo::protocol, and MasterInfo::RTU.

8.1.4 Member Function Documentation

startupServer()   `int startupServer (`

                 `const TCHAR *const portName,`

                 `long baudRate,`

                 `int dataBits,`

                 `int stopBits,`

                 `int parity ) [virtual]`

Puts the Modbus RTU server into operation and opens the associated serial port with specific port parameters.

This function opens the serial port and initialises the server engine.

Parameters

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

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.

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Reimplemented from MbusSerialServerBase.

References FTALK_ILLEGAL_ARGUMENT_ERROR, MbusSerialServerBase::SER_DATABITS←
_8, and MbusSerialServerBase::startupServer().

serverLoop()   `int serverLoop ( ) [virtual]`

Modbus slave server loop.

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) {
  mbusProtocol.serverLoop();
}
```

Returns

> FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Implements MbusSlaveServer.

References FTALK_IO_ERROR, and FTALK_SUCCESS.

---

setFrameTolerance() `int setFrameTolerance (`
`                long frameToleranceMs )`

Configures the tolerance the RTU protocol engine should apply for the detection of inter-frame gaps.

Modbus RTU uses inter-frame gaps (silence periods) to mark start and end of Modbus packets.

Warning

> For strict Modbus compliance this should be set to 0.

Remarks

> There are cases were longer silence periods may appear in the data stream due to the hardware used (USB dongles) or the device driver or the scheduling of the operating system. This value can then be increased to compensate. In a point-to-point communication system (eg RS-232) this value can be safley increased. However in a multi-point system were other slave devices are on the line, this value should be set to the lowest possible value in order to maintain interoperability with other Modbus equipment.

Note

> A protocol must be closed in order to configure it.

Parameters

frameToleranceMs	Additional tolerance time for inter-frame silence detection in ms (Range: 0 - 20)

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_ARGUMENT_ERR↩ OR	Argument out of range

References FTALK_ILLEGAL_ARGUMENT_ERROR, FTALK_ILLEGAL_STATE_ERROR, FTALK_↩
SUCCESS, and MbusSerialServerBase::isStarted().

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

Shuts down the Modbus server.

This function also closes any associated communication resources like serial ports or sockets.

Implements MbusSlaveServer.

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

Returns whether server has been started up.

Return values

true	= started
false	= shutdown

Implements MbusSlaveServer.

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

Enables RS485 mode.

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

Warning

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

Remarks

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

Note

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

Parameters

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

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_ARGUMENT_ERR↩ OR	Argument out of range
FTALK_ILLEGAL_STATE_ERROR	Protocol is already open

References FTALK_ILLEGAL_ARGUMENT_ERROR, FTALK_ILLEGAL_STATE_ERROR, FTALK_↩ SUCCESS, and MbusSerialServerBase::isStarted().

addDataTable()   `int addDataTable (`
                    `int slaveAddr,`
                    `MbusDataTableInterface * dataTablePtr )  [inherited]`

Associates a protocol object with a Data Provider and a Modbus slave ID.

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!
dataTablePtr	Modbus data table pointer. Must point to a Data Provider object derived from the MbusDataTableInterface class. The Data Provider is the interface between your application data and the Modbus network.

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

References FTALK_ILLEGAL_ARGUMENT_ERROR, and FTALK_SUCCESS.

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

Checks if a Modbus master is polling periodically.

Return values

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

Note

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

setTimeout() `int setTimeout (`
`long timeOut )  [inherited]`

Configures master activity time-out supervision.

The slave can monitor whether a master is actually polling or not. This function sets the activity 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. Default value is 1000 ms.

Remarks

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.

Note

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: 0 - 100000), 0 disables time-out

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_ARGUMENT_ERR↩ *OR*	Argument out of range

References FTALK_ILLEGAL_ARGUMENT_ERROR, and FTALK_SUCCESS.

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

Returns the currently set master activity time-out value.

Returns

Timeout value in ms

disableExceptionReplies() `void disableExceptionReplies ( )  [inherited]`

Supress exception replies to be sent.

With this option only positive replies are sent to the master. All failure replies are silently discarded. This option can be useful if redundant Modbus devices are used. In this scenario supressing the reply would trigger a swap-over of the redundant devices.

enableExceptionReplies() `void enableExceptionReplies ( )  [inherited]`

Enables exception replies after they have been turned off.

Sending exception replies in case of slave failures is the normal mode of operation.

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

Returns how often a message transfer has been executed.

Returns

Counter value

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

Returns how often a message transfer was successful.

Returns

Counter value

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

> Returns the library version number.

> **Returns**

>> Version string

8.2 MbusAsciiSlaveProtocol Class Reference

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

Public Types

- enum { SER_DATABITS_7 = 7, SER_DATABITS_8 = 8 }
- enum { SER_STOPBITS_1 = 1, SER_STOPBITS_2 = 2 }
- enum { SER_PARITY_NONE = 0, SER_PARITY_EVEN = 2, SER_PARITY_ODD = 1 }

Public Member Functions

- MbusAsciiSlaveProtocol ()

  Instantiates a Modbus ASCII protocol server object.

- int serverLoop ()

  Modbus slave server loop.

- virtual int startupServer (const char ∗const portName, long baudRate, int dataBits, int stopBits, int parity)

  Puts the Modbus server into operation.

- void shutdownServer ()

  Shuts down the Modbus server.

- int isStarted ()

  Returns whether server has been started up.

- virtual int enableRs485Mode (int rtsDelay)

  Enables RS485 mode.

- int addDataTable (int slaveAddr, MbusDataTableInterface ∗dataTablePtr)

  Associates a protocol object with a Data Provider and a Modbus slave ID.

- int getConnectionStatus ()

  Checks if a Modbus master is polling periodically.

Static Public Member Functions

- static const TCHAR ∗ getPackageVersion ()

  Returns the library version number.

Protocol Configuration

- int setTimeout (long timeOut)

  Configures master activity time-out supervision.
- long getTimeout ()

  Returns the currently set master activity time-out value.
- void disableExceptionReplies ()

  Supress exception replies to be sent.
- void enableExceptionReplies ()

  Enables exception replies after they have been turned off.

Transmission Statistic Functions

- unsigned long getTotalCounter ()

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

  Resets total message transfer counter.
- unsigned long getSuccessCounter ()

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

  Resets successful message transfer counter.

8.2.1   Detailed Description

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

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 Modbus 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

Server Functions common to all Modbus Protocol Flavours, Serial Protocols MbusDataTableInterface, MbusRtuSlaveProtocol, MbusTcpSlaveProtocol

8.2.2   Member Enumeration Documentation

anonymous enum `anonymous enum  [inherited]`

Enumerator

SER_DATABITS↩_7	7 data bits
SER_DATABITS↩_8	8 data bits

anonymous enum `anonymous enum [inherited]`

Enumerator

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

anonymous enum `anonymous enum [inherited]`

Enumerator

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

8.2.3   Constructor & Destructor Documentation

MbusAsciiSlaveProtocol() `MbusAsciiSlaveProtocol ( )`

Instantiates a Modbus ASCII protocol server object.

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

References MasterInfo::ASCII, and MasterInfo::protocol.

8.2.4   Member Function Documentation

serverLoop() `int serverLoop ( )  [virtual]`

Modbus slave server loop.

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) {
  mbusProtocol.serverLoop();
}
```

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Implements MbusSlaveServer.

References FTALK_IO_ERROR, and FTALK_SUCCESS.

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

Puts the Modbus server into operation.

This function opens the serial port. After the port has been opened queries from a Modbus master will be processed.

Parameters

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

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.

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Reimplemented in MbusRtuSlaveProtocol.

References FTALK_ILLEGAL_ARGUMENT_ERROR, FTALK_ILLEGAL_SLAVE_ADDRESS, FTA↩
LK_ILLEGAL_STATE_ERROR, FTALK_NO_DATA_TABLE_ERROR, FTALK_OPEN_ERR, FTALK↩
_PORT_ALREADY_OPEN, FTALK_PORT_NO_ACCESS, FTALK_SUCCESS, and MbusSerial↩
ServerBase::isStarted().

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

Shuts down the Modbus server.

This function also closes any associated communication resources like serial ports or sockets.

Implements MbusSlaveServer.

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

Returns whether server has been started up.

Return values

true	= started
false	= shutdown

Implements MbusSlaveServer.

enableRs485Mode()   `int enableRs485Mode (`

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

Enables RS485 mode.

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

Warning

The use of RTS controlled RS232/RS485 converters should be avoided if possible. It is difficult to determine the exact time when to switch off the transmitter with non real-time operating systems like Windows and Linux. If it is switched off to early

characters might still sit in the FIFO or the transmit register of the UART and these characters will be lost. Hence the slave will not recognize the message. On the other hand if it is switched off too late then the slave's message is corrupted and the master will not recognize the message.

Remarks

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

Note

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

Parameters

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

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_ARGUMENT_ERR↩OR	Argument out of range
FTALK_ILLEGAL_STATE_ERROR	Protocol is already open

References FTALK_ILLEGAL_ARGUMENT_ERROR, FTALK_ILLEGAL_STATE_ERROR, FTALK_↩SUCCESS, and MbusSerialServerBase::isStarted().

addDataTable() `int addDataTable (`
              `int slaveAddr,`
              `MbusDataTableInterface * dataTablePtr ) [inherited]`

Associates a protocol object with a Data Provider and a Modbus slave ID.

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!
dataTablePtr	Modbus data table pointer. Must point to a Data Provider object derived from the MbusDataTableInterface class. The Data Provider is the interface between your application data and the Modbus network.

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

References FTALK_ILLEGAL_ARGUMENT_ERROR, and FTALK_SUCCESS.

---

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

Checks if a Modbus master is polling periodically.

Return values

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

Note

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

---

setTimeout() `int setTimeout (`
`            long timeOut )  [inherited]`

Configures master activity time-out supervision.

The slave can monitor whether a master is actually polling or not. This function sets the activity 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. Default value is 1000 ms.

Remarks

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.

Note

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: 0 - 100000), 0 disables time-out

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_ARGUMENT_ERR↩ OR	Argument out of range

References FTALK_ILLEGAL_ARGUMENT_ERROR, and FTALK_SUCCESS.

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

Returns the currently set master activity time-out value.

> **Returns**
>
> > Timeout value in ms

disableExceptionReplies() `void disableExceptionReplies ( )  [inherited]`

Supress exception replies to be sent.

With this option only positive replies are sent to the master. All failure replies are silently discarded. This option can be useful if redundant Modbus devices are used. In this scenario supressing the reply would trigger a swap-over of the redundant devices.

enableExceptionReplies() `void enableExceptionReplies ( )  [inherited]`

Enables exception replies after they have been turned off.

Sending exception replies in case of slave failures is the normal mode of operation.

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

Returns how often a message transfer has been executed.

> **Returns**
>
> > Counter value

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

Returns how often a message transfer was successful.

> **Returns**
>
> > Counter value

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

Returns the library version number.

Returns

Version string

8.3 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.
- int startupServer ()

    Puts the Modbus server into operation.
- int startupServer (const char ∗const hostName)

    Puts the Modbus server into operation.
- void shutdownServer ()

    Shuts down the Modbus server.
- int serverLoop ()

    Modbus slave server loop.
- int setConnectionTimeOut (long masterTimeOut)

    Configures connection time-out.
- long getConnectionTimeOut ()

    Returns the connection time-out setting.
- void installMasterDisconnectCallBack (void(∗f)(const char ∗masterIpAddrSz))

    This function installs a callback handler to be called in the event of a master disconnection.
- int addDataTable (int slaveAddr, MbusDataTableInterface ∗dataTablePtr)

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

    Checks if a Modbus master is polling periodically.

Static Public Member Functions

- static const TCHAR ∗ getPackageVersion ()

    Returns the library version number.

Protected Member Functions

- int readHeader (int sockIdx)

  Sub-routine to read and validate the Modbus/TCP header.
- int readPayload (int sockIdx, int result)

  Sub-routine to read and validate the Modbus/TCP payload.
- void terminateSocket (int sockIdx)

  Sub-routine to manage termination of connection.

Protocol Configuration

- int setTimeout (long timeOut)

  Configures master activity time-out supervision.
- long getTimeout ()

  Returns the currently set master activity time-out value.
- void disableExceptionReplies ()

  Supress exception replies to be sent.
- void enableExceptionReplies ()

  Enables exception replies after they have been turned off.

Transmission Statistic Functions

- unsigned long getTotalCounter ()

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

  Resets total message transfer counter.
- unsigned long getSuccessCounter ()

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

  Resets successful message transfer counter.

TCP/IP Server Management Functions

- int isStarted ()

  Returns whether server has been started up.
- int setPort (unsigned short portNo)

  Sets the TCP port number to be used by the protocol.
- void installIpAddrValidationCallBack (int(∗f)(const char ∗masterIpAddrSz))

  This function installs a callback handler for validating a master's IP address.
- void installMasterPollNotifyCallBack (int(∗f)(const char ∗masterIpAddrSz))

  This function installs a callback handler to be called everytime a master polls this slave and allows a forced closure of the master connection by returning 0.
- unsigned short getPort ()

  Returns the TCP port number used by the protocol.

8.3.1   Detailed Description

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

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 T↩CP 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 seesection Server Functions common to all Modbus Protocol Flavours.

See also

Server Functions common to all Modbus Protocol Flavours, IP based Protocols
MbusDataTableInterface

8.3.2   Constructor & Destructor Documentation

MbusTcpSlaveProtocol() `MbusTcpSlaveProtocol ( )`

Instantiates a MODBUS/TCP protocol server object.

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

References MAX_CONNECTIONS, MasterInfo::protocol, and MasterInfo::TCP.

8.3.3   Member Function Documentation

startupServer() `[1/2]` `int startupServer ( )  [inline], [virtual]`

Puts the Modbus server into operation.

The server accepts connections on any interface.

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

Note

If the configured TCP port is below IPPORT_RESERVED (usually 1024), the process has to run with root privilege!

Returns

>FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Reimplemented from MbusIpServerBase.

References MbusIpServerBase::startupServer().

startupServer() [2/2]   `int startupServer (`
                `const char *const hostName ) [virtual]`

Puts the Modbus server into operation.

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

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

Note

>If the configured TCP port is below IPPORT_RESERVED (usually 1024), the process has to run with root privilege!

Parameters

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

Returns

>FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Implements MbusIpServerBase.

References FTALK_LISTEN_FAILED, FTALK_OPEN_ERR, FTALK_PORT_ALREADY_BOUND, F↩TALK_PORT_NO_ACCESS, FTALK_PORT_NOT_AVAIL, FTALK_SUCCESS, FTALK_TCPIP_CO↩NNECT_ERR, MAX_CONNECTIONS, and shutdownServer().

shutdownServer()   `void shutdownServer ( ) [virtual]`

Shuts down the Modbus server.

This function closes all TCP/IP connections to MODBUS/TCP masters and releases any system resources associated with the connections.

Reimplemented from MbusIpServerBase.

References MAX_CONNECTIONS, and MbusIpServerBase::shutdownServer().

serverLoop() `int serverLoop ( )  [virtual]`

Modbus slave server loop.

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) {
  mbusProtocol.serverLoop();
}
```

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Implements MbusIpServerBase.

Reimplemented in MbusRtuOverTcpSlaveProtocol.

References FTALK_FILEDES_EXCEEDED, FTALK_IO_ERROR, FTALK_SUCCESS, MAX_CON↩
NECTIONS, readHeader(), readPayload(), and terminateSocket().

setConnectionTimeOut() `int setConnectionTimeOut (`
                  `long masterTimeOut )`

Configures connection time-out.

This allows to detect broken TCP connections. The TCP connection is closed if a valid request is not received within the configured time period. A value of 0 disables the time-out. Default value is 0 (disabled).

Remarks

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.

Parameters

masterTime↩Out	Timeout value in ms (Range: 0 - 3600000), 0 disables time-out

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_ARGUMENT_ERR↩ OR	Argument out of range
FTALK_ILLEGAL_STATE_ERROR	Server already running

References FTALK_ILLEGAL_ARGUMENT_ERROR, FTALK_ILLEGAL_STATE_ERROR, FTALK_↩ SUCCESS, and MbusIpServerBase::isStarted().

getConnectionTimeOut() `long getConnectionTimeOut ( ) [inline]`

Returns the connection time-out setting.

Returns

Timeout value in ms

installMasterDisconnectCallBack() `void installMasterDisconnectCallBack (`
`void(*)(const char *masterIpAddrSz) f )`

This function installs a callback handler to be called in the event of a master disconnection.

This routine can be used to implement custom time-out mechanisms.

Parameters

f	Callback function pointer

readHeader() `int readHeader (`
`int sockIdx ) [protected]`

Sub-routine to read and validate the Modbus/TCP header.

Parameters

sockIdx	Index of socket within the connectionSocketArr

Returns

-1 for error, 0 for graceful socket disconnect or payload data length

readPayload() `int readPayload (`

                `int` *`sockIdx,`*

                `int` *`dataLen`* `)` `[protected]`

Sub-routine to read and validate the Modbus/TCP payload.

Parameters

sockIdx	Index of socket within the connectionSocketArr
dataLen	Length of the payload (read from MPAB header)

Returns

    -1 for error, 0 for graceful socket disconnect

References MasterInfo::transactionId.

terminateSocket() `void terminateSocket (`

                `int` *`sockIdx`* `)` `[protected]`

Sub-routine to manage termination of connection.

Parameters

sockIdx	Index of socket within the connectionSocketArr

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

Returns whether server has been started up.

Return values

true	= started
false	= shutdown

Implements MbusSlaveServer.

setPort() `int setPort (`

                `unsigned short` *`portNo`* `)` `[inherited]`

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

Remarks

Usually the port number remains unchanged and defaults to 502. In this case no call to this function is necessary. However if the port number has to be different from 502 this function must be called *before* starting the server with startupServer().

Note

If the configured TCP port is below IPPORT_RESERVED (usually 1024), the process has to run with root or administrator privilege!
This parameter must be set before starting the server in order to come into effect.

Parameters

port↩ No	Port number the server shall listen on

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_STATE_ERR↩ OR	Server already running

References FTALK_ILLEGAL_STATE_ERROR, FTALK_SUCCESS, and MbusIpServerBase::is↩Started().

---

installIpAddrValidationCallBack() `void installIpAddrValidationCallBack (`
`            int(*)(const char *masterIpAddrSz) f )  [inherited]`

This function installs a callback handler for validating a master's IP address.

Pass a pointer to a function with checks a master's IP address and either accepts or rejects a master's connection.

Parameters

f	Callback function pointer

Returns

Returns 1 to accept a connection or 0 to reject it.

---

installMasterPollNotifyCallBack() `void installMasterPollNotifyCallBack (`
`            int(*)(const char *masterIpAddrSz) f )  [inherited]`

This function installs a callback handler to be called everytime a master polls this slave and allows a forced closure of the master connection by returning 0.

This routine can be used to implement custom time-out mechanisms.

Parameters

f	Callback function pointer

Returns

Returns 1 to process the poll or 0 to reject and drop the connection.

---

getPort()   `unsigned short getPort ( ) [inline], [inherited]`

Returns the TCP port number used by the protocol.

Returns

Port number used by the protocol

---

addDataTable()   `int addDataTable (`
         `int slaveAddr,`
         `MbusDataTableInterface * dataTablePtr ) [inherited]`

Associates a protocol object with a Data Provider and a Modbus slave ID.

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!
dataTablePtr	Modbus data table pointer. Must point to a Data Provider object derived from the MbusDataTableInterface class. The Data Provider is the interface between your application data and the Modbus network.

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

References FTALK_ILLEGAL_ARGUMENT_ERROR, and FTALK_SUCCESS.

---

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

Checks if a Modbus master is polling periodically.

Return values

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

Note

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

setTimeout() `int setTimeout (`
                     `long timeOut )` `[inherited]`

Configures master activity time-out supervision.

The slave can monitor whether a master is actually polling or not. This function sets the activity 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. Default value is 1000 ms.

Remarks

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.

Note

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: 0 - 100000), 0 disables time-out

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_ARGUMENT_ERR↩ OR	Argument out of range

References FTALK_ILLEGAL_ARGUMENT_ERROR, and FTALK_SUCCESS.

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

Returns the currently set master activity time-out value.

Returns

Timeout value in ms

disableExceptionReplies() `void disableExceptionReplies ( ) [inherited]`

Supress exception replies to be sent.

With this option only positive replies are sent to the master. All failure replies are silently discarded. This option can be useful if redundant Modbus devices are used. In this scenario supressing the reply would trigger a swap-over of the redundant devices.

enableExceptionReplies() `void enableExceptionReplies ( ) [inherited]`

Enables exception replies after they have been turned off.

Sending exception replies in case of slave failures is the normal mode of operation.

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

Returns how often a message transfer has been executed.

Returns

Counter value

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

Returns how often a message transfer was successful.

Returns

Counter value

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

Returns the library version number.

Returns

Version string

8.4 MbusRtuOverTcpSlaveProtocol Class Reference

This class realises the Encapsulated Modbus RTU slave protocol.

Public Member Functions

- MbusRtuOverTcpSlaveProtocol ()

    Instantiates an Encapsulated Modbus RTU server object.
- int startupServer ()

    Puts the Modbus server into operation.
- int startupServer (const char ∗const hostName)

    Puts the Modbus server into operation.
- void shutdownServer ()

    Shuts down the Modbus server.
- int setConnectionTimeOut (long masterTimeOut)

    Configures connection time-out.
- long getConnectionTimeOut ()

    Returns the connection time-out setting.
- void installMasterDisconnectCallBack (void(∗f)(const char ∗masterIpAddrSz))

    This function installs a callback handler to be called in the event of a master disconnection.
- int addDataTable (int slaveAddr, MbusDataTableInterface ∗dataTablePtr)

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

    Checks if a Modbus master is polling periodically.

Static Public Member Functions

- static const TCHAR ∗ getPackageVersion ()

    Returns the library version number.

Protected Member Functions

- int readHeader (int sockIdx)

    Sub-routine to read and validate the Modbus/TCP header.
- int readPayload (int sockIdx, int result)

    Sub-routine to read and validate the Modbus/TCP payload.
- void terminateSocket (int sockIdx)

    Sub-routine to manage termination of connection.

Protocol Configuration

- int setTimeout (long timeOut)

    Configures master activity time-out supervision.
- long getTimeout ()

    Returns the currently set master activity time-out value.
- void disableExceptionReplies ()

    Supress exception replies to be sent.
- void enableExceptionReplies ()

    Enables exception replies after they have been turned off.

Transmission Statistic Functions

- unsigned long getTotalCounter ()

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

    Resets total message transfer counter.
- unsigned long getSuccessCounter ()

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

    Resets successful message transfer counter.

TCP/IP Server Management Functions

- int isStarted ()

    Returns whether server has been started up.
- int setPort (unsigned short portNo)

    Sets the TCP port number to be used by the protocol.
- void installIpAddrValidationCallBack (int(∗f)(const char ∗masterIpAddrSz))

    This function installs a callback handler for validating a master's IP address.
- void installMasterPollNotifyCallBack (int(∗f)(const char ∗masterIpAddrSz))

    This function installs a callback handler to be called everytime a master polls this slave and allows a forced closure of the master connection by returning 0.
- unsigned short getPort ()

    Returns the TCP port number used by the protocol.

TCP/IP Server Management Functions

- int serverLoop ()

    Modbus slave server loop.

8.4.1 Detailed Description

This class realises the Encapsulated Modbus RTU slave protocol.

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 T↩CP 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 seesection Server Functions common to all Modbus Protocol Flavours.

See also

Server Functions common to all Modbus Protocol Flavours, IP based Protocols MbusDataTableInterface, MbusRtuSlaveProtocol

8.4.2 Constructor & Destructor Documentation

MbusRtuOverTcpSlaveProtocol() `MbusRtuOverTcpSlaveProtocol ( )`

Instantiates an Encapsulated Modbus RTU server object.

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

References MasterInfo::protocol, and MasterInfo::RTU_OVER_TCP.

8.4.3 Member Function Documentation

serverLoop() `int serverLoop ( )  [virtual]`

Modbus slave server loop.

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) {
  mbusProtocol.serverLoop();
}
```

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Reimplemented from MbusTcpSlaveProtocol.

References FTALK_FILEDES_EXCEEDED, FTALK_IO_ERROR, FTALK_SUCCESS, and MAX_C↩
ONNECTIONS.

startupServer() [1/2] `int startupServer ( )  [inline], [virtual], [inherited]`

Puts the Modbus server into operation.

The server accepts connections on any interface.

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

Note

If the configured TCP port is below IPPORT_RESERVED (usually 1024), the process has to run with root privilege!

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Reimplemented from MbusIpServerBase.

References MbusIpServerBase::startupServer().

startupServer() [2/2] `int startupServer (`
`                const char *const hostName )  [virtual], [inherited]`

Puts the Modbus server into operation.

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

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

Note

If the configured TCP port is below IPPORT_RESERVED (usually 1024), the process has to run with root privilege!

Parameters

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

Returns

> FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Implements MbusIpServerBase.

References FTALK_LISTEN_FAILED, FTALK_OPEN_ERR, FTALK_PORT_ALREADY_BOUND, F↩ TALK_PORT_NO_ACCESS, FTALK_PORT_NOT_AVAIL, FTALK_SUCCESS, FTALK_TCPIP_CO↩ NNECT_ERR, MAX_CONNECTIONS, and MbusTcpSlaveProtocol::shutdownServer().

---

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

Shuts down the Modbus server.

This function closes all TCP/IP connections to MODBUS/TCP masters and releases any system resources associated with the connections.

Reimplemented from MbusIpServerBase.

References MAX_CONNECTIONS, and MbusIpServerBase::shutdownServer().

---

setConnectionTimeOut()   `int setConnectionTimeOut (`
        `long masterTimeOut ) [inherited]`

Configures connection time-out.

This allows to detect broken TCP connections. The TCP connection is closed if a valid request is not received within the configured time period. A value of 0 disables the time-out. Default value is 0 (disabled).

Remarks

> 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.

Parameters

masterTime↩Out	Timeout value in ms (Range: 0 - 3600000), 0 disables time-out

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_ARGUMENT_ERR↩ OR	Argument out of range
FTALK_ILLEGAL_STATE_ERROR	Server already running

References FTALK_ILLEGAL_ARGUMENT_ERROR, FTALK_ILLEGAL_STATE_ERROR, FTALK_↩ SUCCESS, and MbusIpServerBase::isStarted().

getConnectionTimeOut() `long getConnectionTimeOut ( )  [inline], [inherited]`

Returns the connection time-out setting.

Returns

Timeout value in ms

installMasterDisconnectCallBack() `void installMasterDisconnectCallBack (`
`void(*)(const char *masterIpAddrSz) f )  [inherited]`

This function installs a callback handler to be called in the event of a master disconnection.

This routine can be used to implement custom time-out mechanisms.

Parameters

f	Callback function pointer

readHeader() `int readHeader (`
`int sockIdx )  [protected], [inherited]`

Sub-routine to read and validate the Modbus/TCP header.

Parameters

sockIdx	Index of socket within the connectionSocketArr

Returns

-1 for error, 0 for graceful socket disconnect or payload data length

readPayload() `int readPayload (`
`int sockIdx,`
`int dataLen )  [protected], [inherited]`

Sub-routine to read and validate the Modbus/TCP payload.

Parameters

sockIdx	Index of socket within the connectionSocketArr
dataLen	Length of the payload (read from MPAB header)

Returns

-1 for error, 0 for graceful socket disconnect

References MasterInfo::transactionId.

---

terminateSocket() `void terminateSocket (`
`int sockIdx ) [protected], [inherited]`

Sub-routine to manage termination of connection.

Parameters

sockIdx	Index of socket within the connectionSocketArr

---

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

Returns whether server has been started up.

Return values

true	= started
false	= shutdown

Implements MbusSlaveServer.

---

setPort() `int setPort (`
`unsigned short portNo ) [inherited]`

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

Remarks

Usually the port number remains unchanged and defaults to 502. In this case no call to this function is necessary. However if the port number has to be different from 502 this function must be called *before* starting the server with startupServer().

If the configured TCP port is below IPPORT_RESERVED (usually 1024), the process has to run with root or administrator privilege!
This parameter must be set before starting the server in order to come into effect.

Parameters

port↩ *No*	Port number the server shall listen on

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_STATE_ERR↩ *OR*	Server already running

References FTALK_ILLEGAL_STATE_ERROR, FTALK_SUCCESS, and MbusIpServerBase::is↩Started().

installIpAddrValidationCallBack() `void installIpAddrValidationCallBack (`
`            int(*)(const char *masterIpAddrSz) f )  [inherited]`

This function installs a callback handler for validating a master's IP address.

Pass a pointer to a function with checks a master's IP address and either accepts or rejects a master's connection.

Parameters

f	Callback function pointer

Returns

Returns 1 to accept a connection or 0 to reject it.

installMasterPollNotifyCallBack() `void installMasterPollNotifyCallBack (`
`            int(*)(const char *masterIpAddrSz) f )  [inherited]`

This function installs a callback handler to be called everytime a master polls this slave and allows a forced closure of the master connection by returning 0.

This routine can be used to implement custom time-out mechanisms.

Parameters

f	Callback function pointer

Returns

Returns 1 to process the poll or 0 to reject and drop the connection.

---

getPort() `unsigned short getPort ( ) [inline], [inherited]`

Returns the TCP port number used by the protocol.

Returns

Port number used by the protocol

---

addDataTable() `int addDataTable (`
`                int slaveAddr,`
`                MbusDataTableInterface * dataTablePtr ) [inherited]`

Associates a protocol object with a Data Provider and a Modbus slave ID.

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!
dataTablePtr	Modbus data table pointer. Must point to a Data Provider object derived from the MbusDataTableInterface class. The Data Provider is the interface between your application data and the Modbus network.

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

References FTALK_ILLEGAL_ARGUMENT_ERROR, and FTALK_SUCCESS.

---

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

Checks if a Modbus master is polling periodically.

Return values

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

Note

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

setTimeout()　`int setTimeout (`

　　　　　　`long timeOut ) [inherited]`

Configures master activity time-out supervision.

The slave can monitor whether a master is actually polling or not. This function sets the activity 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. Default value is 1000 ms.

Remarks

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.

Note

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: 0 - 100000), 0 disables time-out

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_ARGUMENT_ERR↩ OR	Argument out of range

References FTALK_ILLEGAL_ARGUMENT_ERROR, and FTALK_SUCCESS.

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

Returns the currently set master activity time-out value.

Returns

Timeout value in ms

disableExceptionReplies() `void disableExceptionReplies ( )  [inherited]`

Supress exception replies to be sent.

With this option only positive replies are sent to the master. All failure replies are silently discarded. This option can be useful if redundant Modbus devices are used. In this scenario supressing the reply would trigger a swap-over of the redundant devices.

enableExceptionReplies() `void enableExceptionReplies ( )  [inherited]`

Enables exception replies after they have been turned off.

Sending exception replies in case of slave failures is the normal mode of operation.

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

Returns how often a message transfer has been executed.

Returns

Counter value

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

Returns how often a message transfer was successful.

Returns

Counter value

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

Returns the library version number.

Returns

Version string

8.5   MbusUdpSlaveProtocol Class Reference

This class realises a Modbus server using MODBUS over UDP protocol variant.

Public Member Functions

- MbusUdpSlaveProtocol ()

    Instantiates a MODBUS/UDP protocol server object.
- int startupServer ()

    Puts the Modbus server into operation.
- int startupServer (const char ∗const hostName)

    Puts the Modbus server into operation.
- void shutdownServer ()

    Shuts down the Modbus server.
- int serverLoop ()

    Modbus slave server loop.
- int addDataTable (int slaveAddr, MbusDataTableInterface ∗dataTablePtr)

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

    Checks if a Modbus master is polling periodically.

Static Public Member Functions

- static const TCHAR ∗ getPackageVersion ()

    Returns the library version number.

Protocol Configuration

- int setTimeout (long timeOut)

    Configures master activity time-out supervision.
- long getTimeout ()

    Returns the currently set master activity time-out value.
- void disableExceptionReplies ()

    Supress exception replies to be sent.
- void enableExceptionReplies ()

    Enables exception replies after they have been turned off.

Transmission Statistic Functions

- unsigned long getTotalCounter ()

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

Resets total message transfer counter.

- unsigned long getSuccessCounter ()

    Returns how often a message transfer was successful.

- void resetSuccessCounter ()

    Resets successful message transfer counter.

TCP/IP Server Management Functions

- int isStarted ()

    Returns whether server has been started up.

- int setPort (unsigned short portNo)

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

- void installIpAddrValidationCallBack (int(*f)(const char *masterIpAddrSz))

    This function installs a callback handler for validating a master's IP address.

- void installMasterPollNotifyCallBack (int(*f)(const char *masterIpAddrSz))

    This function installs a callback handler to be called everytime a master polls this slave and allows a forced closure of the master connection by returning 0.

- unsigned short getPort ()

    Returns the TCP port number used by the protocol.

8.5.1 Detailed Description

This class realises a Modbus server using MODBUS over UDP protocol variant.

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 U↩DP 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 seesection Server Functions common to all Modbus Protocol Flavours.

See also

Server Functions common to all Modbus Protocol Flavours, IP based Protocols MbusDataTableInterface

8.5.2 Constructor & Destructor Documentation

MbusUdpSlaveProtocol() `MbusUdpSlaveProtocol ( )`

Instantiates a MODBUS/UDP protocol server object.

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

References MasterInfo::protocol, and MasterInfo::UDP.

8.5.3   Member Function Documentation

startupServer() [1/2]   `int startupServer ( )  [inline], [virtual]`

Puts the Modbus server into operation.

The server accepts connections on any interface.

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

Note

If the configured TCP port is below IPPORT_RESERVED (usually 1024), the process has to run with root privilege!

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Reimplemented from MbusIpServerBase.

References MbusIpServerBase::startupServer().

startupServer() [2/2]   `int startupServer (`
                       `const char *const hostName )  [virtual]`

Puts the Modbus server into operation.

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

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

Note

If the configured TCP port is below IPPORT_RESERVED (usually 1024), the process has to run with root privilege!

Parameters

hostName	String with IP address for a specific host interface or NULL if connections are accepted on any interface. Multihomeing does not work for aliased IP addresses.

Returns

> FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Implements MbusIpServerBase.

References FTALK_OPEN_ERR, FTALK_PORT_ALREADY_BOUND, FTALK_PORT_NO_A↩CCESS, FTALK_PORT_NOT_AVAIL, FTALK_SUCCESS, FTALK_TCPIP_CONNECT_ERR, and shutdownServer().

shutdownServer()   `void shutdownServer ( ) [virtual]`

Shuts down the Modbus server.

This function closes the UDP listeners and releases any system resources associated with the connections.

Reimplemented from MbusIpServerBase.

References MbusIpServerBase::shutdownServer().

serverLoop()   `int serverLoop ( ) [virtual]`

Modbus slave server loop.

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) {
  mbusProtocol.serverLoop();
}
```

Returns

> FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

Implements MbusIpServerBase.

References FTALK_IO_ERROR, FTALK_SUCCESS, and MasterInfo::transactionId.

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

Returns whether server has been started up.

Return values

true	= started
false	= shutdown

Implements MbusSlaveServer.

setPort()   `int setPort (`

`unsigned short` *`portNo`* `)` `[inherited]`

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

Remarks

Usually the port number remains unchanged and defaults to 502. In this case no call to this function is necessary. However if the port number has to be different from 502 this function must be called *before* starting the server with startupServer().

Note

If the configured TCP port is below IPPORT_RESERVED (usually 1024), the process has to run with root or administrator privilege!
This parameter must be set before starting the server in order to come into effect.

Parameters

port↩ No	Port number the server shall listen on

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_STATE_ERR↩ OR	Server already running

References FTALK_ILLEGAL_STATE_ERROR, FTALK_SUCCESS, and MbusIpServerBase::is↩ Started().

installIpAddrValidationCallBack()   `void installIpAddrValidationCallBack (`

`int(*)(const char *masterIpAddrSz)` *`f`* `)` `[inherited]`

This function installs a callback handler for validating a master's IP address.

Pass a pointer to a function with checks a master's IP address and either accepts or rejects a master's connection.

Parameters

f	Callback function pointer

Returns 1 to accept a connection or 0 to reject it.

installMasterPollNotifyCallBack() `void installMasterPollNotifyCallBack (`
`    int(*)(const char *masterIpAddrSz) f ) [inherited]`

This function installs a callback handler to be called everytime a master polls this slave and allows a forced closure of the master connection by returning 0.

This routine can be used to implement custom time-out mechanisms.

Parameters

f	Callback function pointer

Returns

Returns 1 to process the poll or 0 to reject and drop the connection.

getPort() `unsigned short getPort ( ) [inline], [inherited]`

Returns the TCP port number used by the protocol.

Returns

Port number used by the protocol

addDataTable() `int addDataTable (`
`    int slaveAddr,`
`    MbusDataTableInterface * dataTablePtr ) [inherited]`

Associates a protocol object with a Data Provider and a Modbus slave ID.

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!
dataTablePtr	Modbus data table pointer. Must point to a Data Provider object derived from the MbusDataTableInterface class. The Data Provider is the interface between your application data and the Modbus network.

Returns

FTALK_SUCCESS on success or error code. See Error Management for a list of error codes.

References FTALK_ILLEGAL_ARGUMENT_ERROR, and FTALK_SUCCESS.

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

Checks if a Modbus master is polling periodically.

Return values

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

Note

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

setTimeout() `int setTimeout (`

`long timeOut ) [inherited]`

Configures master activity time-out supervision.

The slave can monitor whether a master is actually polling or not. This function sets the activity 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. Default value is 1000 ms.

Remarks

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.

Note

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: 0 - 100000), 0 disables time-out

Return values

FTALK_SUCCESS	Success
FTALK_ILLEGAL_ARGUMENT_ERR↩ OR	Argument out of range

References FTALK_ILLEGAL_ARGUMENT_ERROR, and FTALK_SUCCESS.

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

Returns the currently set master activity time-out value.

Returns

Timeout value in ms

disableExceptionReplies() `void disableExceptionReplies ( )  [inherited]`

Supress exception replies to be sent.

With this option only positive replies are sent to the master. All failure replies are silently discarded. This option can be useful if redundant Modbus devices are used. In this scenario supressing the reply would trigger a swap-over of the redundant devices.

enableExceptionReplies() `void enableExceptionReplies ( )  [inherited]`

Enables exception replies after they have been turned off.

Sending exception replies in case of slave failures is the normal mode of operation.

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

Returns how often a message transfer has been executed.

Returns

Counter value

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

Returns how often a message transfer was successful.

Returns

Counter value

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

Returns the library version number.

Returns

Version string

8.6 MbusDataTableInterface Interface 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 int readHoldingRegistersTable (int startRef, short regArr[ ], int refCnt)
  Override this method to implement a Data Provider function to read Holding Registers.
- virtual int writeHoldingRegistersTable (int startRef, const short regArr[ ], int refCnt)
  Override this method to implement a Data Provider function to write Holding Registers.
- virtual int readEnronRegistersTable (int startRef, long regArr[ ], int refCnt)
  Implement this function only if your slave device has to process register ranges as Daniel/↩ ENRON 32-bit registers.
- virtual int writeEnronRegistersTable (int startRef, const long regArr[ ], int refCnt)
  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.

Input registers cannot be written

- virtual int readInputRegistersTable (int startRef, short regArr[ ], int refCnt)

  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 int readCoilsTable (int startRef, char bitArr[ ], int refCnt)

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

  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 int readInputDiscretesTable (int startRef, char bitArr[ ], int refCnt)

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

Data Access Methods for File Records

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

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

- virtual int readFileRecord (int refType, int fileNo, int startRef, short regArr[ ], int refCnt)

Override this method to implement a Data Provider function to read File Records which is Modbus function code 20 (14 hex).

- virtual int writeFileRecord (int refType, int fileNo, int startRef, short regArr[ ], int refCnt)

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

Data Access for Status and Diagnostic Information

- virtual char readExceptionStatus ()

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

- virtual int reportSlaveId (char bufferArr[ ], int maxBufSize)

  Override this method to implement a function which reports the Slave ID.

- virtual int reportRunIndicatorStatus ()

  Override this method to implement a function which reports the Run Indicator of a device.

- virtual int readDeviceIdentification (int objId, char bufferArr[ ], int maxBufSize)

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

Auxiliary Functions

- virtual void timeOutHandler ()

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

- MasterInfo ∗ getMasterInfo ()

  Retrieves a MasterInfo object which will hold additional information about the currently processed Modbus request.

Data Access Synchronisation Functions

Implementation of these functions may only be required in multithreaded applications, if you are running the server loop in a separate thread and in addition require data consistency over a block of Modbus registers.

Data consistency within a single register is always maintained if the code executes on a 16-bit or 32-bit machine, because the CPU is accessing these data types atomically.

- virtual void lock ()

  You can override this method to implement a mutex locking mechanism to synchronise data access.

- virtual void unlock ()

  You can override this method to implement a mutex un-locking mechanism to synchronise data access.

8.6.1   Detailed Description

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

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

> Server Functions common to all Modbus Protocol Flavours
> MbusRtuSlaveProtocol, MbusAsciiSlaveProtocol, MbusTcpSlaveProtocol

8.6.2   Member Function Documentation

readHoldingRegistersTable()   `virtual int readHoldingRegistersTable (`
                `int startRef,`
                `short regArr[],`
                `int refCnt )  [inline], [virtual]`

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

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

A simple implementation which holds the application data in an array of shorts (`short regData[0x10000]`) could be:

```cpp
int readHoldingRegistersTable(int startRef, short regArr[], int refCnt)
{
    startRef--; // Adjust Modbus reference counting

    if (startRef + refCnt > (int) sizeof(regData) / sizeof(short))
        return 0;

    memcpy(regArr, &regData[startRef], refCnt * sizeof(short));
    return 1;
}
```

Parameters

startRef	Start register (Range: 1 - 65536)
regArr	Buffer which has to be filled with the reply data
refCnt	Number of registers to be retrieved (Range: 0 - 125)

Return values

1	Indicate 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.

Return values

0	Indicate that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

Required:

Yes

Default Implementation:

Returns 0 which indicates to Server Engine that this address range is unsupported.

writeHoldingRegistersTable() `virtual int writeHoldingRegistersTable (`
`        int `*`startRef,`*
`        const short `*`regArr[],`*
`        int `*`refCnt`*` )  [inline], [virtual]`

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

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

A simple implementation which holds the application data in an array of shorts (`short regData[0x10000]`) could be:

```
int writeHoldingRegistersTable(int startRef, const short regArr[], int refCnt)
{
    startRef--; // Adjust Modbus reference counting

    if (startRef + refCnt > (int) sizeof(regData) / sizeof(short))
        return 0;

    memcpy(&regData[startRef], regArr, refCnt * sizeof(short));
    return 1;
}
```

Parameters

startRef	Start register (Range: 1 - 65536)
regArr	Buffer which contains the received data
refCnt	Number of registers received (Range: 0 - 125)

Return values

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

Required:

Yes

Default Implementation:

Returns 0 which indicates to Server Engine that this address range is unsupported.

readEnronRegistersTable() `virtual int readEnronRegistersTable (`
`int startRef,`
`long regArr[],`
`int refCnt ) [inline], [virtual]`

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

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

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
refCnt	Number of registers to be retrieved (Range: 0 - 62)

Return values

1	Indicate 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.
0	Indicate 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.

Required:

No

Default Implementation:

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

writeEnronRegistersTable() `virtual int writeEnronRegistersTable (`

           `int startRef,`

           `const long regArr[],`

           `int refCnt )  [inline], [virtual]`

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

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

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
refCnt	Number of registers received (Range: 0 - 62)

Return values

1	Indicate that a successful access. The Server Engine will send a positive reply to the master.
0	Indicate that the requested range is not to be processed using the Daniel/ENRON frame format. A subsequent call to writeHoldingRegistersTable() will be made to process the range in standard Modbus frame format.

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().

readInputRegistersTable() `virtual int readInputRegistersTable (`

           `int startRef,`

           `short regArr[],`

           `int refCnt )  [inline], [virtual]`

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

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

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

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

Parameters

startRef	Start register (Range: 1 - 65536)
regArr	Buffer which has to be filled with the reply data
refCnt	Number of registers to be retrieved (Range: 0 - 125)

Return values

1	Indicate 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.
0	Indicate that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

Required:

No

Default Implementation:

Returns 0 which indicates to Server Engine that this address range is unsupported.

readCoilsTable()　`virtual int readCoilsTable (`
`        int `*`startRef,`*
`        char `*`bitArr[],`*
`        int `*`refCnt`*` )  [inline], [virtual]`

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

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

A simple implementation which holds the boolean application data in an array of chars (`char bitData[2000]`) could be:

```
int readCoilsTable(int startRef, char bitArr[], int refCnt)
{
    startRef--; // Adjust Modbus reference counting
```

```
    if (startRef + refCnt > (int) sizeof(bitData) / sizeof(char))
        return 0;

    memcpy(bitArr, &bitData[startRef], refCnt * sizeof(char));
    return 1;
}
```

Parameters

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

Return values

1	Indicate 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.
0	Indicate that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

Required:

No

Default Implementation:

Returns 0 which indicates to Server Engine that this address range is unsupported.

writeCoilsTable() `virtual int writeCoilsTable (`
                `int startRef,`
                `const char bitArr[],`
                `int refCnt )  [inline], [virtual]`

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

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

A simple implementation which holds the boolean application data in an array of chars (`char bitData[2000]`) could be:

```
int writeCoilsTable(int startRef, const char bitArr[], int refCnt)
{
    startRef--; // Adjust Modbus reference counting

    if (startRef + refCnt > (int) sizeof(bitData) / sizeof(char))
        return 0;

    memcpy(&bitData[startRef], bitArr, refCnt * sizeof(char));
    return 1;
}
```

Parameters

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

Return values

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

Required:

No

Default Implementation:

Returns 0 which indicates to Server Engine that this address range is unsupported.

readInputDiscretesTable() `virtual int readInputDiscretesTable (`
`        int startRef,`
`        char bitArr[],`
`        int refCnt ) [inline], [virtual]`

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

When a slave receives a poll request for the 1:00000 data table it calls this method to retrieve the data.

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

```
int readInputDiscretesTable(int startRef, char bitArr[], int refCnt)
{
   return readCoilsTable(startRef, bitArr, refCnt);
}
```

Parameters

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

Return values

1	Indicate 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.
0	Indicate that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

Required:

No

Default Implementation:

Returns 0 which indicates to Server Engine that this address range is unsupported.

readFileRecord() `virtual int readFileRecord (`
```
        int refType,
        int fileNo,
        int startRef,
        short regArr[],
        int refCnt )  [inline], [virtual]
```

Override this method to implement a Data Provider function to read File Records which is Modbus function code 20 (14 hex).

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

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
refCnt	Record count (a record is 2 bytes long)

Return values

1	Indicate 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.
0	Indicate that access has been denied or is out of range. The Server Engine will reply to the master with an exception reply message

Required:

No

Default Implementation:

Returns 0 which indicates to Server Engine that this address range is unsupported.

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

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

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

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
refCnt	Record count received (a record is 2 bytes long)

Return values

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

Required:

No

Default Implementation:

Returns 0 which indicates to Server Engine that this address range is unsupported.

readExceptionStatus() `virtual char readExceptionStatus ( ) [inline], [virtual]`

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

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.

Returns

Exception status byte

Required:

No

Default Implementation:

Returns 0 as exception status byte.

reportSlaveId()   `virtual int reportSlaveId (`
`              char bufferArr[],`
`              int maxBufSize )  [inline], [virtual]`

Override this method to implement a function which reports the Slave ID.

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 which must be copied into the supplied buffer array. The string must NOT be 0 terminated.

A simple implementation could be:

```
int reportSlaveId(char bufferArr[], int maxBufSize)
{
   strncpy(bufferArr, PRODUCT_NAME, maxBufSize);
   return strlen(PRODUCT_NAME);
}
```

Returns

The actual ASCII string length in bytes

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.

Required:

No

Default Implementation:

Returns 0 which causes a slave failure exception reply

reportRunIndicatorStatus() `virtual int reportRunIndicatorStatus ( ) [inline], [virtual]`

Override this method to implement a function which reports the Run Indicator of a device.

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

The Run Indicator is one byte which is 0x00 for OFF and 0xFF for ON.

Returns

Run Indicator status byte

Required:

No

Default Implementation:

Returns 0 (OFF) as run indicator status byte.

readDeviceIdentification() `virtual int readDeviceIdentification (`
`                int objId,`
`                char bufferArr[],`
`                int maxBufSize ) [inline], [virtual]`

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

This function allows a master to retrieve various objects with meta information about a slave device. The objects are returned as ASCII string. The string must NOT be 0 terminated.

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

A simple implementation could be:

```
int readDeviceIdentification(int objId, char bufferArr[], int maxBufSize)
{
   switch (objId)
   {
      case 0: // VendorName
         if (bufferArr)
         {
```

```
                strncpy(bufferArr, VENDOR_NAME, maxBufSize);
            }
        return strlen(VENDOR_NAME);
        case 1: // ProductCode
            if (bufferArr)
            {
                strncpy(bufferArr, PRODUCT_CODE, maxBufSize);
            }
        return strlen(PRODUCT_CODE);
        case 4: // ProductName
            if (bufferArr)
            {
                strncpy(bufferArr, PRODUCT_NAME, maxBufSize);
            }
        return strlen(PRODUCT_NAME);
    }
    return 0; // Return 0 for object not found
}
```

Parameters

objId	ID number (0x00 - 0xFF)
bufferArr	0 for size query otherwise a pointer where to store the requested Device ID string.
maxBufSize	Maximum space in the buffer in bytes

Returns

The string length in bytes

Note

bufferArr will be 0 during the size query phase and the implementation must return only the object length in that case and not attempt to copy any Device ID string into bufferArr!

Required:

No

Default Implementation:

Returns 0 which sends an unsupported ID exception reply

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

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

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

getMasterInfo() `MasterInfo* getMasterInfo ( )  [inline]`

>   Retrieves a MasterInfo object which will hold additional information about the currently processed Modbus request.

>   This is the protocol type, the slave ID, the IP address and the transaction ID of the current request.

>   The pointer and data is only valid during the execution of a data table callback method (like readHoldingRegistersTable or writeHoldingRegistersTable)

>   **Returns**

>   >   Pointer to current MasterInfo object

lock() `virtual void lock ( )  [inline], [virtual]`

>   You can override this method to implement a mutex locking mechanism to synchronise data access.

>   This is not needed in single threaded applications but may be necessary in multithreaded applications if you are running the server loop in a separate thread and require data consistency over a block of Modbus registers. Data consistency within a single register is always maintained if the code executes on a 16-bit or 32-bit machine, because the CPU is accessing these data types atomically.

>   This function is called by the server before calling any data read or write functions.

>   **Required:**

>   >   No

>   **Default Implementation:**

>   >   Empty

unlock() `virtual void unlock ( )  [inline], [virtual]`

>   You can override this method to implement a mutex un-locking mechanism to synchronise data access.

This is not needed in single threaded applications but may be necessary in multithreaded applications if you are running the server loop in a separate thread and require data consistency over a block of Modbus registers. Data consistency within a single register is always maintained if the code executes on a 16-bit or 32-bit machine, because the CPU is accessing these data types atomically.

This function is called by the server after calling any data read or write functions.

Required:

> No

Default Implementation:

> Empty

9   License

```
                           Library License
              proconX Pty Ltd, Brisbane/Australia, ACN 104 080 935


      Revision 4, October 2008


      Definitions
      "Software" refers to  the  collection  of files and any part hereof,
      including,  but  not  limited  to,  source  code,  programs,  binary
      executables,  object  files, libraries, images, and scripts, which are
      distributed by proconX.

      "Copyright  Holder" is whoever is named in the copyright or copyrights
      for the Software.

      "You" is you, if you are thinking about using, copying or distributing
      this Software or parts of it.

      "Distributable  Components"  are  dynamic libraries, shared libraries,
      class  files  and  similar  components  supplied  by  proconX  for
      redistribution.  They  must  be  listed in a "README" or "DEPLOY" file
      included with the Software.

      "Application" pertains to Your product be it an application, applet or
      embedded software product.
        ----------------------------------------------------------------

   License Terms

      1. In  consideration of payment of the licence fee and your agreement
         to  abide  by  the  terms  and conditions of this licence, proconX
         grants You the following non-exclusive rights:
           a. You may use the Software on one or more computers by a single
              person who uses the software personally;
           b. You may use the Software nonsimultaneously by multiple people
              if it is installed on a single computer;
           c. You  may  use  the  Software  on a network, provided that the
              network  is  operated  by  the organisation who purchased the
              license and there is no concurrent use of the Software;
           d. You may copy the Software for archival purposes.
      2. You  may  reproduce  and  distribute,  in  executable  form  only,
         Applications  linked with static libraries supplied as part of the
         Software  and Applications incorporating dynamic libraries, shared
         libraries  and  similar  components  supplied  as  Distributable
         Components without royalties provided that:
           a. You paid the license fee;
           b. the purpose of distribution is to execute the Application;
           c. the  Distributable  Components  are not distributed or resold
              apart from the Application;
           d. it  includes  all  of  the  original  Copyright  Notices  and
              associated Disclaimers;
           e. it does not include any Software source code or part thereof.
      3. If  You have received this Software for the purpose of evaluation,
         proconX  grants  You  a  non-exclusive license to use the Software
         free  of  charge for the purpose of evaluating whether to purchase
         an  ongoing  license to use the Software. The evaluation period is
         limited to 30 days and does not include the right to reproduce and
         distribute  Applications  using  the  Software.  At the end of the
         evaluation  period,  if  You  do  not purchase a license, You must
         uninstall the Software from the computers or devices You installed
```

it on.

4. You are not required to accept this License, since You have not signed it. However, nothing else grants You permission to use or distribute the Software or its derivative works. These actions are prohibited by law if You do not accept this License. Therefore, by using or distributing the Software (or any work based on the Software), You indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or using the Software or works based on it.

5. You may not use the Software to develop products which can be used as a replacement or a directly competing product of this Software.

6. Where source code is provided as part of the Software, You may modify the source code for the purpose of improvements and defect fixes. If any modifications are made to any the source code, You will put an additional banner into the code which indicates that modifications were made by You.

7. You may not disclose the Software's software design, source code and documentation or any part thereof to any third party without the expressed written consent from proconX.

8. This License does not grant You any title, ownership rights, rights to patents, copyrights, trade secrets, trademarks, or any other rights in respect to the Software.

9. You may not use, copy, modify, sublicense, or distribute the Software except as expressly provided under this License. Any attempt otherwise to use, copy, modify, sublicense or distribute the Software is void, and will automatically terminate your rights under this License.

10. The License is not transferable without written permission from proconX.

11. proconX may create, from time to time, updated versions of the Software. Updated versions of the Software will be subject to the terms and conditions of this agreement and reference to the Software in this agreement means and includes any version update.

12. THERE IS NO WARRANTY FOR THE SOFTWARE, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING PROCONX, THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE SOFTWARE "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH YOU. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. ANY LIABILITY OF PROCONX WILL BE LIMITED EXCLUSIVELY TO REFUND OF PURCHASE PRICE. IN ADDITION, IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL PROCONX OR ITS PRINCIPALS, SHAREHOLDERS, OFFICERS, EMPLOYEES, AFFILIATES, CONTRACTORS, SUBSIDIARIES, PARENT ORGANIZATIONS AND ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE SOFTWARE AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE SOFTWARE TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

14. IN ADDITION, IN NO EVENT DOES PROCONX AUTHORIZE YOU TO USE THIS SOFTWARE IN APPLICATIONS OR SYSTEMS WHERE IT'S FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO RESULT IN A SIGNIFICANT PHYSICAL INJURY, OR IN LOSS OF LIFE. ANY SUCH USE BY YOU IS ENTIRELY AT YOUR OWN RISK, AND YOU AGREE TO HOLD PROCONX HARMLESS FROM ANY CLAIMS OR LOSSES RELATING TO SUCH UNAUTHORIZED USE.

15. This agreement constitutes the entire agreement between proconX

and You in relation to your use of the Software. Any change will be effective only if in writing signed by proconX and you.

16. This License is governed by the laws of Queensland, Australia, excluding choice of law rules. If any part of this License is found to be in conflict with the law, that part shall be interpreted in its broadest meaning consistent with the law, and no other parts of the License shall be affected.

----

10   Support

We provide electronic support and feedback for our FieldTalk products. Please use the Support web page at: `http://www.modbusdriver.com/support`

Your feedback is always welcome. It helps improving this product.

11 Notices

Disclaimer:

proconX Pty Ltd makes no warranty for the use of its products, other than those expressly contained in the Company's standard warranty which is detailed in the Terms and Conditions located on the Company's Website. The Company assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information contained herein. No licenses to patents or other intellectual property of proconX are granted by the Company in connection with the sale of proconX products, expressly or by implication. proconX products are not authorized for use as critical components in life support devices or systems.

This FieldTalk™ library was developed by:
proconX Pty Ltd, Australia.
Copyright © 2002-2025. All rights reserved.

proconX and FieldTalk are trademarks of proconX Pty Ltd. Modbus is a registered trademark of Schneider Automation Inc. All other product and brand names mentioned in this document may be trademarks or registered trademarks of their respective owners.