FOCUS
SOFTWARE ENGINEERING

FieldTalk™ Modbus® Master Protocol Library / C++ Editions

Last updated: 26 May 2004

FOCUS Software Engineering Pty Ltd, Australia.

Contents

1 General Description

1.1 Introduction

This *FieldTalk*™ Modbus® Master Library - C++ Editions are a C++ class library suite which provides connectivity to Modbus slave compatible devices and applications.

Typical applications are supervisory control systems, data concentrators and gateways, user interfaces and factory information systems.

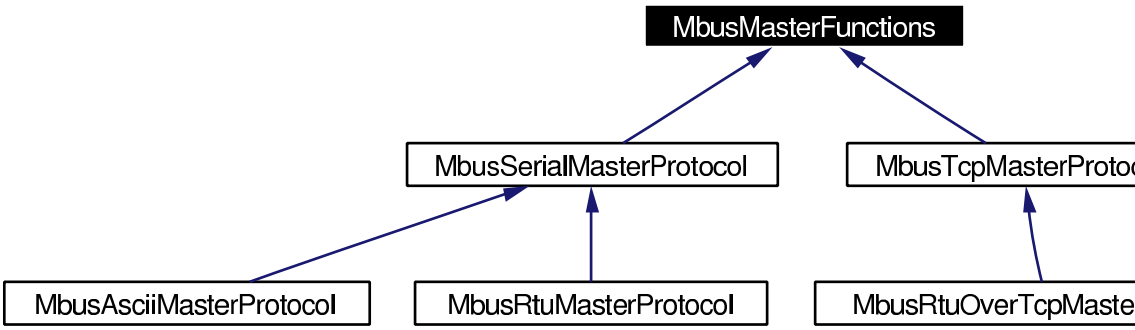
Features:

- Robust design suitable for real-time and industrial applications
- Full implementation of Class 0 and Class 1 Modbus functions as well as a subset of the most commonly used Class 2 functions
- Support for Advantec ADAM 5000/6000 Series Commands

- Standard Modbus boolean and 16-bit integer data types
- Support for 32-bit integer, modulo-10000 and float data types
- Configurable word alignment for 32-bit types (big-endian, little-endian)
- Support of Broadcasting
- Failure and transmission counters
- Transmission and connection time-out supervision
- Detailed transmission and protocol failure reporting using error codes
- Multi-platform design
- Scalable: you can use serial protocols only or TCP/IP or all of them
- Customized modifications and development of add-ons available

1.2 Library Structure

The library is organised into one class for each Modbus master protocol flavour and a common 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 base class MbusMasterFunctions contains all protocol unspecific functions, in particular the data and control functions defined by Modbus. All protocol flavours inherit from this base class.

The class MbusAsciiMasterProtocol implements the Modbus ASCII protocol, the class MbusRtuMasterProtocol implements the Modbus RTU protocol. The class MbusTcpMasterProtocol implements the MODBUS/TCP protocol and the class MbusRtuOverTcpMasterProtocol the Encapsulated Modbus RTU master protocol (also known as RTU over TCP or RTU/IP).

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

```
MbusRtuMasterProtocol mbusProtocol;
```

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

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

1.3 Overview

2 Modbus Protocol Library Documentation Module Documentation

2.1 Data and Control Functions for all Protocol Flavours

2.1.1 Detailed Description

This protocol stack 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 functions of conformance Class 0 and Class 1 have been implemented. In addition the most frequent used functions of conformance Class 2 have been implemented. This rich function set enables a user to solve nearly every Modbus data transfer problem.

The following table lists the available *FieldTalk Master Protocol Pack* functions:

Function Code	Current Terminology	Classic Terminology
Conformance Class 0		
3 (03 hex)	Read Multiple Registers	Read Holding Registers
16 (10 hex)	Write Multiple Registers	Preset Multiple Registers
Conformance Class 1		
1 (01 hex)	Read Coils	Read Coil Status
2 (02 hex)	Read Inputs Discretes	Read Input Status
4 (04 hex)	Read Input Registers	Read Input Registers
5 (05 hex)	Write Coil	Force Single Coil
6 (06 hex)	Write Single Register	Preset Single Register
7 (07 hex)	Read Exception Status	Read Exception Status
Conformance Class 2		
15 (0F hex)	Force Multiple Coils	Force Multiple Coils
22 (16 hex)	Mask Write Register	Mask Write Register
23 (17 hex)	Read/Write Registers	Read/Write Registers

Remarks:

When passing register numbers and discrete numbers to *FieldTalk* functions you have to use the the Modbus register and discrete numbering scheme. See Register and Discrete Numbering Scheme. (Internally the functions will deduct 1 from the start register value before transmitting the value to the slave device.)
Most slave devices are limiting the amount of registers to be exchanged with the ASCII protocol to be 62 registers or 496 discretes. The limitation is based on the fact that the buffer must not exceed 256 bytes.

Class 0 Modbus Functions

- int MbusMasterFunctions::writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)
  Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.

- int MbusMasterFunctions::writeMultipleLongInts (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.

- int MbusMasterFunctions::writeMultipleMod10000 (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

- int MbusMasterFunctions::writeMultipleFloats (int slaveAddr, int startRef, const float

float32Arr[ ], int refCnt)
Modbus function 16 (10 hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

- int MbusMasterFunctions::readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- int MbusMasterFunctions::readMultipleLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

- int MbusMasterFunctions::readMultipleMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

- int MbusMasterFunctions::readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Class 1 Modbus Functions

- int MbusMasterFunctions::readCoils (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 1 (01 hex), Read Coil Status/Read Coils.

- int MbusMasterFunctions::readInputDiscretes (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

- int MbusMasterFunctions::readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 4 (04 hex), Read Input Registers.

- int MbusMasterFunctions::readInputLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.

- int MbusMasterFunctions::readInputMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

- int MbusMasterFunctions::readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

- int MbusMasterFunctions::writeCoil (int slaveAddr, int bitAddr, int bitVal)

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

- int MbusMasterFunctions::writeSingleRegister (int slaveAddr, int regAddr, short regVal)
  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.

- int MbusMasterFunctions::readExceptionStatus (int slaveAddr, unsigned char *statusByte)
  Modbus function 7 (07 hex), Read Exception Status.

Class 2 Modbus Functions

- int MbusMasterFunctions::forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)
  Modbus function 15 (0F hex), Force Multiple Coils.

- int MbusMasterFunctions::maskWriteRegister (int slaveAddr, int regAddr, unsigned short andMask, unsigned short orMask)
  Modbus function 22 (16 hex), Mask Write Register.

- int MbusMasterFunctions::readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)
  Modbus function 23 (17 hex), Read/Write Registers.

Protocol Configuration

- int MbusMasterFunctions::setTimeout (int timeOut)
  Configures time-out.

- int MbusMasterFunctions::getTimeout ()
  Returns the time-out value.

- int MbusMasterFunctions::setPollDelay (int pollDelay)
  Configures poll delay.

- int MbusMasterFunctions::getPollDelay ()
  Returns the poll delay time.

- int MbusMasterFunctions::setRetryCnt (int retryCnt)
  Configures the automatic retry setting.

- int MbusMasterFunctions::getRetryCnt ()
  Returns the automatic retry count.

Transmission Statistic Functions

- unsigned long MbusMasterFunctions::getTotalCounter ()

Returns how often a message transfer has been executed.

- void MbusMasterFunctions::resetTotalCounter ()
  Resets total message transfer counter.

- unsigned long MbusMasterFunctions::getSuccessCounter ()
  Returns how often a message transfer was successful.

- void MbusMasterFunctions::resetSuccessCounter ()
  Resets successful message transfer counter.

Word Order Configuration

- void MbusMasterFunctions::configureBigEndianInts ()
  Configures int data type functions to do a word swap.

- void MbusMasterFunctions::configureSwappedFloats ()
  Configures float data type functions to do a word swap.

- void MbusMasterFunctions::configureLittleEndianInts ()
  Configures int data type functions not to do a word swap.

- void MbusMasterFunctions::configureIeeeFloats ()
  Configures float data type functions not to do a word swap.

Functions

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

2.1.2 Function Documentation

int writeMultipleRegisters (int *slaveAddr*, int *startRef*, const short *regArr*[], int *refCnt*) `[inherited]`

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

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 0x10000)

regArr Buffer with the data to be sent.

refCnt Number of references to be written (Range: 1-100)

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

Note:
Broadcast supported for serial protocols

int readCoils (int *slaveAddr*, int *startRef*, int *bitArr*[ ], int *refCnt*) [inherited]

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

Reads the contents of the discrete outputs (coils, 0:00000 table).

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

startRef Start reference (Range: 1 - 0x10000)

bitArr Buffer which will contain the data read

refCnt Number of references to be read (Range: 1-2000)

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

Note:
No broadcast supported

int forceMultipleCoils (int *slaveAddr*, int *startRef*, const int *bitArr*[ ], int *refCnt*) [inherited]

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

Writes binary values into a sequence of discrete outputs (coils, 0:00000 table).

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

startRef Start reference (Range: 1 - 0x10000)

bitArr Buffer which contains the data to be sent

refCnt Number of references to be written (Range: 1-800)

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

Note:
Broadcast supported for serial protocols

int setTimeout (int *msTime*) [inherited]

Configures time-out.

This function sets the operation or socket time-out to the specified value.

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

Note:
A protocol must be closed in order to configure it.

Parameters:
msTime Timeout value in ms (Range: 1 - 100000)

Return values:
FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

unsigned long getTotalCounter () [inherited]

Returns how often a message transfer has been executed.

Returns:
Counter value

void configureBigEndianInts () [inherited]

Configures int data type functions to do a word swap.

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

char * getPackageVersion () [static, inherited]

Returns the package version number.

Returns:
Package version string

int writeMultipleLongInts (int *slaveAddr*, int *startRef*, const long *int32Arr*[], int *refCnt*) [inherited]

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

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

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

Parameters:
slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 0x10000)

int32Arr Buffer with the data to be sent

refCnt Number of long integers to be sent (Range: 1-50)

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

Note:
Broadcast supported for serial protocols

int writeMultipleMod10000 (int *slaveAddr*, int *startRef*, const long *int32Arr*[], int *refCnt*) [inherited]

Modbus function 16 (10 hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

Writes long int values into pairs of output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value and performs number format conversion.

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 0x10000)

int32Arr Buffer with the data to be sent

refCnt Number of long integer values to be sent (Range: 1-50)

Returns:

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

Note:

Broadcast supported for serial protocols

int writeMultipleFloats (int *slaveAddr*, int *startRef*, const float *float32Arr*[ ], int *refCnt*) [inherited]

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

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

Remarks:

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

Parameters:

slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

startRef Start reference (Range: 1 - 0x10000)

float32Arr Buffer with the data to be sent

refCnt Number of float values to be sent (Range: 1-50)

Returns:

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

Note:

Broadcast supported for serial protocols

int readMultipleRegisters (int *slaveAddr*, int *startRef*, short *regArr*[ ], int *refCnt*) [inherited]

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

Reads the contents of the output registers (holding registers, 4:00000 table).

Parameters:

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

startRef Start reference (Range: 1 - 0x10000)

regArr Buffer which will be filled with the data read

refCnt Number of registers to be read (Range: 1-125)

Returns:

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

Note:

No broadcast supported

int readMultipleLongInts (int *slaveAddr*, int *startRef*, long *int32Arr*[], int *refCnt*) [inherited]

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

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) into 32-bit long int values.

Remarks:

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

Parameters:

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

startRef Start reference (Range: 1 - 0x10000)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

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

Note:

No broadcast supported

int readMultipleMod10000 (int *slaveAddr*, int *startRef*, long *int32Arr*[], int *refCnt*) [inherited]

Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

Reads the contents of pairs of consecutive output registers (holding registers, 4:00000 table) representing a modulo-10000 long int value into 32-bit int values and performs number format conversion.

Remarks:

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

Parameters:

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

startRef Start reference (Range: 1 - 0x10000)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

Returns:

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

Note:

No broadcast supported

int readMultipleFloats (int *slaveAddr*, int *startRef*, float *float32Arr*[ ], int *refCnt*) [inherited]

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

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

Remarks:

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

Parameters:

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

startRef Start reference (Range: 1 - 0x10000)

float32Arr Buffer which will be filled with the data read

refCnt Number of float values to be read (Range: 1-62)

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

Note:
No broadcast supported

int readInputDiscretes (int *slaveAddr*, int *startRef*, int *bitArr*[ ], int *refCnt*) [inherited]

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

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

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

startRef Start reference (Range: 1 - 0x10000)

bitArr Buffer which will contain the data read

refCnt Number of references to be read (Range: 1-2000)

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

Note:
No broadcast supported

int readInputRegisters (int *slaveAddr*, int *startRef*, short *regArr*[ ], int *refCnt*) [inherited]

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

Read the contents of the input registers (3:00000 table).

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

startRef Start reference (Range: 1 - 0x10000)

regArr Buffer which will be filled with the data read.

refCnt Number of references to be read (Range: 1-125)

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

Note:
No broadcast supported

int readInputLongInts (int *slaveAddr*, int *startRef*, long *int32Arr*[], int *refCnt*) [inherited]

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

Reads the contents of pairs of consecutive input registers (3:00000 table) into 32-bit long int values.

Remarks:

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

Parameters:

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

startRef Start reference (Range: 1 - 0x10000)

int32Arr Buffer which will be filled with the data read

refCnt Number of long integers to be read (Range: 1-62)

Returns:

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

Note:

No broadcast supported

int readInputMod10000 (int *slaveAddr*, int *startRef*, long *int32Arr*[], int *refCnt*) [inherited]

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

Reads the contents of pairs of consecutive input registers (3:00000 table) representing a modulo-10000 long int value into 32-bit long int values and performs number format conversion.

Remarks:

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

Parameters:

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

startRef Start reference (Range: 1 - 0x10000)

int32Arr Buffer which will be filled with the data read

refCnt Number of M10K integers to be read (Range: 1-62)

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

Note:
No broadcast supported

int readInputFloats (int *slaveAddr*, int *startRef*, float *float32Arr*[ ], int *refCnt*) [inherited]

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

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

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

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

startRef Start reference (Range: 1 - 0x10000)

float32Arr Buffer which will be filled with the data read

refCnt Number of floats to be read (Range: 1-62)

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

Note:
No broadcast supported

int writeCoil (int *slaveAddr*, int *bitAddr*, int *bitVal*) [inherited]

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

Sets a single discrete output variable (coil, 0:00000 table) to either ON or OFF.

Parameters:
slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

bitAddr Coil address (Range: 1 - 0x10000)

bitVal true sets, false clears discrete output variable

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

Note:
Broadcast supported for serial protocols

int writeSingleRegister (int *slaveAddr*, int *regAddr*, short *regVal*) [inherited]

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

Writes a value into a single output register (holding register, 4:00000 reference).

Parameters:
slaveAddr Modbus address of slave device or unit identifier (Range: 0 - 255)

regAddr Register address (Range: 1 - 0x10000)

regVal Data to be sent

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

Note:
Broadcast supported for serial protocols

int readExceptionStatus (int *slaveAddr*, unsigned char * *statusByte*) [inherited]

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

Reads the eight exception status coils within the slave device.

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

statusByte Slave status byte. The meaning of this status byte is slave specific and varies from device to device. identifier (Range: 1 - 255)

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

Note:
No broadcast supported

int maskWriteRegister (int *slaveAddr*, int *regAddr*, unsigned short *andMask*, unsigned short *orMask*) [inherited]

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

Masks bits according to an AND & an OR mask into a single output register (holding register, 4:00000 reference). Masking is done as follows: result = (currentVal AND andMask) OR (orMask AND andMask)

Parameters:

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

regAddr Register address (Range: 1 - 0x10000)

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

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

Note:

No broadcast supported

int readWriteRegisters (int *slaveAddr*, int *readRef*, short *readArr*[ ], int *readCnt*, int *writeRef*, const short *writeArr*[ ], int *writeCnt*) [inherited]

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

Combines reading and writing of the output registers in one transaction (holding registers, 4:00000 table).

Parameters:

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

readRef Start reference for reading (Range: 1 - 0x10000)

readArr Buffer which will contain the data read

readCnt Number of registers to be read (Range: 1-125)

writeRef Start reference for writing (Range: 1 - 0x10000)

writeArr Buffer with data to be sent

writeCnt Number of registers to be sent (Range: 1-100)

Returns:

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

Note:

No broadcast supported

int getTimeout () [inherited]

Returns the time-out value.

Remarks:

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

Returns:
Timeout value in ms

int setPollDelay (int *msTime*) [inherited]

Configures poll delay.

This function sets the delay time which applies between two consecutive Modbus read/write. A value of 0 disables the poll delay.

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

Note:
A protocol must be closed in order to configure it.

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

Return values:
FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getPollDelay () [inherited]

Returns the poll delay time.

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

int setRetryCnt (int *retries*) [inherited]

Configures the automatic retry setting.

A value of 0 disables any automatic retries.

Note:
A protocol must be closed in order to configure it.

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

Return values:
FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

int getRetryCnt () [inherited]

Returns the automatic retry count.

Returns:
Retry count

unsigned long getSuccessCounter () [inherited]

Returns how often a message transfer was successful.

Returns:
Counter value

void configureSwappedFloats () [inherited]

Configures float data type functions to do a word swap.

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

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

void configureLittleEndianInts () [inherited]

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

This is the default.

void configureIeeeFloats () [inherited]

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

This is the default.

2.2 Device and Vendor Specific Functions

2.2.1 Detailed Description

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

Advantec ADAM 5000/6000 Series Commands

- int MbusTcpMasterProtocol::adamSendReceiveAsciiCmd (const char *const commandSz, char *responseSz)
  Send/Receive ADAM 5000/6000 ASCII command.

2.2.2 Function Documentation

int adamSendReceiveAsciiCmd (const char *const *commandSz*, char * *responseSz*) [inherited]

Send/Receive ADAM 5000/6000 ASCII command.

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

Parameters:

commandSz Buffer which holds command string. Must not be longer than 255 characters.

responseSz Buffer which holds response string. Must be a buffer of 256 bytes. A possible trailing CR is removed.

Returns:

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

Note:

No broadcast supported

2.3 MODBUS/TCP Protocol

2.3.1 Detailed Description

The MODBUS/TCP master protocol is implemented in the class MbusTcpMasterProtocol.

It provides functions to establish and to close a TCP/IP connection to the slave as well as data and control functions which can be used after a connection to a slave device has been established successfully. The data and control functions are organized different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Protocol Flavours.

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

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

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

Compounds

- class MbusTcpMasterProtocol
  MODBUS/TCP Master Protocol class.

2.4 Serial Protocols

2.4.1 Detailed Description

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

These classes provide functions to open and to close serial port as well as data and control functions which can be used at any time after a protocol has been opened. The data and control functions are organized different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Protocol Flavours.

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

See sections The RTU Protocol 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 MbusRtuMasterProtocol class.

Compounds

- class MbusAsciiMasterProtocol
  Modbus ASCII Master Protocol class.

- class MbusRtuMasterProtocol
  Modbus RTU Master Protocol class.

2.5 Encapsulated Modbus RTU Protocol

2.5.1 Detailed Description

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

It provides functions to establish and to close a TCP/IP connection to the slave as well as data and control functions which can be used after a connection to a slave device has been established successfully. The data and control functions are organized different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Protocol Flavours.

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

Compounds

- class MbusRtuOverTcpMasterProtocol
  Encapsulated Modbus RTU Master Protocol class.

2.6 Protocol Errors and Exceptions

I/O error class

- #define FTALK_IO_ERROR_CLASS 0x40
  I/O error class.

- #define FTALK_IO_ERROR (FTALK_IO_ERROR_CLASS | 1)
  I/O error.

- #define FTALK_OPEN_ERR (FTALK_IO_ERROR_CLASS | 2)
  Port or socket open error.

- #define FTALK_PORT_ALREADY_OPEN (FTALK_IO_ERROR_CLASS | 3)

Serial port already open.

- #define FTALK_TCPIP_CONNECT_ERR (FTALK_IO_ERROR_CLASS | 4)
  TCP/IP connection error.
- #define FTALK_CONNECTION_WAS_CLOSED (FTALK_IO_ERROR_CLASS | 5)
  Remote peer closed TCP/IP connection.
- #define FTALK_SOCKET_LIB_ERROR (FTALK_IO_ERROR_CLASS | 6)
  Socket library error.
- #define FTALK_PORT_ALREADY_BOUND (FTALK_IO_ERROR_CLASS | 7)
  TCP port already bound.
- #define FTALK_LISTEN_FAILED (FTALK_IO_ERROR_CLASS | 8)
  Listen failed.
- #define FTALK_FILEDES_EXCEEDED (FTALK_IO_ERROR_CLASS | 9)
  File descriptors exceeded.
- #define FTALK_PORT_NO_ACCESS (FTALK_IO_ERROR_CLASS | 10)
  No permission to access serial port or TCP port.
- #define FTALK_PORT_NOT_AVAIL (FTALK_IO_ERROR_CLASS | 11)
  TCP port not available.

Fieldbus protocol error class

- #define FTALK_BUS_PROTOCOL_ERROR_CLASS 0x80
  Fieldbus protocol error class.
- #define FTALK_CHECKSUM_ERROR (FTALK_BUS_PROTOCOL_ERROR_CLASS | 1)
  Checksum error.
- #define FTALK_INVALID_FRAME_ERROR (FTALK_BUS_PROTOCOL_ERROR_CLASS | 2)
  Invalid frame error.
- #define FTALK_INVALID_REPLY_ERROR (FTALK_BUS_PROTOCOL_ERROR_CLASS | 3)
  Invalid reply error.
- #define FTALK_REPLY_TIMEOUT_ERROR (FTALK_BUS_PROTOCOL_ERROR_CLASS | 4)
  Reply time-out.
- #define FTALK_SEND_TIMEOUT_ERROR (FTALK_BUS_PROTOCOL_ERROR_CLASS | 5)
  Send time-out.
- #define FTALK_MBUS_EXCEPTION_RESPONSE (FTALK_BUS_PROTOCOL_ERROR_CLASS | 0x20)

Modbus® exception response.

- #define FTALK_MBUS_ILLEGAL_FUNCTION_RESPONSE (FTALK_MBUS_EXCEPTION_RESPONSE | 1)
  Illegal Function exception response.
- #define FTALK_MBUS_ILLEGAL_ADDRESS_RESPONSE (FTALK_MBUS_EXCEPTION_RESPONSE | 2)
  Illegal Data Address exception response.
- #define FTALK_MBUS_ILLEGAL_VALUE_RESPONSE (FTALK_MBUS_EXCEPTION_RESPONSE | 3)
  Illegal Data Value exception response.
- #define FTALK_MBUS_SLAVE_FAILURE_RESPONSE (FTALK_MBUS_EXCEPTION_RESPONSE | 4)
  Slave Device Failure exception response.

Defines

- #define FTALK_SUCCESS 0
  Operation was successful.
- #define FTALK_ILLEGAL_ARGUMENT_ERROR 1
  Illegal argument error.
- #define FTALK_ILLEGAL_STATE_ERROR 2
  Illegal state error.
- #define FTALK_EVALUATION_EXPIRED 3
  Evaluation expired.

Functions

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

2.6.1 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_IO_ERROR_CLASS 0x40

I/O error class.

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

#define FTALK_IO_ERROR (FTALK_IO_ERROR_CLASS | 1)

I/O error.

The underlaying I/O system reported an error.

#define FTALK_OPEN_ERR (FTALK_IO_ERROR_CLASS | 2)

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 (FTALK_IO_ERROR_CLASS | 3)

Serial port already open.

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

#define FTALK_TCPIP_CONNECT_ERR (FTALK_IO_ERROR_CLASS | 4)

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 (FTALK_IO_ERROR_CLASS | 5)

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 (FTALK_IO_ERROR_CLASS | 6)

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 (FTALK_IO_ERROR_CLASS | 7)

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 (FTALK_IO_ERROR_CLASS | 8)

Listen failed.

The listen operation on the specified TCP port failed..

#define FTALK_FILEDES_EXCEEDED (FTALK_IO_ERROR_CLASS | 9)

File descriptors exceeded.

Maximum number of usable file descriptors exceeded.

#define FTALK_PORT_NO_ACCESS (FTALK_IO_ERROR_CLASS | 10)

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 (FTALK_IO_ERROR_CLASS | 11)

TCP port not available.

The specified TCP port is not available on this machine.

#define FTALK_BUS_PROTOCOL_ERROR_CLASS 0x80

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 (FTALK_BUS_PROTOCOL_ERROR_CLASS | 1)

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 (FTALK_BUS_PROTOCOL_ERROR_CLASS | 2)

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 (FTALK_BUS_PROTOCOL_ERROR_CLASS | 3)

Invalid reply error.

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

#define FTALK_REPLY_TIMEOUT_ERROR (FTALK_BUS_PROTOCOL_ERROR_CLASS | 4)

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 (FTALK_BUS_PROTOCOL_ERROR_CLASS | 5)

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 (FTALK_BUS_PROTOCOL_ERROR_CLASS | 0x20)

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 (FTALK_MBUS_EXCEPTION_RESPONSE | 1)

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 (FTALK_MBUS_EXCEPTION_RESPONSE | 2)

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 (FTALK_MBUS_EXCEPTION_RESPONSE | 3)

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 (FTALK_MBUS_EXCEPTION_RESPONSE | 4)

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.

2.6.2 Function Documentation

char* getBusProtocolErrorText (int *errCode*)

Returns an error text string for a given error code.

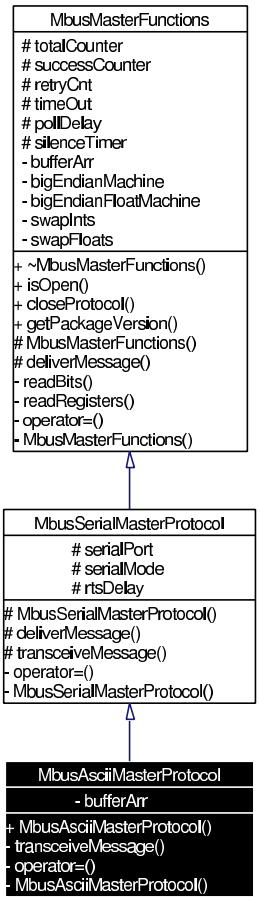
Parameters:
errCode FieldTalk error code

Returns:
Error text string

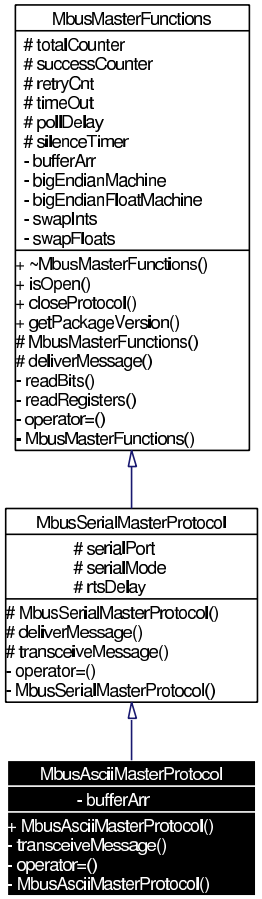
3 Modbus Protocol Library Documentation Class Documentation

3.1 MbusAsciiMasterProtocol Class Reference

Inheritance diagram for MbusAsciiMasterProtocol:



Collaboration diagram for MbusAsciiMasterProtocol:



3.1.1 Detailed Description

Modbus ASCII Master Protocol class.

This class realizes the Modbus ASCII master protocol. It provides functions to open and to close serial port as well as data and control functions which can be used at any time after the protocol has been opened. The data and control functions are organized different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Protocol Flavours.

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

Version:
1.1

See also:
mbusmaster
MbusSerialMasterProtocol, MbusMasterFunctions

Specialised Serial Port Management Functions

- virtual int openProtocol (const char ∗const portName, long baudRate, int dataBits, int stopBits, int parity)
  Opens a Modbus ASCII protocol and the associated serial port with specific port parameters.

- virtual int openProtocol (const char ∗const portName, long baudRate)
  Opens a Modbus ASCII protocol and the associated serial port with default port parameters.

Serial Port Management Functions

- virtual void closeProtocol ()
  Closes the serial port and releases any system resources associated with the port.

- virtual int isOpen ()
  Returns whether the protocol is open or not.

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

Class 0 Modbus Functions

- int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)
  Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.

- int writeMultipleLongInts (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.

- int writeMultipleMod10000 (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

- int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

- int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- int readMultipleLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

- int readMultipleMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding

Registers/Read Multiple Registers as modulo-10000 long int data.

- int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Class 1 Modbus Functions

- int readCoils (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 1 (01 hex), Read Coil Status/Read Coils.
- int readInputDiscretes (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.
- int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 4 (04 hex), Read Input Registers.
- int readInputLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.
- int readInputMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.
- int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.
- int writeCoil (int slaveAddr, int bitAddr, int bitVal)
  Modbus function 5 (05 hex), Force Single Coil/Write Coil.
- int writeSingleRegister (int slaveAddr, int regAddr, short regVal)
  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.
- int readExceptionStatus (int slaveAddr, unsigned char ∗statusByte)
  Modbus function 7 (07 hex), Read Exception Status.

Class 2 Modbus Functions

- int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)
  Modbus function 15 (0F hex), Force Multiple Coils.
- int maskWriteRegister (int slaveAddr, int regAddr, unsigned short andMask, unsigned short orMask)
  Modbus function 22 (16 hex), Mask Write Register.
- int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)

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

Protocol Configuration

- int setTimeout (int timeOut)
  Configures time-out.
- int getTimeout ()
  Returns the time-out value.
- int setPollDelay (int pollDelay)
  Configures poll delay.
- int getPollDelay ()
  Returns the poll delay time.
- int setRetryCnt (int retryCnt)
  Configures the automatic retry setting.
- int getRetryCnt ()
  Returns the automatic retry count.

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.

Word Order Configuration

- void configureBigEndianInts ()
  Configures int data type functions to do a word swap.
- void configureSwappedFloats ()
  Configures float data type functions to do a word swap.
- void configureLittleEndianInts ()
  Configures int data type functions not to do a word swap.

- void configureIeeeFloats ()
  Configures float data type functions not to do a word swap.

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

- MbusAsciiMasterProtocol ()
  Constructs a MbusAsciiMasterProtocol object and initialises its data.

Static Public Member Functions

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

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 Member Function Documentation

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

Opens a Modbus ASCII protocol and the associated serial port with specific port parameters.

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

Note:

The default time-out for the data transfer is 1000 ms.
The default poll delay is 0 ms.
Automatic retries are switched off (retry count is 0).

Parameters:

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

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

dataBits SER_DATABITS_7: 7 data bits, SER_DATABITS_8: data bits

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

parity SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_-EVEN: even parity

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

Reimplemented from MbusSerialMasterProtocol.

int openProtocol (const char *const *portName*, long *baudRate*) [virtual]

Opens a Modbus ASCII protocol and the associated serial port with default port parameters.

This function opens the serial port with 8 databits, 1 stopbit and no parity. After a port has been opened, data and control functions can be used.

Note:
The default time-out for the data transfer is 1000 ms.
The default poll delay is 0 ms.
Automatic retries are switched off (retry count is 0).

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.

Reimplemented from MbusSerialMasterProtocol.

int isOpen () [virtual, inherited]

Returns whether the protocol is open or not.

Return values:
true = open

false = closed

Reimplemented from MbusMasterFunctions.

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:

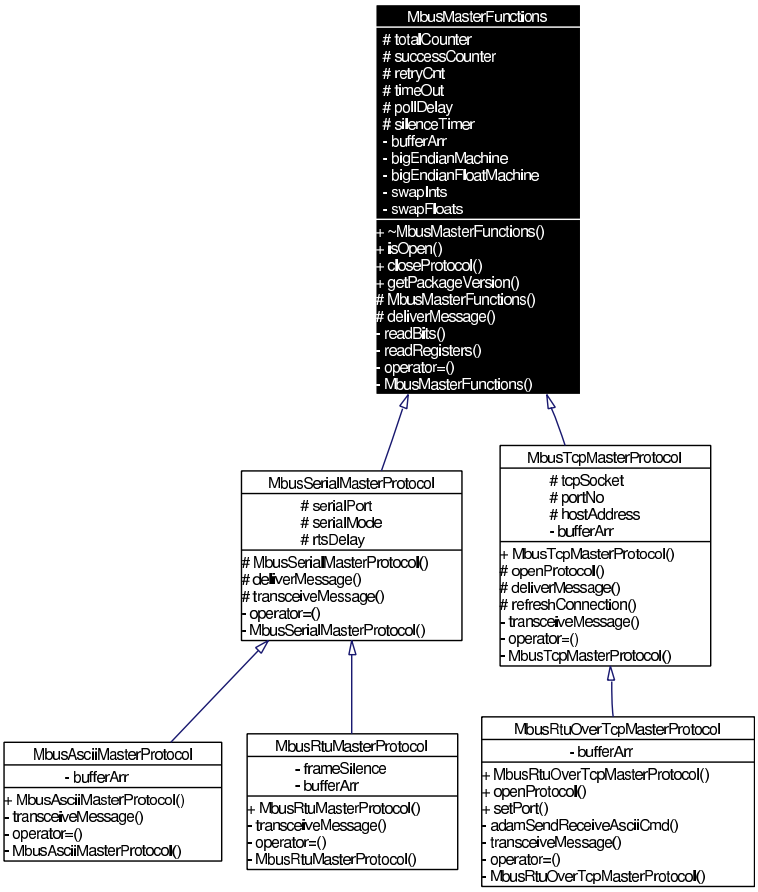
FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

FTALK_ILLEGAL_STATE_ERROR Protocol is already open

3.2 MbusMasterFunctions Class Reference

Inheritance diagram for MbusMasterFunctions:



3.2.1 Detailed Description

Base class which implements Modbus data and control functions.

The functions provided by this base class apply to all protocol flavours via inheritance. For a more detailed description see section Data and Control Functions for all Protocol Flavours.

Version:
1.1

See also:
mbusmaster
MbusSerialMasterProtocol, MbusRtuMasterProtocol
MbusAsciiMasterProtocol, MbusTcpMasterProtocol

Class 0 Modbus Functions

- int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)
  Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.

- int writeMultipleLongInts (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)

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

- int writeMultipleMod10000 (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

- int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

- int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- int readMultipleLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

- int readMultipleMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

- int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Class 1 Modbus Functions

- int readCoils (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 1 (01 hex), Read Coil Status/Read Coils.

- int readInputDiscretes (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

- int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 4 (04 hex), Read Input Registers.

- int readInputLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.

- int readInputMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

- int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

- int writeCoil (int slaveAddr, int bitAddr, int bitVal)
  Modbus function 5 (05 hex), Force Single Coil/Write Coil.

- int writeSingleRegister (int slaveAddr, int regAddr, short regVal)
  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.

- int readExceptionStatus (int slaveAddr, unsigned char *statusByte)
  Modbus function 7 (07 hex), Read Exception Status.

Class 2 Modbus Functions

- int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)
  Modbus function 15 (0F hex), Force Multiple Coils.

- int maskWriteRegister (int slaveAddr, int regAddr, unsigned short andMask, unsigned short orMask)
  Modbus function 22 (16 hex), Mask Write Register.

- int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)
  Modbus function 23 (17 hex), Read/Write Registers.

Protocol Configuration

- int setTimeout (int timeOut)
  Configures time-out.

- int getTimeout ()
  Returns the time-out value.

- int setPollDelay (int pollDelay)
  Configures poll delay.

- int getPollDelay ()
  Returns the poll delay time.

- int setRetryCnt (int retryCnt)
  Configures the automatic retry setting.

- int getRetryCnt ()
  Returns the automatic retry count.

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.

Word Order Configuration

- void configureBigEndianInts ()
  Configures int data type functions to do a word swap.

- void configureSwappedFloats ()
  Configures float data type functions to do a word swap.

- void configureLittleEndianInts ()
  Configures int data type functions not to do a word swap.

- void configureIeeeFloats ()
  Configures float data type functions not to do a word swap.

Public Member Functions

- virtual ~MbusMasterFunctions ()
  Destructor.

- virtual int isOpen ()
  Returns whether the protocol is open or not.

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

Static Public Member Functions

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

Protected Member Functions

- MbusMasterFunctions ()
  Constructs a MbusMasterFunctions object and initialises its data.

3.2.2 Constructor & Destructor Documentation

MbusMasterFunctions () [protected]

Constructs a MbusMasterFunctions object and initialises its data.

It also detects the endianess of the machine it's running on and configures byte swapping if necessary.

~MbusMasterFunctions () [virtual]

Destructor.

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

3.2.3 Member Function Documentation

int isOpen () [virtual]

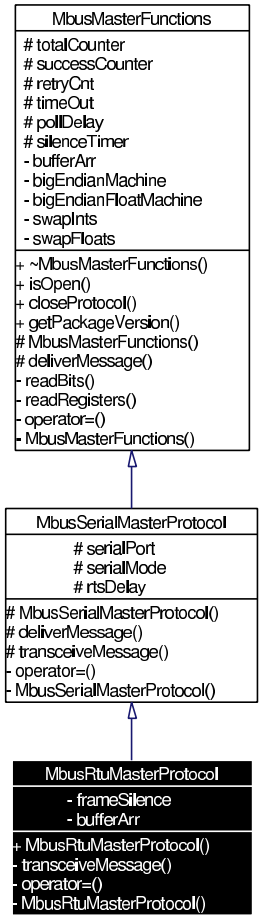
Returns whether the protocol is open or not.

Return values:
true = open
false = closed

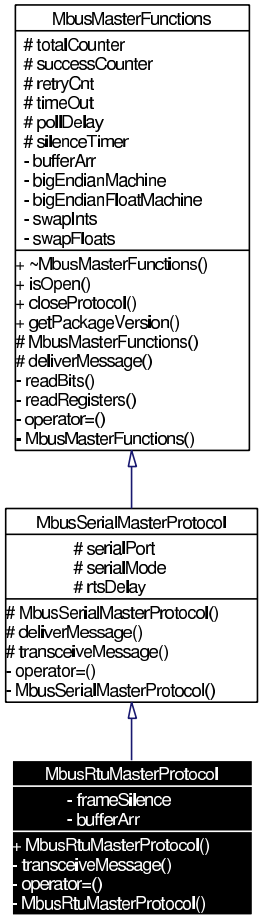
Reimplemented in MbusTcpMasterProtocol, and MbusSerialMasterProtocol.

3.3 MbusRtuMasterProtocol Class Reference

Inheritance diagram for MbusRtuMasterProtocol:



Collaboration diagram for MbusRtuMasterProtocol:



3.3.1 Detailed Description

Modbus RTU Master Protocol class.

This class realizes the Modbus RTU master protocol. It provides functions to open and to close serial port as well as data and control functions which can be used at any time after the protocol has been opened. The data and control functions are organized different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Protocol Flavours.

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

Version:
1.1

See also:
mbusmaster
MbusSerialMasterProtocol, MbusMasterFunctions

Specialised Serial Port Management Functions

- virtual int openProtocol (const char *const portName, long baudRate, int dataBits, int stopBits, int parity)
  Opens a Modbus RTU protocol and the associated serial port with specific port parameters.

- virtual int openProtocol (const char *const portName, long baudRate)
  Opens a Modbus RTU protocol and the associated serial port with default port parameters.

Serial Port Management Functions

- virtual void closeProtocol ()
  Closes the serial port and releases any system resources associated with the port.

- virtual int isOpen ()
  Returns whether the protocol is open or not.

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

Class 0 Modbus Functions

- int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)
  Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.

- int writeMultipleLongInts (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.

- int writeMultipleMod10000 (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

- int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

- int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- int readMultipleLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

- int readMultipleMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding

Registers/Read Multiple Registers as modulo-10000 long int data.

- int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Class 1 Modbus Functions

- int readCoils (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 1 (01 hex), Read Coil Status/Read Coils.
- int readInputDiscretes (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.
- int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 4 (04 hex), Read Input Registers.
- int readInputLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.
- int readInputMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.
- int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.
- int writeCoil (int slaveAddr, int bitAddr, int bitVal)
  Modbus function 5 (05 hex), Force Single Coil/Write Coil.
- int writeSingleRegister (int slaveAddr, int regAddr, short regVal)
  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.
- int readExceptionStatus (int slaveAddr, unsigned char ∗statusByte)
  Modbus function 7 (07 hex), Read Exception Status.

Class 2 Modbus Functions

- int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)
  Modbus function 15 (0F hex), Force Multiple Coils.
- int maskWriteRegister (int slaveAddr, int regAddr, unsigned short andMask, unsigned short orMask)
  Modbus function 22 (16 hex), Mask Write Register.
- int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)

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

Protocol Configuration

- int setTimeout (int timeOut)
  Configures time-out.
- int getTimeout ()
  Returns the time-out value.
- int setPollDelay (int pollDelay)
  Configures poll delay.
- int getPollDelay ()
  Returns the poll delay time.
- int setRetryCnt (int retryCnt)
  Configures the automatic retry setting.
- int getRetryCnt ()
  Returns the automatic retry count.

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.

Word Order Configuration

- void configureBigEndianInts ()
  Configures int data type functions to do a word swap.
- void configureSwappedFloats ()
  Configures float data type functions to do a word swap.
- void configureLittleEndianInts ()
  Configures int data type functions not to do a word swap.

- void configureIeeeFloats ()
  Configures float data type functions not to do a word swap.

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

- MbusRtuMasterProtocol ()
  Constructs a MbusRtuMasterProtocol object and initialises its data.

Static Public Member Functions

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

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 Member Function Documentation

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

Opens a Modbus RTU protocol and the associated serial port with specific port parameters.

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

Note:

The default time-out for the data transfer is 1000 ms.
The default poll delay is 0 ms.
Automatic retries are switched off (retry count is 0).

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 MbusSerialMasterProtocol.

int openProtocol (const char *const *portName*, long *baudRate*) [virtual]

Opens a Modbus RTU protocol and the associated serial port with default port parameters.

This function opens the serial port with 8 databits, 1 stopbit and no parity. After a port has been opened, data and control functions can be used.

Note:
The default time-out for the data transfer is 1000 ms.
The default poll delay is 0 ms.
Automatic retries are switched off (retry count is 0).

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.

Reimplemented from MbusSerialMasterProtocol.

int isOpen () [virtual, inherited]

Returns whether the protocol is open or not.

Return values:
true = open

false = closed

Reimplemented from MbusMasterFunctions.

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:

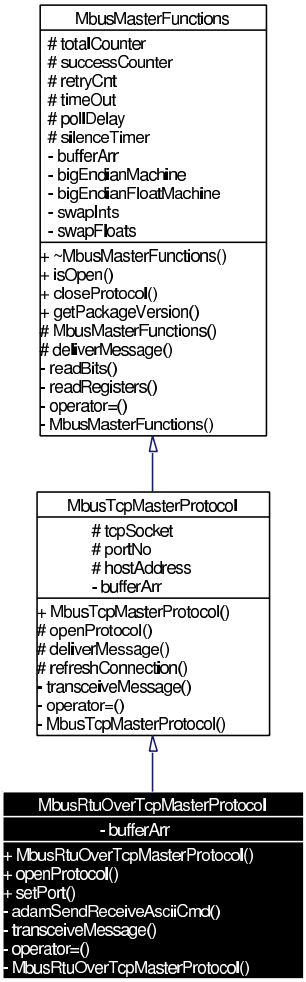
FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

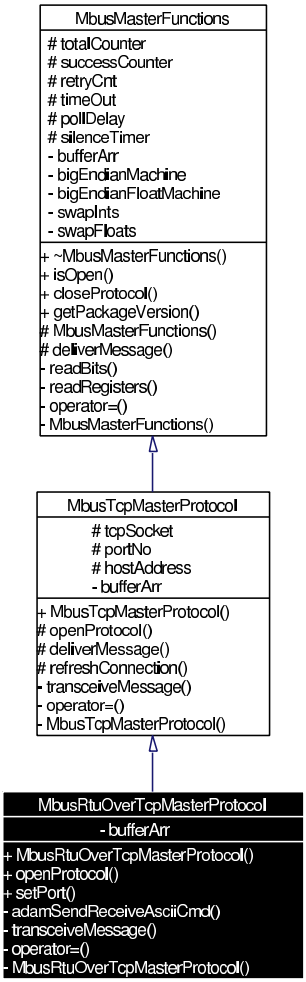
FTALK_ILLEGAL_STATE_ERROR Protocol is already open

3.4 MbusRtuOverTcpMasterProtocol Class Reference

Inheritance diagram for MbusRtuOverTcpMasterProtocol:



Collaboration diagram for MbusRtuOverTcpMasterProtocol:



3.4.1 Detailed Description

Encapsulated Modbus RTU Master Protocol class.

This class realises the Encapsulated Modbus RTU master protocol. This protocol is also known as RTU over TCP or RTU/IP and used for example by ISaGraf174 Soft-PLCs. This class provides functions to establish and to close a TCP/IP connection to the slave as well as data and control functions which can be used after a connection to a slave device has been established successfully. The data and control functions are organized different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Protocol Flavours.

It is also possible to instantiate multiple instances of this class for establishing multiple connections to either the same or different hosts.

Version:
1.0

See also:
mbusmaster

MbusMasterFunctions

TCP/IP Connection Management Functions

- virtual void closeProtocol ()
  Closes a TCP/IP connection to a slave and releases any system resources associated with the connection.

- virtual int isOpen ()
  Returns whether currently connected or not.

- unsigned short getPort ()
  Returns the TCP port number used by the protocol.

Class 0 Modbus Functions

- int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)
  Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.

- int writeMultipleLongInts (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.

- int writeMultipleMod10000 (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

- int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

- int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- int readMultipleLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

- int readMultipleMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

- int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Class 1 Modbus Functions

- int readCoils (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 1 (01 hex), Read Coil Status/Read Coils.

- int readInputDiscretes (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

- int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 4 (04 hex), Read Input Registers.

- int readInputLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.

- int readInputMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

- int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

- int writeCoil (int slaveAddr, int bitAddr, int bitVal)
  Modbus function 5 (05 hex), Force Single Coil/Write Coil.

- int writeSingleRegister (int slaveAddr, int regAddr, short regVal)
  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.

- int readExceptionStatus (int slaveAddr, unsigned char *statusByte)
  Modbus function 7 (07 hex), Read Exception Status.

Class 2 Modbus Functions

- int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)
  Modbus function 15 (0F hex), Force Multiple Coils.

- int maskWriteRegister (int slaveAddr, int regAddr, unsigned short andMask, unsigned short orMask)
  Modbus function 22 (16 hex), Mask Write Register.

- int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)
  Modbus function 23 (17 hex), Read/Write Registers.

Protocol Configuration

- int setTimeout (int timeOut)
  Configures time-out.

- int getTimeout ()
  Returns the time-out value.

- int setPollDelay (int pollDelay)
  Configures poll delay.

- int getPollDelay ()
  Returns the poll delay time.

- int setRetryCnt (int retryCnt)
  Configures the automatic retry setting.

- int getRetryCnt ()
  Returns the automatic retry count.

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.

Word Order Configuration

- void configureBigEndianInts ()
  Configures int data type functions to do a word swap.

- void configureSwappedFloats ()
  Configures float data type functions to do a word swap.

- void configureLittleEndianInts ()
  Configures int data type functions not to do a word swap.

- void configureIeeeFloats ()
  Configures float data type functions not to do a word swap.

Public Member Functions

- MbusRtuOverTcpMasterProtocol ()
  Constructs a MbusRtuOverTcpMasterProtocol object and initialises its data.

- int openProtocol (const char ∗const hostName)
  Connects to a Encapsulated Modbus RTU slave.

- int setPort (unsigned short portNo)
  Sets the TCP port number to be used by the protocol.

Static Public Member Functions

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

3.4.2 Member Function Documentation

int openProtocol (const char ∗const *hostName*)

Connects to a Encapsulated Modbus RTU slave.

This function establishes a logical network connection between master and slave. After a connection has been established data and control functions can be used. A TCP/IP connection should be closed if it is no longer needed.

Note:
The default time-out for the connection is 1000 ms.
The default TCP port number is 1100.

Parameters:
hostName String with IP address or host name

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

Reimplemented from MbusTcpMasterProtocol.

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 1100. In this case no call to this function is necessary. However if the port number has to be different from 1100 this function must be called *before* opening the connection with openProtocol().

Parameters:
portNo Port number to be used when opening the connection

Return values:
FTALK_SUCCESS Success
FTALK_ILLEGAL_STATE_ERROR Protocol already open

Reimplemented from MbusTcpMasterProtocol.

int isOpen () [virtual, inherited]

Returns whether currently connected or not.

Return values:
true = connected
false = not connected

Reimplemented from MbusMasterFunctions.

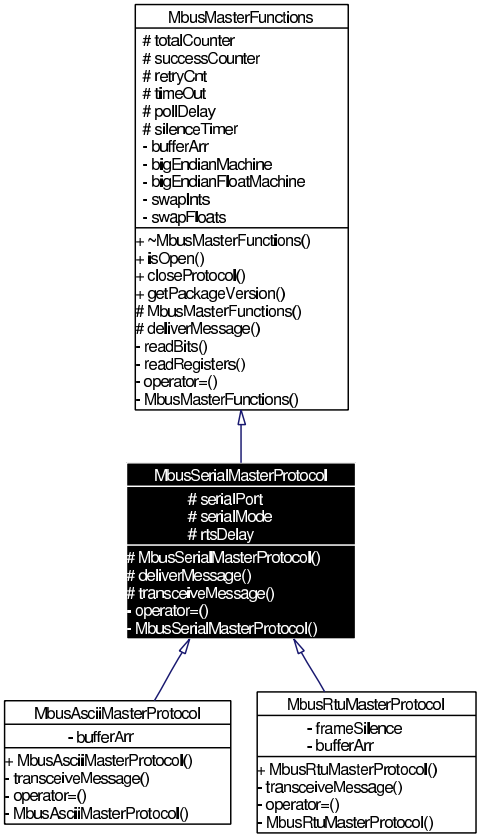
unsigned short getPort () [inherited]

Returns the TCP port number used by the protocol.

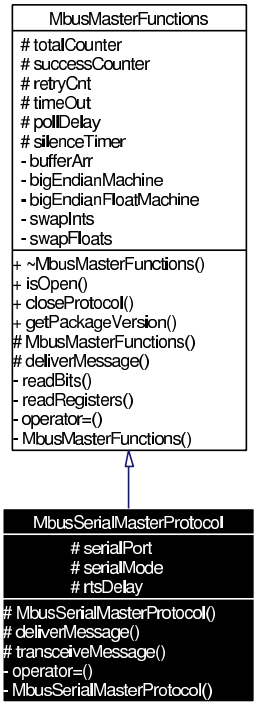
Returns:
Port number used by the protocol

3.5 MbusSerialMasterProtocol Class Reference

Inheritance diagram for MbusSerialMasterProtocol:



Collaboration diagram for MbusSerialMasterProtocol:



3.5.1 Detailed Description

Base class for serial serial master protocols.

This base class realises the Modbus serial master protocols. It provides functions to open and to close serial port as well as data and control functions which can be used at any time after the protocol has been opened. The data and control functions are organized different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Protocol Flavours.

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

Version:
1.1

See also:
mbusmaster
MbusMasterFunctions

Serial Port Management Functions

- virtual int openProtocol (const char ∗const portName, long baudRate, int dataBits, int stopBits, int parity)
  Opens a serial Modbus protocol and the associated serial port with specific port parameters.

- virtual int openProtocol (const char ∗const portName, long baudRate)

Opens a serial Modbus protocol and the associated serial port with default port parameters.

- virtual void closeProtocol ()
  Closes the serial port and releases any system resources associated with the port.

- virtual int isOpen ()
  Returns whether the protocol is open or not.

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

Class 0 Modbus Functions

- int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)
  Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.

- int writeMultipleLongInts (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.

- int writeMultipleMod10000 (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

- int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

- int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- int readMultipleLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

- int readMultipleMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

- int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read Multiple Registers as float data.

Class 1 Modbus Functions

- int readCoils (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 1 (01 hex), Read Coil Status/Read Coils.

- int readInputDiscretes (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.

- int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 4 (04 hex), Read Input Registers.

- int readInputLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.

- int readInputMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.

- int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.

- int writeCoil (int slaveAddr, int bitAddr, int bitVal)
  Modbus function 5 (05 hex), Force Single Coil/Write Coil.

- int writeSingleRegister (int slaveAddr, int regAddr, short regVal)
  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.

- int readExceptionStatus (int slaveAddr, unsigned char *statusByte)
  Modbus function 7 (07 hex), Read Exception Status.

Class 2 Modbus Functions

- int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)
  Modbus function 15 (0F hex), Force Multiple Coils.

- int maskWriteRegister (int slaveAddr, int regAddr, unsigned short andMask, unsigned short orMask)
  Modbus function 22 (16 hex), Mask Write Register.

- int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)
  Modbus function 23 (17 hex), Read/Write Registers.

Protocol Configuration

- int setTimeout (int timeOut)
  Configures time-out.

- int getTimeout ()
  Returns the time-out value.

- int setPollDelay (int pollDelay)
  Configures poll delay.
- int getPollDelay ()
  Returns the poll delay time.
- int setRetryCnt (int retryCnt)
  Configures the automatic retry setting.
- int getRetryCnt ()
  Returns the automatic retry count.

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.

Word Order Configuration

- void configureBigEndianInts ()
  Configures int data type functions to do a word swap.
- void configureSwappedFloats ()
  Configures float data type functions to do a word swap.
- void configureLittleEndianInts ()
  Configures int data type functions not to do a word swap.
- void configureIeeeFloats ()
  Configures float data type functions not to do a word swap.

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 }

Static Public Member Functions

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

Protected Types

- enum { SER_RS232, SER_RS485 }

Protected Member Functions

- MbusSerialMasterProtocol ()
  Constructs a MbusSerialMasterProtocol object and initialises its data.

3.5.2 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.

3.5.3 Member Function Documentation

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

Opens a serial Modbus protocol and the associated serial port with specific port parameters.

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

Note:

The default time-out for the data transfer is 1000 ms.
The default poll delay is 0 ms.
Automatic retries are switched off (retry count is 0).

Parameters:

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

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

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

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

parity SER_PARITY_NONE: no parity, SER_PARITY_ODD: odd parity, SER_PARITY_-EVEN: even parity

Returns:

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

Reimplemented in MbusAsciiMasterProtocol, and MbusRtuMasterProtocol.

int openProtocol (const char *const *portName*, long *baudRate*) [virtual]

Opens a serial Modbus protocol and the associated serial port with default port parameters.

This function opens the serial port with 8 databits, 1 stopbit and no parity. After a port has been opened, data and control functions can be used.

Note:
The default time-out for the data transfer is 1000 ms.
The default poll delay is 0 ms.
Automatic retries are switched off (retry count is 0).

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.

Reimplemented in MbusAsciiMasterProtocol, and MbusRtuMasterProtocol.

int isOpen () [virtual]

Returns whether the protocol is open or not.

Return values:
true = open

false = closed

Reimplemented from MbusMasterFunctions.

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:

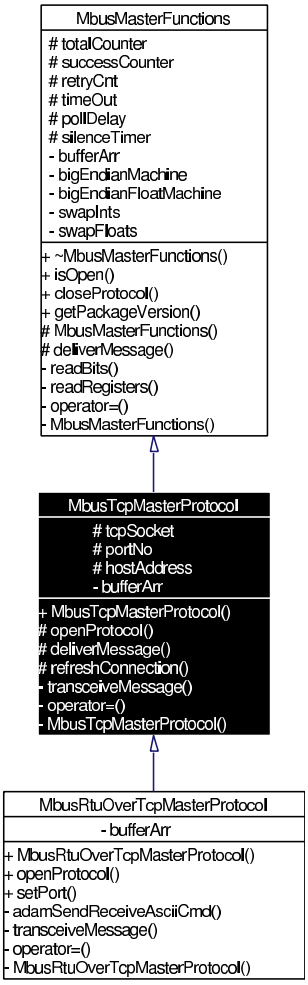
FTALK_SUCCESS Success

FTALK_ILLEGAL_ARGUMENT_ERROR Argument out of range

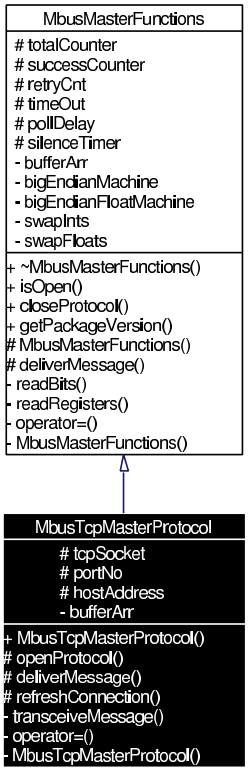
FTALK_ILLEGAL_STATE_ERROR Protocol is already open

3.6 MbusTcpMasterProtocol Class Reference

Inheritance diagram for MbusTcpMasterProtocol:



Collaboration diagram for MbusTcpMasterProtocol:



3.6.1 Detailed Description

MODBUS/TCP Master Protocol class.

This class realises the MODBUS/TCP master protocol. It provides functions to establish and to close a TCP/IP connection to the slave as well as data and control functions which can be used after a connection to a slave device has been established successfully. The data and control functions are organized different conformance classes. For a more detailed description of the data and control functions see section Data and Control Functions for all Protocol Flavours.

It is also possible to instantiate multiple instances of this class for establishing multiple connections to either the same or different hosts.

Version:
1.1

See also:
mbusmaster
MbusMasterFunctions

TCP/IP Connection Management Functions

- int openProtocol (const char *const hostName)
  Connects to a MODBUS/TCP slave.

- virtual void closeProtocol ()
  Closes a TCP/IP connection to a slave and releases any system resources associated with the connection.

- virtual int isOpen ()
  Returns whether currently connected or not.

- int setPort (unsigned short portNo)
  Sets the TCP port number to be used by the protocol.

- unsigned short getPort ()
  Returns the TCP port number used by the protocol.

Advantec ADAM 5000/6000 Series Commands

- int adamSendReceiveAsciiCmd (const char *const commandSz, char *responseSz)
  Send/Receive ADAM 5000/6000 ASCII command.

Class 0 Modbus Functions

- int writeMultipleRegisters (int slaveAddr, int startRef, const short regArr[ ], int refCnt)
  Modbus function 16 (10 hex), Preset Multiple Registers/Write Multiple Registers.

- int writeMultipleLongInts (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit long int data types, Preset Multiple Registers/Write Multiple Registers with long int data.

- int writeMultipleMod10000 (int slaveAddr, int startRef, const long int32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit modulo-10000 long int data types, Preset Multiple Registers/Write Multiple Registers with modulo-10000 long int data.

- int writeMultipleFloats (int slaveAddr, int startRef, const float float32Arr[ ], int refCnt)
  Modbus function 16 (10 hex) for 32-bit float data types, Preset Multiple Registers/Write Multiple Registers with float data.

- int readMultipleRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 3 (03 hex), Read Holding Registers/Read Multiple Registers.

- int readMultipleLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit long int data types, Read Holding Registers/Read Multiple Registers as long int data.

- int readMultipleMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit modulo-10000 long int data types, Read Holding Registers/Read Multiple Registers as modulo-10000 long int data.

- int readMultipleFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 3 (03 hex) for 32-bit float data types, Read Holding Registers/Read

Multiple Registers as float data.

Class 1 Modbus Functions

- int readCoils (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 1 (01 hex), Read Coil Status/Read Coils.
- int readInputDiscretes (int slaveAddr, int startRef, int bitArr[ ], int refCnt)
  Modbus function 2 (02 hex), Read Inputs Status/Read Input Discretes.
- int readInputRegisters (int slaveAddr, int startRef, short regArr[ ], int refCnt)
  Modbus function 4 (04 hex), Read Input Registers.
- int readInputLongInts (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit long int data types, Read Input Registers as long int data.
- int readInputMod10000 (int slaveAddr, int startRef, long int32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit modulo-10000 long int data types, Read Input Registers as modulo-10000 long int data.
- int readInputFloats (int slaveAddr, int startRef, float float32Arr[ ], int refCnt)
  Modbus function 4 (04 hex) for 32-bit float data types, Read Input Registers as float data.
- int writeCoil (int slaveAddr, int bitAddr, int bitVal)
  Modbus function 5 (05 hex), Force Single Coil/Write Coil.
- int writeSingleRegister (int slaveAddr, int regAddr, short regVal)
  Modbus function 6 (06 hex), Preset Single Register/Write Single Register.
- int readExceptionStatus (int slaveAddr, unsigned char *statusByte)
  Modbus function 7 (07 hex), Read Exception Status.

Class 2 Modbus Functions

- int forceMultipleCoils (int slaveAddr, int startRef, const int bitArr[ ], int refCnt)
  Modbus function 15 (0F hex), Force Multiple Coils.
- int maskWriteRegister (int slaveAddr, int regAddr, unsigned short andMask, unsigned short orMask)
  Modbus function 22 (16 hex), Mask Write Register.
- int readWriteRegisters (int slaveAddr, int readRef, short readArr[ ], int readCnt, int writeRef, const short writeArr[ ], int writeCnt)
  Modbus function 23 (17 hex), Read/Write Registers.

Protocol Configuration

- int setTimeout (int timeOut)
  Configures time-out.
- int getTimeout ()
  Returns the time-out value.
- int setPollDelay (int pollDelay)
  Configures poll delay.
- int getPollDelay ()
  Returns the poll delay time.
- int setRetryCnt (int retryCnt)
  Configures the automatic retry setting.
- int getRetryCnt ()
  Returns the automatic retry count.

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.

Word Order Configuration

- void configureBigEndianInts ()
  Configures int data type functions to do a word swap.
- void configureSwappedFloats ()
  Configures float data type functions to do a word swap.
- void configureLittleEndianInts ()
  Configures int data type functions not to do a word swap.
- void configureIeeeFloats ()
  Configures float data type functions not to do a word swap.

Public Member Functions

- MbusTcpMasterProtocol ()
  Constructs a MbusTcpMasterProtocol object and initialises its data.

Static Public Member Functions

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

3.6.2 Member Function Documentation

int openProtocol (const char *const *hostName*)

Connects to a MODBUS/TCP slave.

This function establishes a logical network connection between master and slave. After a connection has been established data and control functions can be used. A TCP/IP connection should be closed if it is no longer needed.

Note:
The default time-out for the connection is 1000 ms.
The default TCP port number is 502.

Parameters:
hostName String with IP address or host name

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

Reimplemented in MbusRtuOverTcpMasterProtocol.

int isOpen () [virtual]

Returns whether currently connected or not.

Return values:
true = connected

false = not connected

Reimplemented from MbusMasterFunctions.

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* opening the connection with openProtocol().

Parameters:
portNo Port number to be used when opening the connection

Return values:
FTALK_SUCCESS Success

FTALK_ILLEGAL_STATE_ERROR Protocol already open

Reimplemented in MbusRtuOverTcpMasterProtocol.

unsigned short getPort ()

Returns the TCP port number used by the protocol.

Returns:
Port number used by the protocol

4 Modbus Protocol Library Documentation Page Documentation

4.1 How to integrate the Protocol in your Application

4.1.1 Using Serial Protocols

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

The registers for reading are in the reference range 4:00100 to 4:00119 and the registers for writing are in the range 4:00200 to 4:00219. The 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 "MbusRtuMasterProtocol.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];
int readBitSet[20];
int writeBitSet[20];
```

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

```
float readFloatSet[10];
float writeFloatSet[10];
```

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

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

```
long readLongSet[10];
long writeLongSet[10];
```

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

3. Declare and instantiate a protocol object

```
MbusRtuMasterProtocol mbusProtocol;
```

4. Open the protocol

```
   int result;

   result = mbusProtocol.openProtocol(portName,
                                      9600L, // Baudrate
                                      8,     // Databits
                                      1,     // Stopbits
                                      0);    // Parity
   if (result != FTALK_SUCCESS)
   {
      fprintf(stderr, "Error opening protocol: %s!\n",
                       getBusProtocolErrorText(result));
      exit(EXIT_FAILURE);
   }
```

5. Perform the data transfer functions

- To read register values:
  `mbusProtocol.readMultipleRegisters(1, 100, readRegSet, sizeof(readRegSet) /`

- To write a single register value:
  `mbusProtocol.writeSingleRegister(1, 200, 1234);`
- To write mutliple register values:
  `mbusProtocol.writeMultipleRegisters(1, 200, writeRegSet, sizeof(writeRegSet)`
- To read discrete values:
  `mbusProtocol.readCoils(1, 10, readBitSet, sizeof(readBitSet) / sizeof(int));`
- To write a single discrete value:
  `mbusProtocol.writeCoil(1, 20, 1);`
- To write multiple discrete values:
  `mbusProtocol.forceMultipleCoils(1, 20, sizeof(writeBitSet) / sizeof(int));`
- To read float values:
  `mbusProtocol.readMultipleFloats(1, 100, readFloatSet, sizeof(readFloatSet) /`
- To read long integer values:
  `mbusProtocol.readMultipleLongInts(1, 100, readLongSet, sizeof(readLongSet) /`

6. Close the protocol port if not needed any more

```
mbusProtocol.closeProtocol();
```

7. Error Handling

Serial protocol errors like slave device failures, transmission failures, checksum errors and time-outs return an error code. The following code snippet can handle and report these errors:

```
int result;

result = mbusProtocol.readMultipleRegisters(1, 100, dataSetArray, 10);
if (result != FTALK_SUCCESS)
{
   fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));
   // Stop for fatal errors
   if (!(result & FTALK_BUS_PROTOCOL_ERROR_CLASS))
      return;
}
```

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

4.1.2 Using MODBUS/TCP Protocol

Let's assume we want to talk to a Modbus slave device with unit address 1 and IP address 10.0.0.11.

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

1. Include the package header files

```
#include "MbusTcpMasterProtocol.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];
int readBitSet[10];
int writeBitSet[10];
```

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

```
float readFloatSet[10];
float writeFloatSet[10];
```

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

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

```
long readLongSet[10];
long writeLongSet[10];
```

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

3. Declare and instantiate a protocol object

```
MbusTcpMasterProtocol mbusProtocol;
```

4. Open the protocol

```
mbusProtocol.openProtocol("10.0.0.11);
```

5. Perform the data transfer functions

- To read register values:
  `mbusProtocol.readMultipleRegisters(1, 100, readRegSet, sizeof(readRegSet) /`
- To write a single register value:
  `mbusProtocol.writeSingleRegister(1, 200, 1234);`
- To write mutliple register values:
  `mbusProtocol.writeMultipleRegisters(1, 200, writeRegSet, sizeof(writeRegSet)`
- To read discrete values:
  `mbusProtocol.readCoils(1, 10, readBitSet, sizeof(readBitSet) / sizeof(int));`
- To write a single discrete value:
  `mbusProtocol.writeCoil(1, 20, 1);`

- To write multiple discrete values:
  `mbusProtocol.forceMultipleCoils(1, 20, writeBitSet, sizeof(writeBitSet) / si`
- To read float values:
  `mbusProtocol.readMultipleFloats(1, 100, readFloatSet, sizeof(readFloatSet) /`
- To read long integer values:
  `mbusProtocol.readMultipleLongInts(1, 100, readLongSet, sizeof(readLongSet) /`

6. Close the connection if not needed any more

```
mbusProtocol.closeProtocol();
```

7. Error Handling

TCP/IP protocol errors like slave failures, TCP/IP connection failures and time-outs return an error code. The following code snippet can handle these errors:

```
   int result;

   result = mbusProtocol.readMultipleRegisters(1, 100, dataSetArray, 10);
   if (result != FTALK_SUCCESS)
   {
      fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));
      // Stop for fatal errors
      if (!(result & FTALK_BUS_PROTOCOL_ERROR_CLASS))
         return;
   }
}
```

If the method returns FTALK_CONNECTION_WAS_CLOSED, it signals that the the TCP/IP connection was lost or closed by the remote end. Before using further data and control functions the connection has to be re-opened succesfully.

4.1.3 Examples

- Serial Example
- MODBUS/TCP Example
- Modpoll application

4.2 Examples

- Serial Example
- MODBUS/TCP Example
- Modpoll application

4.2.1 Serial Example

The following example sersimple.cpp shows how to configure a serial Modbus protocol and read values:

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

// Include FieldTalk package header
#include "MbusAsciiMasterProtocol.hpp"
#include "MbusRtuMasterProtocol.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

//MbusAsciiMasterProtocol mbusProtocol; // Use this declaration for ASCII
MbusRtuMasterProtocol mbusProtocol; // Use this declaration for RTU


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

void openProtocol()
{
   int result;

   result = mbusProtocol.openProtocol(portName,
                                      9600L, // Baudrate
                                      8,     // Databits
                                      1,     // Stopbits
                                      0);    // Parity
   if (result != FTALK_SUCCESS)
   {
```

```
      fprintf(stderr, "Error opening protocol: %s!\n",
                       getBusProtocolErrorText(result));
      exit(EXIT_FAILURE);
   }
}


void closeProtocol()
{
   mbusProtocol.closeProtocol();
}


void runPollLoop()
{
   short dataArr[10];

   for (;;)
   {
      int i;
      int result;

      result = mbusProtocol.readMultipleRegisters(1, 100,
                                                  dataArr,
                                                  sizeof(dataArr) / 2);
      if (result == FTALK_SUCCESS)
         for (i = 0; i < int(sizeof(dataArr) / 2); i++)
            printf("[%d]: %hd\n", 100 + i, dataArr[i]);
      else
      {
         fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));
         // Stop for fatal errors
         if (!(result & FTALK_BUS_PROTOCOL_ERROR_CLASS))
            return;
      }

#ifdef __WIN32__
      Sleep(1000);
#else
      sleep(1);
#endif
   }
}


int main()
{
   openProtocol();

   runPollLoop();

   closeProtocol();
   return (EXIT_SUCCESS);
}
```

4.2.2 MODBUS/TCP Example

The following example tcpsimple.cpp shows how to configure a MODBUS/TCP protocol and read values:

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

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


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

char *hostName = "127.0.0.1";
MbusTcpMasterProtocol mbusProtocol;


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

void openProtocol()
{
   int result;

   result = mbusProtocol.openProtocol(hostName);
   if (result != FTALK_SUCCESS)
   {
      fprintf(stderr, "Error opening protocol: %s!\n",
                       getBusProtocolErrorText(result));
      exit(EXIT_FAILURE);
   }
}


void closeProtocol()
{
   mbusProtocol.closeProtocol();
}


void runPollLoop()
{
   short dataArr[10];

   for (;;)
   {
      int i;
      int result;

      result = mbusProtocol.readMultipleRegisters(1, 100,
                                                  dataArr,
                                                  sizeof(dataArr) / 2);
```

```
      if (result == FTALK_SUCCESS)
         for (i = 0; i < int(sizeof(dataArr) / 2); i++)
            printf("[%d]: %hd\n", 100 + i, dataArr[i]);
      else
      {
         fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));
         // Stop for fatal errors
         if (!(result & FTALK_BUS_PROTOCOL_ERROR_CLASS))
            return;
      }

#ifdef __WIN32__
      Sleep(1000);
#else
      sleep(1);
#endif
   }
}


int main()
{
   openProtocol();

   runPollLoop();

   closeProtocol();
   return (EXIT_SUCCESS);
}
```

4.2.3 Modpoll application

The following more complex example modpoll.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. Modpoll is a very useful master simulator and test tool.

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

// Include FieldTalk package header
#include "MbusRtuMasterProtocol.hpp"
#include "MbusAsciiMasterProtocol.hpp"
#include "MbusTcpMasterProtocol.hpp"
#include "MbusRtuOverTcpMasterProtocol.hpp"

// Provide getopt on Win32 without using a separate lib
#ifdef __WIN32__
#  include "getopt.c"
#else
#  include <unistd.h>
```

```
#endif


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

const char versionStr[]= "$Revision: 1.15 $";
const char progName[] = "modpoll";
const char bannerStr[] =
"%s - FieldTalk(tm) Modbus(R) Polling Utility\n"
"Copyright (c) 2002-2004 FOCUS Software Engineering Pty Ltd\n"
#ifdef __WIN32__
"Getopt Library Copyright (C) 1987-1997   Free Software Foundation, Inc.\n"
#endif
;

const char usageStr[] =
"%s [options] serialport|host \n"
"Arguments: \n"
"serialport    Serial port when using Modbus ASCII or Modbus RTU protocol \n"
"              /dev/ttyS0, /dev/ttyS1 ...    on Linux \n"
"              COM1, COM2 ...                on Win32 \n"
"              /dev/ser1, /dev/ser2 ...      on QNX \n"
"host          Host name or dotted ip address when using MODBUS/TCP protocol \n
"General options: \n"
"-m ascii      Modbus ASCII protocol\n"
"-m rtu        Modbus RTU protocol (default)\n"
"-m tcp        MODBUS/TCP protocol\n"
"-m enc        Encapsulated Modbus RTU over TCP\n"
"-a #          Slave address (1-255 for RTU/ASCII, 0-255 for TCP, 1 is default)
"-r #          Start reference (1-65536, 100 is default)\n"
"-c #          Number of values to poll (1-100, 1 is default)\n"
"-t 0          Discrete output (coil) data type\n"
"-t 1          Discrete input data type\n"
"-t 3          16-bit input register data type\n"
"-t 3:hex      16-bit input register data type with hex display\n"
"-t 3:int      32-bit integer data type in input register table\n"
"-t 3:mod      32-bit module 10000 data type in input register table\n"
"-t 3:float    32-bit float data type in input register table\n"
"-t 4          16-bit output (holding) register data type (default)\n"
"-t 4:hex      16-bit output (holding) register data type with hex display\n"
"-t 4:int      32-bit integer data type in output (holding) register table\n"
"-t 4:mod      32-bit module 10000 type in output (holding) register table\n"
"-t 4:float    32-bit float data type in output (holding) register table\n"
"-i            Slave operates on big-endian 32-bit integers\n"
"-f            Slave operates on big-endian 32-bit floats\n"
"-1            Poll only once, otherwise poll every second\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 (default)\n"
"-p even       Even parity\n"
"-p odd        Odd parity\n"
```

```
"-4 #          RS485 mode, RTS on while transmitting and another # ms after.\n"
"";


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

enum
{
   RTU,
   ASCII,
   TCP,
   RTUOVERTCP
};

enum
{
   T0_BOOL,
   T1_BOOL,
   T3_REG16,
   T3_HEX16,
   T3_INT32,
   T3_MOD10000,
   T3_FLOAT32,
   T4_REG16,
   T4_HEX16,
   T4_INT32,
   T4_MOD10000,
   T4_FLOAT32
};


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

int address = 1;
int ref = 100;
int refCnt = 1;
int pollCnt = -1;
long baudRate = 9600;
int dataBits = 8;
int stopBits = 1;
int parity = MbusSerialMasterProtocol::SER_PARITY_NONE;
int protocol = RTU;
int dataType = T4_REG16;
int swapInts = 0;
int swapFloats = 0;
char *portName = NULL;
int port = 502;
int rs485Mode = 0;


/*****************************************************************************
 * Protocol and data pointers
 *****************************************************************************/
```

```
MbusMasterFunctions *mbusPtr = NULL;
void *dataPtr = 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, MbusMasterFunctions::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;
      case RTUOVERTCP:
         printf("Encapsulated RTU over TCP\n");
      break;
      default:
         printf("unknown\n");
      break;
   }
   printf("Slave configuration: ");
   printf("Address = %d, ", address);
   printf("start reference = %d, ", ref);
   printf("count = %d\n", refCnt);
   if ((protocol == TCP) || (protocol == RTUOVERTCP))
   {
      printf("TCP/IP configuration: ");
      printf("Host = %s, ", portName);
      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 MbusSerialMasterProtocol::SER_PARITY_NONE:
         printf("none\n");
      break;
      case MbusSerialMasterProtocol::SER_PARITY_EVEN:
         printf("even\n");
      break;
      case MbusSerialMasterProtocol::SER_PARITY_ODD:
         printf("odd\n");
      break;
      default:
         printf("unknown\n");
      break;
   }
}
printf("Data type: ");
switch (dataType)
{
   case T0_BOOL:
      printf("discrete output (coil)\n");
   break;
   case T1_BOOL:
      printf("discrete input\n");
   break;
   case T3_REG16:
      printf("16-bit register, input register table\n");
   break;
   case T3_HEX16:
      printf("16-bit register (hex), input register table\n");
   break;
   case T3_INT32:
      printf("32-bit integer, input register table\n");
   break;
   case T3_MOD10000:
      printf("32-bit module 10000, input register table\n");
   break;
   case T3_FLOAT32:
      printf("32-bit float, input register table\n");
   break;
   case T4_REG16:
      printf("16-bit register, output (holding) register table\n");
   break;
   case T4_HEX16:
      printf("16-bit register (hex), output (holding) register table\n");
   break;
   case T4_INT32:
      printf("32-bit integer, output (holding) register table\n");
   break;
   case T4_MOD10000:
```

```
            printf("32-bit module 10000, output (holding) register table\n");
         break;
         case T4_FLOAT32:
            printf("32-bit float, output (holding) register table\n");
         break;
         default:
            printf("unknown\n");
         break;
      }
      if (swapInts || swapFloats)
      {
         printf("Word swapping: Slave configured as big-endian");
         if (swapInts)
            printf(" word");
         if (swapInts && swapFloats)
            printf(" and");
         if (swapFloats)
            printf(" float");
         printf(" machine\n");
      }
      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, "h14:fa:r:c:b:d:s:p:t:m:");
         if (c == -1)
```

```
      break;

   switch (c)
   {
      case '1':
         pollCnt = 1;
      break;
      case '4':
         rs485Mode = (int) strtol(optarg, NULL, 0);
         if ((rs485Mode <= 0) || (rs485Mode > 1000))
            exitBadOption("Invalid RTS delay parameter");
      break;
      case 'i':
         swapInts = 1;
      break;
      case 'f':
         swapFloats = 1;
      break;
      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
                  if (strcmp(optarg, "enc") == 0)
                  {
                     protocol = RTUOVERTCP;
                  }
                  else
                  {
                     exitBadOption("Invalid protocol parameter");
                  }
      break;
      case 'a':
         address = strtol(optarg, NULL, 0);
         if ((address < 0) || (address > 255))
            exitBadOption("Invalid address parameter");
      break;
      case 'r':
         ref = strtol(optarg, NULL, 0);
          if ((ref <= 0) || (ref > 0x10000))
            exitBadOption("Invalid reference parameter");
      break;
      case 'c':
         refCnt = strtol(optarg, NULL, 0);
         if ((refCnt <= 0) || (refCnt >= 100))
            exitBadOption("Invalid count 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 = MbusSerialMasterProtocol::SER_PARITY_NONE;
   }
   else
      if (strcmp(optarg, "odd") == 0)
      {
         parity = MbusSerialMasterProtocol::SER_PARITY_ODD;
      }
      else
         if (strcmp(optarg, "even") == 0)
         {
            parity = MbusSerialMasterProtocol::SER_PARITY_EVEN;
         }
         else
         {
            port = strtol(optarg, NULL, 0);
            if ((port <= 0) || (port > 0xFFFF))
               exitBadOption("Invalid parity or port parameter");
         }
break;
case 't':
   if (strcmp(optarg, "0") == 0)
   {
      dataType = T0_BOOL;
   }
   else
   if (strcmp(optarg, "1") == 0)
   {
      dataType = T1_BOOL;
   }
   else
   if (strcmp(optarg, "3") == 0)
   {
      dataType = T3_REG16;
   }
   else
   if (strcmp(optarg, "3:hex") == 0)
   {
      dataType = T3_HEX16;
```

```
                }
                else
                if (strcmp(optarg, "3:int") == 0)
                {
                   dataType = T3_INT32;
                }
                else
                if (strcmp(optarg, "3:mod") == 0)
                {
                   dataType = T3_MOD10000;
                }
                else
                if (strcmp(optarg, "3:float") == 0)
                {
                   dataType = T3_FLOAT32;
                }
                else
                if (strcmp(optarg, "4") == 0)
                {
                   dataType = T4_REG16;
                }
                else
                if (strcmp(optarg, "4:hex") == 0)
                {
                   dataType = T4_HEX16;
                }
                else
                if (strcmp(optarg, "4:int") == 0)
                {
                   dataType = T4_INT32;
                }
                else
                if (strcmp(optarg, "4:mod") == 0)
                {
                   dataType = T4_MOD10000;
                }
                else
                if (strcmp(optarg, "4:float") == 0)
                {
                   dataType = T4_FLOAT32;
                }
                else
                {
                   exitBadOption("Invalid data type parameter");
                }
             break;
             case 'h':
                printUsage();
             break;
             default:
                exitBadOption("Unrecognized option or missing option parameter");
             break;
          }
       }

       if ((argc - optind) != 1)
             exitBadOption("Invalid number of parameters");
```

```
   else
      portName = argv[optind];
}


void openProtocol()
{
   int result = -1;

   switch (protocol)
   {
      case RTU:
         mbusPtr = new MbusRtuMasterProtocol();
         if (!mbusPtr)
         {
            fprintf(stderr, "Out of memory!\n");
            exit(EXIT_FAILURE);
         }
         if (swapInts)
            mbusPtr->configureBigEndianInts();
         if (swapFloats)
            mbusPtr->configureSwappedFloats();
         mbusPtr->setRetryCnt(2);
         mbusPtr->setPollDelay(1000);
         if (rs485Mode > 0)
            ((MbusAsciiMasterProtocol *) mbusPtr)->enableRs485Mode(rs485Mode);
         result = ((MbusRtuMasterProtocol *) mbusPtr)->openProtocol(
                   portName, baudRate, dataBits, stopBits, parity);
      break;
      case ASCII:
         mbusPtr = new MbusAsciiMasterProtocol();
         if (!mbusPtr)
         {
            fprintf(stderr, "Out of memory!\n");
            exit(EXIT_FAILURE);
         }
         if (swapInts)
            mbusPtr->configureBigEndianInts();
         if (swapFloats)
            mbusPtr->configureSwappedFloats();
         mbusPtr->setRetryCnt(2);
         mbusPtr->setPollDelay(1000);
         if (rs485Mode > 0)
            ((MbusAsciiMasterProtocol *) mbusPtr)->enableRs485Mode(rs485Mode);
         result = ((MbusAsciiMasterProtocol *) mbusPtr)->openProtocol(
                   portName, baudRate, dataBits, stopBits, parity);
      break;
      case TCP:
         mbusPtr = new MbusTcpMasterProtocol();
         if (!mbusPtr)
         {
            fprintf(stderr, "Out of memory!\n");
            exit(EXIT_FAILURE);
         }
         if (swapInts)
            mbusPtr->configureBigEndianInts();
         if (swapFloats)
```

```
            mbusPtr->configureSwappedFloats();
         mbusPtr->setPollDelay(1000);
         ((MbusTcpMasterProtocol *) mbusPtr)->setPort(port);
         result = ((MbusTcpMasterProtocol *) mbusPtr)->openProtocol(portName);
      break;
      case RTUOVERTCP:
         mbusPtr = new MbusRtuOverTcpMasterProtocol();
         if (!mbusPtr)
         {
            fprintf(stderr, "Out of memory!\n");
            exit(EXIT_FAILURE);
         }
         if (swapInts)
            mbusPtr->configureBigEndianInts();
         if (swapFloats)
            mbusPtr->configureSwappedFloats();
         mbusPtr->setPollDelay(1000);
         ((MbusRtuOverTcpMasterProtocol *) mbusPtr)->setPort(port);
         result = ((MbusRtuOverTcpMasterProtocol *) mbusPtr)->openProtocol(port
      break;
   }
   switch (result)
   {
      case FTALK_SUCCESS:
         printf("Protocol opened successfully.\n");
      break;
      case FTALK_ILLEGAL_ARGUMENT_ERROR:
         fprintf(stderr, "Configuration setting not supported!\n");
         exit(EXIT_FAILURE);
      break;
      case FTALK_TCPIP_CONNECT_ERR:
         fprintf(stderr, "Can't reach slave (check ip address)!\n");
         exit(EXIT_FAILURE);
      break;
      default:
         fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));
         exit(EXIT_FAILURE);
      break;
   }

   switch (dataType)
   {
      case T3_HEX16:
      case T3_REG16:
      case T4_HEX16:
      case T4_REG16:
         dataPtr = new short[refCnt];
      break;
      case T0_BOOL:
      case T1_BOOL:
      case T3_INT32:
      case T4_INT32:
      case T3_MOD10000:
      case T4_MOD10000:
         dataPtr = new int[refCnt];
      break;
      case T3_FLOAT32:
```

```
      case T4_FLOAT32:
         dataPtr = new float[refCnt];
      break;
   }
   if (!dataPtr)
   {
      fprintf(stderr, "Out of memory!\n");
      exit(EXIT_FAILURE);
   }
}


void closeProtocol()
{
   delete mbusPtr;
   delete [] dataPtr;
}


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

   while ((pollCnt == -1) || (pollCnt > 0))
   {
      if (pollCnt == -1)
         printf("Polling slave (Ctrl-C to stop) ...\n");
      else
      {
         printf("Polling slave ...\n");
         pollCnt--;
      }
      switch (dataType)
      {
         case T0_BOOL:
            result = mbusPtr->readCoils(address, ref,
                                        (int *) dataPtr, refCnt);
            if (result == FTALK_SUCCESS)
               for (i = 0; i < refCnt; i++)
                  printf("[%d]: %d\n", ref + i, ((int *) dataPtr)[i]);
            break;
         case T1_BOOL:
            result = mbusPtr->readInputDiscretes(address, ref,
                                                 (int *) dataPtr, refCnt);
            if (result == FTALK_SUCCESS)
               for (i = 0; i < refCnt; i++)
                  printf("[%d]: %d\n", ref + i, ((int *) dataPtr)[i]);
            break;
         case T4_REG16:
            result = mbusPtr->readMultipleRegisters(address, ref,
                                                    (short *) dataPtr, refCnt);
            if (result == FTALK_SUCCESS)
               for (i = 0; i < refCnt; i++)
                  printf("[%d]: %hd\n", ref + i, ((short *) dataPtr)[i]);
            break;
         case T4_HEX16:
```

```
   result = mbusPtr->readMultipleRegisters(address, ref,
                                           (short *) dataPtr, refCnt);
   if (result == FTALK_SUCCESS)
      for (i = 0; i < refCnt; i++)
         printf("[%d]: 0x%04hX\n", ref + i, ((short *) dataPtr)[i]);
break;
case T4_INT32:
   result = mbusPtr->readMultipleLongInts(address, ref,
                                          (long *) dataPtr, refCnt);
   if (result == FTALK_SUCCESS)
      for (i = 0; i < refCnt; i++)
         printf("[%d]: %d\n", ref + i * 2, ((int *) dataPtr)[i]);
break;
case T4_MOD10000:
   result = mbusPtr->readMultipleMod10000(address, ref,
                                          (long *) dataPtr, refCnt);
   if (result == FTALK_SUCCESS)
      for (i = 0; i < refCnt; i++)
         printf("[%d]: %d\n", ref + i * 2, ((int *) dataPtr)[i]);
break;
case T4_FLOAT32:
   result = mbusPtr->readMultipleFloats(address, ref,
                                        (float *) dataPtr, refCnt);
   if (result == FTALK_SUCCESS)
      for (i = 0; i < refCnt; i++)
         printf("[%d]: %f\n", ref + i * 2, ((float *) dataPtr)[i]);
break;
case T3_REG16:
   result = mbusPtr->readInputRegisters(address, ref,
                                        (short *) dataPtr, refCnt);
   if (result == FTALK_SUCCESS)
      for (i = 0; i < refCnt; i++)
         printf("[%d]: %hd\n", ref + i, ((short *) dataPtr)[i]);
break;
case T3_HEX16:
   result = mbusPtr->readInputRegisters(address, ref,
                                        (short *) dataPtr, refCnt);
   if (result == FTALK_SUCCESS)
      for (i = 0; i < refCnt; i++)
         printf("[%d]: 0x%04hX\n", ref + i, ((short *) dataPtr)[i]);
break;
case T3_INT32:
   result = mbusPtr->readInputLongInts(address, ref,
                                       (long *) dataPtr, refCnt);
   if (result == FTALK_SUCCESS)
      for (i = 0; i < refCnt; i++)
         printf("[%d]: %d\n", ref + i * 2, ((int *) dataPtr)[i]);
break;
case T3_MOD10000:
   result = mbusPtr->readInputMod10000(address, ref,
                                       (long *) dataPtr, refCnt);
   if (result == FTALK_SUCCESS)
      for (i = 0; i < refCnt; i++)
         printf("[%d]: %d\n", ref + i * 2, ((int *) dataPtr)[i]);
break;
case T3_FLOAT32:
   result = mbusPtr->readInputFloats(address, ref,
```

```
                                                    (float *) dataPtr, refCnt);
                if (result == FTALK_SUCCESS)
                   for (i = 0; i < refCnt; i++)
                      printf("[%d]: %f\n", ref + i * 2, ((float *) dataPtr)[i]);
             break;
          }
          if (result != FTALK_SUCCESS)
          {
             fprintf(stderr, "%s!\n", getBusProtocolErrorText(result));
             // Stop for fatal errors
             if (!(result & FTALK_BUS_PROTOCOL_ERROR_CLASS))
                return;
          }
       }
    }


    int main (int argc, char **argv)
    {
       scanOptions(argc, argv);
       printConfig();
       atexit(closeProtocol);
       openProtocol();
       pollSlave();
       return (EXIT_SUCCESS);
    }
```

4.3 What You should know about Modbus

- Some Background
- Technical Information
- The Protocol Functions
- How Slave Devices are identified
- The Register Model and Data Tables
- Data Encoding
- Register and Discrete Numbering Scheme
- The ASCII Protocol
- The RTU Protocol
- The MODBUS/TCP Protocol

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 serial protocol operates on RS 232, RS 422 and RS 485 physical networks. The TCP/IP 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 functions of conformance Class 0 and Class 1 have been implemented. In addition the most frequently used functions of conformance Class 2 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 functions:

Function Code	Current Terminology	Classic Terminology
Conformance Class 0		
3 (03 hex)	Read Multiple Registers	Read Holding Registers
16 (10 hex)	Write Multiple Registers	Preset Multiple Registers
Conformance Class 1		
1 (01 hex)	Read Coils	Read Coil Status
2 (02 hex)	Read Inputs Discretes	Read Input Status
4 (04 hex)	Read Input Registers	Read Input Registers
5 (05 hex)	Write Coil	Force Single Coil
6 (06 hex)	Write Single Register	Preset Single Register
7 (07 hex)	Read Exception Status	Read Exception Status
Conformance Class 2		
15 (0F hex)	Force Multiple Coils	Force Multiple Coils
22 (16 hex)	Mask Write Register	Mask Write Register
23 (17 hex)	Read/Write Registers	Read/Write Registers

How Slave Devices are identified A slave device is identified with its unique address identifier. Valid address identifiers supported are 1 to 255.

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 table 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.

This *FieldTalk*'s Modbus Master implementation 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

This *FieldTalk*'s Modbus Slave implementation 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**
1	15 (hex 0x8000)
2	14 (hex 0x4000)
3	13 (hex 0x2000)
4	12 (hex 0x1000)
5	11 (hex 0x0800)
6	10 (hex 0x0400)
7	9 (hex 0x0200)
8	8 (hex 0x0100)
9	7 (hex 0x0080)
10	6 (hex 0x0040)
11	5 (hex 0x0020)
12	4 (hex 0x0010)
13	3 (hex 0x0008)
14	2 (hex 0x0004)
15	1 (hex 0x0002)
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.2.0.tar.gz
```

3. Untar the tarball

```
# tar xf FT-MB??-??-ALL.2.0.tar
```

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

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

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.

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
Generic UNIX	lib/unix

Also the necessary header files to link against the library are copied into a include directory.

Your directory structure looks now like:

```
myprj
  |
  +-- fieldtalk
      |
      +-- doc
      +-- src
      +-- include
      +-+ 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.2.0.zip
```

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

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

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
```

If you are using Borland C++ and Borland's make:

```
# cd fieldtalk\src
# make
```

To compile using Visual Studio.net, open the supplied .vcproj project file with Visual Studio.net.

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

Platform	Library Directory
Windows 32-bit Visual C++	lib\win32_vc
Windows 32-bit Borland C++	lib\win32_bc

Your directory structure looks now like:

```
myprj
  |
  +-- fieldtalk
      |
      +-- doc
      +-- src
      +-- include
      +-+ lib
        |
        +-- win32_??   (exact name depends on compiler)
```

5. The library is ready to be used.

4.4.3 Specific Platform Notes

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 a minimum of 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
      +-- 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
      +-- src
      +-- include
      +-+ lib
        |
        +-- win32_vc     (exact name depends on your platform)
```

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/win32_vc/libmbusmaster.lib Ws2_32.lib
```

Borland C++ Example:

```
bcc32 -e myapp myapp.obj fieldtalk/lib/win32_vc/libmbusmaster.lib
```

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

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 SOFTWARE ENGINEERING, THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PACKAGE "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PACKAGE IS WITH YOU. SHOULD THE PACKAGE PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

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

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

15. This agreement constitutes the entire agreement between FOCUS Software Engineering and You in relation to your use of the SOFTWARE. Any change will be effective only if in writing signed by FOCUS Software Engineering and you.

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

-----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 the *FieldTalk* products. Please use the Support web page at: http://www.focus-sw.com/support.html

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

4.9 Notices

FieldTalk™ was developed by:

FOCUS Software Engineering Pty Ltd, Australia.

Index