FOCUS
SOFTWARE ENGINEERING

FieldTalk™ Modbus® Slave C++ Library

Last updated: 20 Oct 2006

FOCUS Software Engineering Pty Ltd, Australia.

Contents

1 General Description

1.1 Introduction

This *FieldTalk*™ Modbus Slave C++ Library allows you to incorporate Modbus slave functionality into your device or application.

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

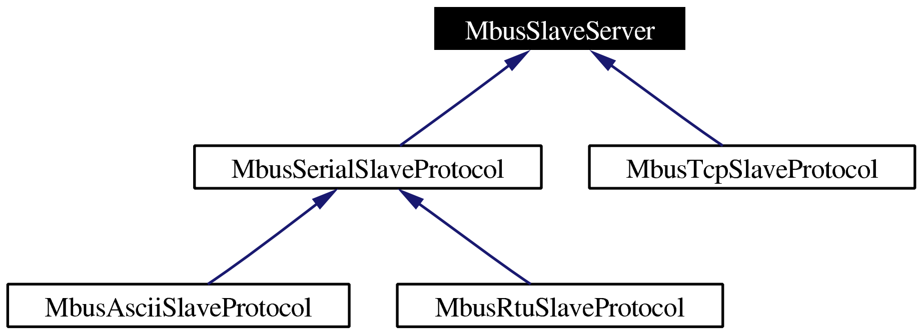
Features:

- Robust design suitable for real-time and industrial applications
- Full implementation of Bit Access and 16 Bits Access Function Codes as well as a subset of the most commonly used Diagnostics Function Codes
- Standard Modbus bit and 16-bit integer data types (coils, discretes & registers)
- Support of Broadcasting
- Master time-out supervision
- Failure and transmission counters
- Supports single or multiple slave addresses
- Scalable: you can use serial-line Modbus protocols only or MODBUS/TCP or all of them

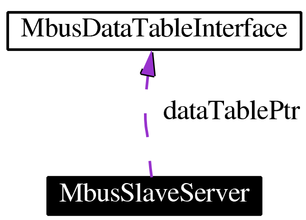
1.2 Library Structure

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 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();
}
```

1.3 Overview

2 Modbus Slave C++ Library Module Documentation

2.1 Server Functions common to all Protocol Flavours

2.1.1 Detailed Description

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:

Modbus Function Code	**Current Terminology**	**Classic Terminology**
16-bit 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 Register
23 (17 Hex)	Read/Write Registers	Read/Write Registers
Bit access		
1	Read Coils	Read Coil Status
2	Read Inputs Discretes	Read Input Status
5	Write Coil	Force Single Coil
15 (0F Hex)	Force Multiple Coils	Force Multiple Coils
Diagnostics		
7	Read Exception Status	Read Exception Status
8 sub code 00	Diagnostics - Return Query Data	Diagnostics - Return Query Data

Server Management Functions

- int MbusSlaveServer::addDataTable (int slaveAddr, MbusDataTableInterface *dataTablePtr)
  Associates a protocol object with a Data Provider and a slave address.
- virtual int MbusSlaveServer::serverLoop ()=0
  Modbus slave server loop.
- virtual void MbusSlaveServer::shutdownServer ()
  Shuts down the Modbus Server.
- virtual int MbusSlaveServer::isStarted ()=0
  Returns if server has been started up.
- virtual int MbusSlaveServer::getConnectionStatus ()=0
  Associates a protocol object with a Data Provider and a slave address.

Protocol Configuration

- long MbusSlaveServer::setTimeout (long timeOut)
  Configures master transmit time-out supervision.
- long MbusSlaveServer::getTimeout ()
  Returns the master time-out supervision value.

Transmission Statistic Functions

- unsigned long MbusSlaveServer::getTotalCounter ()
  Returns how often a message transfer has been executed.
- void MbusSlaveServer::resetTotalCounter ()
  Resets total message transfer counter.
- unsigned long MbusSlaveServer::getSuccessCounter ()
  Returns how often a message transfer was successful.
- void MbusSlaveServer::resetSuccessCounter ()
  Resets successful message transfer counter.

Utility Functions

- static char * MbusSlaveServer::getPackageVersion ()

Returns the package version number.

2.1.2 Function Documentation

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

Associates a protocol object with a Data Provider and a slave address.

Parameters:

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 Protocol Errors and Exceptions for a list of error codes.

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

Configures master transmit time-out supervision.

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

Remarks:

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

Note:

The time-out does not check if a master is sending valid frames.

Parameters:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

unsigned long getTotalCounter () `[inherited]`

Returns how often a message transfer has been executed.

Returns:

Counter value

char * getPackageVersion () `[static, inherited]`

Returns the package version number.

Returns:

Package version string

long getTimeout () `[inherited]`

Returns the master time-out supervision value.

Remarks:

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

Returns:

Timeout value in ms

unsigned long getSuccessCounter () `[inherited]`

Returns how often a message transfer was successful.

Returns:
Counter value

virtual int serverLoop () `[pure virtual, inherited]`

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 endless loop:

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

Returns:
FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Implemented in MbusRtuSlaveProtocol, MbusAsciiSlaveProtocol, and MbusTcpSlaveProtocol.

void shutdownServer () `[virtual, inherited]`

Shuts down the Modbus Server.

This function also closes any associated serial ports or sockets.

Reimplemented in MbusSerialSlaveProtocol, and MbusTcpSlaveProtocol.

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

Returns if server has been started up.

Return values:
true = started

false = shutdown

Implemented in MbusSerialSlaveProtocol, and MbusTcpSlaveProtocol.

virtual int getConnectionStatus () `[pure virtual, inherited]`

Associates a protocol object with a Data Provider and a slave address.

Parameters:

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 Protocol Errors and Exceptions for a list of error codes.

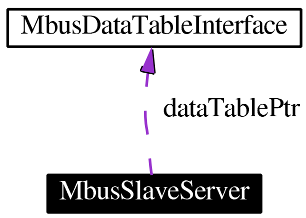
Implemented in MbusSerialSlaveProtocol, and MbusTcpSlaveProtocol.

2.2 Data Provider

2.2.1 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
   }
};
```

Classes

- class MbusDataTableInterface
  This class defines the interface between a Modbus slave Server Engine and your application. Descendants of this class are referred to as Data Providers.

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 MbusDataTableInterface::readHoldingRegistersTable (int startRef, short regArr[ ], int refCnt)
  Override this method to implement a Data Provider function to read Holding Registers.

- virtual int MbusDataTableInterface::writeHoldingRegistersTable (int startRef, const short regArr[ ], int refCnt)
  Override this method to implement a Data Provider function to write Holding Registers.

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

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

This table is accessed by the following Modbus functions:

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

Note:

Input registers cannot be written

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

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

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 MbusDataTableInterface::lock ()
  You can override this method to implement a semaphore locking mechanism to synchronise data access.

- virtual void MbusDataTableInterface::unlock ()
  You can override this method to implement a semaphore un-locking mechanism to synchronise data access.

Auxiliary Functions

- virtual void MbusDataTableInterface::timeOutHandler ()
  Override this method to implement a function to handle master poll time-outs.

- virtual char MbusDataTableInterface::readExceptionStatus ()
  Override this method to implement a function with reports the eight exception status coils (bits) within the slave device.

2.2.2 Function Documentation

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

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 he 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:

```
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 - 0x10000)

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:

Yes

Default Implementation:

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

virtual int readInputRegistersTable (int *startRef*, short *regArr*[], int *refCnt*) `[virtual, inherited]`

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 he 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 - 0x10000)

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.

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

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 he 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 - 0x10000)

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.

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

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 he 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 - 0x10000)

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.

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

You can override this method to implement a semaphore 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

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

Override this method to implement a function to handle master poll timeouts.

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

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

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 he 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 - 0x10000)

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.

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

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 he 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 - 0x10000)

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.

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

You can override this method to implement a semaphore 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

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

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.

2.3 Serial Protocols

2.3.1 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 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.

Classes

- class MbusRtuSlaveProtocol
  Modbus RTU Slave Protocol class.
- class MbusAsciiSlaveProtocol
  Modbus ASCII Slave Protocol class.

2.4 MODBUS/TCP Protocol

2.4.1 Detailed Description

The Server Engine of the MODBUS/TCP slave protocol is implemented in the class MbusTcpSlaveProtocol.

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

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

Classes

- class MbusTcpSlaveProtocol
  MODBUS/TCP Slave Protocol class.

Defines

- #define MAX_CONNECTIONS 16
  Maximum concurrent TCP/IP connections handled by server engine.

2.5 Protocol Errors and Exceptions

Fatal API Errors

Errors of this class typically indicate a programming mistake.

- #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.

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.

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_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.

Defines

- #define FTALK_SUCCESS 0
  Operation was successful.

Functions

- TCHAR * getBusProtocolErrorText (int errCode)
  Returns an error text string for a given error code.

2.5.1 Define Documentation

#define FTALK_SUCCESS 0

Operation was successful.

This return codes indicates no error.

#define FTALK_ILLEGAL_ARGUMENT_ERROR 1

Illegal argument error.

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

#define FTALK_ILLEGAL_STATE_ERROR 2

Illegal state error.

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

#define FTALK_EVALUATION_EXPIRED 3

Evaluation expired.

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

#define FTALK_NO_DATA_TABLE_ERROR 4

No data table configured.

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

#define FTALK_ILLEGAL_SLAVE_ADDRESS 5

Slave address 0 illegal for serial protocols.

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

#define FTALK_IO_ERROR_CLASS 64

I/O error class.

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

#define FTALK_IO_ERROR 65

I/O error.

The underlaying I/O system reported an error.

#define FTALK_OPEN_ERR 66

Port or socket open error.

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

#define FTALK_PORT_ALREADY_OPEN 67

Serial port already open.

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

#define FTALK_TCPIP_CONNECT_ERR 68

TCP/IP connection error.

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

#define FTALK_CONNECTION_WAS_CLOSED 69

Remote peer closed TCP/IP connection.

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

#define FTALK_SOCKET_LIB_ERROR 70

Socket library error.

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

#define FTALK_PORT_ALREADY_BOUND 71

TCP port already bound.

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

#define FTALK_LISTEN_FAILED 72

Listen failed.

The listen operation on the specified TCP port failed..

#define FTALK_FILEDES_EXCEEDED 73

File descriptors exceeded.

Maximum number of usable file descriptors exceeded.

#define FTALK_PORT_NO_ACCESS 74

No permission to access serial port or TCP port.

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

#define FTALK_PORT_NOT_AVAIL 75

TCP port not available.

The specified TCP port is not available on this machine.

#define FTALK_BUS_PROTOCOL_ERROR_CLASS 128

Fieldbus protocol error class.

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

#define FTALK_CHECKSUM_ERROR 129

Checksum error.

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

#define FTALK_INVALID_FRAME_ERROR 130

Invalid frame error.

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

#define FTALK_INVALID_REPLY_ERROR 131

Invalid reply error.

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

#define FTALK_REPLY_TIMEOUT_ERROR 132

Reply time-out.

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

#define FTALK_SEND_TIMEOUT_ERROR 133

Send time-out.

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

#define FTALK_MBUS_EXCEPTION_RESPONSE 160

Modbus® exception response.

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

#define FTALK_MBUS_ILLEGAL_FUNCTION_RESPONSE 161

Illegal Function exception response.

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

#define FTALK_MBUS_ILLEGAL_ADDRESS_RESPONSE 162

Illegal Data Address exception response.

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

#define FTALK_MBUS_ILLEGAL_VALUE_RESPONSE 163

Illegal Data Value exception response.

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

#define FTALK_MBUS_SLAVE_FAILURE_RESPONSE 164

Slave Device Failure exception response.

Signals that a Slave Device Failure exception response (code 04) was received. This exception response is sent by a slave device instead of a normal response message if an unrecoverable error occured while processing the requested action. This response is also sent if the request would generate a response whose size exceeds the allowable data size.

2.5.2 Function Documentation

TCHAR* getBusProtocolErrorText (int *errCode*)

Returns an error text string for a given error code.

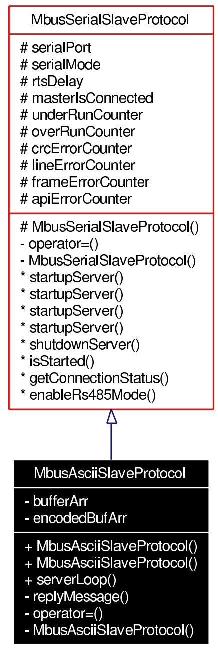
Parameters:
errCode FieldTalk error code

Returns:
Error text string

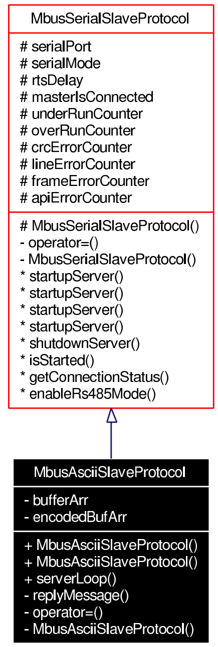
3 Modbus Slave C++ Library Class Documentation

3.1 MbusAsciiSlaveProtocol Class Reference

Inheritance diagram for MbusAsciiSlaveProtocol:



Collaboration diagram for MbusAsciiSlaveProtocol:



3.1.1 Detailed Description

Modbus ASCII Slave Protocol class.

This class realises the Modbus ASCII 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 Protocol Flavours.

It is possible to instantiate multiple instances for establishing multiple connections on different serial ports (They should be executed in separate threads).

See also:
Server Functions common to all Protocol Flavours, MbusSlaveServer

Serial Server Management Functions

- virtual int startupServer (const char ∗const portName, long baudRate, int dataBits, int stopBits, int parity)
  Puts the Modbus server into operation.

- int startupServer (const char *const portName, long baudRate)
  Puts the Modbus RTU server into operation and opens the associated serial port with default port parameters.

- virtual int startupServer (int slaveAddr, const char *const portName, long baudRate, int dataBits, int stopBits, int parity)
  Puts the Modbus server into operation using a single slave address and data table.

- int startupServer (int slaveAddr, const char *const portName, long baudRate)
  Puts the Modbus RTU server into operation and opens the associated serial port with default port parameters.

- void shutdownServer ()
  Shuts down the Modbus server.

- int isStarted ()
  Returns if server has been started up.

- int getConnectionStatus ()
  Checks if a Modbus master is polling periodically.

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

Server Management Functions

- int addDataTable (int slaveAddr, MbusDataTableInterface *dataTablePtr)
  Associates a protocol object with a Data Provider and a slave address.

Protocol Configuration

- long setTimeout (long timeOut)
  Configures master transmit time-out supervision.

- long getTimeout ()
  Returns the master time-out supervision value.

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.

Utility Functions

- static char * getPackageVersion ()
  Returns the package version number.

Public Types

- enum { SER_DATABITS_7 = SerialPort::SER_DATABITS_7, SER_DATABITS_8 = SerialPort::SER_DATABITS_8 }
- enum { SER_STOPBITS_1 = SerialPort::SER_STOPBITS_1, SER_STOPBITS_2 = SerialPort::SER_STOPBITS_2 }
- enum { SER_PARITY_NONE = SerialPort::SER_PARITY_NONE, SER_PARITY_EVEN = SerialPort::SER_PARITY_EVEN, SER_PARITY_ODD = SerialPort::SER_PARITY_ODD }

Public Member Functions

- MbusAsciiSlaveProtocol ()
  Constructs a MbusAsciiSlaveProtocol object.

- MbusAsciiSlaveProtocol (MbusDataTableInterface *dataTablePtr)
  Constructs a MbusAsciiSlaveProtocol object and associates it with a Data Provider.

- int serverLoop ()
  Modbus ASCII slave server loop.

Protected Types

- enum { SER_RS232, SER_RS485 }

3.1.2 Member Enumeration Documentation

anonymous enum `[inherited]`

Enumeration values:

- ***SER_DATABITS_7*** 7 data bits
- ***SER_DATABITS_8*** 8 data bits

anonymous enum `[inherited]`

Enumeration values:

- ***SER_STOPBITS_1*** 1 stop bit
- ***SER_STOPBITS_2*** 2 stop bits

anonymous enum `[inherited]`

Enumeration values:

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

anonymous enum `[protected, inherited]`

Enumeration values:

- ***SER_RS232*** RS232 mode w/o RTS/CTS handshake.
- ***SER_RS485*** RS485 mode: RTS enables/disables transmitter.

3.1.3 Constructor & Destructor Documentation

MbusAsciiSlaveProtocol ()

Constructs a MbusAsciiSlaveProtocol object.

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

Parameters:

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.

MbusAsciiSlaveProtocol (MbusDataTableInterface * *dataTablePtr*)

Constructs a MbusAsciiSlaveProtocol object and associates it with a Data Provider.

Function is kept for compatibility with previous API versions, do not use for new implementations.

Parameters:

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.

Deprecated

This function is deprecated. The preferred way of assigning a dataTable is using the default constructor and configuring data table and slave address using addDataTable method.

3.1.4 Member Function Documentation

int serverLoop () `[virtual]`

Modbus ASCII 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.

Returns:
FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Implements MbusSlaveServer.

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 - 9600).
- *dataBits* Must be SER_DATABITS_8 for RTU
- *stopBits* SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits
- *parity* SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_EVEN: even parity

Returns:
FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Reimplemented in MbusRtuSlaveProtocol.

int startupServer (const char ∗const *portName*, long *baudRate*) `[inherited]`

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

This function opens the serial port with 8 databits, 1 stopbit and even parity 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 - 9600).

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

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

Puts the Modbus server into operation using a single slave address and data table.

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

Function is kept for compatibility with previous API versions, do not use for new implementations.

Parameters:

slaveAddr Modbus slave address for server to listen on (1-255)

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

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

dataBits Must be SER_DATABITS_8 for RTU

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

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

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Deprecated

This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

Reimplemented in MbusRtuSlaveProtocol.

int startupServer (int *slaveAddr*, const char *const *portName*, long *baudRate*) `[inherited]`

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

This function opens the serial port with 8 databits, 1 stopbit and even parity and initialises the server engine.

Function is kept for compatibility with previous API versions, do not use for new implementations.

Parameters:

slaveAddr Modbus slave address for server to listen on (1-255)

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

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

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Deprecated

This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

void shutdownServer () `[virtual, inherited]`

Shuts down the Modbus server.

This function also closes the serial port.

Reimplemented from MbusSlaveServer.

int isStarted () `[virtual, inherited]`

Returns if server has been started up.

Return values:

true = started

false = shutdown

Implements MbusSlaveServer.

int getConnectionStatus () `[virtual, 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.

Implements MbusSlaveServer.

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

Enables RS485 mode.

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

Warning:

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

Remarks:

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

Note:

A protocol must be closed in order to configure it.

Parameters:

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

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

3.2 MbusDataTableInterface Class Reference

3.2.1 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:
MbusSlaveServer
Server Functions common to all Protocol Flavours

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.

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

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

This table is accessed by the following Modbus functions:

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

Note:
Input registers cannot be written

- virtual 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 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 semaphore locking mechanism to synchronise data access.

- virtual void unlock ()
  You can override this method to implement a semaphore un-locking mechanism to synchronise data access.

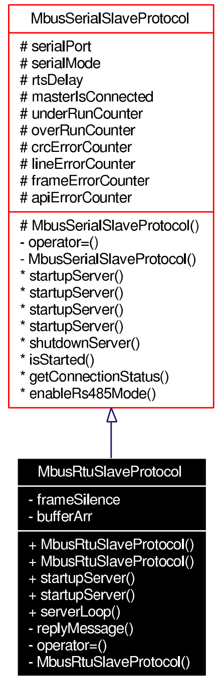
Auxiliary Functions

- virtual void timeOutHandler ()
  Override this method to implement a function to handle master poll time-outs.

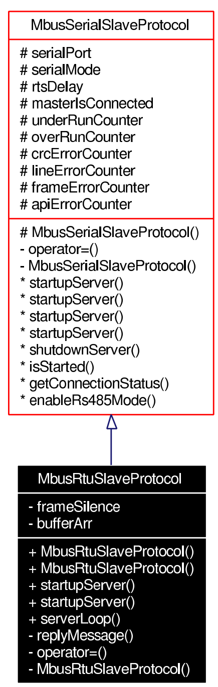
- virtual char readExceptionStatus ()
  Override this method to implement a function with reports the eight exception status coils (bits) within the slave device.

3.3 MbusRtuSlaveProtocol Class Reference

Inheritance diagram for MbusRtuSlaveProtocol:



Collaboration diagram for MbusRtuSlaveProtocol:



3.3.1 Detailed Description

Modbus RTU Slave Protocol class.

This class realises 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 Protocol Flavours.

It is possible to instantiate multiple instances for establishing multiple connections on different serial ports (They should be executed in separate threads).

See also:
Server Functions common to all Protocol Flavours, MbusSlaveServer

Serial Server Management Functions

- int startupServer (const char *const portName, long baudRate)
  Puts the Modbus RTU server into operation and opens the associated serial port with default port parameters.
- int startupServer (int slaveAddr, const char *const portName, long baudRate)
  Puts the Modbus RTU server into operation and opens the associated serial port with default port parameters.
- void shutdownServer ()
  Shuts down the Modbus server.
- int isStarted ()
  Returns if server has been started up.
- int getConnectionStatus ()
  Checks if a Modbus master is polling periodically.
- virtual int enableRs485Mode (int rtsDelay)
  Enables RS485 mode.

Server Management Functions

- int addDataTable (int slaveAddr, MbusDataTableInterface *dataTablePtr)

Associates a protocol object with a Data Provider and a slave address.

Protocol Configuration

- long setTimeout (long timeOut)
  Configures master transmit time-out supervision.
- long getTimeout ()
  Returns the master time-out supervision value.

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.

Utility Functions

- static char ∗ getPackageVersion ()
  Returns the package version number.

Public Types

- enum { SER_DATABITS_7 = SerialPort::SER_DATABITS_7, SER_DATABITS_8 = SerialPort::SER_DATABITS_8 }
- enum { SER_STOPBITS_1 = SerialPort::SER_STOPBITS_1, SER_STOPBITS_2 = SerialPort::SER_STOPBITS_2 }
- enum { SER_PARITY_NONE = SerialPort::SER_PARITY_NONE, SER_PARITY_EVEN = SerialPort::SER_PARITY_EVEN, SER_PARITY_ODD = SerialPort::SER_PARITY_ODD }

Public Member Functions

- MbusRtuSlaveProtocol ()
  Constructs a MbusRtuSlaveProtocol object.
- MbusRtuSlaveProtocol (MbusDataTableInterface *dataTablePtr)
  Constructs a MbusRtuSlaveProtocol object and associates it with a Data Provider.
- int startupServer (const char *const portName, long baudRate, int dataBits, int stopBits, int parity)
  Puts the Modbus RTU server into operation and opens the associated serial port with specific port parameters.
- int startupServer (int slaveAddr, const char *const portName, long baudRate, int dataBits, int stopBits, int parity)
  Puts the Modbus RTU server into operation and opens the associated serial port with specific port parameters.
- int serverLoop ()
  Modbus RTU slave server loop.

Protected Types

- enum { SER_RS232, SER_RS485 }

3.3.2 Member Enumeration Documentation

anonymous enum `[inherited]`

Enumeration values:

SER_DATABITS_7 7 data bits

SER_DATABITS_8 8 data bits

anonymous enum `[inherited]`

Enumeration values:

SER_STOPBITS_1 1 stop bit

SER_STOPBITS_2 2 stop bits

anonymous enum `[inherited]`

Enumeration values:

SER_PARITY_NONE No parity.

SER_PARITY_EVEN Even parity.

SER_PARITY_ODD Odd parity.

anonymous enum `[protected, inherited]`

Enumeration values:

SER_RS232 RS232 mode w/o RTS/CTS handshake.

SER_RS485 RS485 mode: RTS enables/disables transmitter.

3.3.3 Constructor & Destructor Documentation

MbusRtuSlaveProtocol ()

Constructs a MbusRtuSlaveProtocol object.

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

MbusRtuSlaveProtocol (MbusDataTableInterface ∗ *dataTablePtr*)

Constructs a MbusRtuSlaveProtocol object and associates it with a Data Provider.

Function is kept for compatibility with previous API versions, do not use for new implementations.

Parameters:

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.

Deprecated

This function is deprecated. The preferred way of assigning a dataTable is using the default constructor and configuring data table and slave address using addDataTable method.

3.3.4 Member Function Documentation

int startupServer (const char ∗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 - 9600).

dataBits Must be SER_DATABITS_8 for RTU

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

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

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Reimplemented from MbusSerialSlaveProtocol.

int startupServer (int *slaveAddr*, const char ∗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.

Function is kept for compatibility with previous API versions, do not use for new implementations.

Parameters:

slaveAddr Modbus slave address for server to listen on (1-255)

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

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

dataBits Must be SER_DATABITS_8 for RTU

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

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

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Deprecated

This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

Reimplemented from MbusSerialSlaveProtocol.

int serverLoop () `[virtual]`

Modbus RTU 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.

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Implements MbusSlaveServer.

int startupServer (const char ∗const *portName*, long *baudRate*) `[inherited]`

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

This function opens the serial port with 8 databits, 1 stopbit and even parity 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 - 9600).

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

int startupServer (int *slaveAddr*, const char ∗const *portName*, long *baudRate*) `[inherited]`

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

This function opens the serial port with 8 databits, 1 stopbit and even parity and initialises the server engine.

Function is kept for compatibility with previous API versions, do not use for new implementations.

Parameters:

slaveAddr Modbus slave address for server to listen on (1-255)

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

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

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Deprecated

This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

void shutdownServer () [virtual, inherited]

Shuts down the Modbus server.

This function also closes the serial port.

Reimplemented from MbusSlaveServer.

int isStarted () [virtual, inherited]

Returns if server has been started up.

Return values:

true = started

false = shutdown

Implements MbusSlaveServer.

int getConnectionStatus () [virtual, 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.

Implements MbusSlaveServer.

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

Enables RS485 mode.

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

Warning:

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

Remarks:

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

Note:

A protocol must be closed in order to configure it.

Parameters:

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

Return values:

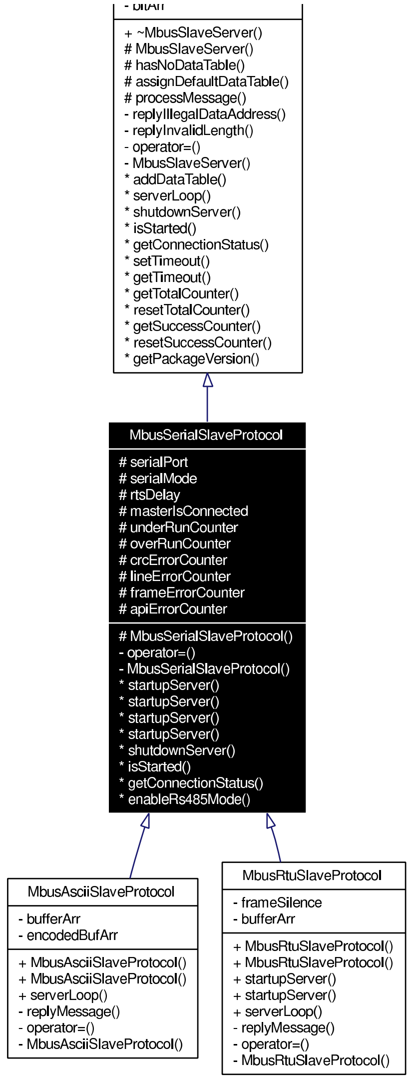
FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

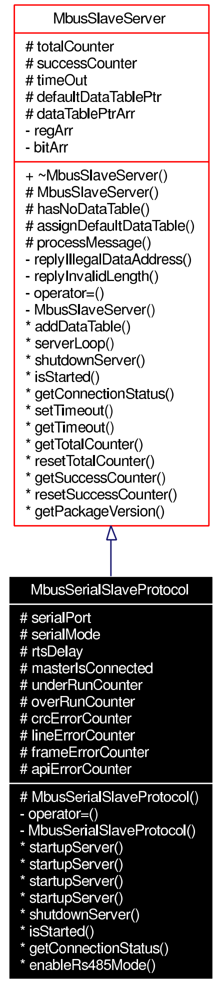
FTALK_ILLEGAL_STATE_ERROR Protocol is already open

3.4 MbusSerialSlaveProtocol Class Reference

Inheritance diagram for MbusSerialSlaveProtocol:



Collaboration diagram for MbusSerialSlaveProtocol:



3.4.1 Detailed Description

This base class realises the Modbus® serial slave protocols.

These methods apply to RTU and ASCII protocol flavours via inheritance. 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 Protocol Flavours.

It is possible to instantiate multiple instances for establishing multiple connections on different serial ports (They should be executed in separate threads).

See also:

Server Functions common to all Protocol Flavours, MbusSlaveServer

Serial Server Management Functions

- virtual int startupServer (const char *const portName, long baudRate, int dataBits, int stopBits, int parity)
  Puts the Modbus server into operation.

- int startupServer (const char *const portName, long baudRate)
  Puts the Modbus RTU server into operation and opens the associated serial port with default port parameters.

- virtual int startupServer (int slaveAddr, const char *const portName, long baudRate, int dataBits, int stopBits, int parity)
  Puts the Modbus server into operation using a single slave address and data table.

- int startupServer (int slaveAddr, const char *const portName, long baudRate)
  Puts the Modbus RTU server into operation and opens the associated serial port with default port parameters.

- void shutdownServer ()
  Shuts down the Modbus server.

- int isStarted ()
  Returns if server has been started up.

- int getConnectionStatus ()
  Checks if a Modbus master is polling periodically.

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

Server Management Functions

- int addDataTable (int slaveAddr, MbusDataTableInterface *dataTablePtr)
  Associates a protocol object with a Data Provider and a slave address.

- virtual int serverLoop ()=0
  Modbus slave server loop.

Protocol Configuration

- long setTimeout (long timeOut)
  Configures master transmit time-out supervision.
- long getTimeout ()
  Returns the master time-out supervision value.

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.

Utility Functions

- static char * getPackageVersion ()
  Returns the package version number.

Public Types

- enum { SER_DATABITS_7 = SerialPort::SER_DATABITS_7, SER_DATABITS_8 = SerialPort::SER_DATABITS_8 }
- enum { SER_STOPBITS_1 = SerialPort::SER_STOPBITS_1, SER_STOPBITS_2 = SerialPort::SER_STOPBITS_2 }
- enum { SER_PARITY_NONE = SerialPort::SER_PARITY_NONE, SER_PARITY_EVEN = SerialPort::SER_PARITY_EVEN, SER_PARITY_ODD = SerialPort::SER_PARITY_ODD }

Protected Types

- enum { SER_RS232, SER_RS485 }

3.4.2 Member Enumeration Documentation

anonymous enum

Enumeration values:

- ***SER_DATABITS_7*** 7 data bits
- ***SER_DATABITS_8*** 8 data bits

anonymous enum

Enumeration values:

- ***SER_STOPBITS_1*** 1 stop bit
- ***SER_STOPBITS_2*** 2 stop bits

anonymous enum

Enumeration values:

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

anonymous enum `[protected]`

Enumeration values:

- ***SER_RS232*** RS232 mode w/o RTS/CTS handshake.
- ***SER_RS485*** RS485 mode: RTS enables/disables transmitter.

3.4.3 Member Function Documentation

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

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 - 9600).
- ***dataBits*** Must be SER_DATABITS_8 for RTU
- ***stopBits*** SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits
- ***parity*** SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_EVEN: even parity

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Reimplemented in MbusRtuSlaveProtocol.

int startupServer (const char ∗const *portName*, long *baudRate*)

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

This function opens the serial port with 8 databits, 1 stopbit and even parity 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 - 9600).

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

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

Puts the Modbus server into operation using a single slave address and data table.

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

Function is kept for compatibility with previous API versions, do not use for new implementations.

Parameters:

- ***slaveAddr*** Modbus slave address for server to listen on (1-255)
- ***portName*** Serial port identifier (e.g. "COM1", "/dev/ser1 or /dev/ttyS0")
- ***baudRate*** The port baudRate in bps (typically 1200 - 9600).
- ***dataBits*** Must be SER_DATABITS_8 for RTU
- ***stopBits*** SER_STOPBITS_1: 1 stop bit, SER_STOPBITS_2: 2 stop bits
- ***parity*** SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_EVEN: even parity

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Deprecated

This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

Reimplemented in MbusRtuSlaveProtocol.

int startupServer (int *slaveAddr*, const char ∗const *portName*, long *baudRate*)

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

This function opens the serial port with 8 databits, 1 stopbit and even parity and initialises the server engine.

Function is kept for compatibility with previous API versions, do not use for new implementations.

Parameters:

slaveAddr Modbus slave address for server to listen on (1-255)

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

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

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Deprecated

This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

void shutdownServer () `[virtual]`

Shuts down the Modbus server.

This function also closes the serial port.

Reimplemented from MbusSlaveServer.

int isStarted () `[virtual]`

Returns if server has been started up.

Return values:

true = started

false = shutdown

Implements MbusSlaveServer.

int getConnectionStatus () `[virtual]`

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.

Implements MbusSlaveServer.

int enableRs485Mode (int *rtsDelay*) `[virtual]`

Enables RS485 mode.

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

Warning:

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

Remarks:

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

Note:

A protocol must be closed in order to configure it.

Parameters:

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

Return values:

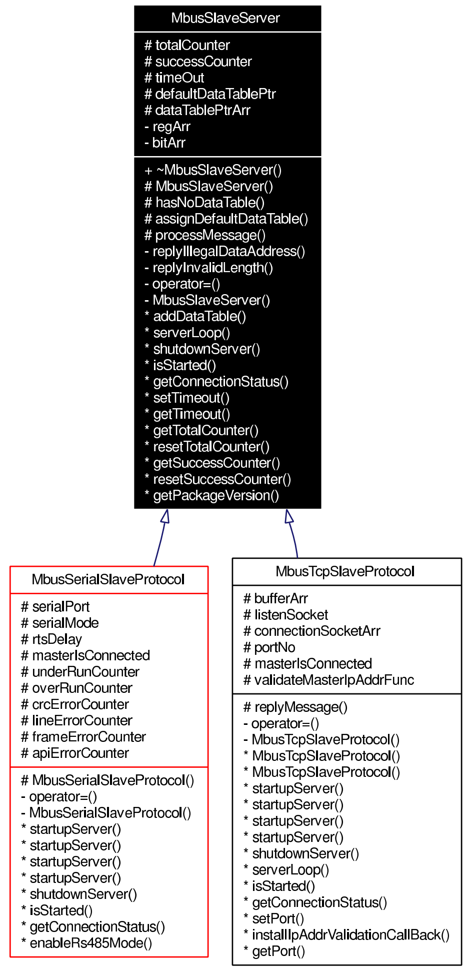
FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

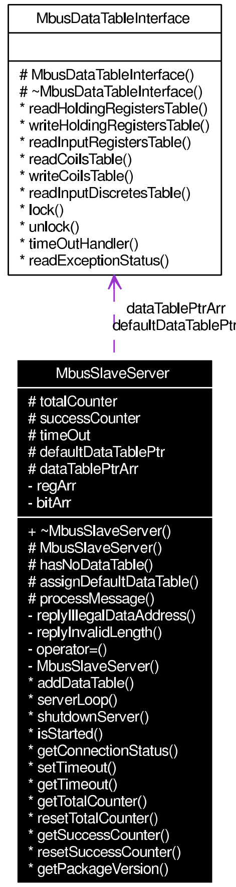
FTALK_ILLEGAL_STATE_ERROR Protocol is already open

3.5 MbusSlaveServer Class Reference

Inheritance diagram for MbusSlaveServer:



Collaboration diagram for MbusSlaveServer:



3.5.1 Detailed Description

Base class which implements the Modbus®server engine.

This class realises the server engine. The server engines processes Modbus messages, parses the function codes and upon receipt of a valid master query it calls Data Provider methods to exchange data with the user application. For a more detailed description which Modbus data and control functions have been implemented in the server engine see section Server Functions common to all Protocol Flavours.

See also:
MbusSlaveServer
Server Functions common to all Protocol Flavours

Server Management Functions

- int addDataTable (int slaveAddr, MbusDataTableInterface *dataTablePtr)

Associates a protocol object with a Data Provider and a slave address.

- virtual int serverLoop ()=0
  Modbus slave server loop.
- virtual void shutdownServer ()
  Shuts down the Modbus Server.
- virtual int isStarted ()=0
  Returns if server has been started up.
- virtual int getConnectionStatus ()=0
  Associates a protocol object with a Data Provider and a slave address.

Protocol Configuration

- long setTimeout (long timeOut)
  Configures master transmit time-out supervision.
- long getTimeout ()
  Returns the master time-out supervision value.

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.

Utility Functions

- static char * getPackageVersion ()
  Returns the package version number.

Public Member Functions

- virtual ~MbusSlaveServer ()
  Destructor.

Protected Member Functions

- MbusSlaveServer (MbusDataTableInterface ∗dataTablePtr=NULL)
  Constructs a MbusSlaveServer object and associates it with a Data Provider.

3.5.2 Constructor & Destructor Documentation

MbusSlaveServer (MbusDataTableInterface ∗ *dataTablePtr* = `NULL`) `[protected]`

Constructs a MbusSlaveServer object and associates it with a Data Provider.

Parameters:

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.

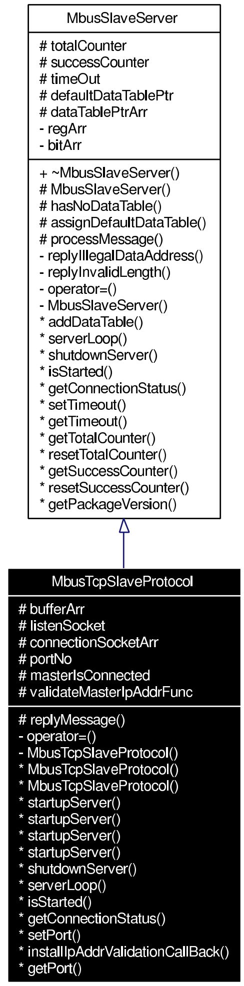
~MbusSlaveServer () `[virtual]`

Destructor.

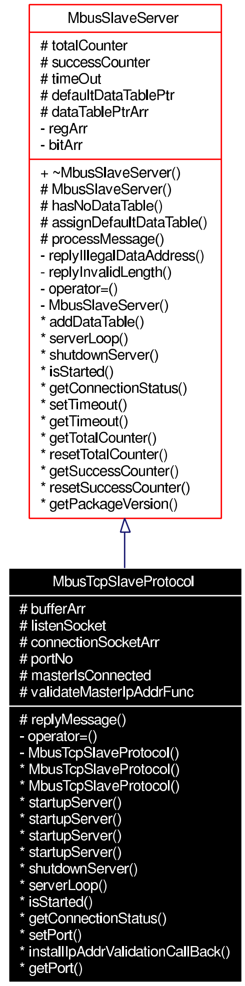
Shuts down server and releases any resources.

3.6 MbusTcpSlaveProtocol Class Reference

Inheritance diagram for MbusTcpSlaveProtocol:



Collaboration diagram for MbusTcpSlaveProtocol:



3.6.1 Detailed Description

MODBUS/TCP Slave Protocol class.

This class realises 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 TCP server. Upon receipt of a valid master query the server engine calls Data Provider methods to exchange data with the user application. For a more detailed description which Modbus data and control functions have been implemented in the server engine see section Server Functions common to all Protocol Flavours.

See also:

Server Functions common to all Protocol Flavours, MbusSlaveServer

MODBUS/TCP Server Management Functions

- MbusTcpSlaveProtocol ()
  Constructs a MbusTcpSlaveProtocol object.
- MbusTcpSlaveProtocol (MbusDataTableInterface *dataTablePtr)
  Constructs a MbusTcpSlaveProtocol object data and associates it with a Data Provider.
- int startupServer ()
  Puts the Modbus server into operation.
- int startupServer (const char *const hostName)
  Puts the Modbus server into operation.
- int startupServer (int slaveAddr)
  Puts the Modbus server into operation using a single slave address and data table.
- int startupServer (int slaveAddr, const char *const hostName)
  Puts the Modbus server into operation using a single slave address and data table.
- void shutdownServer ()
  Shuts down the Modbus server.
- int serverLoop ()
  MODBUS/TCP slave server loop.
- int isStarted ()
  Returns if server has been started up.
- int getConnectionStatus ()
  Checks if a Modbus master is polling periodically.
- int setPort (unsigned short portNo)
  Sets the TCP port number to be used by the protocol.
- void installIpAddrValidationCallBack (int(*f)(char *masterIpAddrSz))
  This function installs a callback handler for validating a master's IP address.
- unsigned short getPort ()
  Returns the TCP port number used by the protocol.

Server Management Functions

- int addDataTable (int slaveAddr, MbusDataTableInterface *dataTablePtr)
  Associates a protocol object with a Data Provider and a slave address.

Protocol Configuration

- long setTimeout (long timeOut)
  Configures master transmit time-out supervision.

- long getTimeout ()
  Returns the master time-out supervision value.

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.

Utility Functions

- static char * getPackageVersion ()
  Returns the package version number.

3.6.2 Constructor & Destructor Documentation

MbusTcpSlaveProtocol ()

Constructs a MbusTcpSlaveProtocol object.

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

MbusTcpSlaveProtocol (MbusDataTableInterface * *dataTablePtr*)

Constructs a MbusTcpSlaveProtocol object data and associates it with a Data Provider.

Function is kept for compatibility with previous API versions, do not use for new implementations.

Parameters:

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.

Deprecated

This function is deprecated. The preferred way of assigning a dataTable is using the default constructor and configuring data table and slave address using addDataTable method.

3.6.3 Member Function Documentation

int startupServer ()

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 Protocol Errors and Exceptions for a list of error codes.

int startupServer (const char ∗const *hostName*)

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 Protocol Errors and Exceptions for a list of error codes.

int startupServer (int *slaveAddr*)

Puts the Modbus server into operation using a single slave address and data table.

The server accepts connections on any interface.

Function is kept for compatibility with previous API versions, do not use for new implementations.

Note:

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

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.

Returns:

FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Deprecated

This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

int startupServer (int *slaveAddr*, const char ∗const *hostName*)

Puts the Modbus server into operation using a single slave address and data table.

Function is kept for compatibility with previous API versions, do not use for new implementations.

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.

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 Protocol Errors and Exceptions for a list of error codes.

Deprecated

This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

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 MbusSlaveServer.

int serverLoop () [virtual]

MODBUS/TCP 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. This server engine can handle multiple TCP/IP connections at the same time.

Returns:
FTALK_SUCCESS on success or error code. See Protocol Errors and Exceptions for a list of error codes.

Implements MbusSlaveServer.

int isStarted () [virtual]

Returns if server has been started up.

Return values:
true = started
false = shutdown

Implements MbusSlaveServer.

int getConnectionStatus () [virtual]

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.

Implements MbusSlaveServer.

int setPort (unsigned short *portNo*)

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

Remarks:

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

Note:

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

Parameters:

portNo Port number the server shall listen on

Return values:

FTALK_SUCCESS Success

FTALK_ILLEGAL_STATE_ERROR Server already running

void installIpAddrValidationCallBack (int(∗)(char ∗masterIpAddrSz) *f*)

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:

masterIpAddrSz IPv4 Internet host address string in the standard numbers-and-dots notation.

Returns:

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

unsigned short getPort ()

Returns the TCP port number used by the protocol.

Returns:
Port number used by the protocol

4 Modbus Slave C++ Library Page Documentation

4.1 How to integrate the Protocol in your Application

4.1.1 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;

   // 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

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

5. Start-up the server

```
   int result;

   result = mbusServer.startupServer(portName,
                                     9600L, // Baudrate
                                     8,     // Databits
                                     1,     // Stopbits
                                     0);    // 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();
```

4.1.2 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();
```

4.1.3 Examples

- A Tiny Slave
- Shared Memory Data Provider example
- Diagnostic Slave

4.2 Examples

- A Tiny Slave
- Shared Memory Data Provider example
- Diagnostic Slave

4.2.1 A Tiny Slave

The following example tinyslave.cpp shows how to implement a small Modbus RTU slave:

```
// Platform header
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Include FieldTalk package header
#include "MbusRtuSlaveProtocol.hpp"


/*****************************************************************************
 * Gobal data
 *****************************************************************************/

#if defined(__LINUX__)
   char *portName = "/dev/ttyS0";
#elif defined(__WIN32__) || defined(__CYGWIN__)
   char *portName = "COM1";
#elif defined(__FREEBSD__) || defined(__NETBSD__) || defined(__OPENBSD__)
   char *portName = "/dev/ttyd0";
#elif defined(__QNX__)
   char *portName = "/dev/ser1";
#elif defined(__VXWORKS__)
   char *portName = "/tyCo/0";
#elif defined(__IRIX__)
   char *portName = "/dev/ttyf1";
#elif defined(__SOLARIS__)
   char *portName = "/dev/ttya";
#elif defined(__OSF__)
   char *portName = "/dev/tty00";
#else
#  error Unknown platform, please add an entry for portName
#endif


/*****************************************************************************
 * Modbus data table
 *****************************************************************************/
```

```
typedef struct
{
   short actTemp;           // Register 1
   short minTemp;           // Register 2
   long  scanCounter;       // Register 3 and 4
   float setPoint;          // Register 5 and 6
   short statusReg;         // Register 7
   short configType;        // Register 8
} MyDeviceData;

MyDeviceData deviceData;


/*****************************************************************************
 * Data provider
 *****************************************************************************/

class MyDataProvider: public MbusDataTableInterface
{

  public:

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

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

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


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

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

      //
      // Copy data
```

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

} dataProvider;


/*****************************************************************************
 * Modbus protocol declaration
 *****************************************************************************/

MbusRtuSlaveProtocol mbusServer;


/*****************************************************************************
 * Function implementation
 *****************************************************************************/

void startupServer()
{
   int result;

   result = mbusServer.addDataTable(1, &dataProvider);
   if (result == FTALK_SUCCESS)
      result = mbusServer.startupServer(portName,
                                        9600L, // Baudrate
                                        8,     // Databits
                                        1,     // Stopbits
                                        0);    // Parity
   if (result != FTALK_SUCCESS)
   {
      fprintf(stderr, "Error starting server: %s!\n",
              getBusProtocolErrorText(result));
      exit(EXIT_FAILURE);
   }
}


void shutdownServer()
{
   mbusServer.shutdownServer();
}


void runServer()
{
   int result = FTALK_SUCCESS;

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

```
int main()
{
   atexit(shutdownServer);
   startupServer();
   runServer();
   return (EXIT_FAILURE);
}
```

4.2.2 Shared Memory Data Provider example

The following example shows how to implement a Data Provider which serves it's data from shared memory:

```
class ShmemMbusDataTable: public MbusDataTableInterface
{
  public:

   ShmemMbusDataTable(int table0Size, int table1Size, int table3Size, int table4Si
   {
      int i;
      int fd;

      for (i = 0; i < 5; i++)
      {
         //
         // Determine table sizes
         //
         switch (i)
         {
           case 0:
              if (table0Size <= 0)
                 break; // No table will be created for this type
              dataTableArr[i].size = table0Size * sizeof(char);
           break;
           case 1:
              if (table1Size <= 0)
                 break; // No table will be created for this type
              dataTableArr[i].size = table1Size * sizeof(char);
           break;
           case 2:
              dataTableArr[i].size = sizeof(SlaveStatusInfo);
           break;
           case 3:
              if (table3Size <= 0)
                 break; // No table will be created for this type
              dataTableArr[i].size = table3Size * sizeof(short);
           break;
           case 4:
              if (table4Size <= 0)
```

```
              break; // No table will be created for this type
           dataTableArr[i].size = table4Size * sizeof(short);
        break;
      }

      //
      // Open shared memory tables
      //
      shm_unlink(dataTableArr[i].name);
      if (dataTableArr[i].size == 0)
         continue;
      fd = shm_open(dataTableArr[i].name, O_CREAT | O_RDWR | O_EXCL, S_IRWXU);
      if (fd < 0)
      {
         perror("Shared memory open failed");
         abort();
      }

      //
      // Size it
      //
      if (ftruncate(fd, dataTableArr[i].size) < 0)
      {
         perror("Shared memory ftruncate failed");
         abort();
      }

      //
      // Map shared memory into address space
      //
      dataTableArr[i].ptr = (short *) mmap(0, dataTableArr[i].size,
                                           PROT_READ | PROT_WRITE,
                                           MAP_SHARED, fd, 0L);
      if (dataTableArr[i].ptr == NULL)
      {
         perror("Shared memory mmap failed");
         abort();
      }

      close  (fd);  // No we can close the file descriptor
      memset(dataTableArr[i].ptr, 0, dataTableArr[i].size);
   }

   //
   // Handle special cases where we map table 3 to point to table 4 and
   // table 1 to point to table 0
   //
   if (table3Size == -1) // Map table 3 to table 4
   {
      dataTableArr[3].size = dataTableArr[4].size;
      dataTableArr[3].ptr = dataTableArr[4].ptr;
   }
   if (table1Size == -1) // Map table 1 to table 0
   {
      dataTableArr[1].size = dataTableArr[0].size;
      dataTableArr[1].ptr = dataTableArr[0].ptr;
   }
```

```
   // Setup pointer to status table
   slaveStatusInfoPtr = (SlaveStatusInfo *) dataTableArr[2].ptr;

}


~ShmemMbusDataTable()
{
   int i;

   for (i = 0; i < 5; i++)
   {
      if (dataTableArr[i].ptr != NULL)
      {
         munmap(dataTableArr[i].ptr, dataTableArr[i].size);
         shm_unlink(dataTableArr[i].name);
      }
   }
}


void timeOutHandler()
{
   slaveStatusInfoPtr->timeoutCnt++;
}


int readInputDiscretesTable(int startRef,
                            char bitArr[],
                            int refCnt)
{
   printf("\nreadInputDiscretesTable: %d, %d\n", startRef, refCnt);

   // Adjust Modbus reference counting
   startRef--;

   //
   // Validate range
   //
   if (startRef + refCnt > (int) (dataTableArr[1].size / sizeof(char)))
      return (0);

   //
   // Copy data
   //
   memcpy(bitArr, &((char *) dataTableArr[1].ptr)[startRef], refCnt * sizeof(ch
   slaveStatusInfoPtr->readDiscretesCnt++;
   return (1);
}


int readCoilsTable(int startRef,
                   char bitArr[],
                   int refCnt)
{
   printf("\nreadCoilsTable: %d, %d\n", startRef, refCnt);
```

```
   // Adjust Modbus reference counting
   startRef--;

   //
   // Validate range
   //
   if (startRef + refCnt > (int) (dataTableArr[0].size / sizeof(char)))
      return (0);

   //
   // Copy data
   //
   memcpy(bitArr, &((char *) dataTableArr[0].ptr)[startRef], refCnt * sizeof(ch
   slaveStatusInfoPtr->readDiscretesCnt++;
   return (1);
}


int writeCoilsTable(int startRef,
                    const char bitArr[],
                    int refCnt)
{
   printf("\nriteCoilsTable: %d, %d\n", startRef, refCnt);

   // Adjust Modbus reference counting
   startRef--;

   //
   // Validate range
   //
   if (startRef + refCnt > (int) (dataTableArr[0].size / sizeof(char)))
      return (0);

   //
   // Copy data
   //
   memcpy(&((char *) dataTableArr[0].ptr)[startRef], bitArr, refCnt * sizeof(ch
   slaveStatusInfoPtr->writeDiscretesCnt++;
   return (1);
}


int readInputRegistersTable(int startRef,
                            short regArr[],
                            int refCnt)
{
   printf("\nreadInputRegistersTable: %d, %d\n", startRef, refCnt);

   // Adjust Modbus reference counting
   startRef--;

   //
   // Validate range
   //
   if (startRef + refCnt > (int) (dataTableArr[3].size / sizeof(short)))
      return (0);
```

```
   //
   // Copy data
   //
   memcpy(regArr, &((short *) dataTableArr[3].ptr)[startRef], refCnt * sizeof(s
   slaveStatusInfoPtr->readRegistersCnt++;
   return (1);
}


int readHoldingRegistersTable(int startRef,
                              short regArr[],
                              int refCnt)
{
   printf("\nreadHoldingRegistersTable: %d, %d\n", startRef, refCnt);

   // Adjust Modbus reference counting
   startRef--;

   //
   // Validate range
   //
   if (startRef + refCnt > (int) (dataTableArr[4].size / sizeof(short)))
      return (0);

   //
   // Copy data
   //
   memcpy(regArr, &((short *) dataTableArr[4].ptr)[startRef], refCnt * sizeof(s
   slaveStatusInfoPtr->readRegistersCnt++;
   return (1);
}


int writeHoldingRegistersTable(int startRef,
                               const short regArr[],
                               int refCnt)
{
   printf("\nwriteHoldingRegistersTable: %d, %d\n", startRef, refCnt);

   // Adjust Modbus reference counting
   startRef--;

   //
   // Validate range
   //
   if (startRef + refCnt > (int) (dataTableArr[4].size / sizeof(short)))
      return (0);

   //
   // Copy data
   //
   memcpy(&((short *) dataTableArr[4].ptr)[startRef], regArr, refCnt * sizeof(s
   slaveStatusInfoPtr->writeRegistersCnt++;
   return (1);
}
```

```
  private:

   SlaveStatusInfo *slaveStatusInfoPtr;

};
```

4.2.3 Diagnostic Slave

The following more complex example diagslave.cpp shows how to use the protocol stack in a context where the user can select the protocol type (TCP, RTU and ASCII) and other parameters. Diagslave is a slave simulator and test tool.

```
// Platform header
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Include FieldTalk package header
#include "MbusRtuSlaveProtocol.hpp"
#include "MbusAsciiSlaveProtocol.hpp"
#include "MbusTcpSlaveProtocol.hpp"
#include "DiagnosticDataTable.hpp"

#ifdef _WIN32
#  include "getopt.h"
#else
#  include <unistd.h>
#endif


/*****************************************************************************
 * String constants
 *****************************************************************************/

const char versionStr[]= "$Revision: 1.14 $";
const char progName[] = "diagslave";
const char bannerStr[] =
"\n"
"%s - FieldTalk(tm) Modbus(R) Diagnostic Slave\n"
"Copyright (c) 2002-2006 FOCUS Software Engineering Pty Ltd\n"
"Visit http://www.modbusdriver.com for Modbus libraries and tools.\n"
"\n";

const char usageStr[] =
"%s [options] [serialport]\n"
"Arguments: \n"
"serialport    Serial port when using Modbus ASCII or Modbus RTU protocol \n"
"              COM1, COM2 ...                on Windows \n"
```

```
"               /dev/ttyS0, /dev/ttyS1 ...    on Linux \n"
"               /dev/ser1, /dev/ser2 ...      on QNX \n"
"General options:\n"
"-m ascii      Modbus ASCII protocol\n"
"-m rtu        Modbus RTU protocol (default)\n"
"-m tcp        MODBUS/TCP protocol\n"
"-t #          Master poll time-out in ms (0-100000, 3000 is default)\n"
"-a #          Slave address (1-255 for RTU/ASCII, 0-255 for TCP)\n"
"Options for MODBUS/TCP:\n"
"-p #          TCP port number (502 is default)\n"
"Options for Modbus ASCII and Modbus RTU:\n"
"-b #          Baudrate (e.g. 9600, 19200, ...) (9600 is default)\n"
"-d #          Databits (7 or 8 for ASCII protocol, 8 for RTU)\n"
"-s #          Stopbits (1 or 2, 1 is default)\n"
"-p none       No parity\n"
"-p even       Even parity (default)\n"
"-p odd        Odd parity\n"
"";


/*****************************************************************************
 * Enums
 *****************************************************************************/

enum
{
   RTU,
   ASCII,
   TCP
};


/*****************************************************************************
 * Gobal configuration data
 *****************************************************************************/

int address = -1;
long timeout = 3000;
long baudRate = 9600;
int dataBits = 8;
int stopBits = 1;
int parity = MbusSerialSlaveProtocol::SER_PARITY_EVEN;
int protocol = RTU;
char *portName = NULL;
int port = 502;


/*****************************************************************************
 * Protocol and data table
 *****************************************************************************/

DiagnosticMbusDataTable *dataTablePtrArr[256];
MbusSlaveServer *mbusServerPtr = NULL;


/*****************************************************************************
 * Function implementation
```

```
 ****************************************************************************/

void printUsage()
{
   printf("Usage: ");
   printf(usageStr, progName);
   exit(EXIT_SUCCESS);
}


void printVersion()
{
   printf(bannerStr, progName);
   printf("Version: %s using FieldTalk package version %s\n",
          versionStr, MbusSlaveServer::getPackageVersion());
}


void printConfig()
{
   printf(bannerStr, progName);
   printf("Protocol configuration: ");
   switch (protocol)
   {
      case RTU:
         printf("Modbus RTU\n");
      break;
      case ASCII:
         printf("Modbus ASCII\n");
      break;
      case TCP:
         printf("MODBUS/TCP\n");
      break;
      default:
         printf("unknown\n");
      break;
   }
   printf("Slave configuration: ");
   printf("Address = %d, ", address);
   printf("Master Time-out = %ld\n", timeout);
   if (protocol == TCP)
   {
      printf("TCP configuration: ");
      printf("Port = %d\n", port);
   }
   else
   {
      printf("Serial port configuration: ");
      printf("%s, ", portName);
      printf("%ld, ", baudRate);
      printf("%d, ", dataBits);
      printf("%d, ", stopBits);
      switch (parity)
      {
         case MbusSerialSlaveProtocol::SER_PARITY_NONE:
            printf("none\n");
         break;
```

```
            case MbusSerialSlaveProtocol::SER_PARITY_EVEN:
               printf("even\n");
            break;
            case MbusSerialSlaveProtocol::SER_PARITY_ODD:
               printf("odd\n");
            break;
            default:
               printf("unknown\n");
            break;
         }
      }
      printf("\n");
   }


   void exitBadOption(const char *const text)
   {
      fprintf(stderr, "%s: %s! Try -h for help.\n", progName, text);
      exit(EXIT_FAILURE);
   }


   void scanOptions(int argc, char **argv)
   {
      int c;

      // Check for --version option
      for (c = 1; c < argc; c++)
      {
         if (strcmp (argv[c], "--version") == 0)
         {
            printVersion();
            exit(EXIT_SUCCESS);
         }
      }

      // Check for --help option
      for (c = 1; c < argc; c++)
      {
         if (strcmp (argv[c], "--help") == 0)
            printUsage();
      }

      opterr = 0; // Disable getopt’s error messages
      for(;;)
      {
         c = getopt(argc, argv, "ha:b:d:s:p:m:");
         if (c == -1)
            break;

         switch (c)
         {
            case ’m’:
               if (strcmp(optarg, "tcp") == 0)
               {
                  protocol = TCP;
               }
```

```
            else
               if (strcmp(optarg, "rtu") == 0)
               {
                  protocol = RTU;
               }
               else
                  if (strcmp(optarg, "ascii") == 0)
                  {
                     protocol = ASCII;
                  }
                  else
                  {
                     exitBadOption("Invalid protocol parameter");
                  }
         break;
         case ’a’:
            address = strtol(optarg, NULL, 0);
            if ((address < -1) || (address > 255))
               exitBadOption("Invalid address parameter");
         break;
         case ’t’:
            timeout = strtol(optarg, NULL, 0);
            if ((timeout < 0) || (timeout > 100000))
               exitBadOption("Invalid time-out parameter");
         break;
         case ’b’:
            baudRate = strtol(optarg, NULL, 0);
            if (baudRate == 0)
               exitBadOption("Invalid baudrate parameter");
         break;
         case ’d’:
            dataBits = (int) strtol(optarg, NULL, 0);
            if ((dataBits != 7) || (dataBits != 8))
               exitBadOption("Invalid databits parameter");
         break;
         case ’s’:
            stopBits = (int) strtol(optarg, NULL, 0);
            if ((stopBits != 1) || (stopBits != 2))
               exitBadOption("Invalid stopbits parameter");
         break;
         case ’p’:
            if (strcmp(optarg, "none") == 0)
            {
               parity = MbusSerialSlaveProtocol::SER_PARITY_NONE;
            }
            else
               if (strcmp(optarg, "odd") == 0)
               {
                  parity = MbusSerialSlaveProtocol::SER_PARITY_ODD;
               }
               else
                  if (strcmp(optarg, "even") == 0)
                  {
                     parity = MbusSerialSlaveProtocol::SER_PARITY_EVEN;
                  }
                  else
                  {
```

```
                        port = strtol(optarg, NULL, 0);
                        if ((port <= 0) || (port > 0xFFFF))
                           exitBadOption("Invalid parity or port parameter");
                     }
            break;
            case ’h’:
               printUsage();
            break;
            default:
               exitBadOption("Unrecognized option or missing option parameter");
            break;
         }
      }

      if (protocol == TCP)
      {
         if ((argc - optind) != 0)
            exitBadOption("Invalid number of parameters");
      }
      else
      {
         if ((argc - optind) != 1)
            exitBadOption("Invalid number of parameters");
         else
            portName = argv[optind];
      }
   }



   int validateMasterIpAddr(char* masterIpAddrSz)
   {
      printf("\nvalidateMasterIpAddr: accepting connection from %s\n",
              masterIpAddrSz);
      return (1);
   }


   void startupServer()
   {
      int i;
      int result = -1;

      switch (protocol)
      {
         case RTU:
            mbusServerPtr = new MbusRtuSlaveProtocol();
            if (address == -1)
            {
               for (i = 1; i < 255; i++)
               mbusServerPtr->addDataTable(i, dataTablePtrArr[i]);
            }
            else
               mbusServerPtr->addDataTable(address, dataTablePtrArr[address]);
            mbusServerPtr->setTimeout(timeout);
            result = ((MbusRtuSlaveProtocol *) mbusServerPtr)->startupServer(
```

```
                     portName, baudRate, dataBits, stopBits, parity);
         break;
         case ASCII:
            mbusServerPtr = new MbusAsciiSlaveProtocol();
            if (address == -1)
            {
               for (i = 1; i < 255; i++)
               mbusServerPtr->addDataTable(i, dataTablePtrArr[i]);
            }
            else
               mbusServerPtr->addDataTable(address, dataTablePtrArr[address]);
            mbusServerPtr->setTimeout(timeout);
            result = ((MbusAsciiSlaveProtocol *) mbusServerPtr)->startupServer(
                      portName, baudRate, dataBits, stopBits, parity);
         break;
         case TCP:
            mbusServerPtr = new MbusTcpSlaveProtocol();
            if (address == -1)
            {
               for (i = 0; i < 255; i++) // Note: TCP support slave addres of 0
                  mbusServerPtr->addDataTable(i, dataTablePtrArr[i]);
            }
            else
               mbusServerPtr->addDataTable(address, dataTablePtrArr[address]);
            mbusServerPtr->setTimeout(timeout);
            ((MbusTcpSlaveProtocol *) mbusServerPtr)->installIpAddrValidationCallBack
            ((MbusTcpSlaveProtocol *) mbusServerPtr)->setPort((unsigned short) port);
            result = ((MbusTcpSlaveProtocol *) mbusServerPtr)->startupServer();
         break;
      }
      switch (result)
      {
         case FTALK_SUCCESS:
            printf("Server started up successfully.\n");
         break;
         case FTALK_ILLEGAL_ARGUMENT_ERROR:
            fprintf(stderr, "Configuration setting not supported!\n");
            exit(EXIT_FAILURE);
         break;
         default:
            fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));
            exit(EXIT_FAILURE);
         break;
      }
   }


   void shutdownServer()
   {
      printf("Shutting down server.\n");
      delete mbusServerPtr;
   }


   void runServer()
   {
      int result = FTALK_SUCCESS;
```

```
   printf("Listening to network (Ctrl-C to stop)\n");
   while (result == FTALK_SUCCESS)
   {
      result = mbusServerPtr->serverLoop();
      if (result != FTALK_SUCCESS)
         fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));\
      else
      {
         printf(".");
         fflush(stdout);
      }
   }
}


int main(int argc, char **argv)
{
   int i;

   // Construct data tables
   for (i = 0; i < 255; i++)
   {
      dataTablePtrArr[i] = new DiagnosticMbusDataTable(i);
   }

   scanOptions(argc, argv);
   printConfig();
   atexit(shutdownServer);
   startupServer();
   runServer();
   return (EXIT_FAILURE);
}
```

Diagslave uses the following diagnostic data table as Data Provider:

```
#ifndef _DIAGNOSTICDATATABLE_H_INCLUDED
#define _DIAGNOSTICDATATABLE_H_INCLUDED


// Platform header
#include <stdio.h>
#include <string.h>

// Package header
#include "MbusDataTableInterface.hpp"


/*****************************************************************************
 * DiagnosticMbusDataTable class declaration
 *****************************************************************************/

class DiagnosticMbusDataTable: public MbusDataTableInterface
{

public:
```

```
DiagnosticMbusDataTable(int slaveAddr)
{
   this->slaveAddr = slaveAddr;
   memset(regData, 0, sizeof(regData));
   memset(bitData, 0, sizeof(bitData));
}


~DiagnosticMbusDataTable()
{
}


char readExceptionStatus()
{
   printf("\rSlave %3d: readExceptionStatus\n", slaveAddr);
   return (0x55);
}


int readInputDiscretesTable(int startRef,
                            char bitArr[],
                            int refCnt)
{
   printf("\rSlave %3d: readInputDiscretes from %d, %d references\n",
          slaveAddr, startRef, refCnt);

   // Adjust Modbus reference counting
   startRef--;

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

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


int readCoilsTable(int startRef,
                   char bitArr[],
                   int refCnt)
{
   printf("\rSlave %3d: readCoils from %d, %d references\n",
          slaveAddr, startRef, refCnt);

   // Adjust Modbus reference counting
   startRef--;

   //
   // Validate range
```

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

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


int writeCoilsTable(int startRef,
                    const char bitArr[],
                    int refCnt)
{
   printf("\rSlave %3d: writeCoils from %d, %d references\n",
          slaveAddr, startRef, refCnt);

   // Adjust Modbus reference counting
   startRef--;

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

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


int readInputRegistersTable(int startRef,
                            short regArr[],
                            int refCnt)
{
   printf("\rSlave %3d: readInputRegisters from %d, %d references\n",
          slaveAddr, startRef, refCnt);

   // Adjust Modbus reference counting
   startRef--;

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

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

```
}


int readHoldingRegistersTable(int startRef,
                              short regArr[],
                              int refCnt)
{
   printf("\rSlave %3d: readHoldingRegisters from %d, %d references\n",
          slaveAddr, startRef, refCnt);

   // Adjust Modbus reference counting
   startRef--;

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

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


int writeHoldingRegistersTable(int startRef,
                               const short regArr[],
                               int refCnt)
{
   printf("\rSlave %3d: writeHoldingRegisters from %d, %d references\n",
          slaveAddr, startRef, refCnt);

   // Adjust Modbus reference counting
   startRef--;

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

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


int validateMasterIpAddr(char* masterIpAddrSz)
{
   printf("\nvalidateMasterIpAddr: accepting connection from %s\n",
          masterIpAddrSz);
   return (1);
}
```

```
  private:

   int slaveAddr;
   short regData[0x10000];
   char bitData[2000];

};


#endif // ifdef ..._H_INCLUDED
```

4.3 What You should know about Modbus

4.3.1 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.

4.3.2 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 RS485 line driver and support enabling and disabling of the RS485 transmitter via the RTS signal. Some FieldTalk C++ editions support this RTS driven RS485 mode.

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:

Function Code	Current Terminology	Classic Terminology
Bit Access		
1	Read Coils	Read Coil Status
2	Read Inputs Discretes	Read Input Status
5 (05 hex)	Write Coil	Force Single Coil
15 (0F hex)	Force 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 Register
23 (17 hex)	Read/Write Registers	Read/Write Registers
Diagnostics		
7 (07 hex)	Read Exception Status	Read Exception Status
8 sub code 00	Diagnostics - Return Query Data	Diagnostics - Return Query Data

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.

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	16-bit quantity, alterable by an application program, read-write
Discrete inputs	Inputs	1:00000	Single bit, provided by an I/O system, read-only
Input registers	Input registers	3:00000	16-bit quantity, provided by an I/O system, read-only
Output registers	Holding registers	4:00000	Single bit, 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.

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.

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

- Reading and Writing bit values
- Reading and Writing 16-bit integers
- Reading and Writing 32-bit integers
- Reading and Writing 32-bit floats
- Configuring the word order and representation for 32-bit values

The FieldTalk Modbus Slave Library defines services to

- Read and Write bit values
- Read and Write 16-bit integers

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

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

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

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

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

The following table shows the correlation between Discrete Numbers and

Bit Numbers:

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

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

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.

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.

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.

4.4 Installation and Source Code Compilation

4.4.1 Linux, UNIX and QNX Systems: Unpacking and Compiling the Source

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

2. Uncompress the zipped tarball using gzip:

```
# gunzip FT-MB??-??-ALL.2.4.0.tar.gz
```

3. Untar the tarball

```
# tar xf FT-MB??-??-ALL.2.4.0.tar
```

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

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

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

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

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

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

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

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

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

Platform	Library Directory
Linux	lib/linux
QNX 6	lib/qnx6
QNX 4	lib/qnx4
Irix	lib/irix
OSF1/True 64/Digital UNIX	lib/osf
Solaris	lib/solaris
HP-UX	lib/hpux
IBM AIX	lib/aix

Your directory structure looks now like:

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

6. The library is ready to be used.

4.4.2 Windows Systems: Unpacking and Compiling the Source

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

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

```
# unzip FT-MB??-WIN-ALL.2.4.0.zip
```

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

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

3. Compile the library from the source code.

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

If you are using Microsoft C++ and nmake:

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

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

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

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

Your directory structure looks now like:

```
myprj
  |
  +-- fieldtalk
      |
      +-- doc
      +-- src
      +-- include
      +-- samples
      +-+ lib
        |
        +-- win
        |   |
        |   +-- win32
        |       |
        |       +-- release
        |
        +-- wce
            |
            +-- [platformname]
                |
                +-- release
```

5. The library is ready to be used.

4.4.3 Specific Platform Notes

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

```
./make uclinux
```

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

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

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

QNX 4 In order to get proper control over Modebus timing, you have to adjust the system's clock rate. The standard ticksize is not suitable for Modbus RTU and needs to be adjusted. Configure the ticksize to be <= 1 ms.

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.

4.5 Linking your Applications against the Library

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

Let's assume the following project directory structure:

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

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

Example:

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

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

Example:

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

4.5.2 Windows Systems: Compiling and Linking Applications

Let's assume the following project directory structure:

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

```
+-- src
+-- include
+-+ lib
  |
  +-- win
      |
      +-- win32
          |
          +-- release
```

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

Visual C++ Example:

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

Borland C++ Example:

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

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

Visual C++ Example:

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

4.6 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 Java language, Object Pascal and for C++ 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*.

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

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

4.7 License

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

FOCUS Software Engineering Pty Ltd, Brisbane/Australia, ACN 104 080 935
Library License
Version 3, May 2003
Copyright (c) 2002-2003 FOCUS Software Engineering Pty Ltd. All rights reserved.
```

```
Definitions:

   "Package" 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 FOCUS Software Engineering.

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

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

   "Distributable Components" are dynamic libraries, shared libraries,
   class files and similar components supplied by FOCUS Software
   Engineering for redistribution. They must be listed in a "readme"
   or "deploy" file included with the Package.

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


1.  In consideration of payment of the licence fee and your agreement
    to abide by the terms and conditions of this licence, FOCUS
    Software Engineering grants You the following non-exclusive
    rights:

    a) You may use the Package on one or more computers by a
       single person who uses the software personally;
    b) You may use the Package nonsimultaneously by multiple people if
       it is installed on a single computer;
    c) You may use the Package on a network, provided that the network
       is operated by the organisation who purchased the license and
       there is no concurrent use of the Package;
    d) You may copy the Package for archival purposes.

2.  You may reproduce and distribute, in executable form only,
    Applications linked with static libraries supplied as part of the
    Package 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 Package source code or part thereof.

3. If You have received this Package for the purpose of evaluation, FOCUS Software Engineering grants You a non-exclusive license to use the Package free of charge for the purpose of evaluating whether to purchase an ongoing license to use the Package. The evaluation period is limited to 30 days and does not include the right to reproduce and distribute Applications using the Package. At the end of the evaluation period, if You do not purchase a license, You must uninstall the Package 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 Package or its derivative works. These actions are prohibited by law if You do not accept this License. Therefore, by using or distributing the Package (or any work based on the Package), You indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or using the Package or works based on it.

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

6. Where source code is provided as part of the Package, 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 Package’s software design, source code and documentation or any part thereof to any third party without the expressed written consent from FOCUS Software Engineering.

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 Package.

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

10. The License is not transferable without written permission from FOCUS Software Engineering.

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

12. THERE IS NO WARRANTY FOR THE PACKAGE, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING FOCUS

-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.2.1 (MingW32)

iD8DBQE+yb3kCO2PJievT8IRAla5AKDQforpJqjriti20XFTYe5REbV7dgCfSEV2
u0c3NnXlrOMkayn4txSMgv4=
=WAla
-----END PGP SIGNATURE-----

4.8 Support

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

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

4.9 Notices

Disclaimer: FOCUS Software Engineering makes no warranty for the use of its products, other than those expressly contained in the Company146s standard warranty which is detailed in FOCUS Software Engineering146s Terms and Conditions located on the Company146s 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 FOCUS Software Engineering are granted by the Company in connection with the sale of FOCUS Software Engineering's products, expressly or by implication. FOCUS Software Engineering146s products are not authorized for use as critical components in life support devices or systems.

FieldTalk™was developed by:

FOCUS Software Engineering Pty Ltd, Australia.

4.10 Deprecated List

Member startupServer(int slaveAddr, const char *const portName, long baudRate, int dat
This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

Member startupServer(int slaveAddr, const char *const portName, long baudRate)
This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

Member MbusAsciiSlaveProtocol(MbusDataTableInterface *dataTablePtr)
This function is deprecated. The preferred way of assigning a dataTable is using the default constructor and configuring data table and slave address using addDataTable method.

Member MbusRtuSlaveProtocol(MbusDataTableInterface *dataTablePtr)
This function is deprecated. The preferred way of assigning a dataTable is using the default constructor and configuring data table and slave address using addDataTable method.

Member startupServer(int slaveAddr, const char *const portName, long baudRate, int dat
This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

Member MbusTcpSlaveProtocol(MbusDataTableInterface *dataTablePtr)
This function is deprecated. The preferred way of assigning a dataTable is using the default constructor and configuring data table and slave address using addDataTable method.

Member startupServer(int slaveAddr) This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

Member startupServer(int slaveAddr, const char *const hostName) This function is deprecated. The preferred way of assigning a slave address is using the default constructor and configuring data table and slave address using addDataTable method.

Index