# ModbusLib

0.4.4

Generated by Doxygen 1.12.0

# Chapter 1

# ModbusLib

### 1.0.1 Overview

ModbusLib is a free, open-source Modbus library written in C++. It implements client and server functions for TCP, RTU and ASCII versions of Modbus Protocol. It has interface for C language (implements in cModbus.h header file). Also it has optional wrapper to use with Qt (implements in ModbusQt.h header file). Library can work in both blocking and non-blocking mode.

Library implements such Modbus functions as:

- 1 (0x01) - READ_COILS
- 2 (0x02) - READ_DISCRETE_INPUTS
- 3 (0x03) - READ_HOLDING_REGISTERS
- 4 (0x04) - READ_INPUT_REGISTERS
- 5 (0x05) - WRITE_SINGLE_COIL
- 6 (0x06) - WRITE_SINGLE_REGISTER
- 7 (0x07) - READ_EXCEPTION_STATUS
- 8 (0x08) - DIAGNOSTICS
- 11 (0x0B) - GET_COMM_EVENT_COUNTER
- 12 (0x0C) - GET_COMM_EVENT_LOG
- 15 (0x0F) - WRITE_MULTIPLE_COILS
- 16 (0x10) - WRITE_MULTIPLE_REGISTERS
- 17 (0x11) - REPORT_SERVER_ID
- 22 (0x16) - MASK_WRITE_REGISTER
- 23 (0x17) - WRITE_MULTIPLE_REGISTERS
- 24 (0x18) - READ_FIFO_QUEUE

### 1.0.2 Using Library

#### 1.0.2.1 Common usage (C++)

Library was written in C++ and it is the main language to use it. To start using this library you must include `ModbusClientPort.h` (`ModbusClient.h`) or `ModbusServerPort.h` header files (of course after add include path to the compiler). This header directly or indirectly include `Modbus.h` main header file. `Modbus.h` header file contains declarations of main data types, functions and class interfaces to work with the library.

It contains definition of `Modbus::StatusCode` enumeration that defines result of library operations, `ModbusInterface` class interface that contains list of functions which the library implements, `Modbus::createClientPort` and `Modbus::createServerPort` functions, that creates corresponding `ModbusClientPort` and `ModbusServerPort` main working classes. Those classes that implements Modbus functions for the library for client and server version of protocol, respectively.

#### 1.0.2.2 Client

`ModbusClientPort` implements Modbus interface directly and can be used very simple:

```
#include <ModbusClientPort.h>
//...
void main()
{
    Modbus::TcpSettings settings;
    settings.host = "someadr.plc";
    settings.port = Modbus::STANDARD_TCP_PORT;
    settings.timeout = 3000;
    ModbusClientPort *port = Modbus::createClientPort(Modbus::TCP, &settings, true);
    const uint8_t unit = 1;
    const uint16_t offset = 0;
    const uint16_t count = 10;
    uint16_t values[count];
    Modbus::StatusCode status = port->readHoldingRegisters(unit, offset, count, values);
    if (Modbus::StatusIsGood(status))
    {
        // process out array `values' ...
    }
    else
        std::cout « "Error: " « port->lastErrorText() « '\n';
    delete port;
}
//...
```

User don't need to create any connection or open any port, library makes it automatically.

User can use `ModbusClient` class to simplify Modbus function's interface (don't need to use `unit` parameter):

```
#include <ModbusClientPort.h>
//...
void main()
{
    //...
    ModbusClient c1(1, port);
    ModbusClient c2(2, port);
    ModbusClient c3(3, port);
    Modbus::StatusCode s1, s2, s3;
    while(1)
    {
        s1 = c1.readHoldingRegisters(0, 10, values);
        s2 = c2.readHoldingRegisters(0, 10, values);
        s3 = c3.readHoldingRegisters(0, 10, values);
        Modbus::msleep(1);
    }
    //...
}
//...
```

In this example 3 clients with unit address 1, 2, 3 are used. User don't need to manage its common resource `port`. Library make it automatically. First `c1` client owns `port`, than when finished resource transferred to `c2` and so on.

#### 1.0.2.3 Server

Unlike client the server do not implement `ModbusInterface` directly. It accepts pointer to `ModbusInterface` in its constructor as parameter and transfer all requests to this interface. So user can define by itself how incoming Modbus-request will be processed:

```
#include <ModbusServerPort.h>
//...
class MyModbusDevice : public ModbusInterface
{
#define MEM_SIZE 16
    uint16_t mem4x[MEM_SIZE];
public:
    MyModbusDevice() { memset(mem4x, 0, sizeof(mem4x)); }
    uint16_t getValue(uint16_t offset) { return mem4x[offset]; }
    void setValue(uint16_t offset, uint16_t value) { mem4x[offset] = value; }
    Modbus::StatusCode readHoldingRegisters(uint8_t  unit,
                                            uint16_t offset,
                                            uint16_t count,
                                            uint16_t *values) override
    {
        if (unit != 1)
            return Modbus::Status_BadGatewayPathUnavailable;
        if ((offset + count) <= MEM_SIZE)
        {
            memcpy(values, &mem4x[offset], count*sizeof(uint16_t));
            return Modbus::Status_Good;
        }
        return Modbus::Status_BadIllegalDataAddress;
    }
};

void main()
{
    MyModbusDevice device;
    Modbus::TcpSettings settings;
    settings.port = Modbus::STANDARD_TCP_PORT;
    settings.timeout = 3000;
    ModbusServerPort *port = Modbus::createServerPort(&device, Modbus::TCP, &settings, false);
    int c = 0;
    while (1)
    {
        port->process();
        Modbus::msleep(1);
        if (c % 1000 == 0) setValue(0, getValue(0)+1);
    }
}
//...
```

In this example `MyModbusDevice` ModbusInterface class was created. It imlements only single function: `readHoldingRegisters` (`0x03`). All other functions will return `Modbus::Status_BadIllegalFunction` by default.

This example creates Modbus TCP server that process connections and increment first 4x register by 1 every second. This example uses non blocking mode.

##### 1.0.2.3.1 Non blocking mode

In non blocking mode Modbus function exits immediately even if remote connection processing is not finished. In this case function returns `Modbus::Status_Processing`. This is 'Arduino'-style of programing, when function must not be blocked and return intermediate value that indicates that function is not finished. Then external code call this function again and again until Good or Bad status will not be returned.

Example of non blocking client:

```
#include <ModbusClientPort.h>
//...
void main()
{
    //...
    ModbusClientPort *port = Modbus::createClientPort(Modbus::TCP, &settings, false);
    //...
    while(1)
    {
        s1 = c1.readHoldingRegisters(0, 10, values);
        s2 = c2.readHoldingRegisters(0, 10, values);
```

```
        s3 = c3.readHoldingRegisters(0, 10, values);
        doSomeOtherStuffInCurrentThread();
        Modbus::msleep(1);
    }
    //...
}
//...
```

So if user needs to check is function finished he can write:

```
//...
s1 = c1.readHoldingRegisters(0, 10, values);
if (!Modbus::StatusIsProcessing(s1)) {
    // ...
}
//...
```

#### 1.0.2.3.2 Signal/slot mechanism

Library has simplified Qt-like signal/slot mechanism that can use callbacks when some signal is occured. User can connect function(s) or class method(s) to the predefined signal. Callbacks will be called in the order in which they were connected.

For example `ModbusClientPort` signal/slot mechanism:

```
#include <ModbusClientPort.h>

class Printable
{
public:
    void printTx(const Modbus::Char *source, const uint8_t* buff, uint16_t size)
    {
        std::cout « source « " Tx: " « Modbus::bytesToString(buff, size) « '\n';
    }
};

void printRx(const Modbus::Char *source, const uint8_t* buff, uint16_t size)
{
    std::cout « source « " Rx: " « Modbus::bytesToString(buff, size) « '\n';
}

void main()
{
    //...
    ModbusClientPort *port = Modbus::createClientPort(Modbus::TCP, &settings, false);
    Printable print;
    port->connect(&ModbusClientPort::signalTx, &print, &Printable::printTx);
    port->connect(&ModbusClientPort::signalRx, printRx);
    //...
}
```

### 1.0.2.4 Using with C

To use the library with pure C language user needs to include only one header: `cModbus.h`. This header includes functions that wraps Modbus interface classes and its methods.

```
#include <cModbus.h>
//...
void printTx(const Char *source, const uint8_t* buff, uint16_t size)
{
    Char s[1000];
    printf("%s Tx: %s\n", source, sbytes(buff, size, s, sizeof(s)));
}

void printRx(const Char *source, const uint8_t* buff, uint16_t size)
{
    Char s[1000];
    printf("%s Rx: %s\n", source, sbytes(buff, size, s, sizeof(s)));
}

void main()
{
    TcpSettings settings;
    settings.host = "someadr.plc";
    settings.port = STANDARD_TCP_PORT;
    settings.timeout = 3000;
    const uint8_t unit = 1;
```

```
    cModbusClient client = cCliCreate(unit, TCP, &settings, true);
    cModbusClientPort cpo = cCliGetPort(client);
    StatusCode s;
    cCpoConnectTx(cpo, printTx);
    cCpoConnectRx(cpo, printRx);
    while(1)
    {
        s = cReadHoldingRegisters(client, 0, 10, values);
        //...
        msleep(1);
    }
}
//...
```

#### 1.0.2.5 Using with Qt

When including `ModbusQt.h` user can use ModbusLib in convinient way in Qt framework. It has wrapper functions for Qt library to use it together with Qt core objects:

```
#include <ModbusQt.h>
```

### 1.0.3 Examples

Examples is located in `examples` folder or root directory.

#### 1.0.3.1 `democlient`

`democlient` example demonstrate all implemented functions for client one by one begining from function with lowest number and then increasing this number with predefined period and other parameters. To see list of available parameters you can print next commands:

```
$ ./democlient -?
$ ./democlient -help
```

#### 1.0.3.2 `mbclient`

`mbclient` is a simple example that can work like command-line Modbus Client Tester. It can use only single function at a time but user can change parameters of every supported function. To see list of available parameters you can print next commands:

```
$ ./mbclient -?
$ ./mbclient -help
```

Usage example:

```
$ ./mbclient -func 3 -offset 0 -count 10 -period 500 -n inf
```

#### 1.0.3.3 `demoserver`

`demoserver` example demonstrate all implemented functions for server. It uses single block for every type of Modbus memory (0x, 1x, 3x and 4x) and emulates value change for the first 16 bit register by inceremting it by 1 every 1000 milliseconds. So user can run Modbus Client to check first 16 bit of 000001 (100001) or first register 400001 (300001) changing every 1 second. To see list of available parameters you can print next commands:

```
$ ./demoserver -?
$ ./demoserver -help
```

#### 1.0.3.4 mbserver

`mbserver` is a simple example that can work like command-line Modbus Server Tester. It implements all function of Modbus library. So remote client can work with server reading and writting values to it. To see list of available parameters you can print next commands:

```
$ ./mbserver -?
$ ./mbserver -help
```

Usage example:

```
$ ./mbserver -c0 256 -c1 256 -c3 16 -c4 16 -type RTU -serial /dev/ttyS0
```

### 1.0.4 Tests

Unit Tests using googletest library. Googletest source library must be located in external/googletest

### 1.0.5 Documenations

Documentation is located in `docs` directory. Documentation is automatically generated by doxygen.

### 1.0.6 Building

#### 1.0.6.1 Build using CMake

1. Build Tools

   Previously you need to install c++ compiler kit, git and cmake itself (qt tools if needed).

   Then set PATH env variable to find compliler, cmake, git etc.

   Don't forget to use appropriate version of compiler, linker (x86|x64).

2. Create project directory, move to it and clone repository:
   ```
   $ cd ~
   $ mkdir src
   $ cd src
   $ git clone https://github.com/serhmarch/ModbusLib.git
   ```

3. Create and/or move to directory for build output, e.g. `~/bin/ModbusLib`:
   ```
   $ cd ~
   $ mkdir -p bin/ModbusLib
   $ cd bin/ModbusLib
   ```

4. Run cmake to generate project (make) files.
   ```
   $ cmake -S ~/src/ModbusLib -B .
   ```

   To make Qt-compatibility (switch off by default for cmake build) you can use next command (e.g. for Windows 64):
   ```
   >cmake -DMB_QT_ENABLED=ON -DCMAKE_PREFIX_PATH:PATH=C:/Qt/5.15.2/msvc2019_64 -S <path\to\src\ModbusLib>
   -B .
   ```

5. Make binaries (+ debug|release config):
   ```
   $ cmake --build .
   $ cmake --build . --config Debug
   $ cmake --build . --config Release
   ```

6. Resulting bin files is located in `./bin` directory.

#### 1.0.6.2 Build using qmake

1. Update package list:
```
$ sudo apt-get update
```
2. Install main build tools like g++, make etc:
```
$ sudo apt-get install build-essential
```
3. Install Qt tools:
```
$ sudo apt-get install qtbase5-dev qttools5-dev
```
4. Check for correct instalation:
```
$ whereis qmake
qmake: /usr/bin/qmake
$ whereis libQt5Core*
libQt5Core.prl: /usr/lib/x86_64-linux-gnu/libQt5Core.prl
libQt5Core.so: /usr/lib/x86_64-linux-gnu/libQt5Core.so
libQt5Core.so.5: /usr/lib/x86_64-linux-gnu/libQt5Core.so.5
libQt5Core.so.5.15: /usr/lib/x86_64-linux-gnu/libQt5Core.so.5.15
libQt5Core.so.5.15.3: /usr/lib/x86_64-linux-gnu/libQt5Core.so.5.15.3
$ whereis libQt5Help*
libQt5Help.prl: /usr/lib/x86_64-linux-gnu/libQt5Help.prl
libQt5Help.so: /usr/lib/x86_64-linux-gnu/libQt5Help.so
libQt5Help.so.5: /usr/lib/x86_64-linux-gnu/libQt5Help.so.5
libQt5Help.so.5.15: /usr/lib/x86_64-linux-gnu/libQt5Help.so.5.15
libQt5Help.so.5.15.3: /usr/lib/x86_64-linux-gnu/libQt5Help.so.5.15.3
```
5. Install git:
```
$ sudo apt-get install git
```
6. Create project directory, move to it and clone repository:
```
$ cd ~
$ mkdir src
$ cd src
$ git clone https://github.com/serhmarch/ModbusLib.git
```
7. Create and/or move to directory for build output, e.g. ~/bin/ModbusLib:
```
$ cd ~
$ mkdir -p bin/ModbusLib
$ cd bin/ModbusLib
```
8. Run qmake to create Makefile for build:
```
$ qmake ~/src/ModbusLib/src/ModbusLib.pro -spec linux-g++
```
9. To ensure Makefile was created print:
```
$ ls -l
total 36
-rw-r--r-- 1 march march 35001 May  6 18:41 Makefile
```
10. Finaly to make current set of programs print:
```
$ make
```
11. After build step move to <build_folder>/bin to ensure everything is correct:
```
$ cd bin
$ pwd
~/bin/ModbusLib/bin
```

# Chapter 2

# Namespace Index

## 2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

# Chapter 3

# Hierarchical Index

## 3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

# Chapter 4

# Class Index

## 4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

# Chapter 5

# File Index

## 5.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 6

# Namespace Documentation

## 6.1 Modbus Namespace Reference

Main Modbus namespace. Contains classes, functions and constants to work with Modbus-protocol.

### Classes

- class Address

  *Modbus Data Address class. Represents Modbus Data Address.*
- class Defaults

  *Holds the default values of the settings.*
- struct SerialSettings

  *Struct to define settings for Serial Port.*
- class Strings

  *Sets constant key values for the map of settings.*
- struct TcpSettings

  *Struct to define settings for TCP connection.*

### Typedefs

- typedef std::string **String**

  *Modbus::String class for strings.*
- template<class T >
  using **List** = std::list<T>

  *Modbus::List template class.*
- typedef void * **Handle**

  *Handle type for native OS values.*
- typedef char **Char**

  *Type for Modbus character.*
- typedef uint32_t **Timer**

  *Type for Modbus timer.*
- typedef int64_t **Timestamp**

  *Type for Modbus timestamp (in UNIX millisec format)*
- typedef enum Modbus::_MemoryType **MemoryType**

  *Defines type of memory used in Modbus protocol.*
- typedef enum Modbus::_Color **Color**

  *Enum of color (used for console text color).*
- typedef QHash< QString, QVariant > **Settings**

  *Map for settings of Modbus protocol where key has type `QString` and value is `QVariant`.*

### Enumerations

- enum Constants { VALID_MODBUS_ADDRESS_BEGIN = 1 , VALID_MODBUS_ADDRESS_END = 247 , STANDARD_TCP_PORT = 502 }

  *Define list of contants of Modbus protocol.*
- enum _MemoryType {
  Memory_Unknown = 0xFFFF , Memory_0x = 0 , Memory_Coils = Memory_0x , Memory_1x = 1 ,
  Memory_DiscreteInputs = Memory_1x , Memory_3x = 3 , Memory_InputRegisters = Memory_3x ,
  Memory_4x = 4 ,
  Memory_HoldingRegisters = Memory_4x }

  *Defines type of memory used in Modbus protocol.*
- enum _Color {
  **Color_Black** , **Color_Red** , **Color_Green** , **Color_Yellow** ,
  **Color_Blue** , **Color_Magenta** , **Color_Cyan** , **Color_White** ,
  **Color_Default** }

  *Enum of color (used for console text color).*
- enum StatusCode {
  Status_Processing = 0x80000000 , Status_Good = 0x00000000 , Status_Bad = 0x01000000 ,
  Status_Uncertain = 0x02000000 ,
  Status_BadIllegalFunction = Status_Bad | 0x01 , Status_BadIllegalDataAddress = Status_Bad | 0x02 ,
  Status_BadIllegalDataValue = Status_Bad | 0x03 , Status_BadServerDeviceFailure = Status_Bad | 0x04 ,
  Status_BadAcknowledge = Status_Bad | 0x05 , Status_BadServerDeviceBusy = Status_Bad | 0x06 ,
  Status_BadNegativeAcknowledge = Status_Bad | 0x07 , Status_BadMemoryParityError = Status_Bad | 0x08
  ,
  Status_BadGatewayPathUnavailable = Status_Bad | 0x0A , Status_BadGatewayTargetDeviceFailedToRespond
  = Status_Bad | 0x0B , Status_BadEmptyResponse = Status_Bad | 0x101 , Status_BadNotCorrectRequest ,
  Status_BadNotCorrectResponse , Status_BadWriteBufferOverflow , Status_BadReadBufferOverflow ,
  Status_BadSerialOpen = Status_Bad | 0x201 ,
  Status_BadSerialWrite , Status_BadSerialRead , Status_BadSerialReadTimeout , Status_BadSerialWriteTimeout
  ,
  Status_BadAscMissColon = Status_Bad | 0x301 , Status_BadAscMissCrLf , Status_BadAscChar ,
  Status_BadLrc ,
  Status_BadCrc = Status_Bad | 0x401 , Status_BadTcpCreate = Status_Bad | 0x501 , Status_BadTcpConnect
  , Status_BadTcpWrite ,
  Status_BadTcpRead , Status_BadTcpBind , Status_BadTcpListen , Status_BadTcpAccept ,
  Status_BadTcpDisconnect }

  *Defines status of executed Modbus functions.*
- enum ProtocolType { ASC , RTU , TCP }

  *Defines type of Modbus protocol.*
- enum Parity {
  NoParity , EvenParity , OddParity , SpaceParity ,
  MarkParity }

  *Defines Parity for serial port.*
- enum StopBits { OneStop , OneAndHalfStop , TwoStop }

  *Defines Stop Bits for serial port.*
- enum FlowControl { NoFlowControl , HardwareControl , SoftwareControl }

  *FlowControl Parity for serial port.*

### Functions

- MODBUS_EXPORT String getLastErrorText ()
- MODBUS_EXPORT String trim (const String &str)
- template<class StringT , class T >
  StringT toBinString (T value)

- template<class StringT , class T >
  StringT toOctString (T value)
- template<class StringT , class T >
  StringT toHexString (T value)
- template<class StringT , class T >
  StringT toDecString (T value)
- template<class StringT , class T >
  StringT toDecString (T value, int c, char fillChar='0')
- template<typename StringT >
  bool startsWith (const StringT &s, const char *prefix)
- int decDigitValue (int ch)
- int hexDigitValue (int ch)
- String toModbusString (int val)
- MODBUS_EXPORT String bytesToString (const uint8_t *buff, uint32_t count)
- MODBUS_EXPORT String asciiToString (const uint8_t *buff, uint32_t count)
- MODBUS_EXPORT List< String > availableSerialPorts ()
- MODBUS_EXPORT List< int32_t > availableBaudRate ()
- MODBUS_EXPORT List< int8_t > availableDataBits ()
- MODBUS_EXPORT List< Parity > availableParity ()
- MODBUS_EXPORT List< StopBits > availableStopBits ()
- MODBUS_EXPORT List< FlowControl > availableFlowControl ()
- MODBUS_EXPORT ModbusPort * createPort (ProtocolType type, const void *settings, bool blocking)
- MODBUS_EXPORT ModbusClientPort * createClientPort (ProtocolType type, const void *settings, bool blocking)
- MODBUS_EXPORT ModbusServerPort * createServerPort (ModbusInterface *device, ProtocolType type, const void *settings, bool blocking)
- StatusCode readMemRegs (uint32_t offset, uint32_t count, void *values, const void *memBuff, uint32_t memRegCount)
- StatusCode writeMemRegs (uint32_t offset, uint32_t count, const void *values, void *memBuff, uint32_t memRegCount)
- StatusCode readMemBits (uint32_t offset, uint32_t count, void *values, const void *memBuff, uint32_t memBitCount)
- StatusCode writeMemBits (uint32_t offset, uint32_t count, const void *values, void *memBuff, uint32_t memBitCount)
- bool StatusIsProcessing (StatusCode status)
- bool StatusIsGood (StatusCode status)
- bool StatusIsBad (StatusCode status)
- bool StatusIsUncertain (StatusCode status)
- bool StatusIsStandardError (StatusCode status)
- bool getBit (const void *bitBuff, uint16_t bitNum)
- bool getBitS (const void *bitBuff, uint16_t bitNum, uint16_t maxBitCount)
- void setBit (void *bitBuff, uint16_t bitNum, bool value)
- void setBitS (void *bitBuff, uint16_t bitNum, bool value, uint16_t maxBitCount)
- bool * getBits (const void *bitBuff, uint16_t bitNum, uint16_t bitCount, bool *boolBuff)
- bool * getBitsS (const void *bitBuff, uint16_t bitNum, uint16_t bitCount, bool *boolBuff, uint16_t maxBitCount)
- void * setBits (void *bitBuff, uint16_t bitNum, uint16_t bitCount, const bool *boolBuff)
- void * setBitsS (void *bitBuff, uint16_t bitNum, uint16_t bitCount, const bool *boolBuff, uint16_t maxBitCount)
- MODBUS_EXPORT uint32_t modbusLibVersion ()
- MODBUS_EXPORT const Char * modbusLibVersionStr ()
- uint16_t toModbusOffset (uint32_t adr)
- MODBUS_EXPORT uint16_t crc16 (const uint8_t *byteArr, uint32_t count)
- MODBUS_EXPORT uint8_t lrc (const uint8_t *byteArr, uint32_t count)
- MODBUS_EXPORT StatusCode readMemRegs (uint32_t offset, uint32_t count, void *values, const void *memBuff, uint32_t memRegCount, uint32_t *outCount)
- MODBUS_EXPORT StatusCode writeMemRegs (uint32_t offset, uint32_t count, const void *values, void *memBuff, uint32_t memRegCount, uint32_t *outCount)

- MODBUS_EXPORT StatusCode readMemBits (uint32_t offset, uint32_t count, void *values, const void *memBuff, uint32_t memBitCount, uint32_t *outCount)
- MODBUS_EXPORT StatusCode writeMemBits (uint32_t offset, uint32_t count, const void *values, void *memBuff, uint32_t memBitCount, uint32_t *outCount)
- MODBUS_EXPORT uint32_t bytesToAscii (const uint8_t *bytesBuff, uint8_t *asciiBuff, uint32_t count)
- MODBUS_EXPORT uint32_t asciiToBytes (const uint8_t *asciiBuff, uint8_t *bytesBuff, uint32_t count)
- MODBUS_EXPORT Char * sbytes (const uint8_t *buff, uint32_t count, Char *str, uint32_t strmaxlen)
- MODBUS_EXPORT Char * sascii (const uint8_t *buff, uint32_t count, Char *str, uint32_t strmaxlen)
- MODBUS_EXPORT const Char * sprotocolType (ProtocolType type)
- MODBUS_EXPORT ProtocolType toprotocolType (const Char *s)
- MODBUS_EXPORT const Char * sbaudRate (int32_t baudRate)
- MODBUS_EXPORT int32_t tobaudRate (const Char *s)
- MODBUS_EXPORT const Char * sdataBits (int8_t dataBits)
- MODBUS_EXPORT int8_t todataBits (const Char *s)
- MODBUS_EXPORT const Char * sparity (Parity parity)
- MODBUS_EXPORT Parity toparity (const Char *s)
- MODBUS_EXPORT const Char * sstopBits (StopBits stopBits)
- MODBUS_EXPORT StopBits tostopBits (const Char *s)
- MODBUS_EXPORT const Char * sflowControl (FlowControl flowControl)
- MODBUS_EXPORT FlowControl toflowControl (const Char *s)
- MODBUS_EXPORT Timer timer ()
- MODBUS_EXPORT Timestamp currentTimestamp ()
- MODBUS_EXPORT void setConsoleColor (Color color)
- MODBUS_EXPORT void msleep (uint32_t msec)
- MODBUS_EXPORT uint8_t getSettingUnit (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT ProtocolType getSettingType (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint32_t getSettingTries (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT QString getSettingHost (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint16_t getSettingPort (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint32_t getSettingTimeout (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint32_t getSettingMaxconn (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT QString getSettingSerialPortName (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT int32_t getSettingBaudRate (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT int8_t getSettingDataBits (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT Parity getSettingParity (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT StopBits getSettingStopBits (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT FlowControl getSettingFlowControl (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint32_t getSettingTimeoutFirstByte (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint32_t getSettingTimeoutInterByte (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT bool getSettingBroadcastEnabled (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT void setSettingUnit (Settings &s, uint8_t v)
- MODBUS_EXPORT void setSettingType (Settings &s, ProtocolType v)
- MODBUS_EXPORT void setSettingTries (Settings &s, uint32_t)
- MODBUS_EXPORT void setSettingHost (Settings &s, const QString &v)
- MODBUS_EXPORT void setSettingPort (Settings &s, uint16_t v)
- MODBUS_EXPORT void setSettingTimeout (Settings &s, uint32_t v)
- MODBUS_EXPORT void setSettingMaxconn (Settings &s, uint32_t v)
- MODBUS_EXPORT void setSettingSerialPortName (Settings &s, const QString &v)
- MODBUS_EXPORT void setSettingBaudRate (Settings &s, int32_t v)
- MODBUS_EXPORT void setSettingDataBits (Settings &s, int8_t v)
- MODBUS_EXPORT void setSettingParity (Settings &s, Parity v)
- MODBUS_EXPORT void setSettingStopBits (Settings &s, StopBits v)
- MODBUS_EXPORT void setSettingFlowControl (Settings &s, FlowControl v)
- MODBUS_EXPORT void setSettingTimeoutFirstByte (Settings &s, uint32_t v)
- MODBUS_EXPORT void setSettingTimeoutInterByte (Settings &s, uint32_t v)

- MODBUS_EXPORT void setSettingBroadcastEnabled (Settings &s, bool v)
- Address addressFromQString (const QString &s)
- template<class EnumType >
  QString enumKey (int value)
- template<class EnumType >
  QString enumKey (EnumType value, const QString &byDef=QString())
- template<class EnumType >
  EnumType enumValue (const QString &key, bool *ok=nullptr, EnumType defaultValue=static_cast< EnumType >(-1))
- template<class EnumType >
  EnumType enumValue (const QVariant &value, bool *ok=nullptr, EnumType defaultValue=static_cast< EnumType >(-1))
- template<class EnumType >
  EnumType enumValue (const QVariant &value, EnumType defaultValue)
- template<class EnumType >
  EnumType enumValue (const QVariant &value)
- MODBUS_EXPORT ProtocolType toProtocolType (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT ProtocolType toProtocolType (const QVariant &v, bool *ok=nullptr)
- MODBUS_EXPORT int32_t toBaudRate (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT int32_t toBaudRate (const QVariant &v, bool *ok=nullptr)
- MODBUS_EXPORT int8_t toDataBits (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT int8_t toDataBits (const QVariant &v, bool *ok=nullptr)
- MODBUS_EXPORT Parity toParity (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT Parity toParity (const QVariant &v, bool *ok=nullptr)
- MODBUS_EXPORT StopBits toStopBits (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT StopBits toStopBits (const QVariant &v, bool *ok=nullptr)
- MODBUS_EXPORT FlowControl toFlowControl (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT FlowControl toFlowControl (const QVariant &v, bool *ok=nullptr)
- MODBUS_EXPORT QString toString (StatusCode v)
- MODBUS_EXPORT QString toString (ProtocolType v)
- MODBUS_EXPORT QString toString (Parity v)
- MODBUS_EXPORT QString toString (StopBits v)
- MODBUS_EXPORT QString toString (FlowControl v)
- QString bytesToString (const QByteArray &v)
- QString asciiToString (const QByteArray &v)
- MODBUS_EXPORT QStringList availableSerialPortList ()
- MODBUS_EXPORT ModbusPort * createPort (const Settings &settings, bool blocking=false)
- MODBUS_EXPORT ModbusClientPort * createClientPort (const Settings &settings, bool blocking=false)
- MODBUS_EXPORT ModbusServerPort * createServerPort (ModbusInterface *device, const Settings &settings, bool blocking=false)

### 6.1.1 Detailed Description

Main Modbus namespace. Contains classes, functions and constants to work with Modbus-protocol.

### 6.1.2 Enumeration Type Documentation

#### 6.1.2.1 _MemoryType

```
enum Modbus::_MemoryType
```

Defines type of memory used in Modbus protocol.

**Enumerator**

| | |
|---|---|
| Memory_Unknown | Invalid memory type. |
| Memory_0x | Memory allocated for coils/discrete outputs. |
| Memory_Coils | Same as `Memory_0x`. |
| Memory_1x | Memory allocated for discrete inputs. |
| Memory_DiscreteInputs | Same as `Memory_1x`. |
| Memory_3x | Memory allocated for analog inputs. |
| Memory_InputRegisters | Same as `Memory_3x`. |
| Memory_4x | Memory allocated for holding registers/analog outputs. |
| Memory_HoldingRegisters | Same as `Memory_4x`. |

#### 6.1.2.2 Constants

`enum Modbus::Constants`

Define list of contants of Modbus protocol.

**Enumerator**

| | |
|---|---|
| VALID_MODBUS_ADDRESS_BEGIN | Start of Modbus device address range according to specification. |
| VALID_MODBUS_ADDRESS_END | End of the Modbus protocol device address range according to the specification. |
| STANDARD_TCP_PORT | Standard TCP port of the Modbus protocol. |

#### 6.1.2.3 FlowControl

`enum Modbus::FlowControl`

FlowControl Parity for serial port.

**Enumerator**

| | |
|---|---|
| NoFlowControl | No flow control. |
| HardwareControl | Hardware flow control (RTS/CTS). |
| SoftwareControl | Software flow control (XON/XOFF). |

#### 6.1.2.4 Parity

`enum Modbus::Parity`

Defines Parity for serial port.

**Enumerator**

| | |
|---|---|
| NoParity | No parity bit it sent. This is the most common parity setting. |
| EvenParity | The number of 1 bits in each character, including the parity bit, is always even. |

| | |
|---|---|
| OddParity | The number of 1 bits in each character, including the parity bit, is always odd. It ensures that at least one state transition occurs in each character. |
| SpaceParity | Space parity. The parity bit is sent in the space signal condition. It does not provide error detection information. |
| MarkParity | Mark parity. The parity bit is always set to the mark signal condition (logical 1). It does not provide error detection information. |

#### 6.1.2.5 ProtocolType

`enum Modbus::ProtocolType`

Defines type of Modbus protocol.

**Enumerator**

| | |
|---|---|
| ASC | ASCII version of Modbus communication protocol. |
| RTU | RTU version of Modbus communication protocol. |
| TCP | TCP version of Modbus communication protocol. |

#### 6.1.2.6 StatusCode

`enum Modbus::StatusCode`

Defines status of executed Modbus functions.

**Enumerator**

| | |
|---|---|
| Status_Processing | The operation is not complete. Further operation is required. |
| Status_Good | Successful result. |
| Status_Bad | Error. General. |
| Status_Uncertain | The status is undefined. |
| Status_BadIllegalFunction | Standard error. The feature is not supported. |
| Status_BadIllegalDataAddress | Standard error. Invalid data address. |
| Status_BadIllegalDataValue | Standard error. Invalid data value. |
| Status_BadServerDeviceFailure | Standard error. Failure during a specified operation. |
| Status_BadAcknowledge | Standard error. The server has accepted the request and is processing it, but it will take a long time. |
| Status_BadServerDeviceBusy | Standard error. The server is busy processing a long command. The request must be repeated later. |
| Status_BadNegativeAcknowledge | Standard error. The programming function cannot be performed. |
| Status_BadMemoryParityError | Standard error. The server attempted to read a record file but detected a parity error in memory. |
| Status_BadGatewayPathUnavailable | Standard error. Indicates that the gateway was unable to allocate an internal communication path from the input port o the output port for processing the request. Usually means that the gateway is misconfigured or overloaded. |

**Enumerator**

| | |
|---|---|
| Status_BadGatewayTargetDeviceFailedToRespond | Standard error. Indicates that no response was obtained from the target device. Usually means that the device is not present on the network. |
| Status_BadEmptyResponse | Error. Empty request/response body. |
| Status_BadNotCorrectRequest | Error. Invalid request. |
| Status_BadNotCorrectResponse | Error. Invalid response. |
| Status_BadWriteBufferOverflow | Error. Write buffer overflow. |
| Status_BadReadBufferOverflow | Error. Request receive buffer overflow. |
| Status_BadSerialOpen | Error. Serial port cannot be opened. |
| Status_BadSerialWrite | Error. Cannot send a parcel to the serial port. |
| Status_BadSerialRead | Error. Reading the serial port (timeout) |
| Status_BadSerialReadTimeout | Error. Reading the serial port (timeout) |
| Status_BadSerialWriteTimeout | Error. Writing the serial port (timeout) |
| Status_BadAscMissColon | Error (ASC). Missing packet start character ':'. |
| Status_BadAscMissCrLf | Error (ASC). '\r\n' end of packet character missing. |
| Status_BadAscChar | Error (ASC). Invalid ASCII character. |
| Status_BadLrc | Error (ASC). Invalid checksum. |
| Status_BadCrc | Error (RTU). Wrong checksum. |
| Status_BadTcpCreate | Error. Unable to create a TCP socket. |
| Status_BadTcpConnect | Error. Unable to create a TCP connection. |
| Status_BadTcpWrite | Error. Unable to send a TCP packet. |
| Status_BadTcpRead | Error. Unable to receive a TCP packet. |
| Status_BadTcpBind | Error. Unable to bind a TCP socket (server side) |
| Status_BadTcpListen | Error. Unable to listen a TCP socket (server side) |
| Status_BadTcpAccept | Error. Unable accept bind a TCP socket (server side) |
| Status_BadTcpDisconnect | Error. Bad disconnection result. |

#### 6.1.2.7 StopBits

```
enum Modbus::StopBits
```

Defines Stop Bits for serial port.

**Enumerator**

| | |
|---|---|
| OneStop | 1 stop bit. |
| OneAndHalfStop | 1.5 stop bit. |
| TwoStop | 2 stop bits. |

### 6.1.3 Function Documentation

#### 6.1.3.1 addressFromQString()

```
Address Modbus::addressFromQString (
            const QString & s)  [inline]
```

Convert String repr to Modbus::Address

#### 6.1.3.2 asciiToBytes()

```
MODBUS_EXPORT uint32_t Modbus::asciiToBytes (
            const uint8_t * asciiBuff,
            uint8_t * bytesBuff,
            uint32_t count)
```

Function converts ASCII repr `asciiBuff` to binary byte array. Every byte of output `bytesBuff` are repr as two bytes in `asciiBuff`, where most signified tetrabits represented as leading byte in hex digit in ASCII encoding (upper) and less signified tetrabits represented as tailing byte in hex digit in ASCII encoding (upper). `count` is a size of input array `asciiBuff`.

**Note**

Output array `bytesBuff` must be at least twice smaller than input array `asciiBuff`.

**Returns**

Returns size of `bytesBuff` in bytes which calc as {`output` = count / 2}

#### 6.1.3.3 asciiToString() [1/2]

```
QString Modbus::asciiToString (
            const QByteArray & v)  [inline]
```

Make string representation of ASCII array and separate bytes by space

#### 6.1.3.4 asciiToString() [2/2]

```
MODBUS_EXPORT String Modbus::asciiToString (
            const uint8_t * buff,
            uint32_t count)
```

Make string representation of ASCII array and separate bytes by space

#### 6.1.3.5 availableBaudRate()

```
MODBUS_EXPORT List< int32_t > Modbus::availableBaudRate ()
```

Return list of baud rates

#### 6.1.3.6 availableDataBits()

```
MODBUS_EXPORT List< int8_t > Modbus::availableDataBits ()
```

Return list of data bits

#### 6.1.3.7 availableFlowControl()

```
MODBUS_EXPORT List< FlowControl > Modbus::availableFlowControl ()
```

Return list of `FlowControl` values

#### 6.1.3.8 availableParity()

```
MODBUS_EXPORT List< Parity > Modbus::availableParity ()
```

Return list of `Parity` values

#### 6.1.3.9 availableSerialPortList()

```
MODBUS_EXPORT QStringList Modbus::availableSerialPortList ()
```

Returns list of string that represent names of serial ports

#### 6.1.3.10 availableSerialPorts()

```
MODBUS_EXPORT List< String > Modbus::availableSerialPorts ()
```

Return list of names of available serial ports

#### 6.1.3.11 availableStopBits()

```
MODBUS_EXPORT List< StopBits > Modbus::availableStopBits ()
```

Return list of `StopBits` values

#### 6.1.3.12 bytesToAscii()

```
MODBUS_EXPORT uint32_t Modbus::bytesToAscii (
            const uint8_t * bytesBuff,
            uint8_t * asciiBuff,
            uint32_t count)
```

Function converts byte array `bytesBuff` to ASCII repr of byte array. Every byte of `bytesBuff` are repr as two bytes in `asciiBuff`, where most signified tetrabits represented as leading byte in hex digit in ASCII encoding (upper) and less signified tetrabits represented as tailing byte in hex digit in ASCII encoding (upper). `count` is count bytes of `bytesBuff`.

**Note**

Output array `asciiBuff` must be at least twice bigger than input array `bytesBuff`.

**Returns**

Returns size of `asciiBuff` in bytes which calc as {`output` = count * 2}

#### 6.1.3.13 bytesToString() [1/2]

```
QString Modbus::bytesToString (
            const QByteArray & v)  [inline]
```

Make string representation of bytes array and separate bytes by space

#### 6.1.3.14 bytesToString() [2/2]

```
MODBUS_EXPORT String Modbus::bytesToString (
            const uint8_t * buff,
            uint32_t count)
```

Make string representation of bytes array and separate bytes by space

#### 6.1.3.15 crc16()

```
MODBUS_EXPORT uint16_t Modbus::crc16 (
            const uint8_t * byteArr,
            uint32_t count)
```

CRC16 checksum hash function (for Modbus RTU).

**Returns**

Returns a 16-bit unsigned integer value of the checksum

#### 6.1.3.16 createClientPort() [1/2]

```
MODBUS_EXPORT ModbusClientPort * Modbus::createClientPort (
            const Settings & settings,
            bool blocking = false)
```

Same as `Modbus::createClientPort(ProtocolType type, const void *settings, bool blocking)` but `ProtocolType type` and `const void *settings` are defined by `Modbus::Settings` key-value map.

#### 6.1.3.17 createClientPort() [2/2]

```
MODBUS_EXPORT ModbusClientPort * Modbus::createClientPort (
            ProtocolType type,
            const void * settings,
            bool blocking)
```

Function for creation `ModbusClientPort` with defined parameters:

**Parameters**

| | | |
|---|---|---|
| in | *type* | Protocol type: TCP, RTU, ASC. |
| in | *settings* | For TCP must be pointer: `TcpSettings*`, `SerialSettings*` otherwise. |
| in | *blocking* | If true blocking will be set, non blocking otherwise. |

#### 6.1.3.18 createPort() [1/2]

```
MODBUS_EXPORT ModbusPort * Modbus::createPort (
            const Settings & settings,
            bool blocking = false)
```

Same as `Modbus::createPort(ProtocolType type, const void *settings, bool blocking)` but `ProtocolType type` and `const void *settings` are defined by `Modbus::Settings` key-value map.

#### 6.1.3.19 createPort() [2/2]

```
MODBUS_EXPORT ModbusPort * Modbus::createPort (
            ProtocolType type,
            const void * settings,
            bool blocking)
```

Function for creation `ModbusPort` with defined parameters:

**Parameters**

| | | |
|---|---|---|
| in | *type* | Protocol type: TCP, RTU, ASC. |
| in | *settings* | For TCP must be pointer: `TcpSettings*`, `SerialSettings*` otherwise. |
| in | *blocking* | If true blocking will be set, non blocking otherwise. |

#### 6.1.3.20 createServerPort() [1/2]

```
MODBUS_EXPORT ModbusServerPort * Modbus::createServerPort (
            ModbusInterface * device,
            const Settings & settings,
            bool blocking = false)
```

Same as `Modbus::createServerPort(ProtocolType type, const void *settings, bool blocking)` but `ProtocolType type` and `const void *settings` are defined by `Modbus::Settings` key-value map.

#### 6.1.3.21 createServerPort() [2/2]

```
MODBUS_EXPORT ModbusServerPort * Modbus::createServerPort (
            ModbusInterface * device,
            ProtocolType type,
            const void * settings,
            bool blocking)
```

Function for creation `ModbusServerPort` with defined parameters:

**Parameters**

| | | |
|---|---|---|
| in | *device* | Pointer to the `ModbusInterface` implementation to which all requests for Modbus functions are forwarded. |
| in | *type* | Protocol type: TCP, RTU, ASC. |
| in | *settings* | For TCP must be pointer: `TcpSettings*`, `SerialSettings*` otherwise. |
| in | *blocking* | If true blocking will be set, non blocking otherwise. |

#### 6.1.3.22 currentTimestamp()

```
MODBUS_EXPORT Timestamp Modbus::currentTimestamp ()
```

Get current timestamp in UNIX format in milliseconds.

#### 6.1.3.23 decDigitValue()

```
int Modbus::decDigitValue (
            int ch)  [inline]
```

Returns value of decimal digit [0-9] for ASCII code `ch` or -1 if the value cannot be converted.

#### 6.1.3.24 enumKey() [1/2]

```
template<class EnumType >
QString Modbus::enumKey (
            EnumType value,
            const QString & byDef = QString())  [inline]
```

Convert value to QString key for type

#### 6.1.3.25 enumKey() [2/2]

```
template<class EnumType >
QString Modbus::enumKey (
            int value)  [inline]
```

Convert value to QString key for type

#### 6.1.3.26 enumValue() [1/4]

```
template<class EnumType >
EnumType Modbus::enumValue (
            const QString & key,
            bool * ok = nullptr,
            EnumType defaultValue = static_cast<EnumType>(-1))  [inline]
```

Convert key to value for enumeration by QString key

#### 6.1.3.27 enumValue() [2/4]

```
template<class EnumType >
EnumType Modbus::enumValue (
            const QVariant & value)  [inline]
```

Convert `QVariant` value to enumeration value (int - value, string - key).

#### 6.1.3.28 enumValue() [3/4]

```
template<class EnumType >
EnumType Modbus::enumValue (
            const QVariant & value,
            bool * ok = nullptr,
            EnumType defaultValue = static_cast<EnumType>(-1))  [inline]
```

Convert QVariant value to enumeration value (int - value, string - key). Stores result of convertion in output parameter ok. If value can't be converted, defaultValue is returned.

#### 6.1.3.29 enumValue() [4/4]

```
template<class EnumType >
EnumType Modbus::enumValue (
            const QVariant & value,
            EnumType defaultValue)  [inline]
```

Convert QVariant value to enumeration value (int - value, string - key). If value can't be converted, default↩ Value is returned.

#### 6.1.3.30 getBit()

```
bool Modbus::getBit (
            const void * bitBuff,
            uint16_t bitNum)  [inline]
```

Returns the value of the bit with number 'bitNum' from the bit array 'bitBuff'.

#### 6.1.3.31 getBitS()

```
bool Modbus::getBitS (
            const void * bitBuff,
            uint16_t bitNum,
            uint16_t maxBitCount)  [inline]
```

Returns the value of the bit with the number 'bitNum' from the bit array 'bitBuff', if the bit number is greater than or equal to 'maxBitCount', then 'false' is returned.

#### 6.1.3.32 getBits()

```
bool * Modbus::getBits (
            const void * bitBuff,
            uint16_t bitNum,
            uint16_t bitCount,
            bool * boolBuff)  [inline]
```

Gets the values of bits with number bitNum and count bitCount from the bit array bitBuff and stores their values in the boolean array boolBuff, where the value of each bit is stored as a separate bool value.

**Returns**

A pointer to the boolBuff array.

#### 6.1.3.33 getBitsS()

```
bool * Modbus::getBitsS (
            const void * bitBuff,
            uint16_t bitNum,
            uint16_t bitCount,
            bool * boolBuff,
            uint16_t maxBitCount)  [inline]
```

Similar to the `Modbus::getBits(const void*,uint16_t,uint16_t,bool*)` function, but it is controlled that the size does not exceed the maximum number of bits `maxBitCount`.

**Returns**

A pointer to the `boolBuff` array.

#### 6.1.3.34 getLastErrorText()

```
MODBUS_EXPORT String Modbus::getLastErrorText ()
```

Returns string representation of the last error

#### 6.1.3.35 getSettingBaudRate()

```
MODBUS_EXPORT int32_t Modbus::getSettingBaudRate (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the serial port's baud rate. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.36 getSettingBroadcastEnabled()

```
MODBUS_EXPORT bool Modbus::getSettingBroadcastEnabled (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the serial port enables broadcast mode for `0` unit address. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.37 getSettingDataBits()

```
MODBUS_EXPORT int8_t Modbus::getSettingDataBits (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the serial port's data bits. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.38 getSettingFlowControl()

```
MODBUS_EXPORT FlowControl Modbus::getSettingFlowControl (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the serial port's flow control. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.39 getSettingHost()

```
MODBUS_EXPORT QString Modbus::getSettingHost (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the IP address or DNS name of the remote device. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.40 getSettingMaxconn()

```
MODBUS_EXPORT uint32_t Modbus::getSettingMaxconn (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the maximum number of simultaneous connections to the server. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.41 getSettingParity()

```
MODBUS_EXPORT Parity Modbus::getSettingParity (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the serial port's parity. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.42 getSettingPort()

```
MODBUS_EXPORT uint16_t Modbus::getSettingPort (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the TCP port of the remote device. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.43 getSettingSerialPortName()

```
MODBUS_EXPORT QString Modbus::getSettingSerialPortName (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the serial port name. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.44 getSettingStopBits()

```
MODBUS_EXPORT StopBits Modbus::getSettingStopBits (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the serial port's stop bits. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.45 getSettingTimeout()

```
MODBUS_EXPORT uint32_t Modbus::getSettingTimeout (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for connection timeout (milliseconds). If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.46 getSettingTimeoutFirstByte()

```
MODBUS_EXPORT uint32_t Modbus::getSettingTimeoutFirstByte (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the serial port's timeout waiting first byte of packet. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.47 getSettingTimeoutInterByte()

```
MODBUS_EXPORT uint32_t Modbus::getSettingTimeoutInterByte (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the serial port's timeout waiting next byte of packet. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.48 getSettingTries()

```
MODBUS_EXPORT uint32_t Modbus::getSettingTries (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for number of tries a Modbus request is repeated if it fails. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.49 getSettingType()

```
MODBUS_EXPORT ProtocolType Modbus::getSettingType (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the type of Modbus protocol. If value can't be retrieved that default value is returned and ∗ok = false (if provided).

#### 6.1.3.50 getSettingUnit()

```
MODBUS_EXPORT uint8_t Modbus::getSettingUnit (
            const Settings & s,
            bool * ok = nullptr)
```

Get settings value for the unit number of remote device. If value can't be retrieved that default value is returned and *ok = false (if provided).

#### 6.1.3.51 hexDigitValue()

```
int Modbus::hexDigitValue (
            int ch)  [inline]
```

Returns value of hex digit [0-15] for ASCII code `ch`. or -1 if the value cannot be converted.

#### 6.1.3.52 lrc()

```
MODBUS_EXPORT uint8_t Modbus::lrc (
            const uint8_t * byteArr,
            uint32_t count)
```

LRC checksum hash function (for Modbus ASCII).

**Returns**

Returns an 8-bit unsigned integer value of the checksum

#### 6.1.3.53 modbusLibVersion()

```
MODBUS_EXPORT uint32_t Modbus::modbusLibVersion ()
```

Returns version of current lib like (major << 16) + (minor << 8) + patch.

#### 6.1.3.54 modbusLibVersionStr()

```
MODBUS_EXPORT const Char * Modbus::modbusLibVersionStr ()
```

Returns version of current lib as string constant pointer like "major.minor.patch".

#### 6.1.3.55 msleep()

```
MODBUS_EXPORT void Modbus::msleep (
            uint32_t msec)
```

Make current thread sleep with 'msec' milliseconds.

#### 6.1.3.56 readMemBits() [1/2]

```
StatusCode Modbus::readMemBits (
            uint32_t offset,
            uint32_t count,
            void * values,
            const void * memBuff,
            uint32_t memBitCount)  [inline]
```

Overloaded function

#### 6.1.3.57 readMemBits() [2/2]

```
MODBUS_EXPORT StatusCode Modbus::readMemBits (
            uint32_t offset,
            uint32_t count,
            void * values,
            const void * memBuff,
            uint32_t memBitCount,
            uint32_t * outCount)
```

Function for copy (read) values from memory input `memBuff` and put it to the output buffer `values` for discretes (bits):

**Parameters**

| | | |
|---|---|---|
| in | *offset* | Memory offset to read from `memBuff` in bit size. |
| in | *count* | Count of bits to read from memory `memBuff`. |
| out | *values* | Output buffer to store data. |
| in | *memBuff* | Pointer to the memory which holds data. |
| in | *memBitCount* | Size of memory buffer `memBuff` in bits. |
| out | *outCount* | Optional, can be NULL. If specified, then if the requested amount of memory exceeds the limits of this memory, the error is not returned, and the amount of memory read is reduced to the memory limits and this new amount is returned in `outCount` |

#### 6.1.3.58 readMemRegs() [1/2]

```
StatusCode Modbus::readMemRegs (
            uint32_t offset,
            uint32_t count,
            void * values,
            const void * memBuff,
            uint32_t memRegCount)  [inline]
```

Overloaded function

#### 6.1.3.59 readMemRegs() [2/2]

```
MODBUS_EXPORT StatusCode Modbus::readMemRegs (
            uint32_t offset,
            uint32_t count,
            void * values,
            const void * memBuff,
            uint32_t memRegCount,
            uint32_t * outCount)
```

Function for copy (read) values from memory input `memBuff` and put it to the output buffer `values` for 16 bit registers:

**Parameters**

| | | |
|---|---|---|
| in | *offset* | Memory offset to read from `memBuff` in 16-bit registers size. |
| in | *count* | Count of 16-bit registers to read from memory `memBuff`. |
| out | *values* | Output buffer to store data. |
| in | *memBuff* | Pointer to the memory which holds data. |
| in | *memRegCount* | Size of memory buffer `memBuff` in 16-bit registers. |
| out | *outCount* | Optional, can be NULL. If specified, then if the requested amount of memory exceeds the limits of this memory, the error is not returned, and the amount of memory read is reduced to the memory limits and this new amount is returned in `outCount` |

#### 6.1.3.60 sascii()

```
MODBUS_EXPORT Char * Modbus::sascii (
            const uint8_t * buff,
            uint32_t count,
            Char * str,
            uint32_t strmaxlen)
```

Make string representation of ASCII array and separate bytes by space

#### 6.1.3.61 sbaudRate()

```
MODBUS_EXPORT const Char * Modbus::sbaudRate (
            int32_t baudRate)
```

Returns pointer to constant string value that represent `int32_t` baud rate value or nullptr (NULL) if the value is invalid.

**See also**

availableBaudRate()

#### 6.1.3.62 sbytes()

```
MODBUS_EXPORT Char * Modbus::sbytes (
            const uint8_t * buff,
            uint32_t count,
            Char * str,
            uint32_t strmaxlen)
```

Make string representation of bytes array and separate bytes by space

#### 6.1.3.63 sdataBits()

```
MODBUS_EXPORT const Char * Modbus::sdataBits (
            int8_t dataBits)
```

Returns pointer to constant string value that represent name of the data bits value ("8", "7", "6" or "5") or nullptr (NULL) if the value is invalid.

**See also**

availableDataBits()

#### 6.1.3.64 setBit()

```
void Modbus::setBit (
            void * bitBuff,
            uint16_t bitNum,
            bool value)  [inline]
```

Sets the value of the bit with the number 'bitNum' to the bit array 'bitBuff'.

#### 6.1.3.65 setBitS()

```
void Modbus::setBitS (
            void * bitBuff,
            uint16_t bitNum,
            bool value,
            uint16_t maxBitCount)  [inline]
```

Sets the value of the bit with the number 'bitNum' to the bit array 'bitBuff', controlling the size of the array 'maxBit↩ Count' in bits.

#### 6.1.3.66 setBits()

```
void * Modbus::setBits (
            void * bitBuff,
            uint16_t bitNum,
            uint16_t bitCount,
            const bool * boolBuff)  [inline]
```

Sets the values of the bits in the `bitBuff` array starting with the number `bitNum` and the count `bitCount` from the `boolBuff` array, where the value of each bit is stored as a separate `bool` value.

**Returns**

A pointer to the `bitBuff` array.

#### 6.1.3.67 setBitsS()

```
void * Modbus::setBitsS (
            void * bitBuff,
            uint16_t bitNum,
            uint16_t bitCount,
            const bool * boolBuff,
            uint16_t maxBitCount)  [inline]
```

Similar to the `Modbus::setBits(void*,uint16_t,uint16_t,const bool*)` function, but it is controlled that the size does not exceed the maximum number of bits `maxBitCount`.

**Returns**

A pointer to the `bitBuff` array.

#### 6.1.3.68 setConsoleColor()

```
MODBUS_EXPORT void Modbus::setConsoleColor (
            Color color)
```

Set color of console text.

#### 6.1.3.69 setSettingBaudRate()

```
MODBUS_EXPORT void Modbus::setSettingBaudRate (
            Settings & s,
            int32_t v)
```

Set settings value for the serial port's baud rate.

#### 6.1.3.70 setSettingBroadcastEnabled()

```
MODBUS_EXPORT void Modbus::setSettingBroadcastEnabled (
            Settings & s,
            bool v)
```

Set settings value for the serial port enables broadcast mode for `0` unit address.

#### 6.1.3.71 setSettingDataBits()

```
MODBUS_EXPORT void Modbus::setSettingDataBits (
            Settings & s,
            int8_t v)
```

Set settings value for the serial port's data bits.

#### 6.1.3.72 setSettingFlowControl()

```
MODBUS_EXPORT void Modbus::setSettingFlowControl (
            Settings & s,
            FlowControl v)
```

Set settings value for the serial port's flow control.

#### 6.1.3.73 setSettingHost()

```
MODBUS_EXPORT void Modbus::setSettingHost (
            Settings & s,
            const QString & v)
```

Set settings value for the IP address or DNS name of the remote device.

#### 6.1.3.74 setSettingMaxconn()

```
MODBUS_EXPORT void Modbus::setSettingMaxconn (
            Settings & s,
            uint32_t v)
```

Set settings value for maximum number of simultaneous connections to the server.

#### 6.1.3.75 setSettingParity()

```
MODBUS_EXPORT void Modbus::setSettingParity (
            Settings & s,
            Parity v)
```

Set settings value for the serial port's parity.

#### 6.1.3.76 setSettingPort()

```
MODBUS_EXPORT void Modbus::setSettingPort (
            Settings & s,
            uint16_t v)
```

Set settings value for the TCP port number of the remote device.

#### 6.1.3.77 setSettingSerialPortName()

```
MODBUS_EXPORT void Modbus::setSettingSerialPortName (
            Settings & s,
            const QString & v)
```

Set settings value for the serial port name.

#### 6.1.3.78 setSettingStopBits()

```
MODBUS_EXPORT void Modbus::setSettingStopBits (
            Settings & s,
            StopBits v)
```

Set settings value for the serial port's stop bits.

#### 6.1.3.79 setSettingTimeout()

```
MODBUS_EXPORT void Modbus::setSettingTimeout (
            Settings & s,
            uint32_t v)
```

Set settings value for connection timeout (milliseconds).

#### 6.1.3.80 setSettingTimeoutFirstByte()

```
MODBUS_EXPORT void Modbus::setSettingTimeoutFirstByte (
            Settings & s,
            uint32_t v)
```

Set settings value for the serial port's timeout waiting first byte of packet.

#### 6.1.3.81 setSettingTimeoutInterByte()

```
MODBUS_EXPORT void Modbus::setSettingTimeoutInterByte (
            Settings & s,
            uint32_t v)
```

Set settings value for the serial port's timeout waiting next byte of packet.

#### 6.1.3.82 setSettingTries()

```
MODBUS_EXPORT void Modbus::setSettingTries (
            Settings & s,
            uint32_t )
```

Set settings value for number of tries a Modbus request is repeated if it fails.

#### 6.1.3.83 setSettingType()

```
MODBUS_EXPORT void Modbus::setSettingType (
            Settings & s,
            ProtocolType v)
```

Set settings value the type of Modbus protocol.

#### 6.1.3.84 setSettingUnit()

```
MODBUS_EXPORT void Modbus::setSettingUnit (
            Settings & s,
            uint8_t v)
```

Set settings value for the unit number of remote device.

#### 6.1.3.85 sflowControl()

```
MODBUS_EXPORT const Char * Modbus::sflowControl (
            FlowControl flowControl)
```

Returns pointer to constant string value that represent name of the `FlowControl` parameter or nullptr (NULL) if the value is invalid.

**See also**

availableFlowControl()

#### 6.1.3.86 sparity()

```
MODBUS_EXPORT const Char * Modbus::sparity (
            Parity parity)
```

Returns pointer to constant string value that represent name of the `Parity` value or nullptr (NULL) if the value is invalid.

**See also**

availableParity()

#### 6.1.3.87 sprotocolType()

```
MODBUS_EXPORT const Char * Modbus::sprotocolType (
            ProtocolType type)
```

Returns pointer to constant string value that represent name of the `ProtocolType` value or nullptr (NULL) if the value is invalid.

#### 6.1.3.88 sstopBits()

```
MODBUS_EXPORT const Char * Modbus::sstopBits (
            StopBits stopBits)
```

Returns pointer to constant string value that represent name of the `StopBits` value or nullptr (NULL) if the value is invalid.

**See also**

availableStopBits()

#### 6.1.3.89 startsWith()

```
template<typename StringT >
bool Modbus::startsWith (
            const StringT & s,
            const char * prefix)
```

Returns true if string `s` starts with `prefix`.

#### 6.1.3.90 StatusIsBad()

```
bool Modbus::StatusIsBad (
            StatusCode status)  [inline]
```

Returns a general indication that the operation result is unsuccessful.

#### 6.1.3.91 StatusIsGood()

```
bool Modbus::StatusIsGood (
            StatusCode status)  [inline]
```

Returns a general indication that the operation result is successful.

#### 6.1.3.92 StatusIsProcessing()

```
bool Modbus::StatusIsProcessing (
            StatusCode status)  [inline]
```

Returns a general indication that the result of the operation is incomplete.

#### 6.1.3.93 StatusIsStandardError()

```
bool Modbus::StatusIsStandardError (
            StatusCode status)  [inline]
```

Returns a general sign that the result is standard error.

#### 6.1.3.94 StatusIsUncertain()

```
bool Modbus::StatusIsUncertain (
            StatusCode status)  [inline]
```

Returns a general sign that the result of the operation is undefined.

#### 6.1.3.95 timer()

```
MODBUS_EXPORT Timer Modbus::timer ()
```

Get timer value in milliseconds.

#### 6.1.3.96 toBaudRate() [1/2]

```
MODBUS_EXPORT int32_t Modbus::toBaudRate (
            const QString & s,
            bool * ok = nullptr)
```

Converts string representation to `BaudRate` value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.97 toBaudRate() [2/2]

```
MODBUS_EXPORT int32_t Modbus::toBaudRate (
            const QVariant & v,
            bool * ok = nullptr)
```

Converts QVariant value to `DataBits` value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.98 tobaudRate()

```
MODBUS_EXPORT int32_t Modbus::tobaudRate (
            const Char * s)
```

Converts string representation to `int32_t` baud rate value or returns -1 if value cannot be converted.

#### 6.1.3.99 toBinString()

```
template<class StringT , class T >
StringT Modbus::toBinString (
            T value)
```

Convert integer value to oct string representation withleading zeroes.

#### 6.1.3.100 toDataBits() [1/2]

```
MODBUS_EXPORT int8_t Modbus::toDataBits (
            const QString & s,
            bool * ok = nullptr)
```

Converts string representation to `DataBits` value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.101 toDataBits() [2/2]

```
MODBUS_EXPORT int8_t Modbus::toDataBits (
            const QVariant & v,
            bool * ok = nullptr)
```

Converts QVariant value to `DataBits` value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.102 todataBits()

```
MODBUS_EXPORT int8_t Modbus::todataBits (
            const Char * s)
```

Converts string representation to data bits `int8_t` value or returns -1 if value cannot be converted.

#### 6.1.3.103 toDecString() [1/2]

```
template<class StringT , class T >
StringT Modbus::toDecString (
            T value)
```

Convert integer value to dec string representation.

#### 6.1.3.104 toDecString() [2/2]

```
template<class StringT , class T >
StringT Modbus::toDecString (
            T value,
            int c,
            char fillChar = '0')
```

Convert integer value to dec string representation for `c`-count symbols with left digits filled with `fillChar`.

#### 6.1.3.105 toFlowControl() [1/2]

```
MODBUS_EXPORT FlowControl Modbus::toFlowControl (
            const QString & s,
            bool * ok = nullptr)
```

Converts string representation to `FlowControl` enum value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.106 toFlowControl() [2/2]

```
MODBUS_EXPORT FlowControl Modbus::toFlowControl (
            const QVariant & v,
            bool * ok = nullptr)
```

Converts QVariant value to `FlowControl` enum value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.107 toflowControl()

```
MODBUS_EXPORT FlowControl Modbus::toflowControl (
            const Char * s)
```

Converts string representation to `FlowControl` value or returns -1 if value cannot be converted.

#### 6.1.3.108 toHexString()

```
template<class StringT , class T >
StringT Modbus::toHexString (
            T value)
```

Convert integer value to hex string representation with upper case and leading zeroes.

#### 6.1.3.109 toModbusOffset()

```
uint16_t Modbus::toModbusOffset (
            uint32_t adr)  [inline]
```

Function extract only offset part from Modbus address and returns it.

#### 6.1.3.110 toModbusString()

```
String Modbus::toModbusString (
            int val)  [inline]
```

Convert interger value to Modbus::String

**Returns**

Returns new Modbus::String value

#### 6.1.3.111 toOctString()

```
template<class StringT , class T >
StringT Modbus::toOctString (
            T value)
```

Convert integer value to oct string representation withleading zeroes.

#### 6.1.3.112 toParity() [1/2]

```
MODBUS_EXPORT Parity Modbus::toParity (
            const QString & s,
            bool * ok = nullptr)
```

Converts string representation to `Parity` enum value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.113 toParity() [2/2]

```
MODBUS_EXPORT Parity Modbus::toParity (
            const QVariant & v,
            bool * ok = nullptr)
```

Converts QVariant value to `Parity` enum value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.114 toparity()

```
MODBUS_EXPORT Parity Modbus::toparity (
            const Char * s)
```

Converts string representation to `Parity` value or returns -1 if value cannot be converted.

#### 6.1.3.115 toProtocolType() [1/2]

```
MODBUS_EXPORT ProtocolType Modbus::toProtocolType (
            const QString & s,
            bool * ok = nullptr)
```

Converts string representation to `ProtocolType` enum value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.116 toProtocolType() [2/2]

```
MODBUS_EXPORT ProtocolType Modbus::toProtocolType (
            const QVariant & v,
            bool * ok = nullptr)
```

Converts QVariant value to `ProtocolType` enum value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.117 toprotocolType()

```
MODBUS_EXPORT ProtocolType Modbus::toprotocolType (
            const Char * s)
```

Converts string representation to `ProtocolType` value or returns -1 if value cannot be converted.

#### 6.1.3.118 toStopBits() [1/2]

```
MODBUS_EXPORT StopBits Modbus::toStopBits (
            const QString & s,
            bool * ok = nullptr)
```

Converts string representation to `StopBits` enum value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.119 toStopBits() [2/2]

```
MODBUS_EXPORT StopBits Modbus::toStopBits (
            const QVariant & v,
            bool * ok = nullptr)
```

Converts QVariant value to `StopBits` enum value. If ok is not nullptr, failure is reported by setting ∗ok to false, and success by setting ∗ok to true.

#### 6.1.3.120 tostopBits()

```
MODBUS_EXPORT StopBits Modbus::tostopBits (
            const Char * s)
```

Converts string representation to `StopBits` value or returns -1 if value cannot be converted.

#### 6.1.3.121 toString() [1/5]

```
MODBUS_EXPORT QString Modbus::toString (
            FlowControl v)
```

Returns string representation of `FlowControl` enum value

#### 6.1.3.122 toString() [2/5]

```
MODBUS_EXPORT QString Modbus::toString (
            Parity v)
```

Returns string representation of `Parity` enum value

#### 6.1.3.123 toString() [3/5]

```
MODBUS_EXPORT QString Modbus::toString (
            ProtocolType v)
```

Returns string representation of `ProtocolType` enum value

#### 6.1.3.124 toString() [4/5]

```
MODBUS_EXPORT QString Modbus::toString (
            StatusCode v)
```

Returns string representation of `StatusCode` enum value

#### 6.1.3.125 toString() [5/5]

```
MODBUS_EXPORT QString Modbus::toString (
            StopBits v)
```

Returns string representation of `StopBits` enum value

#### 6.1.3.126 trim()

```
MODBUS_EXPORT String Modbus::trim (
            const String & str)
```

Returns trim white spaces from the left and right side of the string `str`

#### 6.1.3.127 writeMemBits() [1/2]

```
StatusCode Modbus::writeMemBits (
            uint32_t offset,
            uint32_t count,
            const void * values,
            void * memBuff,
            uint32_t memBitCount)  [inline]
```

Overloaded function

#### 6.1.3.128 writeMemBits() [2/2]

```
MODBUS_EXPORT StatusCode Modbus::writeMemBits (
            uint32_t offset,
            uint32_t count,
            const void * values,
            void * memBuff,
            uint32_t memBitCount,
            uint32_t * outCount)
```

Function for copy (write) values from input buffer `values` to memory `memBuff` for discretes (bits):

**Parameters**

| | | |
|---|---|---|
| in | *offset* | Memory offset to write to `memBuff` in bit size. |
| in | *count* | Count of bits to write into memory `memBuff`. |
| out | *values* | Input buffer that holds data to write. |
| in | *memBuff* | Pointer to the memory buffer. |
| in | *memBitCount* | Size of memory buffer `memBuff` in bits. |
| out | *outCount* | Optional, can be NULL. If specified, then if the requested amount of memory exceeds the limits of this memory, the error is not returned, and the amount of memory write is reduced to the memory limits and this new amount is returned in `outCount` |

#### 6.1.3.129 writeMemRegs() [1/2]

```
StatusCode Modbus::writeMemRegs (
            uint32_t offset,
            uint32_t count,
            const void * values,
            void * memBuff,
            uint32_t memRegCount)  [inline]
```

Overloaded function

#### 6.1.3.130 writeMemRegs() [2/2]

```
MODBUS_EXPORT StatusCode Modbus::writeMemRegs (
            uint32_t offset,
            uint32_t count,
            const void * values,
            void * memBuff,
            uint32_t memRegCount,
            uint32_t * outCount)
```

Function for copy (write) values from input buffer `values` to memory `memBuff` for 16 bit registers:

**Parameters**

| | | |
|---|---|---|
| in | *offset* | Memory offset to write to `memBuff` in 16-bit registers size. |
| in | *count* | Count of 16-bit registers to write into memory `memBuff`. |
| out | *values* | Input buffer that holds data to write. |
| in | *memBuff* | Pointer to the memory buffer. |
| in | *memRegCount* | Size of memory buffer `memBuff` in 16-bit registers. |
| out | *outCount* | Optional, can be NULL. If specified, then if the requested amount of memory exceeds the limits of this memory, the error is not returned, and the amount of memory write is reduced to the memory limits and this new amount is returned in `outCount` |

# Chapter 7

# Class Documentation

## 7.1 Modbus::Address Class Reference

Modbus Data Address class. Represents Modbus Data Address.

`#include <Modbus.h>`

### Public Types

- enum Notation { Notation_Default , Notation_Modbus , Notation_IEC61131 , Notation_IEC61131Hex }
  *Type of Modbus Data Address notation.*

### Public Member Functions

- Address ()
- Address (Modbus::MemoryType type, uint16_t offset)
- Address (uint32_t adr)
- bool isValid () const
- Modbus::MemoryType type () const
- uint16_t offset () const
- void setOffset (uint16_t offset)
- uint32_t number () const
- void setNumber (uint16_t number)
- int toInt () const
- operator uint32_t () const
- Address & operator= (uint32_t v)
- Address & operator+= (uint16_t c)
- template<class StringT >
  StringT toString (Notation notation) const

### Static Public Member Functions

- template<class StringT >
  static Address **fromString** (const StringT &s)
  *Make modbus address from string representaion.*

### 7.1.1 Detailed Description

Modbus Data Address class. Represents Modbus Data Address.

`Address` class is used to represent Modbus Data Address. It contains memory type and offset. E.g. `Address(Modbus::Memory_4x, 0)` creates `400001` standard address. E.g. `Address(400001)` creates `Address` with type `Modbus::Memory_4x` and offset `0`, and `Address(1)` creates `Address` with type `Modbus::Memory_0x` and offset `0`. Class provides convertions from/to String using template methods for this. `template <class StringT> - StringT` can be `std::basic_string` or `QString`.

### 7.1.2 Member Enumeration Documentation

#### 7.1.2.1 Notation

`enum Modbus::Address::Notation`

Type of Modbus Data Address notation.

**Enumerator**

| | |
|---|---|
| Notation_Default | Default notation which is equal to Modbus notation. |
| Notation_Modbus | Standard Modbus address notation like `000001`, `100001`, `300001`, `400001` |
| Notation_IEC61131 | IEC-61131 address notation like `%Q0`, `%I0`, `%IW0`, `%MW0` |
| Notation_IEC61131Hex | IEC-61131 Hex address notation like `%Q0000h`, `%I0000h`, `%IW0000h`, `%MW0000h` |

### 7.1.3 Constructor & Destructor Documentation

#### 7.1.3.1 Address() [1/3]

```
Modbus::Address::Address ()  [inline]
```

Defaullt constructor ot the class. Creates invalid Modbus Data Address

#### 7.1.3.2 Address() [2/3]

```
Modbus::Address::Address (
            Modbus::MemoryType type,
            uint16_t offset)  [inline]
```

Constructor ot the class. E.g. `Address(Modbus::Memory_4x, 0)` creates `400001` standard address.

#### 7.1.3.3 Address() [3/3]

```
Modbus::Address::Address (
            uint32_t adr)  [inline]
```

Constructor ot the class. E.g. `Address(400001)` creates `Address` with type `Modbus::Memory_4x` and offset `0`, and `Address(1)` creates `Address` with type `Modbus::Memory_0x` and offset `0`.

### 7.1.4 Member Function Documentation

#### 7.1.4.1 isValid()

```
bool Modbus::Address::isValid () const  [inline]
```

Returns `true` if memory type is not `Modbus::Memory_Unknown`, `false` otherwise

#### 7.1.4.2 number()

```
uint32_t Modbus::Address::number () const  [inline]
```

Returns memory number (offset+1) of Modbus Data Address

#### 7.1.4.3 offset()

```
uint16_t Modbus::Address::offset () const  [inline]
```

Returns memory offset of Modbus Data Address

#### 7.1.4.4 operator uint32_t()

```
Modbus::Address::operator uint32_t () const  [inline]
```

Converts current Modbus Data Address to `quint32`, e.g. `Address(Modbus::Memory_4x, 0)` will be converted to `400001`.

#### 7.1.4.5 operator+=()

```
Address & Modbus::Address::operator+= (
            uint16_t c)  [inline]
```

Add operator definition. Increase address offset by `c` value

#### 7.1.4.6 operator=()

```
Address & Modbus::Address::operator= (
            uint32_t v)  [inline]
```

Assigment operator definition.

#### 7.1.4.7 setNumber()

```
void Modbus::Address::setNumber (
            uint16_t number)  [inline]
```

Set memory number of Modbus Data Address

#### 7.1.4.8 setOffset()

```
void Modbus::Address::setOffset (
            uint16_t offset)  [inline]
```

Set memory offset of Modbus Data Address

#### 7.1.4.9 toInt()

```
int Modbus::Address::toInt () const  [inline]
```

Returns int repr of Modbus Data Address e.g. `Address(Modbus::Memory_4x, 0)` will be converted to `400001`.

#### 7.1.4.10 toString()

```
template<class StringT >
StringT Modbus::Address::toString (
            Notation notation) const  [inline]
```

Returns string repr of Modbus Data Address e.g. `Address(Modbus::Memory_4x, 0)` will be converted to `QString("400001")`.

#### 7.1.4.11 type()

```
Modbus::MemoryType Modbus::Address::type () const  [inline]
```

Returns memory type of Modbus Data Address

The documentation for this class was generated from the following file:

- c:/Users/march/Dropbox/PRJ/ModbusLib/src/Modbus.h

## 7.2 Modbus::Defaults Class Reference

Holds the default values of the settings.

`#include <ModbusQt.h>`

### Public Member Functions

- Defaults ()

### Static Public Member Functions

- static const Defaults & instance ()

### Public Attributes

- const uint8_t **unit**

  *Default value for the unit number of remote device.*
- const ProtocolType **type**

  *Default value for the type of Modbus protocol.*
- const uint32_t **tries**

  *Default value for number of tries a Modbus request is repeated if it fails.*
- const QString **host**

  *Default value for the IP address or DNS name of the remote device.*
- const uint16_t **port**

  *Default value for the TCP port number of the remote device.*
- const uint32_t **timeout**

  *Default value for connection timeout (milliseconds)*
- const uint32_t **maxconn**

  *Default value for the maximum number of simultaneous connections to the server.*
- const QString **serialPortName**

  *Default value for the serial port name.*
- const int32_t **baudRate**

  *Default value for the serial port's baud rate.*
- const int8_t **dataBits**

  *Default value for the serial port's data bits.*
- const Parity **parity**

  *Default value for the serial port's parity.*
- const StopBits **stopBits**

  *Default value for the serial port's stop bits.*
- const FlowControl **flowControl**

  *Default value for the serial port's flow control.*
- const uint32_t **timeoutFirstByte**

  *Default value for the serial port's timeout waiting first byte of packet.*
- const uint32_t **timeoutInterByte**

  *Default value for the serial port's timeout waiting next byte of packet.*
- const bool **isBroadcastEnabled**

  *Default value for the serial port enables broadcast mode for `0` unit address.*

## 7.2.1 Detailed Description

Holds the default values of the settings.

## 7.2.2 Constructor & Destructor Documentation

### 7.2.2.1 Defaults()

```
Modbus::Defaults::Defaults ()
```

Constructor ot the class.

### 7.2.3 Member Function Documentation

#### 7.2.3.1 instance()

```
static const Defaults & Modbus::Defaults::instance ()  [static]
```

Returns a reference to the global `Modbus::Defaults` object.

The documentation for this class was generated from the following file:

- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusQt.h

## 7.3 ModbusSerialPort::Defaults Struct Reference

Holds the default values of the settings.

```
#include <ModbusSerialPort.h>
```

#### Public Member Functions

- Defaults ()

#### Static Public Member Functions

- static const Defaults & instance ()

#### Public Attributes

- const Modbus::Char * **portName**

  *Default value for the serial port name.*
- const int32_t **baudRate**

  *Default value for the serial port's baud rate.*
- const int8_t **dataBits**

  *Default value for the serial port's data bits.*
- const Modbus::Parity **parity**

  *Default value for the serial port's patiry.*
- const Modbus::StopBits **stopBits**

  *Default value for the serial port's stop bits.*
- const Modbus::FlowControl **flowControl**

  *Default value for the serial port's flow control.*
- const uint32_t **timeoutFirstByte**

  *Default value for the serial port's timeout waiting first byte of packet.*
- const uint32_t **timeoutInterByte**

  *Default value for the serial port's timeout waiting next byte of packet.*

### 7.3.1 Detailed Description

Holds the default values of the settings.

### 7.3.2 Constructor & Destructor Documentation

#### 7.3.2.1 Defaults()

```
ModbusSerialPort::Defaults::Defaults ()
```

Constructor ot the class.

### 7.3.3 Member Function Documentation

#### 7.3.3.1 instance()

```
static const Defaults & ModbusSerialPort::Defaults::instance ()  [static]
```

Returns a reference to the global `ModbusSerialPort::Defaults` object.

The documentation for this struct was generated from the following file:

- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusSerialPort.h

## 7.4 ModbusTcpPort::Defaults Struct Reference

`Defaults` class constain default settings values for `ModbusTcpPort`.

```
#include <ModbusTcpPort.h>
```

#### Public Member Functions

- Defaults ()

#### Static Public Member Functions

- static const Defaults & instance ()

#### Public Attributes

- const Modbus::Char * **host**

  *Default setting 'TCP host name (DNS or IP address)'.*
- const uint16_t **port**

  *Default setting 'TCP port number' for the listening server.*
- const uint32_t **timeout**

  *Default setting for the read timeout of every single conncetion.*

### 7.4.1 Detailed Description

`Defaults` class constain default settings values for `ModbusTcpPort`.

### 7.4.2 Constructor & Destructor Documentation

#### 7.4.2.1 Defaults()

```
ModbusTcpPort::Defaults::Defaults ()
```

Constructor of the class.

### 7.4.3 Member Function Documentation

#### 7.4.3.1 instance()

```
static const Defaults & ModbusTcpPort::Defaults::instance ()  [static]
```

Returns a reference to the global default value object.

The documentation for this struct was generated from the following file:

- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusTcpPort.h

## 7.5 ModbusTcpServer::Defaults Struct Reference

`Defaults` class constain default settings values for `ModbusTcpServer`.

```
#include <ModbusTcpServer.h>
```

#### Public Member Functions

- Defaults ()

#### Static Public Member Functions

- static const Defaults & instance ()

#### Public Attributes

- const uint16_t **port**

  *Default setting 'TCP port number' for the listening server.*
- const uint32_t **timeout**

  *Default setting for the read timeout of every single conncetion.*
- const uint32_t **maxconn**

  *Default setting for the maximum number of simultaneous connections to the server.*

### 7.5.1 Detailed Description

`Defaults` class constain default settings values for `ModbusTcpServer`.

### 7.5.2 Constructor & Destructor Documentation

#### 7.5.2.1 Defaults()

```
ModbusTcpServer::Defaults::Defaults ()
```

Constructor of the class.

### 7.5.3 Member Function Documentation

#### 7.5.3.1 instance()

```
static const Defaults & ModbusTcpServer::Defaults::instance ()  [static]
```

Returns a reference to the global default value object.

The documentation for this struct was generated from the following file:

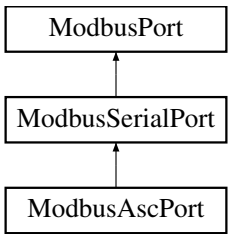
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusTcpServer.h

## 7.6 ModbusAscPort Class Reference

Implements ASCII version of the Modbus communication protocol.

```
#include <ModbusAscPort.h>
```

Inheritance diagram for ModbusAscPort:




#### Public Member Functions

- ModbusAscPort (bool blocking=false)
- ~ModbusAscPort ()
- Modbus::ProtocolType type () const override

## Public Member Functions inherited from ModbusSerialPort

- ~ModbusSerialPort ()
- Modbus::Handle handle () const override
- Modbus::StatusCode open () override
- Modbus::StatusCode close () override
- bool isOpen () const override
- const Modbus::Char * portName () const
- void setPortName (const Modbus::Char *portName)
- int32_t baudRate () const
- void setBaudRate (int32_t baudRate)
- int8_t dataBits () const
- void setDataBits (int8_t dataBits)
- Modbus::Parity parity () const
- void setParity (Modbus::Parity parity)
- Modbus::StopBits stopBits () const
- void setStopBits (Modbus::StopBits stopBits)
- Modbus::FlowControl flowControl () const
- void setFlowControl (Modbus::FlowControl flowControl)
- uint32_t timeoutFirstByte () const
- void setTimeoutFirstByte (uint32_t timeout)
- uint32_t timeoutInterByte () const
- void setTimeoutInterByte (uint32_t timeout)
- const uint8_t * readBufferData () const override
- uint16_t readBufferSize () const override
- const uint8_t * writeBufferData () const override
- uint16_t writeBufferSize () const override

## Public Member Functions inherited from ModbusPort

- virtual ~ModbusPort ()
- virtual void setNextRequestRepeated (bool v)
- bool isChanged () const
- bool isServerMode () const
- virtual void setServerMode (bool mode)
- bool isBlocking () const
- bool isNonBlocking () const
- uint32_t timeout () const
- void setTimeout (uint32_t timeout)
- Modbus::StatusCode lastErrorStatus () const
- const Modbus::Char * lastErrorText () const

### Protected Member Functions

- Modbus::StatusCode writeBuffer (uint8_t unit, uint8_t func, uint8_t *buff, uint16_t szInBuff) override
- Modbus::StatusCode readBuffer (uint8_t &unit, uint8_t &func, uint8_t *buff, uint16_t maxSzBuff, uint16_t *szOutBuff) override

## Protected Member Functions inherited from ModbusSerialPort

- Modbus::StatusCode write () override
- Modbus::StatusCode read () override

### Protected Member Functions inherited from ModbusPort

- Modbus::StatusCode setError (Modbus::StatusCode status, const Modbus::Char *text)

## 7.6.1 Detailed Description

Implements ASCII version of the Modbus communication protocol.

ModbusAscPort derived from ModbusSerialPort and implements writeBuffer and readBuffer for ASCII version of Modbus communication protocol.

## 7.6.2 Constructor & Destructor Documentation

### 7.6.2.1 ModbusAscPort()

```
ModbusAscPort::ModbusAscPort (
            bool blocking = false)
```

Constructor of the class. if blocking = true then defines blocking mode, non blocking otherwise.

### 7.6.2.2 ~ModbusAscPort()

```
ModbusAscPort::~ModbusAscPort ()
```

Destructor of the class.

## 7.6.3 Member Function Documentation

### 7.6.3.1 readBuffer()

```
Modbus::StatusCode ModbusAscPort::readBuffer (
            uint8_t & unit,
            uint8_t & func,
            uint8_t * buff,
            uint16_t maxSzBuff,
            uint16_t * szOutBuff)  [override], [protected], [virtual]
```

The function parses the packet that the read() function puts into the buffer, checks it for correctness, extracts its parameters, and returns the status of the operation.

Implements ModbusPort.

### 7.6.3.2 type()

```
Modbus::ProtocolType ModbusAscPort::type () const  [inline], [override], [virtual]
```

Returns the Modbus protocol type. For ModbusAscPort returns Modbus::ASC.

Implements ModbusPort.

#### 7.6.3.3 writeBuffer()

```
Modbus::StatusCode ModbusAscPort::writeBuffer (
            uint8_t unit,
            uint8_t func,
            uint8_t * buff,
            uint16_t szInBuff)  [override], [protected], [virtual]
```

The function directly generates a packet and places it in the buffer for further sending. Returns the status of the operation.

Implements ModbusPort.

The documentation for this class was generated from the following file:

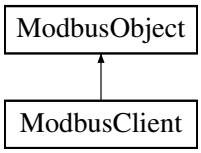
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusAscPort.h

## 7.7 ModbusClient Class Reference

The `ModbusClient` class implements the interface of the client part of the Modbus protocol.

`#include <ModbusClient.h>`

Inheritance diagram for ModbusClient:




### Public Member Functions

- ModbusClient (uint8_t unit, ModbusClientPort *port)
- Modbus::ProtocolType type () const
- uint8_t unit () const
- void setUnit (uint8_t unit)
- bool isOpen () const
- ModbusClientPort * port () const
- Modbus::StatusCode readCoils (uint16_t offset, uint16_t count, void *values)
- Modbus::StatusCode readDiscreteInputs (uint16_t offset, uint16_t count, void *values)
- Modbus::StatusCode readHoldingRegisters (uint16_t offset, uint16_t count, uint16_t *values)
- Modbus::StatusCode readInputRegisters (uint16_t offset, uint16_t count, uint16_t *values)
- Modbus::StatusCode writeSingleCoil (uint16_t offset, bool value)
- Modbus::StatusCode writeSingleRegister (uint16_t offset, uint16_t value)
- Modbus::StatusCode readExceptionStatus (uint8_t *value)
- Modbus::StatusCode diagnostics (uint16_t subfunc, uint8_t insize, const uint8_t *indata, uint8_t *outsize, uint8_t *outdata)
- Modbus::StatusCode getCommEventCounter (uint16_t *status, uint16_t *eventCount)
- Modbus::StatusCode getCommEventLog (uint16_t *status, uint16_t *eventCount, uint16_t *messageCount, uint8_t *eventBuffSize, uint8_t *eventBuff)
- Modbus::StatusCode writeMultipleCoils (uint16_t offset, uint16_t count, const void *values)

- Modbus::StatusCode writeMultipleRegisters (uint16_t offset, uint16_t count, const uint16_t *values)
- Modbus::StatusCode reportServerID (uint8_t *count, uint8_t *data)
- Modbus::StatusCode maskWriteRegister (uint16_t offset, uint16_t andMask, uint16_t orMask)
- Modbus::StatusCode readWriteMultipleRegisters (uint16_t readOffset, uint16_t readCount, uint16_t *read↩ Values, uint16_t writeOffset, uint16_t writeCount, const uint16_t *writeValues)
- Modbus::StatusCode readFIFOQueue (uint16_t fifoadr, uint16_t *count, uint16_t *values)
- Modbus::StatusCode readCoilsAsBoolArray (uint16_t offset, uint16_t count, bool *values)
- Modbus::StatusCode readDiscreteInputsAsBoolArray (uint16_t offset, uint16_t count, bool *values)
- Modbus::StatusCode writeMultipleCoilsAsBoolArray (uint16_t offset, uint16_t count, const bool *values)
- Modbus::StatusCode lastPortStatus () const
- Modbus::StatusCode lastPortErrorStatus () const
- const Modbus::Char * lastPortErrorText () const

## Public Member Functions inherited from ModbusObject

- ModbusObject ()
- virtual ~ModbusObject ()
- const Modbus::Char * objectName () const
- void setObjectName (const Modbus::Char *name)
- template<class SignalClass , class T , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class SignalClass , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- template<class ReturnType , class ... Args>
  void disconnect (ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- void disconnectFunc (void *funcPtr)
- template<class T , class ReturnType , class ... Args>
  void disconnect (T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class T >
  void disconnect (T *object)

### Additional Inherited Members

## Static Public Member Functions inherited from ModbusObject

- static ModbusObject * sender ()

## Protected Member Functions inherited from ModbusObject

- template<class T , class ... Args>
  void emitSignal (const char *thisMethodId, ModbusMethodPointer< T, void, Args ... > thisMethod, Args ... args)

### 7.7.1 Detailed Description

The `ModbusClient` class implements the interface of the client part of the Modbus protocol.

`ModbusClient` contains a list of Modbus functions that are implemented by the Modbus client program. It implements functions for reading and writing different types of Modbus memory that are defined by the specification. The operations that return `Modbus::StatusCode` are asynchronous, that is, if the operation is not completed, it returns the intermediate status `Modbus::Status_Processing`, and then it must be called until it is successfully completed or returns an error status.

## 7.7.2 Constructor & Destructor Documentation

### 7.7.2.1 ModbusClient()

```
ModbusClient::ModbusClient (
            uint8_t unit,
            ModbusClientPort * port)
```

Class constructor.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | The address of the remote Modbus device to which this client is bound. |
| in | *port* | A pointer to the port object to which this client object belongs. |

## 7.7.3 Member Function Documentation

### 7.7.3.1 diagnostics()

```
Modbus::StatusCode ModbusClient::diagnostics (
            uint16_t subfunc,
            uint8_t insize,
            const uint8_t * indata,
            uint8_t * outsize,
            uint8_t * outdata)
```

Same as `ModbusClientPort::readInputRegisters(uint8_t unit, uint16_t offset, uint16_t count` but the `unit` address of the remote Modbus device is missing. It is preset in the constructor.

### 7.7.3.2 getCommEventCounter()

```
Modbus::StatusCode ModbusClient::getCommEventCounter (
            uint16_t * status,
            uint16_t * eventCount)
```

Same as `ModbusClientPort::getCommEventCounter(uint8_t unit, uint16_t offset, uint16_t count, uint16_t *values)`, but the `unit` address of the remote Modbus device is missing. It is preset in the constructor.

### 7.7.3.3 getCommEventLog()

```
Modbus::StatusCode ModbusClient::getCommEventLog (
            uint16_t * status,
            uint16_t * eventCount,
            uint16_t * messageCount,
            uint8_t * eventBuffSize,
            uint8_t * eventBuff)
```

Same as `ModbusClientPort::getCommEventLog(uint8_t unit, uint16_t *status, uint16_t *eventCount, uint16_t *messageCount, uint8_t *events)`, but the `unit` address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.4 isOpen()

```
bool ModbusClient::isOpen () const
```

Returns `true` if communication with the remote device is established, `false` otherwise.

#### 7.7.3.5 lastPortErrorStatus()

```
Modbus::StatusCode ModbusClient::lastPortErrorStatus () const
```

Returns the status of the last error of the performed operation.

#### 7.7.3.6 lastPortErrorText()

```
const Modbus::Char * ModbusClient::lastPortErrorText () const
```

Returns text repr of the last error of the performed operation.

#### 7.7.3.7 lastPortStatus()

```
Modbus::StatusCode ModbusClient::lastPortStatus () const
```

Returns the status of the last operation performed.

#### 7.7.3.8 maskWriteRegister()

```
Modbus::StatusCode ModbusClient::maskWriteRegister (
            uint16_t offset,
            uint16_t andMask,
            uint16_t orMask)
```

Same as `ModbusClientPort::writeMultipleRegisters(uint8_t unit, uint16_t offset, uint16_t andMask, uint16_t orMask)`, but the `unit` address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.9 port()

```
ModbusClientPort * ModbusClient::port () const
```

Returns a pointer to the port object to which this client object belongs.

#### 7.7.3.10 readCoils()

```
Modbus::StatusCode ModbusClient::readCoils (
            uint16_t offset,
            uint16_t count,
            void * values)
```

Same as `ModbusInterface::readCoils(uint8_t unit, uint16_t offset, uint16_t count, void *va` but the `unit` address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.11 readCoilsAsBoolArray()

```
Modbus::StatusCode ModbusClient::readCoilsAsBoolArray (
            uint16_t offset,
            uint16_t count,
            bool * values)
```

Same as ModbusClientPort::readCoilsAsBoolArray(uint8_t unit, uint16_t offset, uint16_t cou but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.12 readDiscreteInputs()

```
Modbus::StatusCode ModbusClient::readDiscreteInputs (
            uint16_t offset,
            uint16_t count,
            void * values)
```

Same as ModbusInterface::readDiscreteInputs(uint8_t unit, uint16_t offset, uint16_t count, but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.13 readDiscreteInputsAsBoolArray()

```
Modbus::StatusCode ModbusClient::readDiscreteInputsAsBoolArray (
            uint16_t offset,
            uint16_t count,
            bool * values)
```

Same as ModbusClientPort::readWriteMultipleRegisters(uint8_t unit, uint16_t offset, readOffset, uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t *writeValues), but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.14 readExceptionStatus()

```
Modbus::StatusCode ModbusClient::readExceptionStatus (
            uint8_t * value)
```

Same as ModbusInterface::readExceptionStatus(uint8_t unit, uint8_t *status), but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.15 readFIFOQueue()

```
Modbus::StatusCode ModbusClient::readFIFOQueue (
            uint16_t fifoadr,
            uint16_t * count,
            uint16_t * values)
```

Same as ModbusClientPort::readFIFOQueue(uint8_t unit, uint16_t fifoadr, uint16_t *count, u but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.16 readHoldingRegisters()

```
Modbus::StatusCode ModbusClient::readHoldingRegisters (
            uint16_t offset,
            uint16_t count,
            uint16_t * values)
```

Same as ModbusInterface::readHoldingRegisters(uint8_t unit, uint16_t offset, uint16_t cour but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.17 readInputRegisters()

```
Modbus::StatusCode ModbusClient::readInputRegisters (
            uint16_t offset,
            uint16_t count,
            uint16_t * values)
```

Same as ModbusInterface::readInputRegisters(uint8_t unit, uint16_t offset, uint16_t count, but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.18 readWriteMultipleRegisters()

```
Modbus::StatusCode ModbusClient::readWriteMultipleRegisters (
            uint16_t readOffset,
            uint16_t readCount,
            uint16_t * readValues,
            uint16_t writeOffset,
            uint16_t writeCount,
            const uint16_t * writeValues)
```

Same as ModbusClientPort::readWriteMultipleRegisters(uint8_t unit, uint16_↩ t offset, uint16_t count, const uint16_t *values), but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.19 reportServerID()

```
Modbus::StatusCode ModbusClient::reportServerID (
            uint8_t * count,
            uint8_t * data)
```

Same as ModbusClientPort::reportServerID(uint8_t unit, uint8_t *count, uint8_t *data), but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.20 setUnit()

```
void ModbusClient::setUnit (
            uint8_t unit)
```

Sets the address of the remote Modbus device to which this client is bound.

#### 7.7.3.21 type()

```
Modbus::ProtocolType ModbusClient::type () const
```

Returns the type of the Modbus protocol.

#### 7.7.3.22 unit()

```
uint8_t ModbusClient::unit () const
```

Returns the address of the remote Modbus device to which this client is bound.

#### 7.7.3.23 writeMultipleCoils()

```
Modbus::StatusCode ModbusClient::writeMultipleCoils (
            uint16_t offset,
            uint16_t count,
            const void * values)
```

Same as ModbusInterface::writeMultipleCoils(uint8_t unit, uint16_t offset, uint16_t count, but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.24 writeMultipleCoilsAsBoolArray()

```
Modbus::StatusCode ModbusClient::writeMultipleCoilsAsBoolArray (
            uint16_t offset,
            uint16_t count,
            const bool * values)
```

Same as ModbusClientPort::writeMultipleCoilsAsBoolArray(uint8_t unit, uint16_t offset, uir but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.25 writeMultipleRegisters()

```
Modbus::StatusCode ModbusClient::writeMultipleRegisters (
            uint16_t offset,
            uint16_t count,
            const uint16_t * values)
```

Same as ModbusInterface::writeMultipleRegisters(uint8_t unit, uint16_t offset, uint16_t co but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.26 writeSingleCoil()

```
Modbus::StatusCode ModbusClient::writeSingleCoil (
            uint16_t offset,
            bool value)
```

Same as ModbusInterface::writeSingleCoil(uint8_t unit, uint16_t offset, bool value), but the unit address of the remote Modbus device is missing. It is preset in the constructor.

#### 7.7.3.27 writeSingleRegister()

```
Modbus::StatusCode ModbusClient::writeSingleRegister (
            uint16_t offset,
            uint16_t value)
```

Same as `ModbusInterface::writeSingleRegister(uint8_t unit, uint16_t offset, uint16_t value` but the `unit` address of the remote Modbus device is missing. It is preset in the constructor.

The documentation for this class was generated from the following file:

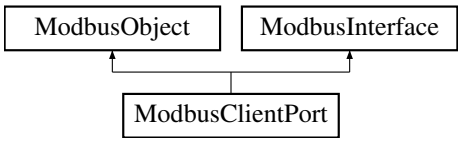
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusClient.h

## 7.8 ModbusClientPort Class Reference

The `ModbusClientPort` class implements the algorithm of the client part of the Modbus communication protocol port.

`#include <ModbusClientPort.h>`

Inheritance diagram for ModbusClientPort:




### Public Types

- enum RequestStatus { **Enable** , **Disable** , **Process** }

  *Sets the status of the request for the client.*

### Public Member Functions

- ModbusClientPort (ModbusPort *port)
- Modbus::ProtocolType type () const
- ModbusPort * port () const
- void setPort (ModbusPort *port)
- Modbus::StatusCode close ()
- bool isOpen () const
- uint32_t tries () const
- void setTries (uint32_t v)
- uint32_t repeatCount () const
- void setRepeatCount (uint32_t v)
- bool isBroadcastEnabled () const
- void setBroadcastEnabled (bool enable)
- Modbus::StatusCode readCoils (ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t count, void *values)

- Modbus::StatusCode readDiscreteInputs (ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t count, void *values)
- Modbus::StatusCode readHoldingRegisters (ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t count, uint16_t *values)
- Modbus::StatusCode readInputRegisters (ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t count, uint16_t *values)
- Modbus::StatusCode writeSingleCoil (ModbusObject *client, uint8_t unit, uint16_t offset, bool value)
- Modbus::StatusCode writeSingleRegister (ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t value)
- Modbus::StatusCode readExceptionStatus (ModbusObject *client, uint8_t unit, uint8_t *value)
- Modbus::StatusCode diagnostics (ModbusObject *client, uint8_t unit, uint16_t subfunc, uint8_t insize, const uint8_t *indata, uint8_t *outsize, uint8_t *outdata)
- Modbus::StatusCode getCommEventCounter (ModbusObject *client, uint8_t unit, uint16_t *status, uint16_t *eventCount)
- Modbus::StatusCode getCommEventLog (ModbusObject *client, uint8_t unit, uint16_t *status, uint16_t *eventCount, uint16_t *messageCount, uint8_t *eventBuffSize, uint8_t *eventBuff)
- Modbus::StatusCode writeMultipleCoils (ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t count, const void *values)
- Modbus::StatusCode writeMultipleRegisters (ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t count, const uint16_t *values)
- Modbus::StatusCode reportServerID (ModbusObject *client, uint8_t unit, uint8_t *count, uint8_t *data)
- Modbus::StatusCode maskWriteRegister (ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t andMask, uint16_t orMask)
- Modbus::StatusCode readWriteMultipleRegisters (ModbusObject *client, uint8_t unit, uint16_t readOffset, uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t *writeValues)
- Modbus::StatusCode readFIFOQueue (ModbusObject *client, uint8_t unit, uint16_t fifoadr, uint16_t *count, uint16_t *values)
- Modbus::StatusCode readCoilsAsBoolArray (ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t count, bool *values)
- Modbus::StatusCode readDiscreteInputsAsBoolArray (ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t count, bool *values)
- Modbus::StatusCode writeMultipleCoilsAsBoolArray (ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t count, const bool *values)
- Modbus::StatusCode readCoils (uint8_t unit, uint16_t offset, uint16_t count, void *values) override
- Modbus::StatusCode readDiscreteInputs (uint8_t unit, uint16_t offset, uint16_t count, void *values) override
- Modbus::StatusCode readHoldingRegisters (uint8_t unit, uint16_t offset, uint16_t count, uint16_t *values) override
- Modbus::StatusCode readInputRegisters (uint8_t unit, uint16_t offset, uint16_t count, uint16_t *values) override
- Modbus::StatusCode writeSingleCoil (uint8_t unit, uint16_t offset, bool value) override
- Modbus::StatusCode writeSingleRegister (uint8_t unit, uint16_t offset, uint16_t value) override
- Modbus::StatusCode readExceptionStatus (uint8_t unit, uint8_t *value) override
- Modbus::StatusCode diagnostics (uint8_t unit, uint16_t subfunc, uint8_t insize, const uint8_t *indata, uint8_t *outsize, uint8_t *outdata) override
- Modbus::StatusCode getCommEventCounter (uint8_t unit, uint16_t *status, uint16_t *eventCount) override
- Modbus::StatusCode getCommEventLog (uint8_t unit, uint16_t *status, uint16_t *eventCount, uint16_t *messageCount, uint8_t *eventBuffSize, uint8_t *eventBuff) override
- Modbus::StatusCode writeMultipleCoils (uint8_t unit, uint16_t offset, uint16_t count, const void *values) override
- Modbus::StatusCode writeMultipleRegisters (uint8_t unit, uint16_t offset, uint16_t count, const uint16_t *values) override
- Modbus::StatusCode reportServerID (uint8_t unit, uint8_t *count, uint8_t *data) override
- Modbus::StatusCode maskWriteRegister (uint8_t unit, uint16_t offset, uint16_t andMask, uint16_t orMask) override
- Modbus::StatusCode readWriteMultipleRegisters (uint8_t unit, uint16_t readOffset, uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t *writeValues) override

- Modbus::StatusCode readFIFOQueue (uint8_t unit, uint16_t fifoadr, uint16_t *count, uint16_t *values) override
- Modbus::StatusCode readCoilsAsBoolArray (uint8_t unit, uint16_t offset, uint16_t count, bool *values)
- Modbus::StatusCode readDiscreteInputsAsBoolArray (uint8_t unit, uint16_t offset, uint16_t count, bool *values)
- Modbus::StatusCode writeMultipleCoilsAsBoolArray (uint8_t unit, uint16_t offset, uint16_t count, const bool *values)
- Modbus::StatusCode lastStatus () const
- Modbus::Timestamp lastStatusTimestamp () const
- Modbus::StatusCode lastErrorStatus () const
- const Modbus::Char * lastErrorText () const
- uint32_t lastTries () const
- uint32_t lastRepeatCount () const
- const ModbusObject * currentClient () const
- RequestStatus getRequestStatus (ModbusObject *client)
- void cancelRequest (ModbusObject *client)
- void signalOpened (const Modbus::Char *source)
- void signalClosed (const Modbus::Char *source)
- void signalTx (const Modbus::Char *source, const uint8_t *buff, uint16_t size)
- void signalRx (const Modbus::Char *source, const uint8_t *buff, uint16_t size)
- void signalError (const Modbus::Char *source, Modbus::StatusCode status, const Modbus::Char *text)

## Public Member Functions inherited from ModbusObject

- ModbusObject ()
- virtual ~ModbusObject ()
- const Modbus::Char * objectName () const
- void setObjectName (const Modbus::Char *name)
- template<class SignalClass , class T , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class SignalClass , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- template<class ReturnType , class ... Args>
  void disconnect (ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- void disconnectFunc (void *funcPtr)
- template<class T , class ReturnType , class ... Args>
  void disconnect (T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class T >
  void disconnect (T *object)

## Public Member Functions inherited from ModbusInterface

### Friends

- class **ModbusClient**

### Additional Inherited Members

## Static Public Member Functions inherited from ModbusObject

- static ModbusObject * sender ()

## Protected Member Functions inherited from ModbusObject

- template<class T , class ... Args>
  void emitSignal (const char *thisMethodId, ModbusMethodPointer< T, void, Args ... > thisMethod, Args ... args)

### 7.8.1 Detailed Description

The `ModbusClientPort` class implements the algorithm of the client part of the Modbus communication protocol port.

`ModbusClient` contains a list of Modbus functions that are implemented by the Modbus client program. It implements functions for reading and writing various types of Modbus memory defined by the specification. In the non blocking mode if the operation is not completed it returns the intermediate status `Modbus::Status_Processing`, and then it must be called until it is successfully completed or returns an error status.

`ModbusClientPort` has number of Modbus functions with interface like `readCoils(ModbusObject *client, ...)`. Several clients can automatically share a current `ModbusClientPort` resource. The first one to access the port seizes the resource until the operation with the remote device is completed. Then the first client will release the resource and the next client in the queue will capture it, and so on in a circle.

```
#include <ModbusClient.h>
//...
void main()
{
    //...
    ModbusClientPort *port = Modbus::createClientPort(Modbus::TCP, &settings, false);
    ModbusClient c1(1, port);
    ModbusClient c2(2, port);
    ModbusClient c3(3, port);
    Modbus::StatusCode s1, s2, s3;
    //...
    while(1)
    {
        s1 = c1.readHoldingRegisters(0, 10, values);
        s2 = c2.readHoldingRegisters(0, 10, values);
        s3 = c3.readHoldingRegisters(0, 10, values);
        doSomeOtherStuffInCurrentThread();
        Modbus::msleep(1);
    }
    //...
}
//...
```

### 7.8.2 Constructor & Destructor Documentation

#### 7.8.2.1 ModbusClientPort()

```
ModbusClientPort::ModbusClientPort (
            ModbusPort * port)
```

Constructor of the class.

**Parameters**

| | | |
|---|---|---|
| in | *port* | A pointer to the port object which belongs to this client object. Lifecycle of the `port` object is managed by this `ModbusClientPort`-object |

## 7.8.3 Member Function Documentation

### 7.8.3.1 cancelRequest()

```
void ModbusClientPort::cancelRequest (
            ModbusObject * client)
```

Cancels the previous request specified by the `*rp` pointer for the client.

### 7.8.3.2 close()

```
Modbus::StatusCode ModbusClientPort::close ()
```

Closes connection and returns status of the operation.

### 7.8.3.3 currentClient()

```
const ModbusObject * ModbusClientPort::currentClient () const
```

Returns a pointer to the client object whose request is currently being processed by the current port.

### 7.8.3.4 diagnostics() [1/2]

```
Modbus::StatusCode ModbusClientPort::diagnostics (
            ModbusObject * client,
            uint8_t unit,
            uint16_t subfunc,
            uint8_t insize,
            const uint8_t * indata,
            uint8_t * outsize,
            uint8_t * outdata)
```

Same as `ModbusClientPort::readInputRegisters(uint8_t unit, uint16_t offset, uint16_t count` but has `client` as first parameter to seize current `ModbusClientPort` resource.

### 7.8.3.5 diagnostics() [2/2]

```
Modbus::StatusCode ModbusClientPort::diagnostics (
            uint8_t unit,
            uint16_t subfunc,
            uint8_t insize,
            const uint8_t * indata,
            uint8_t * outsize,
            uint8_t * outdata)  [override], [virtual]
```

Function provides a series of tests for checking the communication system between a client device and a server, or for checking various internal error conditions within a server.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *subfunc* | Address of the remote Modbus device. |
| in | *insize* | Size of the input buffer (in bytes). |
| in | *indata* | Pointer to the buffer where the input (request) data is stored. |
| out | *outsize* | Size of the buffer (in bytes) where the output data is stored. |
| out | *outdata* | Pointer to the buffer where the output data is stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented from ModbusInterface.

#### 7.8.3.6 getCommEventCounter() [1/2]

```
Modbus::StatusCode ModbusClientPort::getCommEventCounter (
            ModbusObject * client,
            uint8_t unit,
            uint16_t * status,
            uint16_t * eventCount)
```

Same as `ModbusClientPort::getCommEventCounter(uint8_t unit, uint16_t offset, uint16_t count, uint16_t *values)` but has `client` as first parameter to seize current `ModbusClientPort` resource.

#### 7.8.3.7 getCommEventCounter() [2/2]

```
Modbus::StatusCode ModbusClientPort::getCommEventCounter (
            uint8_t unit,
            uint16_t * status,
            uint16_t * eventCount)  [override], [virtual]
```

Function is used to get a status word and an event count from the remote device's communication event counter.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| out | *status* | Returned status word. |
| out | *eventCount* | Returned event counter. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented from ModbusInterface.

#### 7.8.3.8 getCommEventLog() [1/2]

```
Modbus::StatusCode ModbusClientPort::getCommEventLog (
            ModbusObject * client,
            uint8_t unit,
            uint16_t * status,
            uint16_t * eventCount,
            uint16_t * messageCount,
            uint8_t * eventBuffSize,
            uint8_t * eventBuff)
```

Same as `ModbusClientPort::getCommEventLog(uint8_t unit, uint16_t *status, uint16_t *eventCount, uint16_t *messageCount, uint8_t *events)` but has `client` as first parameter to seize current `ModbusClientPort` resource.

#### 7.8.3.9 getCommEventLog() [2/2]

```
Modbus::StatusCode ModbusClientPort::getCommEventLog (
            uint8_t unit,
            uint16_t * status,
            uint16_t * eventCount,
            uint16_t * messageCount,
            uint8_t * eventBuffSize,
            uint8_t * eventBuff)  [override], [virtual]
```

Function is used to get a status word and an event count from the remote device's communication event counter.

**Parameters**

| | | |
|---|---|---|
| `in` | *unit* | Address of the remote Modbus device. |
| `out` | *status* | Returned status word. |
| `out` | *eventCount* | Returned event counter. |
| `out` | *messageCount* | Returned message counter. |
| `out` | *eventBuffSize* | Size of the buffer where the output events (bytes) is stored. |
| `out` | *eventBuff* | Pointer to the buffer where the output events (bytes) is stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented from ModbusInterface.

#### 7.8.3.10 getRequestStatus()

```
RequestStatus ModbusClientPort::getRequestStatus (
            ModbusObject * client)
```

Returns status the current request for `client`.
The client usually calls this function to determine whether its request is pending/finished/blocked. If function returns `Enable`, `client` has just became current and can make request to the port, `Process` - current `client` is already processing, `Disable` - other client owns the port.

#### 7.8.3.11 isBroadcastEnabled()

```
bool ModbusClientPort::isBroadcastEnabled () const
```

Returns `true` if broadcast mode for `0` unit address is enabled, `false` otherwise. Broadcast mode for `0` unit address is required by Modbus protocol so it is enabled by default

#### 7.8.3.12 isOpen()

```
bool ModbusClientPort::isOpen () const
```

Returns `true` if the connection with the remote device is established, `false` otherwise.

#### 7.8.3.13 lastErrorStatus()

```
Modbus::StatusCode ModbusClientPort::lastErrorStatus () const
```

Returns the status of the last error of the performed operation.

#### 7.8.3.14 lastErrorText()

```
const Modbus::Char * ModbusClientPort::lastErrorText () const
```

Returns the text of the last error of the performed operation.

#### 7.8.3.15 lastRepeatCount()

```
uint32_t ModbusClientPort::lastRepeatCount () const  [inline]
```

Same as `lastTries()`.

#### 7.8.3.16 lastStatus()

```
Modbus::StatusCode ModbusClientPort::lastStatus () const
```

Returns the status of the last operation performed.

#### 7.8.3.17 lastStatusTimestamp()

```
Modbus::Timestamp ModbusClientPort::lastStatusTimestamp () const
```

Returns the timestamp of the last operation performed.

#### 7.8.3.18 lastTries()

```
uint32_t ModbusClientPort::lastTries () const
```

Returns statistics of the count of tries already processed.

#### 7.8.3.19 maskWriteRegister() [1/2]

```
Modbus::StatusCode ModbusClientPort::maskWriteRegister (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            uint16_t andMask,
            uint16_t orMask)
```

Same as `ModbusClientPort::writeMultipleRegisters(uint8_t unit, uint16_t offset, uint16_t andMask, uint16_t orMask)` but has `client` as first parameter to seize current `ModbusClientPort` resource.

#### 7.8.3.20 maskWriteRegister() [2/2]

```
Modbus::StatusCode ModbusClientPort::maskWriteRegister (
            uint8_t unit,
            uint16_t offset,
            uint16_t andMask,
            uint16_t orMask)  [override], [virtual]
```

Function is used to modify the contents of a specified holding register using a combination of an AND mask, an OR mask, and the register's current contents. The function's algorithm is: `Result = (Current Contents AND And_Mask) OR (Or_Mask AND (NOT And_Mask))`

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *andMask* | 16-bit unsigned integer value AND mask. |
| in | *orMask* | 16-bit unsigned integer value OR mask. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented from ModbusInterface.

#### 7.8.3.21 port()

```
ModbusPort * ModbusClientPort::port () const
```

Returns a pointer to the port object that is used by this algorithm.

#### 7.8.3.22 readCoils() [1/2]

```
Modbus::StatusCode ModbusClientPort::readCoils (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            void * values)
```

Same as ModbusClientPort::readCoils(uint8_t unit, uint16_t offset, uint16_t count, void *v but has client as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.23 readCoils() [2/2]

```
Modbus::StatusCode ModbusClientPort::readCoils (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            void * values)  [override], [virtual]
```

Function for read discrete outputs (coils, 0x bits).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of coils (bits). |
| out | *values* | Pointer to the output buffer (bit array) where the read values are stored. |

**Returns**

The result Modbus::StatusCode of the operation. Default implementation returns Status_Bad↩ IllegalFunction.

Reimplemented from ModbusInterface.

#### 7.8.3.24 readCoilsAsBoolArray() [1/2]

```
Modbus::StatusCode ModbusClientPort::readCoilsAsBoolArray (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            bool * values)
```

Same as ModbusClientPort::readCoilsAsBoolArray(uint8_t unit, uint16_t offset, uint16_t cou but has client as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.25 readCoilsAsBoolArray() [2/2]

```
Modbus::StatusCode ModbusClientPort::readCoilsAsBoolArray (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            bool * values)  [inline]
```

Same as ModbusClientPort::readCoils(uint8_t unit, uint16_t offset, uint16_t count, void *v but the output buffer of values `values` is an array, where each discrete value is located in a separate element of the array of type `bool`.

#### 7.8.3.26 readDiscreteInputs() [1/2]

```
Modbus::StatusCode ModbusClientPort::readDiscreteInputs (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            void * values)
```

Same as ModbusClientPort::readDiscreteInputs(uint8_t unit, uint16_t offset, uint16_t count but has `client` as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.27 readDiscreteInputs() [2/2]

```
Modbus::StatusCode ModbusClientPort::readDiscreteInputs (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            void * values)  [override], [virtual]
```

Function for read digital inputs (1x bits).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of inputs (bits). |
| out | *values* | Pointer to the output buffer (bit array) where the read values are stored. |

**Returns**

The result Modbus::StatusCode of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented from ModbusInterface.

#### 7.8.3.28 readDiscreteInputsAsBoolArray() [1/2]

```
Modbus::StatusCode ModbusClientPort::readDiscreteInputsAsBoolArray (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            bool * values)
```

Same as ModbusClientPort::readDiscreteInputsAsBoolArray(uint8_t unit, uint16_t offset, uir
but has client as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.29 readDiscreteInputsAsBoolArray() [2/2]

```
Modbus::StatusCode ModbusClientPort::readDiscreteInputsAsBoolArray (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            bool * values)  [inline]
```

Same as ModbusClientPort::readDiscreteInputs(uint8_t unit, uint16_t offset, uint16_t count
but the output buffer of values values is an array, where each discrete value is located in a separate element of the array of type bool.

#### 7.8.3.30 readExceptionStatus() [1/2]

```
Modbus::StatusCode ModbusClientPort::readExceptionStatus (
            ModbusObject * client,
            uint8_t unit,
            uint8_t * value)
```

Same as ModbusClientPort::readExceptionStatus(uint8_t unit, uint8_t *status)
but has client as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.31 readExceptionStatus() [2/2]

```
Modbus::StatusCode ModbusClientPort::readExceptionStatus (
            uint8_t unit,
            uint8_t * status)  [override], [virtual]
```

Function to read ExceptionStatus.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| out | *status* | Pointer to the byte (bit array) where the exception status is stored. |

**Returns**

The result Modbus::StatusCode of the operation. Default implementation returns Status_Bad↩IllegalFunction.

Reimplemented from ModbusInterface.

#### 7.8.3.32 readFIFOQueue() [1/2]

```
Modbus::StatusCode ModbusClientPort::readFIFOQueue (
            ModbusObject * client,
            uint8_t unit,
            uint16_t fifoadr,
            uint16_t * count,
            uint16_t * values)
```

Same as ModbusClientPort::readFIFOQueue(uint8_t unit, uint16_t fifoadr, uint16_t *count, u but has client as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.33 readFIFOQueue() [2/2]

```
Modbus::StatusCode ModbusClientPort::readFIFOQueue (
            uint8_t unit,
            uint16_t fifoadr,
            uint16_t * count,
            uint16_t * values)  [override], [virtual]
```

Function for read the contents of a First-In-First-Out (FIFO) queue of register in a remote device.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *fifoadr* | Address of FIFO (0-based). |
| in | *count* | Count of registers. |
| out | *values* | Pointer to the output buffer where the read values are stored. |

**Returns**

The result Modbus::StatusCode of the operation. Default implementation returns Status_Bad↩ IllegalFunction.

Reimplemented from ModbusInterface.

#### 7.8.3.34 readHoldingRegisters() [1/2]

```
Modbus::StatusCode ModbusClientPort::readHoldingRegisters (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            uint16_t * values)
```

Same as ModbusClientPort::readHoldingRegisters(uint8_t unit, uint16_t offset, uint16_t cou but has client as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.35 readHoldingRegisters() [2/2]

```
Modbus::StatusCode ModbusClientPort::readHoldingRegisters (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            uint16_t * values)  [override], [virtual]
```

Function for read holding (output) 16-bit registers (4x regs).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of registers. |
| out | *values* | Pointer to the output buffer (bit array) where the read values are stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented from ModbusInterface.

#### 7.8.3.36 readInputRegisters() [1/2]

```
Modbus::StatusCode ModbusClientPort::readInputRegisters (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            uint16_t * values)
```

Same as `ModbusClientPort::readInputRegisters(uint8_t unit, uint16_t offset, uint16_t count` but has `client` as first parameter to seize current `ModbusClientPort` resource.

#### 7.8.3.37 readInputRegisters() [2/2]

```
Modbus::StatusCode ModbusClientPort::readInputRegisters (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            uint16_t * values)  [override], [virtual]
```

Function for read input 16-bit registers (3x regs).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of registers. |
| out | *values* | Pointer to the output buffer (bit array) where the read values are stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented from ModbusInterface.

#### 7.8.3.38 readWriteMultipleRegisters() [1/2]

```
Modbus::StatusCode ModbusClientPort::readWriteMultipleRegisters (
            ModbusObject * client,
            uint8_t unit,
            uint16_t readOffset,
            uint16_t readCount,
            uint16_t * readValues,
            uint16_t writeOffset,
            uint16_t writeCount,
            const uint16_t * writeValues)
```

Same as `ModbusClientPort::readWriteMultipleRegisters(uint8_t unit, uint16_t offset, readOffset, uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t *writeValues)` but has `client` as first parameter to seize current `ModbusClientPort` resource.

#### 7.8.3.39 readWriteMultipleRegisters() [2/2]

```
Modbus::StatusCode ModbusClientPort::readWriteMultipleRegisters (
            uint8_t unit,
            uint16_t readOffset,
            uint16_t readCount,
            uint16_t * readValues,
            uint16_t writeOffset,
            uint16_t writeCount,
            const uint16_t * writeValues)  [override], [virtual]
```

This function code performs a combination of one read operation and one write operation in a single MODBUS transaction.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *readOffset* | Starting offset for read(0-based). |
| in | *readCount* | Count of registers to read. |
| out | *readValues* | Pointer to the output buffer which values must be read. |
| in | *writeOffset* | Starting offset for write(0-based). |
| in | *writeCount* | Count of registers to write. |
| in | *writeValues* | Pointer to the input buffer which values must be written. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented from ModbusInterface.

#### 7.8.3.40 repeatCount()

```
uint32_t ModbusClientPort::repeatCount () const  [inline]
```

Same as `tries()`. Used for backward compatibility.

#### 7.8.3.41 reportServerID() [1/2]

```
Modbus::StatusCode ModbusClientPort::reportServerID (
            ModbusObject * client,
            uint8_t unit,
            uint8_t * count,
            uint8_t * data)
```

Same as ModbusClientPort::reportServerID(uint8_t unit, uint8_t *count, uint8_t *data) but has client as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.42 reportServerID() [2/2]

```
Modbus::StatusCode ModbusClientPort::reportServerID (
            uint8_t unit,
            uint8_t * count,
            uint8_t * data)  [override], [virtual]
```

Function to read the description of the type, the current status, and other information specific to a remote device.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *count* | Count of bytes returned. |
| in | *data* | Pointer to the output buffer where the read data are stored. |

**Returns**

The result Modbus::StatusCode of the operation. Default implementation returns Status_Bad↩ IllegalFunction.

Reimplemented from ModbusInterface.

#### 7.8.3.43 setBroadcastEnabled()

```
void ModbusClientPort::setBroadcastEnabled (
            bool enable)
```

Enables broadcast mode for 0 unit address. It is enabled by default.

**See also**

isBroadcastEnabled()

#### 7.8.3.44 setPort()

```
void ModbusClientPort::setPort (
            ModbusPort * port)
```

Set new port object for current client port control. Previous port object is deleted.

#### 7.8.3.45 setRepeatCount()

```
void ModbusClientPort::setRepeatCount (
            uint32_t v)  [inline]
```

Same as `setTries()`. Used for backward compatibility.

#### 7.8.3.46 setTries()

```
void ModbusClientPort::setTries (
            uint32_t v)
```

Sets the number of tries a Modbus request is repeated if it fails.

#### 7.8.3.47 signalClosed()

```
void ModbusClientPort::signalClosed (
            const Modbus::Char * source)
```

Calls each callback of the port when the port is closed. `source` - current port's name

#### 7.8.3.48 signalError()

```
void ModbusClientPort::signalError (
            const Modbus::Char * source,
            Modbus::StatusCode status,
            const Modbus::Char * text)
```

Calls each callback of the port when error is occured with error's status and text.

#### 7.8.3.49 signalOpened()

```
void ModbusClientPort::signalOpened (
            const Modbus::Char * source)
```

Calls each callback of the port when the port is opened. `source` - current port's name

#### 7.8.3.50 signalRx()

```
void ModbusClientPort::signalRx (
            const Modbus::Char * source,
            const uint8_t * buff,
            uint16_t size)
```

Calls each callback of the incoming packet 'Rx' from the internal list of callbacks, passing them the input array 'buff' and its size 'size'.

#### 7.8.3.51 signalTx()

```
void ModbusClientPort::signalTx (
            const Modbus::Char * source,
            const uint8_t * buff,
            uint16_t size)
```

Calls each callback of the original packet 'Tx' from the internal list of callbacks, passing them the original array 'buff' and its size 'size'.

#### 7.8.3.52 tries()

```
uint32_t ModbusClientPort::tries () const
```

Returns the setting of the number of tries of the Modbus request if it fails.

#### 7.8.3.53 type()

```
Modbus::ProtocolType ModbusClientPort::type () const
```

Returns type of Modbus protocol.

#### 7.8.3.54 writeMultipleCoils() [1/2]

```
Modbus::StatusCode ModbusClientPort::writeMultipleCoils (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            const void * values)
```

Same as `ModbusClientPort::writeMultipleCoils(uint8_t unit, uint16_t offset, uint16_t count` but has `client` as first parameter to seize current `ModbusClientPort` resource.

#### 7.8.3.55 writeMultipleCoils() [2/2]

```
Modbus::StatusCode ModbusClientPort::writeMultipleCoils (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            const void * values)  [override], [virtual]
```

Function is used to modify the contents of a specified holding register using a combination of an AND mask, an OR mask, and the register's current contents.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of coils. |
| in | *values* | Pointer to the input buffer (bit array) which values must be written. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented from ModbusInterface.

#### 7.8.3.56 writeMultipleCoilsAsBoolArray() [1/2]

```
Modbus::StatusCode ModbusClientPort::writeMultipleCoilsAsBoolArray (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            const bool * values)
```

Same as ModbusClientPort::writeMultipleCoilsAsBoolArray(uint8_t unit, uint16_t offset, uir but has client as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.57 writeMultipleCoilsAsBoolArray() [2/2]

```
Modbus::StatusCode ModbusClientPort::writeMultipleCoilsAsBoolArray (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            const bool * values)  [inline]
```

Same as ModbusClientPort::writeMultipleCoils(uint8_t unit, uint16_t offset, uint16_t count but the input buffer of values values is an array, where each discrete value is located in a separate element of the array of type bool.

#### 7.8.3.58 writeMultipleRegisters() [1/2]

```
Modbus::StatusCode ModbusClientPort::writeMultipleRegisters (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            const uint16_t * values)
```

Same as ModbusClientPort::writeMultipleRegisters(uint8_t unit, uint16_t offset, uint16_t c but has client as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.59 writeMultipleRegisters() [2/2]

```
Modbus::StatusCode ModbusClientPort::writeMultipleRegisters (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            const uint16_t * values)  [override], [virtual]
```

Function for write holding (output) 16-bit registers (4x regs).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of registers. |
| in | *values* | Pointer to the input buffer which values must be written. |

**Returns**

The result Modbus::StatusCode of the operation. Default implementation returns Status_Bad↩ IllegalFunction.

Reimplemented from ModbusInterface.

#### 7.8.3.60 writeSingleCoil() [1/2]

```
Modbus::StatusCode ModbusClientPort::writeSingleCoil (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            bool value)
```

Same as ModbusClientPort::writeSingleCoil(uint8_t unit, uint16_t offset, bool value) but has client as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.61 writeSingleCoil() [2/2]

```
Modbus::StatusCode ModbusClientPort::writeSingleCoil (
            uint8_t unit,
            uint16_t offset,
            bool value)  [override], [virtual]
```

Function for write one separate discrete output (0x coil).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *value* | Boolean value to be set. |

**Returns**

The result Modbus::StatusCode of the operation. Default implementation returns Status_Bad↩ IllegalFunction.

Reimplemented from ModbusInterface.

#### 7.8.3.62 writeSingleRegister() [1/2]

```
Modbus::StatusCode ModbusClientPort::writeSingleRegister (
            ModbusObject * client,
            uint8_t unit,
            uint16_t offset,
            uint16_t value)
```

Same as ModbusClientPort::writeSingleRegister(uint8_t unit, uint16_t offset, uint16_t valu but has client as first parameter to seize current ModbusClientPort resource.

#### 7.8.3.63 writeSingleRegister() [2/2]

```
Modbus::StatusCode ModbusClientPort::writeSingleRegister (
            uint8_t unit,
            uint16_t offset,
            uint16_t value)  [override], [virtual]
```

Function for write one separate 16-bit holding register (4x).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *value* | 16-bit unsigned integer value to be set. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩ IllegalFunction`.

Reimplemented from ModbusInterface.

The documentation for this class was generated from the following file:

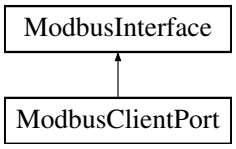
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusClientPort.h

## 7.9 ModbusInterface Class Reference

Main interface of Modbus communication protocol.

`#include <Modbus.h>`

Inheritance diagram for ModbusInterface:




### Public Member Functions

- virtual Modbus::StatusCode readCoils (uint8_t unit, uint16_t offset, uint16_t count, void ∗values)
- virtual Modbus::StatusCode readDiscreteInputs (uint8_t unit, uint16_t offset, uint16_t count, void ∗values)
- virtual Modbus::StatusCode readHoldingRegisters (uint8_t unit, uint16_t offset, uint16_t count, uint16_↩ t ∗values)
- virtual Modbus::StatusCode readInputRegisters (uint8_t unit, uint16_t offset, uint16_t count, uint16_t ∗values)
- virtual Modbus::StatusCode writeSingleCoil (uint8_t unit, uint16_t offset, bool value)
- virtual Modbus::StatusCode writeSingleRegister (uint8_t unit, uint16_t offset, uint16_t value)
- virtual Modbus::StatusCode readExceptionStatus (uint8_t unit, uint8_t ∗status)
- virtual Modbus::StatusCode diagnostics (uint8_t unit, uint16_t subfunc, uint8_t insize, const uint8_t ∗indata, uint8_t ∗outsize, uint8_t ∗outdata)
- virtual Modbus::StatusCode getCommEventCounter (uint8_t unit, uint16_t ∗status, uint16_t ∗eventCount)
- virtual Modbus::StatusCode getCommEventLog (uint8_t unit, uint16_t ∗status, uint16_t ∗eventCount, uint16_t ∗messageCount, uint8_t ∗eventBuffSize, uint8_t ∗eventBuff)
- virtual Modbus::StatusCode writeMultipleCoils (uint8_t unit, uint16_t offset, uint16_t count, const void ∗values)
- virtual Modbus::StatusCode writeMultipleRegisters (uint8_t unit, uint16_t offset, uint16_t count, const uint16_t ∗values)
- virtual Modbus::StatusCode reportServerID (uint8_t unit, uint8_t ∗count, uint8_t ∗data)
- virtual Modbus::StatusCode maskWriteRegister (uint8_t unit, uint16_t offset, uint16_t andMask, uint16_t or↩ Mask)
- virtual Modbus::StatusCode readWriteMultipleRegisters (uint8_t unit, uint16_t readOffset, uint16_t read↩ Count, uint16_t ∗readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t ∗writeValues)
- virtual Modbus::StatusCode readFIFOQueue (uint8_t unit, uint16_t fifoadr, uint16_t ∗count, uint16_t ∗values)

### 7.9.1 Detailed Description

Main interface of Modbus communication protocol.

ModbusInterface constains list of functions that ModbusLib is supported. There are such functions as: 1 (0x01) - READ_COILS 2 (0x02) - READ_DISCRETE_INPUTS 3 (0x03) - READ_HOLDING_REGISTERS 4 (0x04) - READ_INPUT_REGISTERS 5 (0x05) - WRITE_SINGLE_COIL 6 (0x06) - WRITE_SINGLE_REGISTER 7 (0x07) - READ_EXCEPTION_STATUS 8 (0x08) - DIAGNOSTICS 11 (0x0B) - GET_COMM_EVENT_COUNTER 12 (0x0C) - GET_COMM_EVENT_LOG 15 (0x0F) - WRITE_MULTIPLE_COILS 16 (0x10) - WRITE_MULTIPLE_REGISTERS 17 (0x11) - REPORT_SERVER_ID 22 (0x16) - MASK_WRITE_REGISTER 23 (0x17) - READ_WRITE_MULTIPLE_REGISTERS 24 (0x18) - READ_FIFO_QUEUE

Default implementation of every Modbus function returns Modbus::Status_BadIllegalFunction.

### 7.9.2 Member Function Documentation

#### 7.9.2.1 diagnostics()

```
virtual Modbus::StatusCode ModbusInterface::diagnostics (
            uint8_t unit,
            uint16_t subfunc,
            uint8_t insize,
            const uint8_t * indata,
            uint8_t * outsize,
            uint8_t * outdata)  [virtual]
```

Function provides a series of tests for checking the communication system between a client device and a server, or for checking various internal error conditions within a server.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *subfunc* | Address of the remote Modbus device. |
| in | *insize* | Size of the input buffer (in bytes). |
| in | *indata* | Pointer to the buffer where the input (request) data is stored. |
| out | *outsize* | Size of the buffer (in bytes) where the output data is stored. |
| out | *outdata* | Pointer to the buffer where the output data is stored. |

**Returns**

The result Modbus::StatusCode of the operation. Default implementation returns Status_BadIllegalFunction.

Reimplemented in ModbusClientPort.

#### 7.9.2.2 getCommEventCounter()

```
virtual Modbus::StatusCode ModbusInterface::getCommEventCounter (
            uint8_t unit,
            uint16_t * status,
            uint16_t * eventCount)  [virtual]
```

Function is used to get a status word and an event count from the remote device's communication event counter.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| out | *status* | Returned status word. |
| out | *eventCount* | Returned event counter. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.3 getCommEventLog()

```
virtual Modbus::StatusCode ModbusInterface::getCommEventLog (
            uint8_t unit,
            uint16_t * status,
            uint16_t * eventCount,
            uint16_t * messageCount,
            uint8_t * eventBuffSize,
            uint8_t * eventBuff)  [virtual]
```

Function is used to get a status word and an event count from the remote device's communication event counter.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| out | *status* | Returned status word. |
| out | *eventCount* | Returned event counter. |
| out | *messageCount* | Returned message counter. |
| out | *eventBuffSize* | Size of the buffer where the output events (bytes) is stored. |
| out | *eventBuff* | Pointer to the buffer where the output events (bytes) is stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.4 maskWriteRegister()

```
virtual Modbus::StatusCode ModbusInterface::maskWriteRegister (
            uint8_t unit,
            uint16_t offset,
            uint16_t andMask,
            uint16_t orMask)  [virtual]
```

Function is used to modify the contents of a specified holding register using a combination of an AND mask, an OR mask, and the register's current contents. The function's algorithm is: `Result = (Current Contents AND And_Mask) OR (Or_Mask AND (NOT And_Mask))`

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *andMask* | 16-bit unsigned integer value AND mask. |
| in | *orMask* | 16-bit unsigned integer value OR mask. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.5 readCoils()

```
virtual Modbus::StatusCode ModbusInterface::readCoils (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            void * values)  [virtual]
```

Function for read discrete outputs (coils, 0x bits).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of coils (bits). |
| out | *values* | Pointer to the output buffer (bit array) where the read values are stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.6 readDiscreteInputs()

```
virtual Modbus::StatusCode ModbusInterface::readDiscreteInputs (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            void * values)  [virtual]
```

Function for read digital inputs (1x bits).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of inputs (bits). |
| out | *values* | Pointer to the output buffer (bit array) where the read values are stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩ IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.7 readExceptionStatus()

```
virtual Modbus::StatusCode ModbusInterface::readExceptionStatus (
            uint8_t unit,
            uint8_t * status)  [virtual]
```

Function to read ExceptionStatus.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| out | *status* | Pointer to the byte (bit array) where the exception status is stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩ IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.8 readFIFOQueue()

```
virtual Modbus::StatusCode ModbusInterface::readFIFOQueue (
            uint8_t unit,
            uint16_t fifoadr,
            uint16_t * count,
            uint16_t * values)  [virtual]
```

Function for read the contents of a First-In-First-Out (FIFO) queue of register in a remote device.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *fifoadr* | Address of FIFO (0-based). |
| in | *count* | Count of registers. |
| out | *values* | Pointer to the output buffer where the read values are stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩ IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.9 readHoldingRegisters()

```
virtual Modbus::StatusCode ModbusInterface::readHoldingRegisters (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            uint16_t * values)  [virtual]
```

Function for read holding (output) 16-bit registers (4x regs).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of registers. |
| out | *values* | Pointer to the output buffer (bit array) where the read values are stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.10 readInputRegisters()

```
virtual Modbus::StatusCode ModbusInterface::readInputRegisters (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            uint16_t * values)  [virtual]
```

Function for read input 16-bit registers (3x regs).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of registers. |
| out | *values* | Pointer to the output buffer (bit array) where the read values are stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.11 readWriteMultipleRegisters()

```
virtual Modbus::StatusCode ModbusInterface::readWriteMultipleRegisters (
            uint8_t unit,
            uint16_t readOffset,
            uint16_t readCount,
            uint16_t * readValues,
            uint16_t writeOffset,
            uint16_t writeCount,
            const uint16_t * writeValues)  [virtual]
```

This function code performs a combination of one read operation and one write operation in a single MODBUS transaction.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *readOffset* | Starting offset for read(0-based). |
| in | *readCount* | Count of registers to read. |
| out | *readValues* | Pointer to the output buffer which values must be read. |
| in | *writeOffset* | Starting offset for write(0-based). |
| in | *writeCount* | Count of registers to write. |
| in | *writeValues* | Pointer to the input buffer which values must be written. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.12 reportServerID()

```
virtual Modbus::StatusCode ModbusInterface::reportServerID (
            uint8_t unit,
            uint8_t * count,
            uint8_t * data)  [virtual]
```

Function to read the description of the type, the current status, and other information specific to a remote device.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *count* | Count of bytes returned. |
| in | *data* | Pointer to the output buffer where the read data are stored. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.13 writeMultipleCoils()

```
virtual Modbus::StatusCode ModbusInterface::writeMultipleCoils (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            const void * values)  [virtual]
```

Function is used to modify the contents of a specified holding register using a combination of an AND mask, an OR mask, and the register's current contents.

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of coils. |
| in | *values* | Pointer to the input buffer (bit array) which values must be written. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.14 writeMultipleRegisters()

```
virtual Modbus::StatusCode ModbusInterface::writeMultipleRegisters (
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            const uint16_t * values)  [virtual]
```

Function for write holding (output) 16-bit registers (4x regs).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *count* | Count of registers. |
| in | *values* | Pointer to the input buffer which values must be written. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.15 writeSingleCoil()

```
virtual Modbus::StatusCode ModbusInterface::writeSingleCoil (
            uint8_t unit,
            uint16_t offset,
            bool value)  [virtual]
```

Function for write one separate discrete output (0x coil).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *value* | Boolean value to be set. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩ IllegalFunction`.

Reimplemented in ModbusClientPort.

#### 7.9.2.16 writeSingleRegister()

```
virtual Modbus::StatusCode ModbusInterface::writeSingleRegister (
            uint8_t unit,
            uint16_t offset,
            uint16_t value)  [virtual]
```

Function for write one separate 16-bit holding register (4x).

**Parameters**

| | | |
|---|---|---|
| in | *unit* | Address of the remote Modbus device. |
| in | *offset* | Starting offset (0-based). |
| in | *value* | 16-bit unsigned integer value to be set. |

**Returns**

The result `Modbus::StatusCode` of the operation. Default implementation returns `Status_Bad↩ IllegalFunction`.

Reimplemented in ModbusClientPort.

The documentation for this class was generated from the following file:

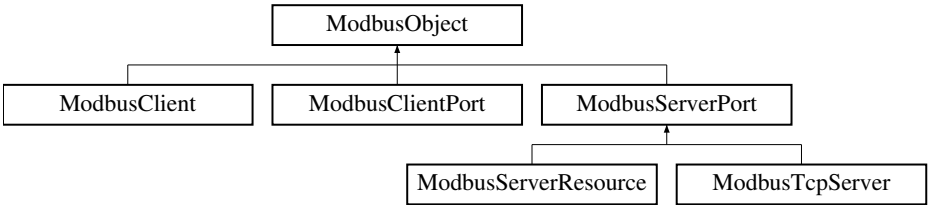
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/Modbus.h

## 7.10 ModbusObject Class Reference

The `ModbusObject` class is the base class for objects that use signal/slot mechanism.

`#include <ModbusObject.h>`

Inheritance diagram for ModbusObject:



### Public Member Functions

- ModbusObject ()
- virtual ~ModbusObject ()
- const Modbus::Char * objectName () const
- void setObjectName (const Modbus::Char *name)
- template<class SignalClass , class T , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class SignalClass , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- template<class ReturnType , class ... Args>
  void disconnect (ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- void disconnectFunc (void *funcPtr)
- template<class T , class ReturnType , class ... Args>
  void disconnect (T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class T >
  void disconnect (T *object)

### Static Public Member Functions

- static ModbusObject * sender ()

### Protected Member Functions

- template<class T , class ... Args>
  void emitSignal (const char *thisMethodId, ModbusMethodPointer< T, void, Args ... > thisMethod, Args ... args)

## 7.10.1 Detailed Description

The `ModbusObject` class is the base class for objects that use signal/slot mechanism.

`ModbusObject` is designed to be a base class for objects that need to use simplified Qt-like signal/slot mechanism. User can connect signal of the object he want to listen to his own function or method of his own class and then it can be disconnected if he is not interesting of this signal anymore. Callbacks will be called in order which it were connected.

`ModbusObject` has a map which key means signal identifier (pointer to signal) and value is a list of callbacks functions/methods connected to this signal.

`ModbusObject` has `objectName()` and `setObjectName` methods. This methods can be used to simply identify object which is signal's source (e.g. to print info in console).

**Note**

`ModbusObject` class is not thread safe

## 7.10.2 Constructor & Destructor Documentation

### 7.10.2.1 ModbusObject()

```
ModbusObject::ModbusObject ()
```

Constructor of the class.

### 7.10.2.2 ~ModbusObject()

```
virtual ModbusObject::~ModbusObject ()  [virtual]
```

Virtual destructor of the class.

## 7.10.3 Member Function Documentation

### 7.10.3.1 connect() [1/2]

```
template<class SignalClass , class ReturnType , class ...  Args>
void ModbusObject::connect (
            ModbusMethodPointer< SignalClass, ReturnType, Args ...  > signalMethodPtr,
            ModbusFunctionPointer< ReturnType, Args ...  > funcPtr)  [inline]
```

Same as `ModbusObject::connect(ModbusMethodPointer, T*, ModbusMethodPointer)` but connects `ModbusFunctionPointer` to current object's signal `signalMethodPtr`.

### 7.10.3.2 connect() [2/2]

```
template<class SignalClass , class T , class ReturnType , class ...  Args>
void ModbusObject::connect (
            ModbusMethodPointer< SignalClass, ReturnType, Args ...  > signalMethodPtr,
            T * object,
            ModbusMethodPointer< T, ReturnType, Args ...  > objectMethodPtr)  [inline]
```

Connect `this` object's signal `signalMethodPtr` to the `objects` method `objectMethodPtr`.

```
class MyClass : public ModbusObject { public: void signalSomething(int a, int b) {
     emitSignal(&MyClass::signalSomething, a, b); } };
class MyReceiver { public: void slotSomething(int a, int b) { doSomething(); } };
MyClass c;
MyReceiver r;
c.connect(&MyClass::signalSomething, r, &MyReceiver::slotSomething);
```

**Note**

`SignalClass` template type refers to any class but it must be `this` or derived class. It makes separate `SignalClass` to easely refers signal of the derived class.

### 7.10.3.3 disconnect() [1/3]

```
template<class ReturnType , class ...  Args>
void ModbusObject::disconnect (
            ModbusFunctionPointer< ReturnType, Args ...  > funcPtr)  [inline]
```

Disconnects function `funcPtr` from all signals of current object.

#### 7.10.3.4 disconnect() [2/3]

```
template<class T >
void ModbusObject::disconnect (
            T * object)  [inline]
```

Disconnect all slots of `T *object` from all signals of current object.

#### 7.10.3.5 disconnect() [3/3]

```
template<class T , class ReturnType , class ...  Args>
void ModbusObject::disconnect (
            T * object,
            ModbusMethodPointer< T, ReturnType, Args ...  > objectMethodPtr)  [inline]
```

Disconnects slot represented by pair `(object, objectMethodPtr)` from all signals of current object.

#### 7.10.3.6 disconnectFunc()

```
void ModbusObject::disconnectFunc (
            void * funcPtr)  [inline]
```

Disconnects function `funcPtr` from all signals of current object, but `funcPtr` is a void pointer.

#### 7.10.3.7 emitSignal()

```
template<class T , class ...  Args>
void ModbusObject::emitSignal (
            const char * thisMethodId,
            ModbusMethodPointer< T, void, Args ...  > thisMethod,
            Args ...  args)  [inline], [protected]
```

Template method for emit signal. Must be called from within of the signal method.

#### 7.10.3.8 objectName()

```
const Modbus::Char * ModbusObject::objectName () const
```

Returns a pointer to current object's name string.

#### 7.10.3.9 sender()

```
static ModbusObject * ModbusObject::sender ()  [static]
```

Returns a pointer to the object that sent the signal. This pointer is valid in thread where signal was occured only. So this function must be called only within the slot that is a callback of signal occured.

#### 7.10.3.10 setObjectName()

```
void ModbusObject::setObjectName (
            const Modbus::Char * name)
```

Set name of current object.

The documentation for this class was generated from the following file:

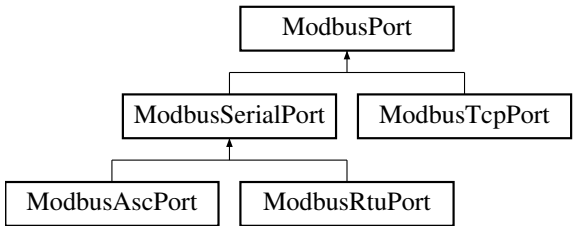
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusObject.h

## 7.11 ModbusPort Class Reference

The abstract class `ModbusPort` is the base class for a specific implementation of the Modbus communication protocol.

`#include <ModbusPort.h>`

Inheritance diagram for ModbusPort:




### Public Member Functions

- virtual ~ModbusPort ()
- virtual Modbus::ProtocolType type () const =0
- virtual Modbus::Handle handle () const =0
- virtual Modbus::StatusCode open ()=0
- virtual Modbus::StatusCode close ()=0
- virtual bool isOpen () const =0
- virtual void setNextRequestRepeated (bool v)
- bool isChanged () const
- bool isServerMode () const
- virtual void setServerMode (bool mode)
- bool isBlocking () const
- bool isNonBlocking () const
- uint32_t timeout () const
- void setTimeout (uint32_t timeout)
- Modbus::StatusCode lastErrorStatus () const
- const Modbus::Char * lastErrorText () const
- virtual Modbus::StatusCode writeBuffer (uint8_t unit, uint8_t func, uint8_t *buff, uint16_t szInBuff)=0
- virtual Modbus::StatusCode readBuffer (uint8_t &unit, uint8_t &func, uint8_t *buff, uint16_t maxSzBuff, uint16_t *szOutBuff)=0
- virtual Modbus::StatusCode write ()=0
- virtual Modbus::StatusCode read ()=0
- virtual const uint8_t * readBufferData () const =0
- virtual uint16_t readBufferSize () const =0
- virtual const uint8_t * writeBufferData () const =0
- virtual uint16_t writeBufferSize () const =0

**Protected Member Functions**

- Modbus::StatusCode setError (Modbus::StatusCode status, const Modbus::Char *text)

## 7.11.1 Detailed Description

The abstract class ModbusPort is the base class for a specific implementation of the Modbus communication protocol.

ModbusPort contains general functions for working with a specific port, implementing a specific version of the Modbus communication protocol. For example, versions for working with a TCP port or a serial port.

## 7.11.2 Constructor & Destructor Documentation

### 7.11.2.1 ~ModbusPort()

```
virtual ModbusPort::~ModbusPort ()  [virtual]
```

Virtual destructor.

## 7.11.3 Member Function Documentation

### 7.11.3.1 close()

```
virtual Modbus::StatusCode ModbusPort::close ()  [pure virtual]
```

Closes the port (breaks the connection) and returns the status the result status.

Implemented in ModbusSerialPort, and ModbusTcpPort.

### 7.11.3.2 handle()

```
virtual Modbus::Handle ModbusPort::handle () const  [pure virtual]
```

Returns the native handle value that depenp on OS used. For TCP it socket handle, for serial port - file handle.

Implemented in ModbusSerialPort, and ModbusTcpPort.

### 7.11.3.3 isBlocking()

```
bool ModbusPort::isBlocking () const
```

Returns `true` if the port works in synch (blocking) mode, `false` otherwise.

#### 7.11.3.4 isChanged()

```
bool ModbusPort::isChanged () const
```

Returns `true` if the port settings have been changed and the port needs to be reopened/reestablished communication with the remote device, `false` otherwise.

#### 7.11.3.5 isNonBlocking()

```
bool ModbusPort::isNonBlocking () const
```

Returns `true` if the port works in asynch (nonblocking) mode, `false` otherwise.

#### 7.11.3.6 isOpen()

```
virtual bool ModbusPort::isOpen () const  [pure virtual]
```

Returns `true` if the port is open/communication with the remote device is established, `false` otherwise.

Implemented in ModbusSerialPort, and ModbusTcpPort.

#### 7.11.3.7 isServerMode()

```
bool ModbusPort::isServerMode () const
```

Returns `true` if the port works in server mode, `false` otherwise.

#### 7.11.3.8 lastErrorStatus()

```
Modbus::StatusCode ModbusPort::lastErrorStatus () const
```

Returns the status of the last error of the performed operation.

#### 7.11.3.9 lastErrorText()

```
const Modbus::Char * ModbusPort::lastErrorText () const
```

Returns the pointer to `const Char` text buffer of the last error of the performed operation.

#### 7.11.3.10 open()

```
virtual Modbus::StatusCode ModbusPort::open ()  [pure virtual]
```

Opens port (create connection) for further operations and returns the result status.

Implemented in ModbusSerialPort, and ModbusTcpPort.

#### 7.11.3.11 read()

```
virtual Modbus::StatusCode ModbusPort::read ()  [pure virtual]
```

Implements the algorithm for reading from the port and returns the status of the operation.

Implemented in ModbusSerialPort, and ModbusTcpPort.

#### 7.11.3.12 readBuffer()

```
virtual Modbus::StatusCode ModbusPort::readBuffer (
            uint8_t & unit,
            uint8_t & func,
            uint8_t * buff,
            uint16_t maxSzBuff,
            uint16_t * szOutBuff)  [pure virtual]
```

The function parses the packet that the `read()` function puts into the buffer, checks it for correctness, extracts its parameters, and returns the status of the operation.

Implemented in ModbusAscPort, ModbusRtuPort, and ModbusTcpPort.

#### 7.11.3.13 readBufferData()

```
virtual const uint8_t * ModbusPort::readBufferData () const  [pure virtual]
```

Returns pointer to data of read buffer.

Implemented in ModbusSerialPort, and ModbusTcpPort.

#### 7.11.3.14 readBufferSize()

```
virtual uint16_t ModbusPort::readBufferSize () const  [pure virtual]
```

Returns size of data of read buffer.

Implemented in ModbusSerialPort, and ModbusTcpPort.

#### 7.11.3.15 setError()

```
Modbus::StatusCode ModbusPort::setError (
            Modbus::StatusCode status,
            const Modbus::Char * text)  [protected]
```

Sets the error parameters of the last operation performed.

#### 7.11.3.16 setNextRequestRepeated()

```
virtual void ModbusPort::setNextRequestRepeated (
            bool v)  [virtual]
```

For the TCP version of the Modbus protocol. The identifier of each subsequent parcel is automatically increased by 1. If you set `setNextRequestRepeated(true)` then the next ID will not be increased by 1 but for only one next parcel.

Reimplemented in ModbusTcpPort.

#### 7.11.3.17 setServerMode()

```
virtual void ModbusPort::setServerMode (
            bool mode)  [virtual]
```

Sets server mode if `true`, `false` for client mode.

#### 7.11.3.18 setTimeout()

```
void ModbusPort::setTimeout (
            uint32_t timeout)
```

Sets the setting for the connection timeout of the remote device.

#### 7.11.3.19 timeout()

```
uint32_t ModbusPort::timeout () const
```

Returns the setting for the connection timeout of the remote device.

#### 7.11.3.20 type()

```
virtual Modbus::ProtocolType ModbusPort::type () const  [pure virtual]
```

Returns the Modbus protocol type.

Implemented in ModbusAscPort, ModbusRtuPort, and ModbusTcpPort.

#### 7.11.3.21 write()

```
virtual Modbus::StatusCode ModbusPort::write ()  [pure virtual]
```

Implements the algorithm for writing to the port and returns the status of the operation.

Implemented in ModbusSerialPort, and ModbusTcpPort.

#### 7.11.3.22 writeBuffer()

```
virtual Modbus::StatusCode ModbusPort::writeBuffer (
            uint8_t unit,
            uint8_t func,
            uint8_t * buff,
            uint16_t szInBuff)  [pure virtual]
```

The function directly generates a packet and places it in the buffer for further sending. Returns the status of the operation.

Implemented in ModbusAscPort, ModbusRtuPort, and ModbusTcpPort.

#### 7.11.3.23 writeBufferData()

```
virtual const uint8_t * ModbusPort::writeBufferData () const  [pure virtual]
```

Returns pointer to data of write buffer.

Implemented in ModbusSerialPort, and ModbusTcpPort.

#### 7.11.3.24 writeBufferSize()

```
virtual uint16_t ModbusPort::writeBufferSize () const  [pure virtual]
```

Returns size of data of write buffer.

Implemented in ModbusSerialPort, and ModbusTcpPort.

The documentation for this class was generated from the following file:

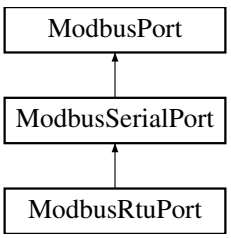
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusPort.h

## 7.12 ModbusRtuPort Class Reference

Implements RTU version of the Modbus communication protocol.

```
#include <ModbusRtuPort.h>
```

Inheritance diagram for ModbusRtuPort:



### Public Member Functions

- ModbusRtuPort (bool blocking=false)
- ~ModbusRtuPort ()
- Modbus::ProtocolType type () const override

## Public Member Functions inherited from ModbusSerialPort

- ~ModbusSerialPort ()
- Modbus::Handle handle () const override
- Modbus::StatusCode open () override
- Modbus::StatusCode close () override
- bool isOpen () const override
- const Modbus::Char * portName () const
- void setPortName (const Modbus::Char *portName)
- int32_t baudRate () const
- void setBaudRate (int32_t baudRate)
- int8_t dataBits () const
- void setDataBits (int8_t dataBits)
- Modbus::Parity parity () const
- void setParity (Modbus::Parity parity)
- Modbus::StopBits stopBits () const
- void setStopBits (Modbus::StopBits stopBits)
- Modbus::FlowControl flowControl () const
- void setFlowControl (Modbus::FlowControl flowControl)
- uint32_t timeoutFirstByte () const
- void setTimeoutFirstByte (uint32_t timeout)
- uint32_t timeoutInterByte () const
- void setTimeoutInterByte (uint32_t timeout)
- const uint8_t * readBufferData () const override
- uint16_t readBufferSize () const override
- const uint8_t * writeBufferData () const override
- uint16_t writeBufferSize () const override

## Public Member Functions inherited from ModbusPort

- virtual ~ModbusPort ()
- virtual void setNextRequestRepeated (bool v)
- bool isChanged () const
- bool isServerMode () const
- virtual void setServerMode (bool mode)
- bool isBlocking () const
- bool isNonBlocking () const
- uint32_t timeout () const
- void setTimeout (uint32_t timeout)
- Modbus::StatusCode lastErrorStatus () const
- const Modbus::Char * lastErrorText () const

### Protected Member Functions

- Modbus::StatusCode writeBuffer (uint8_t unit, uint8_t func, uint8_t *buff, uint16_t szInBuff) override
- Modbus::StatusCode readBuffer (uint8_t &unit, uint8_t &func, uint8_t *buff, uint16_t maxSzBuff, uint16_t *szOutBuff) override

### Protected Member Functions inherited from ModbusSerialPort

- Modbus::StatusCode write () override
- Modbus::StatusCode read () override

### Protected Member Functions inherited from ModbusPort

- Modbus::StatusCode setError (Modbus::StatusCode status, const Modbus::Char *text)

## 7.12.1 Detailed Description

Implements RTU version of the Modbus communication protocol.

`ModbusRtuPort` derived from `ModbusSerialPort` and implements `writeBuffer` and `readBuffer` for RTU version of Modbus communication protocol.

## 7.12.2 Constructor & Destructor Documentation

### 7.12.2.1 ModbusRtuPort()

```
ModbusRtuPort::ModbusRtuPort (
            bool blocking = false)
```

Constructor of the class. if `blocking = true` then defines blocking mode, non blocking otherwise.

### 7.12.2.2 ~ModbusRtuPort()

```
ModbusRtuPort::~ModbusRtuPort ()
```

Destructor of the class.

## 7.12.3 Member Function Documentation

### 7.12.3.1 readBuffer()

```
Modbus::StatusCode ModbusRtuPort::readBuffer (
            uint8_t & unit,
            uint8_t & func,
            uint8_t * buff,
            uint16_t maxSzBuff,
            uint16_t * szOutBuff)  [override], [protected], [virtual]
```

The function parses the packet that the `read()` function puts into the buffer, checks it for correctness, extracts its parameters, and returns the status of the operation.

Implements ModbusPort.

#### 7.12.3.2 type()

```
Modbus::ProtocolType ModbusRtuPort::type () const  [inline], [override], [virtual]
```

Returns the Modbus protocol type. For `ModbusAscPort` returns `Modbus::RTU`.

Implements ModbusPort.

#### 7.12.3.3 writeBuffer()

```
Modbus::StatusCode ModbusRtuPort::writeBuffer (
            uint8_t unit,
            uint8_t func,
            uint8_t * buff,
            uint16_t szInBuff)  [override], [protected], [virtual]
```

The function directly generates a packet and places it in the buffer for further sending. Returns the status of the operation.

Implements ModbusPort.

The documentation for this class was generated from the following file:

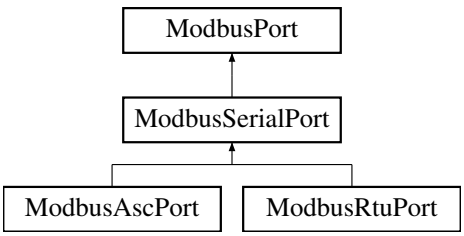
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusRtuPort.h

## 7.13 ModbusSerialPort Class Reference

The abstract class `ModbusSerialPort` is the base class serial port Modbus communications.

```
#include <ModbusSerialPort.h>
```

Inheritance diagram for ModbusSerialPort:




### Classes

- struct Defaults

  *Holds the default values of the settings.*

### Public Member Functions

- ~ModbusSerialPort ()
- Modbus::Handle handle () const override
- Modbus::StatusCode open () override
- Modbus::StatusCode close () override
- bool isOpen () const override
- const Modbus::Char * portName () const
- void setPortName (const Modbus::Char *portName)
- int32_t baudRate () const
- void setBaudRate (int32_t baudRate)
- int8_t dataBits () const
- void setDataBits (int8_t dataBits)
- Modbus::Parity parity () const
- void setParity (Modbus::Parity parity)
- Modbus::StopBits stopBits () const
- void setStopBits (Modbus::StopBits stopBits)
- Modbus::FlowControl flowControl () const
- void setFlowControl (Modbus::FlowControl flowControl)
- uint32_t timeoutFirstByte () const
- void setTimeoutFirstByte (uint32_t timeout)
- uint32_t timeoutInterByte () const
- void setTimeoutInterByte (uint32_t timeout)
- const uint8_t * readBufferData () const override
- uint16_t readBufferSize () const override
- const uint8_t * writeBufferData () const override
- uint16_t writeBufferSize () const override

## Public Member Functions inherited from ModbusPort

- virtual ~ModbusPort ()
- virtual Modbus::ProtocolType type () const =0
- virtual void setNextRequestRepeated (bool v)
- bool isChanged () const
- bool isServerMode () const
- virtual void setServerMode (bool mode)
- bool isBlocking () const
- bool isNonBlocking () const
- uint32_t timeout () const
- void setTimeout (uint32_t timeout)
- Modbus::StatusCode lastErrorStatus () const
- const Modbus::Char * lastErrorText () const
- virtual Modbus::StatusCode writeBuffer (uint8_t unit, uint8_t func, uint8_t *buff, uint16_t szInBuff)=0
- virtual Modbus::StatusCode readBuffer (uint8_t &unit, uint8_t &func, uint8_t *buff, uint16_t maxSzBuff, uint16_t *szOutBuff)=0

### Protected Member Functions

- Modbus::StatusCode write () override
- Modbus::StatusCode read () override

### Protected Member Functions inherited from ModbusPort

- Modbus::StatusCode setError (Modbus::StatusCode status, const Modbus::Char *text)

## 7.13.1 Detailed Description

The abstract class `ModbusSerialPort` is the base class serial port Modbus communications.

The abstract class `ModbusSerialPort` is the base class for a specific implementation of the Modbus communication protocol that using Serial Port. It implements functions which are common for the serial port: `open`, `close`, `read` and `write`.

## 7.13.2 Constructor & Destructor Documentation

### 7.13.2.1 ~ModbusSerialPort()

```
ModbusSerialPort::~ModbusSerialPort ()
```

Virtual destructor. Closes serial port before destruction.

## 7.13.3 Member Function Documentation

### 7.13.3.1 baudRate()

```
int32_t ModbusSerialPort::baudRate () const
```

Returns current serial port baud rate, e.g. 1200, 2400, 9600, 115200 etc.

### 7.13.3.2 close()

```
Modbus::StatusCode ModbusSerialPort::close ()  [override], [virtual]
```

Close serial port and returns `Modbus::Status_Good`.

Implements ModbusPort.

### 7.13.3.3 dataBits()

```
int8_t ModbusSerialPort::dataBits () const
```

Returns current serial port data bits, e.g. 5, 6, 7 or 8.

### 7.13.3.4 flowControl()

```
Modbus::FlowControl ModbusSerialPort::flowControl () const
```

Returns current serial port `Modbus::FlowControl` enum value.

#### 7.13.3.5 handle()

```
Modbus::Handle ModbusSerialPort::handle () const  [override], [virtual]
```

Returns native OS serial port handle, e.g. `HANDLE` value for Windows.

Implements ModbusPort.

#### 7.13.3.6 isOpen()

```
bool ModbusSerialPort::isOpen () const  [override], [virtual]
```

Returns `true` if the serial port is open, `false` otherwise.

Implements ModbusPort.

#### 7.13.3.7 open()

```
Modbus::StatusCode ModbusSerialPort::open ()  [override], [virtual]
```

Try to open serial port and returns `Modbus::Status_Good` if success or `Modbus::Status_BadSerialOpen` otherwise.

Implements ModbusPort.

#### 7.13.3.8 parity()

```
Modbus::Parity ModbusSerialPort::parity () const
```

Returns current serial port `Modbus::Parity` enum value.

#### 7.13.3.9 portName()

```
const Modbus::Char * ModbusSerialPort::portName () const
```

Returns current serial port name, e.g. `COM1` for Windows or `/dev/ttyS0` for Unix.

#### 7.13.3.10 read()

```
Modbus::StatusCode ModbusSerialPort::read ()  [override], [protected], [virtual]
```

Implements the algorithm for reading from the port and returns the status of the operation.

Implements ModbusPort.

#### 7.13.3.11 readBufferData()

```
const uint8_t * ModbusSerialPort::readBufferData () const  [override], [virtual]
```

Returns pointer to data of read buffer.

Implements ModbusPort.

#### 7.13.3.12 readBufferSize()

```
uint16_t ModbusSerialPort::readBufferSize () const  [override], [virtual]
```

Returns size of data of read buffer.

Implements ModbusPort.

#### 7.13.3.13 setBaudRate()

```
void ModbusSerialPort::setBaudRate (
            int32_t baudRate)
```

Set current serial port baud rate.

#### 7.13.3.14 setDataBits()

```
void ModbusSerialPort::setDataBits (
            int8_t dataBits)
```

Set current serial port baud data bits.

#### 7.13.3.15 setFlowControl()

```
void ModbusSerialPort::setFlowControl (
            Modbus::FlowControl flowControl)
```

Set current serial port `Modbus::FlowControl` enum value.

#### 7.13.3.16 setParity()

```
void ModbusSerialPort::setParity (
            Modbus::Parity parity)
```

Set current serial port `Modbus::Parity` enum value.

#### 7.13.3.17 setPortName()

```
void ModbusSerialPort::setPortName (
            const Modbus::Char * portName)
```

Set current serial port name.

#### 7.13.3.18 setStopBits()

```
void ModbusSerialPort::setStopBits (
            Modbus::StopBits stopBits)
```

Set current serial port Modbus::StopBits enum value.

#### 7.13.3.19 setTimeoutFirstByte()

```
void ModbusSerialPort::setTimeoutFirstByte (
            uint32_t timeout)  [inline]
```

Set current serial port timeout of waiting first byte of incomming packet (in milliseconds).

#### 7.13.3.20 setTimeoutInterByte()

```
void ModbusSerialPort::setTimeoutInterByte (
            uint32_t timeout)
```

Set current serial port timeout of waiting next byte (inter byte waiting tgimeout) of incomming packet (in milliseconds).

#### 7.13.3.21 stopBits()

```
Modbus::StopBits ModbusSerialPort::stopBits () const
```

Returns current serial port Modbus::StopBits enum value.

#### 7.13.3.22 timeoutFirstByte()

```
uint32_t ModbusSerialPort::timeoutFirstByte () const  [inline]
```

Returns current serial port timeout of waiting first byte of incomming packet (in milliseconds).

#### 7.13.3.23 timeoutInterByte()

```
uint32_t ModbusSerialPort::timeoutInterByte () const
```

Returns current serial port timeout of waiting next byte (inter byte waiting tgimeout) of incomming packet (in milliseconds).

#### 7.13.3.24 write()

```
Modbus::StatusCode ModbusSerialPort::write ()  [override], [protected], [virtual]
```

Implements the algorithm for writing to the port and returns the status of the operation.

Implements ModbusPort.

#### 7.13.3.25 writeBufferData()

```
const uint8_t * ModbusSerialPort::writeBufferData () const  [override], [virtual]
```

Returns pointer to data of write buffer.

Implements ModbusPort.

#### 7.13.3.26 writeBufferSize()

```
uint16_t ModbusSerialPort::writeBufferSize () const  [override], [virtual]
```

Returns size of data of write buffer.

Implements ModbusPort.

The documentation for this class was generated from the following file:

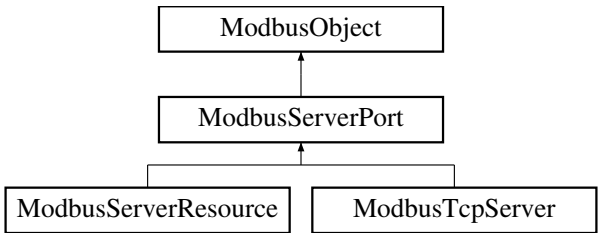
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusSerialPort.h

## 7.14 ModbusServerPort Class Reference

Abstract base class for direct control of `ModbusPort` derived classes (TCP or serial) for server side.

`#include <ModbusServerPort.h>`

Inheritance diagram for ModbusServerPort:




### Public Member Functions

- ModbusInterface * device () const
- void setDevice (ModbusInterface *device)
- virtual Modbus::ProtocolType type () const =0
- virtual bool isTcpServer () const
- virtual Modbus::StatusCode open ()=0
- virtual Modbus::StatusCode close ()=0
- virtual bool isOpen () const =0
- bool isBroadcastEnabled () const
- virtual void setBroadcastEnabled (bool enable)
- const void * unitMap () const
- virtual void setUnitMap (const void *unitmap)
- void * context () const
- void setContext (void *context)
- virtual Modbus::StatusCode process ()=0
- bool isStateClosed () const
- void signalOpened (const Modbus::Char *source)
- void signalClosed (const Modbus::Char *source)
- void signalTx (const Modbus::Char *source, const uint8_t *buff, uint16_t size)
- void signalRx (const Modbus::Char *source, const uint8_t *buff, uint16_t size)
- void signalError (const Modbus::Char *source, Modbus::StatusCode status, const Modbus::Char *text)

## Public Member Functions inherited from ModbusObject

- ModbusObject ()
- virtual ~ModbusObject ()
- const Modbus::Char * objectName () const
- void setObjectName (const Modbus::Char *name)
- template<class SignalClass , class T , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class SignalClass , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- template<class ReturnType , class ... Args>
  void disconnect (ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- void disconnectFunc (void *funcPtr)
- template<class T , class ReturnType , class ... Args>
  void disconnect (T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class T >
  void disconnect (T *object)

### Protected Member Functions

- ModbusObject ()

## Protected Member Functions inherited from ModbusObject

- template<class T , class ... Args>
  void emitSignal (const char *thisMethodId, ModbusMethodPointer< T, void, Args ... > thisMethod, Args ... args)

### Additional Inherited Members

## Static Public Member Functions inherited from ModbusObject

- static ModbusObject * sender ()

### 7.14.1 Detailed Description

Abstract base class for direct control of `ModbusPort` derived classes (TCP or serial) for server side.

Pointer to `ModbusPort` object must be passed to `ModbusServerPort` derived class constructor.

Also assumed that `ModbusServerPort` derived classes must accept `ModbusInterface` object in its constructor to process every Modbus function request.

### 7.14.2 Member Function Documentation

#### 7.14.2.1 close()

```
virtual Modbus::StatusCode ModbusServerPort::close ()  [pure virtual]
```

Closes port/connection and returns status of the operation.

Implemented in ModbusServerResource, and ModbusTcpServer.

#### 7.14.2.2 context()

```
void * ModbusServerPort::context () const
```

Return context of the port previously set by `setContext` function or `nullptr` by default.

#### 7.14.2.3 device()

```
ModbusInterface * ModbusServerPort::device () const
```

Returns pointer to `ModbusInterface` object/device that was previously passed in constructor. This device must process every input Modbus function request for this server port.

#### 7.14.2.4 isBroadcastEnabled()

```
bool ModbusServerPort::isBroadcastEnabled () const
```

Returns `true` if broadcast mode for `0` unit address is enabled, `false` otherwise. Broadcast mode for `0` unit address is required by Modbus protocol so it is enabled by default

#### 7.14.2.5 isOpen()

```
virtual bool ModbusServerPort::isOpen () const  [pure virtual]
```

Returns `true` if inner port is open, `false` otherwise.

Implemented in ModbusServerResource, and ModbusTcpServer.

#### 7.14.2.6 isStateClosed()

```
bool ModbusServerPort::isStateClosed () const
```

Returns `true` if current port has closed inner state, `false` otherwise.

#### 7.14.2.7 isTcpServer()

```
virtual bool ModbusServerPort::isTcpServer () const  [virtual]
```

Returns `true` if current server port is TCP server, `false` otherwise.

Reimplemented in ModbusTcpServer.

#### 7.14.2.8 ModbusObject()

```
ModbusObject::ModbusObject ()  [protected]
```

Constructor of the class.

#### 7.14.2.9 open()

```
virtual Modbus::StatusCode ModbusServerPort::open ()  [pure virtual]
```

Open inner port/connection to begin working and returns status of the operation. User do not need to call this method directly.

Implemented in ModbusServerResource, and ModbusTcpServer.

#### 7.14.2.10 process()

```
virtual Modbus::StatusCode ModbusServerPort::process ()  [pure virtual]
```

Main function of the class. Must be called in the cycle. Return statuc code is not very useful but can indicate that inner server operations are good, bad or in process.

Implemented in ModbusServerResource, and ModbusTcpServer.

#### 7.14.2.11 setBroadcastEnabled()

```
virtual void ModbusServerPort::setBroadcastEnabled (
            bool enable)  [virtual]
```

Enables broadcast mode for 0 unit address. It is enabled by default.

**See also**

isBroadcastEnabled()

Reimplemented in ModbusTcpServer.

#### 7.14.2.12 setContext()

```
void ModbusServerPort::setContext (
            void * context)
```

Set context of the port.

#### 7.14.2.13 setDevice()

```
void ModbusServerPort::setDevice (
            ModbusInterface * device)
```

Set pointer to ModbusInterface object/device to transfer all request ot it. This device must process every input Modbus function request for this server port.

#### 7.14.2.14 setUnitMap()

```
virtual void ModbusServerPort::setUnitMap (
            const void * unitmap)  [virtual]
```

Set units map of current server. Server make a copy of units map data.

**See also**

`unitMap()`

Reimplemented in ModbusTcpServer.

#### 7.14.2.15 signalClosed()

```
void ModbusServerPort::signalClosed (
            const Modbus::Char * source)
```

Signal occured when inner port was closed. `source` - current port name.

#### 7.14.2.16 signalError()

```
void ModbusServerPort::signalError (
            const Modbus::Char * source,
            Modbus::StatusCode status,
            const Modbus::Char * text)
```

Signal occured when error is occured with error's `status` and `text`. `source` - current port name.

#### 7.14.2.17 signalOpened()

```
void ModbusServerPort::signalOpened (
            const Modbus::Char * source)
```

Signal occured when inner port was opened. `source` - current port name.

#### 7.14.2.18 signalRx()

```
void ModbusServerPort::signalRx (
            const Modbus::Char * source,
            const uint8_t * buff,
            uint16_t size)
```

Signal occured when the incoming packet 'Rx' from the internal list of callbacks, passing them the input array 'buff' and its size 'size'. `source` - current port name.

#### 7.14.2.19 signalTx()

```
void ModbusServerPort::signalTx (
            const Modbus::Char * source,
            const uint8_t * buff,
            uint16_t size)
```

Signal occured when the original packet 'Tx' from the internal list of callbacks, passing them the original array 'buff' and its size 'size'. `source` - current port name.

#### 7.14.2.20 type()

```
virtual Modbus::ProtocolType ModbusServerPort::type () const  [pure virtual]
```

Returns type of Modbus protocol.

Implemented in ModbusServerResource, and ModbusTcpServer.

#### 7.14.2.21 unitMap()

```
const void * ModbusServerPort::unitMap () const
```

Return pointer to the units map byte array of the current server. By default unit map is not set so return value is `nullptr`. Unit map is data type with size of 32 bytes in which every bit represents unit address from `0` to `255`. So bit 0 of byte 0 represents unit address `0`, bit 1 of byte 0 represents unit address `1` and so on. Bit 0 of byte 1 represnt unit address `8`, bit 7 of byte 31 represents unit address `255`. If set unit map can enable or disable (depends on respecting 1/0 bit value) unit address for further processing. It is not set by default and function returns `nullptr`.

The documentation for this class was generated from the following file:

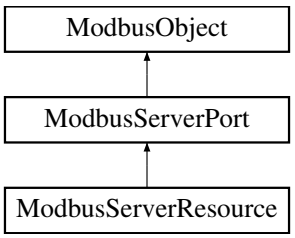
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusServerPort.h

## 7.15 ModbusServerResource Class Reference

Implements direct control for `ModbusPort` derived classes (TCP or serial) for server side.

`#include <ModbusServerResource.h>`

Inheritance diagram for ModbusServerResource:



### Public Member Functions

- ModbusServerResource (ModbusPort *port, ModbusInterface *device)
- ModbusPort * port () const
- Modbus::ProtocolType type () const override
- Modbus::StatusCode open () override
- Modbus::StatusCode close () override
- bool isOpen () const override
- Modbus::StatusCode process () override

## Public Member Functions inherited from ModbusServerPort

- ModbusInterface * device () const
- void setDevice (ModbusInterface *device)
- virtual bool isTcpServer () const
- bool isBroadcastEnabled () const
- virtual void setBroadcastEnabled (bool enable)
- const void * unitMap () const
- virtual void setUnitMap (const void *unitmap)
- void * context () const
- void setContext (void *context)
- bool isStateClosed () const
- void signalOpened (const Modbus::Char *source)
- void signalClosed (const Modbus::Char *source)
- void signalTx (const Modbus::Char *source, const uint8_t *buff, uint16_t size)
- void signalRx (const Modbus::Char *source, const uint8_t *buff, uint16_t size)
- void signalError (const Modbus::Char *source, Modbus::StatusCode status, const Modbus::Char *text)

## Public Member Functions inherited from ModbusObject

- ModbusObject ()
- virtual ~ModbusObject ()
- const Modbus::Char * objectName () const
- void setObjectName (const Modbus::Char *name)
- template<class SignalClass , class T , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class SignalClass , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- template<class ReturnType , class ... Args>
  void disconnect (ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- void disconnectFunc (void *funcPtr)
- template<class T , class ReturnType , class ... Args>
  void disconnect (T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class T >
  void disconnect (T *object)

### Protected Member Functions

- virtual Modbus::StatusCode processInputData (const uint8_t *buff, uint16_t sz)
- virtual Modbus::StatusCode processDevice ()
- virtual Modbus::StatusCode processOutputData (uint8_t *buff, uint16_t &sz)

## Protected Member Functions inherited from ModbusServerPort

- ModbusObject ()

## Protected Member Functions inherited from ModbusObject

- template<class T , class ... Args>
  void emitSignal (const char *thisMethodId, ModbusMethodPointer< T, void, Args ... > thisMethod, Args ... args)

### Additional Inherited Members

## Static Public Member Functions inherited from ModbusObject

- static ModbusObject * sender ()

### 7.15.1 Detailed Description

Implements direct control for ModbusPort derived classes (TCP or serial) for server side.

ModbusServerResource derived from ModbusServerPort and makes ModbusPort object behaves like server port. Pointer to ModbusPort object is passed to ModbusServerResource constructor.

Also ModbusServerResource have ModbusInterface object as second parameter of constructor which process every Modbus function request.

### 7.15.2 Constructor & Destructor Documentation

#### 7.15.2.1 ModbusServerResource()

```
ModbusServerResource::ModbusServerResource (
            ModbusPort * port,
            ModbusInterface * device)
```

Constructor of the class.

**Parameters**

| | | |
|---|---|---|
| in | *port* | Pointer to the ModbusPort which is managed by the current class object. |
| in | *device* | Pointer to the ModbusInterface implementation to which all requests for Modbus functions are forwarded. |

### 7.15.3 Member Function Documentation

#### 7.15.3.1 close()

```
Modbus::StatusCode ModbusServerResource::close ()  [override], [virtual]
```

Closes port/connection and returns status of the operation.

Implements ModbusServerPort.

#### 7.15.3.2 isOpen()

```
bool ModbusServerResource::isOpen () const  [override], [virtual]
```

Returns `true` if inner port is open, `false` otherwise.

Implements ModbusServerPort.

#### 7.15.3.3 open()

```
Modbus::StatusCode ModbusServerResource::open ()  [override], [virtual]
```

Open inner port/connection to begin working and returns status of the operation. User do not need to call this method directly.

Implements ModbusServerPort.

#### 7.15.3.4 port()

```
ModbusPort * ModbusServerResource::port () const
```

Returns pointer to inner port which was previously passed in constructor.

#### 7.15.3.5 process()

```
Modbus::StatusCode ModbusServerResource::process ()  [override], [virtual]
```

Main function of the class. Must be called in the cycle. Return statuc code is not very useful but can indicate that inner server operations are good, bad or in process.

Implements ModbusServerPort.

#### 7.15.3.6 processDevice()

```
virtual Modbus::StatusCode ModbusServerResource::processDevice ()  [protected], [virtual]
```

Transfer input request Modbus function to inner device and returns status of the operation.

#### 7.15.3.7 processInputData()

```
virtual Modbus::StatusCode ModbusServerResource::processInputData (
            const uint8_t * buff,
            uint16_t sz)  [protected], [virtual]
```

Process input data `buff` with `size` and returns status of the operation.

#### 7.15.3.8 processOutputData()

```
virtual Modbus::StatusCode ModbusServerResource::processOutputData (
            uint8_t * buff,
            uint16_t & sz)  [protected], [virtual]
```

Process output data `buff` with `size` and returns status of the operation.

#### 7.15.3.9 type()

```
Modbus::ProtocolType ModbusServerResource::type () const  [override], [virtual]
```

Returns type of Modbus protocol. Same as `port()->type()`.

Implements ModbusServerPort.

The documentation for this class was generated from the following file:

- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusServerResource.h

## 7.16 ModbusSlotBase< ReturnType, Args > Class Template Reference

`ModbusSlotBase` base template for slot (method or function)

`#include <ModbusObject.h>`

### Public Member Functions

- virtual ~ModbusSlotBase ()
- virtual void * object () const
- virtual void * methodOrFunction () const =0
- virtual ReturnType exec (Args ... args)=0

### 7.16.1 Detailed Description

**template<class ReturnType, class ... Args>**
**class ModbusSlotBase< ReturnType, Args >**

`ModbusSlotBase` base template for slot (method or function)

### 7.16.2 Constructor & Destructor Documentation

#### 7.16.2.1 ~ModbusSlotBase()

```
template<class ReturnType , class ...  Args>
virtual ModbusSlotBase< ReturnType, Args >::~ModbusSlotBase ()  [inline], [virtual]
```

Virtual destructor of the class

### 7.16.3 Member Function Documentation

#### 7.16.3.1 exec()

```
template<class ReturnType , class ...  Args>
virtual ReturnType ModbusSlotBase< ReturnType, Args >::exec (
            Args ...  args)  [pure virtual]
```

Execute method or function slot

Implemented in ModbusSlotFunction< ReturnType, Args >, and ModbusSlotMethod< T, ReturnType, Args >.

#### 7.16.3.2 methodOrFunction()

```
template<class ReturnType , class ...  Args>
virtual void * ModbusSlotBase< ReturnType, Args >::methodOrFunction () const  [pure virtual]
```

Return pointer to method (in case of method slot) or function (in case of function slot)

Implemented in ModbusSlotFunction< ReturnType, Args >, and ModbusSlotMethod< T, ReturnType, Args >.

#### 7.16.3.3 object()

```
template<class ReturnType , class ...  Args>
virtual void * ModbusSlotBase< ReturnType, Args >::object () const  [inline], [virtual]
```

Return pointer to object which method belongs to (in case of method slot) or `nullptr` in case of function slot

Reimplemented in ModbusSlotMethod< T, ReturnType, Args >.

The documentation for this class was generated from the following file:

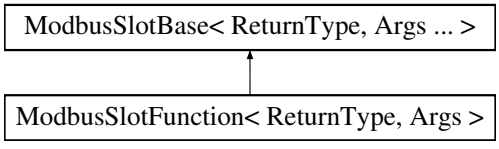
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusObject.h

## 7.17 ModbusSlotFunction< ReturnType, Args > Class Template Reference

`ModbusSlotFunction` template class hold pointer to slot function

```
#include <ModbusObject.h>
```

Inheritance diagram for ModbusSlotFunction< ReturnType, Args >:



**Public Member Functions**

- ModbusSlotFunction (ModbusFunctionPointer< ReturnType, Args... > funcPtr)
- void * methodOrFunction () const override
- ReturnType exec (Args ... args) override

## Public Member Functions inherited from ModbusSlotBase< ReturnType, Args ... >

- virtual ~ModbusSlotBase ()
- virtual void * object () const

### 7.17.1 Detailed Description

**template<class ReturnType, class ... Args>**
**class ModbusSlotFunction< ReturnType, Args >**

`ModbusSlotFunction` template class hold pointer to slot function

### 7.17.2 Constructor & Destructor Documentation

#### 7.17.2.1 ModbusSlotFunction()

```
template<class ReturnType , class ...  Args>
ModbusSlotFunction< ReturnType, Args >::ModbusSlotFunction (
            ModbusFunctionPointer< ReturnType, Args...  > funcPtr)  [inline]
```

Constructor of the slot.

**Parameters**

| | | |
|---|---|---|
| in | *funcPtr* | Pointer to slot function. |

### 7.17.3 Member Function Documentation

#### 7.17.3.1 exec()

```
template<class ReturnType , class ...  Args>
ReturnType ModbusSlotFunction< ReturnType, Args >::exec (
            Args ...  args)  [inline], [override], [virtual]
```

Execute method or function slot

Implements ModbusSlotBase< ReturnType, Args ... >.

#### 7.17.3.2 methodOrFunction()

```
template<class ReturnType , class ...  Args>
void * ModbusSlotFunction< ReturnType, Args >::methodOrFunction () const  [inline], [override],
[virtual]
```

Return pointer to method (in case of method slot) or function (in case of function slot)

Implements ModbusSlotBase< ReturnType, Args ... >.

The documentation for this class was generated from the following file:

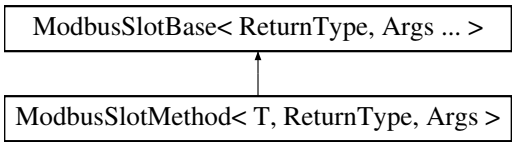
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusObject.h

## 7.18 ModbusSlotMethod< T, ReturnType, Args > Class Template Reference

ModbusSlotMethod template class hold pointer to object and its method

`#include <ModbusObject.h>`

Inheritance diagram for ModbusSlotMethod< T, ReturnType, Args >:




### Public Member Functions

- ModbusSlotMethod (T *object, ModbusMethodPointer< T, ReturnType, Args... > methodPtr)
- void * object () const override
- void * methodOrFunction () const override
- ReturnType exec (Args ... args) override

### Public Member Functions inherited from ModbusSlotBase< ReturnType, Args ... >

- virtual ~ModbusSlotBase ()

### 7.18.1 Detailed Description

**template<class T, class ReturnType, class ... Args>**
**class ModbusSlotMethod< T, ReturnType, Args >**

ModbusSlotMethod template class hold pointer to object and its method

### 7.18.2 Constructor & Destructor Documentation

#### 7.18.2.1 ModbusSlotMethod()

```
template<class T , class ReturnType , class ...  Args>
ModbusSlotMethod< T, ReturnType, Args >::ModbusSlotMethod (
            T * object,
            ModbusMethodPointer< T, ReturnType, Args...  > methodPtr)  [inline]
```

Constructor of the slot.

**Parameters**

| | | |
|---|---|---|
| in | *object* | Pointer to object. |
| in | *methodPtr* | Pointer to object's method. |

### 7.18.3 Member Function Documentation

#### 7.18.3.1 exec()

```
template<class T , class ReturnType , class ... Args>
ReturnType ModbusSlotMethod< T, ReturnType, Args >::exec (
            Args ... args) [inline], [override], [virtual]
```

Execute method or function slot

Implements ModbusSlotBase< ReturnType, Args ... >.

#### 7.18.3.2 methodOrFunction()

```
template<class T , class ReturnType , class ... Args>
void * ModbusSlotMethod< T, ReturnType, Args >::methodOrFunction () const [inline], [override],
[virtual]
```

Return pointer to method (in case of method slot) or function (in case of function slot)

Implements ModbusSlotBase< ReturnType, Args ... >.

#### 7.18.3.3 object()

```
template<class T , class ReturnType , class ... Args>
void * ModbusSlotMethod< T, ReturnType, Args >::object () const [inline], [override], [virtual]
```

Return pointer to object which method belongs to (in case of method slot) or `nullptr` in case of function slot

Reimplemented from ModbusSlotBase< ReturnType, Args ... >.

The documentation for this class was generated from the following file:

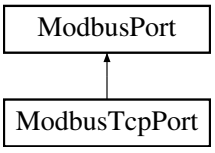
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusObject.h

## 7.19 ModbusTcpPort Class Reference

Class `ModbusTcpPort` implements TCP version of Modbus protocol.

`#include <ModbusTcpPort.h>`

Inheritance diagram for ModbusTcpPort:



### Classes

- struct Defaults

  *Defaults class constain default settings values for ModbusTcpPort.*

### Public Member Functions

- ModbusTcpPort (ModbusTcpSocket *socket, bool blocking=false)
- ModbusTcpPort (bool blocking=false)
- ~ModbusTcpPort ()
- Modbus::ProtocolType type () const override
- Modbus::Handle handle () const override
- Modbus::StatusCode open () override
- Modbus::StatusCode close () override
- bool isOpen () const override
- const Modbus::Char * host () const
- void setHost (const Modbus::Char *host)
- uint16_t port () const
- void setPort (uint16_t port)
- void setNextRequestRepeated (bool v) override
- bool autoIncrement () const
- const uint8_t * readBufferData () const override
- uint16_t readBufferSize () const override
- const uint8_t * writeBufferData () const override
- uint16_t writeBufferSize () const override

## Public Member Functions inherited from ModbusPort

- virtual ~ModbusPort ()
- bool isChanged () const
- bool isServerMode () const
- virtual void setServerMode (bool mode)
- bool isBlocking () const
- bool isNonBlocking () const
- uint32_t timeout () const
- void setTimeout (uint32_t timeout)
- Modbus::StatusCode lastErrorStatus () const
- const Modbus::Char * lastErrorText () const

### Protected Member Functions

- Modbus::StatusCode write () override
- Modbus::StatusCode read () override
- Modbus::StatusCode writeBuffer (uint8_t unit, uint8_t func, uint8_t *buff, uint16_t szInBuff) override
- Modbus::StatusCode readBuffer (uint8_t &unit, uint8_t &func, uint8_t *buff, uint16_t maxSzBuff, uint16_t *szOutBuff) override

## Protected Member Functions inherited from ModbusPort

- Modbus::StatusCode setError (Modbus::StatusCode status, const Modbus::Char *text)

### 7.19.1 Detailed Description

Class ModbusTcpPort implements TCP version of Modbus protocol.

ModbusPort contains function to work with TCP-port (connection).

### 7.19.2 Constructor & Destructor Documentation

#### 7.19.2.1 ModbusTcpPort() [1/2]

```
ModbusTcpPort::ModbusTcpPort (
            ModbusTcpSocket * socket,
            bool blocking = false)
```

Constructor of the class.

#### 7.19.2.2 ModbusTcpPort() [2/2]

```
ModbusTcpPort::ModbusTcpPort (
            bool blocking = false)
```

Constructor of the class.

#### 7.19.2.3 ~ModbusTcpPort()

```
ModbusTcpPort::~ModbusTcpPort ()
```

Destructor of the class. Close socket if it was not closed previously

### 7.19.3 Member Function Documentation

#### 7.19.3.1 autoIncrement()

```
bool ModbusTcpPort::autoIncrement () const
```

Returns 'true' if the identifier of each subsequent parcel is automatically incremented by 1, 'false' otherwise.

#### 7.19.3.2 close()

```
Modbus::StatusCode ModbusTcpPort::close ()  [override], [virtual]
```

Closes the port (breaks the connection) and returns the status the result status.

Implements ModbusPort.

#### 7.19.3.3 handle()

```
Modbus::Handle ModbusTcpPort::handle () const  [override], [virtual]
```

Native OS handle for the socket.

Implements ModbusPort.

#### 7.19.3.4 host()

```
const Modbus::Char * ModbusTcpPort::host () const
```

Returns the settings for the IP address or DNS name of the remote device.

#### 7.19.3.5 isOpen()

```
bool ModbusTcpPort::isOpen () const  [override], [virtual]
```

Returns `true` if the port is open/communication with the remote device is established, `false` otherwise.

Implements ModbusPort.

#### 7.19.3.6 open()

```
Modbus::StatusCode ModbusTcpPort::open ()  [override], [virtual]
```

Opens port (create connection) for further operations and returns the result status.

Implements ModbusPort.

#### 7.19.3.7 port()

```
uint16_t ModbusTcpPort::port () const
```

Returns the setting for the TCP port number of the remote device.

#### 7.19.3.8 read()

```
Modbus::StatusCode ModbusTcpPort::read ()  [override], [protected], [virtual]
```

Implements the algorithm for reading from the port and returns the status of the operation.

Implements ModbusPort.

#### 7.19.3.9 readBuffer()

```
Modbus::StatusCode ModbusTcpPort::readBuffer (
            uint8_t & unit,
            uint8_t & func,
            uint8_t * buff,
            uint16_t maxSzBuff,
            uint16_t * szOutBuff)  [override], [protected], [virtual]
```

The function parses the packet that the `read()` function puts into the buffer, checks it for correctness, extracts its parameters, and returns the status of the operation.

Implements ModbusPort.

#### 7.19.3.10 readBufferData()

```
const uint8_t * ModbusTcpPort::readBufferData () const  [override], [virtual]
```

Returns pointer to data of read buffer.

Implements ModbusPort.

#### 7.19.3.11 readBufferSize()

```
uint16_t ModbusTcpPort::readBufferSize () const  [override], [virtual]
```

Returns size of data of read buffer.

Implements ModbusPort.

#### 7.19.3.12 setHost()

```
void ModbusTcpPort::setHost (
            const Modbus::Char * host)
```

Sets the settings for the IP address or DNS name of the remote device.

#### 7.19.3.13 setNextRequestRepeated()

```
void ModbusTcpPort::setNextRequestRepeated (
            bool v)  [override], [virtual]
```

Repeat next request parameters (for Modbus TCP transaction Id).

Reimplemented from ModbusPort.

#### 7.19.3.14 setPort()

```
void ModbusTcpPort::setPort (
            uint16_t port)
```

Sets the settings for the TCP port number of the remote device.

#### 7.19.3.15 type()

```
Modbus::ProtocolType ModbusTcpPort::type () const  [inline], [override], [virtual]
```

Returns the Modbus protocol type. In this case it is `Modbus::TCP`.

Implements ModbusPort.

#### 7.19.3.16 write()

```
Modbus::StatusCode ModbusTcpPort::write ()  [override], [protected], [virtual]
```

Implements the algorithm for writing to the port and returns the status of the operation.

Implements ModbusPort.

#### 7.19.3.17 writeBuffer()

```
Modbus::StatusCode ModbusTcpPort::writeBuffer (
            uint8_t unit,
            uint8_t func,
            uint8_t * buff,
            uint16_t szInBuff)  [override], [protected], [virtual]
```

The function directly generates a packet and places it in the buffer for further sending. Returns the status of the operation.

Implements ModbusPort.

#### 7.19.3.18 writeBufferData()

```
const uint8_t * ModbusTcpPort::writeBufferData () const  [override], [virtual]
```

Returns pointer to data of write buffer.

Implements ModbusPort.

#### 7.19.3.19 writeBufferSize()

```
uint16_t ModbusTcpPort::writeBufferSize () const  [override], [virtual]
```

Returns size of data of write buffer.

Implements ModbusPort.

The documentation for this class was generated from the following file:

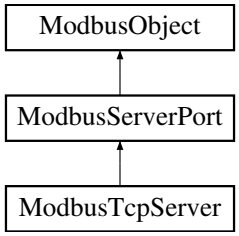
- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusTcpPort.h

## 7.20 ModbusTcpServer Class Reference

The `ModbusTcpServer` class implements TCP server part of the Modbus protocol.

`#include <ModbusTcpServer.h>`

Inheritance diagram for ModbusTcpServer:




### Classes

- struct Defaults

  *`Defaults` class constain default settings values for `ModbusTcpServer`.*

### Public Member Functions

- ModbusTcpServer (ModbusInterface *device)
- ~ModbusTcpServer ()
- uint16_t port () const
- void setPort (uint16_t port)
- uint32_t timeout () const
- void setTimeout (uint32_t timeout)
- uint32_t maxConnections () const
- void setMaxConnections (uint32_t maxconn)
- Modbus::ProtocolType type () const override
- bool isTcpServer () const override
- Modbus::StatusCode open () override
- Modbus::StatusCode close () override
- bool isOpen () const override
- void setBroadcastEnabled (bool enable) override
- void setUnitMap (const void *unitmap) override
- Modbus::StatusCode process () override
- virtual ModbusServerPort * createTcpPort (ModbusTcpSocket *socket)
- virtual void deleteTcpPort (ModbusServerPort *port)
- void signalNewConnection (const Modbus::Char *source)
- void signalCloseConnection (const Modbus::Char *source)

## Public Member Functions inherited from ModbusServerPort

- ModbusInterface * device () const
- void setDevice (ModbusInterface *device)
- bool isBroadcastEnabled () const
- const void * unitMap () const
- void * context () const
- void setContext (void *context)
- bool isStateClosed () const
- void signalOpened (const Modbus::Char *source)
- void signalClosed (const Modbus::Char *source)
- void signalTx (const Modbus::Char *source, const uint8_t *buff, uint16_t size)
- void signalRx (const Modbus::Char *source, const uint8_t *buff, uint16_t size)
- void signalError (const Modbus::Char *source, Modbus::StatusCode status, const Modbus::Char *text)

## Public Member Functions inherited from ModbusObject

- ModbusObject ()
- virtual ~ModbusObject ()
- const Modbus::Char * objectName () const
- void setObjectName (const Modbus::Char *name)
- template<class SignalClass , class T , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class SignalClass , class ReturnType , class ... Args>
  void connect (ModbusMethodPointer< SignalClass, ReturnType, Args ... > signalMethodPtr, ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- template<class ReturnType , class ... Args>
  void disconnect (ModbusFunctionPointer< ReturnType, Args ... > funcPtr)
- void disconnectFunc (void *funcPtr)
- template<class T , class ReturnType , class ... Args>
  void disconnect (T *object, ModbusMethodPointer< T, ReturnType, Args ... > objectMethodPtr)
- template<class T >
  void disconnect (T *object)

### Protected Member Functions

- ModbusTcpSocket * nextPendingConnection ()
- void clearConnections ()

## Protected Member Functions inherited from ModbusServerPort

- ModbusObject ()

## Protected Member Functions inherited from ModbusObject

- template<class T , class ... Args>
  void emitSignal (const char *thisMethodId, ModbusMethodPointer< T, void, Args ... > thisMethod, Args ... args)

**Additional Inherited Members**

### Static Public Member Functions inherited from ModbusObject

- static ModbusObject * sender ()

## 7.20.1 Detailed Description

The ModbusTcpServer class implements TCP server part of the Modbus protocol.

ModbusTcpServer ...

## 7.20.2 Constructor & Destructor Documentation

### 7.20.2.1 ModbusTcpServer()

```
ModbusTcpServer::ModbusTcpServer (
            ModbusInterface * device)
```

Constructor of the class. device param is object which might process incoming requests for read/write memory.

### 7.20.2.2 ~ModbusTcpServer()

```
ModbusTcpServer::~ModbusTcpServer ()
```

Destructor of the class. Clear all unclosed connections.

## 7.20.3 Member Function Documentation

### 7.20.3.1 clearConnections()

```
void ModbusTcpServer::clearConnections ()  [protected]
```

Clear all allocated memory for previously established connections.

### 7.20.3.2 close()

```
Modbus::StatusCode ModbusTcpServer::close ()  [override], [virtual]
```

Stop listening for incoming connections and close all previously opened connections.

**Returns**

- Modbus::Status_Good on success
- Modbus::Status_Processing when operation is not complete

Implements ModbusServerPort.

#### 7.20.3.3 createTcpPort()

```
virtual ModbusServerPort * ModbusTcpServer::createTcpPort (
            ModbusTcpSocket * socket)  [virtual]
```

Creates ModbusServerPort for new incoming connection defined by ModbusTcpSocket pointer May be reimplemented in subclasses.

#### 7.20.3.4 deleteTcpPort()

```
virtual void ModbusTcpServer::deleteTcpPort (
            ModbusServerPort * port)  [virtual]
```

Deletes ModbusServerPort by default. May be reimplemented in subclasses.

#### 7.20.3.5 isOpen()

```
bool ModbusTcpServer::isOpen () const  [override], [virtual]
```

Returns true if the server is currently listening for incoming connections, false otherwise.

Implements ModbusServerPort.

#### 7.20.3.6 isTcpServer()

```
bool ModbusTcpServer::isTcpServer () const  [inline], [override], [virtual]
```

Returns true.

Reimplemented from ModbusServerPort.

#### 7.20.3.7 maxConnections()

```
uint32_t ModbusTcpServer::maxConnections () const
```

Returns setting for the maximum number of simultaneous connections to the server.

#### 7.20.3.8 nextPendingConnection()

```
ModbusTcpSocket * ModbusTcpServer::nextPendingConnection ()  [protected]
```

Checks for incoming connections and returns pointer ModbusTcpSocket if new connection established, nullptr otherwise.

#### 7.20.3.9 open()

```
Modbus::StatusCode ModbusTcpServer::open ()  [override], [virtual]
```

Try to listen for incoming connections on TCP port that was previously set (port()).

**Returns**

- Modbus::Status_Good on success
- Modbus::Status_Processing when operation is not complete
- Modbus::Status_BadTcpCreate when can't create TCP socket
- Modbus::Status_BadTcpBind when can't bind TCP socket
- Modbus::Status_BadTcpListen when can't listen TCP socket

Implements ModbusServerPort.

#### 7.20.3.10 port()

```
uint16_t ModbusTcpServer::port () const
```

Returns the setting for the TCP port number of the server.

#### 7.20.3.11 process()

```
Modbus::StatusCode ModbusTcpServer::process ()  [override], [virtual]
```

Main function of TCP server. Must be called in cycle to perform all incoming TCP connections.

Implements ModbusServerPort.

#### 7.20.3.12 setBroadcastEnabled()

```
void ModbusTcpServer::setBroadcastEnabled (
            bool enable)  [override], [virtual]
```

Enables broadcast mode for 0 unit address. It is enabled by default.

**See also**

isBroadcastEnabled()

Reimplemented from ModbusServerPort.

#### 7.20.3.13 setMaxConnections()

```
void ModbusTcpServer::setMaxConnections (
            uint32_t maxconn)
```

Sets the setting for the maximum number of simultaneous connections to the server.

#### 7.20.3.14 setPort()

```
void ModbusTcpServer::setPort (
            uint16_t port)
```

Sets the settings for the TCP port number of the server.

#### 7.20.3.15 setTimeout()

```
void ModbusTcpServer::setTimeout (
            uint32_t timeout)
```

Sets the setting for the read timeout of every single conncetion.

#### 7.20.3.16 setUnitMap()

```
void ModbusTcpServer::setUnitMap (
            const void * unitmap)  [override], [virtual]
```

Set units map of current server. Server make a copy of units map data.

**See also**

`unitMap()`

Reimplemented from ModbusServerPort.

#### 7.20.3.17 signalCloseConnection()

```
void ModbusTcpServer::signalCloseConnection (
            const Modbus::Char * source)
```

Signal occured when TCP connection was closed. `source` - name of the current connection.

#### 7.20.3.18 signalNewConnection()

```
void ModbusTcpServer::signalNewConnection (
            const Modbus::Char * source)
```

Signal occured when new TCP connection was accepted. `source` - name of the current connection.

#### 7.20.3.19 timeout()

```
uint32_t ModbusTcpServer::timeout () const
```

Returns the setting for the read timeout of every single conncetion.

#### 7.20.3.20 type()

```
Modbus::ProtocolType ModbusTcpServer::type () const  [inline], [override], [virtual]
```

Returns the Modbus protocol type. In this case it is `Modbus::TCP`.

Implements ModbusServerPort.

The documentation for this class was generated from the following file:

- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusTcpServer.h

## 7.21 Modbus::SerialSettings Struct Reference

Struct to define settings for Serial Port.

```
#include <ModbusGlobal.h>
```

### Public Attributes

- const Char * **portName**

  *Value for the serial port name.*
- int32_t **baudRate**

  *Value for the serial port's baud rate.*
- int8_t **dataBits**

  *Value for the serial port's data bits.*
- Parity **parity**

  *Value for the serial port's patiry.*
- StopBits **stopBits**

  *Value for the serial port's stop bits.*
- FlowControl **flowControl**

  *Value for the serial port's flow control.*
- uint32_t **timeoutFirstByte**

  *Value for the serial port's timeout waiting first byte of packet.*
- uint32_t **timeoutInterByte**

  *Value for the serial port's timeout waiting next byte of packet.*

### 7.21.1 Detailed Description

Struct to define settings for Serial Port.

The documentation for this struct was generated from the following file:

- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusGlobal.h

## 7.22 Modbus::Strings Class Reference

Sets constant key values for the map of settings.

`#include <ModbusQt.h>`

### Public Member Functions

- Strings ()

### Static Public Member Functions

- static const Strings & instance ()

### Public Attributes

- const QString **unit**

  *Setting key for the unit number of remote device.*
- const QString **type**

  *Setting key for the type of Modbus protocol.*
- const QString **tries**

  *Setting key for the number of tries a Modbus request is repeated if it fails.*
- const QString **host**

  *Setting key for the IP address or DNS name of the remote device.*
- const QString **port**

  *Setting key for the TCP port number of the remote device.*
- const QString **timeout**

  *Setting key for connection timeout (milliseconds)*
- const QString **maxconn**

  *Setting key for the maximum number of simultaneous connections to the server.*
- const QString **serialPortName**

  *Setting key for the serial port name.*
- const QString **baudRate**

  *Setting key for the serial port's baud rate.*
- const QString **dataBits**

  *Setting key for the serial port's data bits.*
- const QString **parity**

  *Setting key for the serial port's parity.*
- const QString **stopBits**

  *Setting key for the serial port's stop bits.*
- const QString **flowControl**

  *Setting key for the serial port's flow control.*
- const QString **timeoutFirstByte**

  *Setting key for the serial port's timeout waiting first byte of packet.*
- const QString **timeoutInterByte**

  *Setting key for the serial port's timeout waiting next byte of packet.*
- const QString **isBroadcastEnabled**

  *Setting key for the serial port enables broadcast mode for `0` unit address.*
- const QString **NoParity**

*String constant for repr of `NoParity` enum value.*

- const QString **EvenParity**

  *String constant for repr of `EvenParity` enum value.*
- const QString **OddParity**

  *String constant for repr of `OddParity` enum value.*
- const QString **SpaceParity**

  *String constant for repr of `SpaceParity` enum value.*
- const QString **MarkParity**

  *String constant for repr of `MarkParity` enum value.*
- const QString **OneStop**

  *String constant for repr of `OneStop` enum value.*
- const QString **OneAndHalfStop**

  *String constant for repr of `OneAndHalfStop` enum value.*
- const QString **TwoStop**

  *String constant for repr of `TwoStop` enum value.*
- const QString **NoFlowControl**

  *String constant for repr of `NoFlowControl` enum value.*
- const QString **HardwareControl**

  *String constant for repr of `HardwareControl` enum value.*
- const QString **SoftwareControl**

  *String constant for repr of `SoftwareControl` enum value.*

## 7.22.1 Detailed Description

Sets constant key values for the map of settings.

## 7.22.2 Constructor & Destructor Documentation

### 7.22.2.1 Strings()

```
Modbus::Strings::Strings ()
```

Constructor ot the class.

## 7.22.3 Member Function Documentation

### 7.22.3.1 instance()

```
static const Strings & Modbus::Strings::instance ()  [static]
```

Returns a reference to the global `Modbus::Strings` object.

The documentation for this class was generated from the following file:

- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusQt.h

## 7.23 Modbus::TcpSettings Struct Reference

Struct to define settings for TCP connection.

`#include <ModbusGlobal.h>`

### Public Attributes

- const Char * **host**

  *Value for the IP address or DNS name of the remote device.*
- uint16_t **port**

  *Value for the TCP port number of the remote device.*
- uint32_t **timeout**

  *Value for connection timeout (milliseconds)*
- uint32_t **maxconn**

  *Maximum number of simultaneous connections to the server (for server side only)*

### 7.23.1 Detailed Description

Struct to define settings for TCP connection.

The documentation for this struct was generated from the following file:

- c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusGlobal.h

# Chapter 8

# File Documentation

## 8.1 c:/Users/march/Dropbox/PRJ/ModbusLib/src/cModbus.h File Reference

Contains library interface for C language.

```
#include <stdbool.h>
#include "ModbusGlobal.h"
```

### Typedefs

- typedef ModbusPort * **cModbusPort**

  *Handle (pointer) of `ModbusPort` for C interface.*
- typedef ModbusClientPort * **cModbusClientPort**

  *Handle (pointer) of `ModbusClientPort` for C interface.*
- typedef ModbusClient * **cModbusClient**

  *Handle (pointer) of `ModbusClient` for C interface.*
- typedef ModbusServerPort * **cModbusServerPort**

  *Handle (pointer) of `ModbusServerPort` for C interface.*
- typedef ModbusInterface * **cModbusInterface**

  *Handle (pointer) of `ModbusInterface` for C interface.*
- typedef void * **cModbusDevice**

  *Handle (pointer) of `ModbusDevice` for C interface.*
- typedef StatusCode(* pfReadCoils) (cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t count, void *values)
- typedef StatusCode(* pfReadDiscreteInputs) (cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t count, void *values)
- typedef StatusCode(* pfReadHoldingRegisters) (cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t count, uint16_t *values)
- typedef StatusCode(* pfReadInputRegisters) (cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t count, uint16_t *values)
- typedef StatusCode(* pfWriteSingleCoil) (cModbusDevice dev, uint8_t unit, uint16_t offset, bool value)
- typedef StatusCode(* pfWriteSingleRegister) (cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t value)
- typedef StatusCode(* pfReadExceptionStatus) (cModbusDevice dev, uint8_t unit, uint8_t *status)

- typedef StatusCode(* pfDiagnostics) (cModbusDevice dev, uint8_t unit, uint16_t subfunc, uint8_t insize, const uint8_t *indata, uint8_t *outsize, uint8_t *outdata)
- typedef StatusCode(* pfGetCommEventCounter) (cModbusDevice dev, uint8_t unit, uint16_t *status, uint16_t *eventCount)
- typedef StatusCode(* pfGetCommEventLog) (cModbusDevice dev, uint8_t unit, uint16_t *status, uint16_t *eventCount, uint16_t *messageCount, uint8_t *eventBuffSize, uint8_t *eventBuff)
- typedef StatusCode(* pfWriteMultipleCoils) (cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t count, const void *values)
- typedef StatusCode(* pfWriteMultipleRegisters) (cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t count, const uint16_t *values)
- typedef StatusCode(* pfReportServerID) (cModbusDevice dev, uint8_t unit, uint8_t *count, uint8_t *data)
- typedef StatusCode(* pfMaskWriteRegister) (cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t andMask, uint16_t orMask)
- typedef StatusCode(* pfReadWriteMultipleRegisters) (cModbusDevice dev, uint8_t unit, uint16_t readOffset, uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t *writeValues)
- typedef StatusCode(* pfReadFIFOQueue) (cModbusDevice dev, uint8_t unit, uint16_t fifoadr, uint16_t *count, uint16_t *values)
- typedef void(* pfSlotOpened) (const Char *source)
- typedef void(* pfSlotClosed) (const Char *source)
- typedef void(* pfSlotTx) (const Char *source, const uint8_t *buff, uint16_t size)
- typedef void(* pfSlotRx) (const Char *source, const uint8_t *buff, uint16_t size)
- typedef void(* pfSlotError) (const Char *source, StatusCode status, const Char *text)
- typedef void(* pfSlotNewConnection) (const Char *source)
- typedef void(* pfSlotCloseConnection) (const Char *source)

### Functions

- MODBUS_EXPORT cModbusInterface cCreateModbusDevice (cModbusDevice device, pfReadCoils readCoils, pfReadDiscreteInputs readDiscreteInputs, pfReadHoldingRegisters readHoldingRegisters, pfReadInputRegisters readInputRegisters, pfWriteSingleCoil writeSingleCoil, pfWriteSingleRegister writeSingleRegister, pfReadExceptionStatus readExceptionStatus, pfDiagnostics diagnostics, pfGetCommEventCounter getCommEventCounter, pfGetCommEventLog getCommEventLog, pfWriteMultipleCoils writeMultipleCoils, pfWriteMultipleRegisters writeMultipleRegisters, pfReportServerID reportServerID, pfMaskWriteRegister maskWriteRegister, pfReadWriteMultipleRegisters readWriteMultipleRegisters, pfReadFIFOQueue readFIFOQueue)
- MODBUS_EXPORT void cDeleteModbusDevice (cModbusInterface dev)
- MODBUS_EXPORT cModbusPort cPortCreate (ProtocolType type, const void *settings, bool blocking)
- MODBUS_EXPORT void cPortDelete (cModbusPort port)
- MODBUS_EXPORT cModbusClientPort cCpoCreate (ProtocolType type, const void *settings, bool blocking)
- MODBUS_EXPORT cModbusClientPort cCpoCreateForPort (cModbusPort port)
- MODBUS_EXPORT void cCpoDelete (cModbusClientPort clientPort)
- MODBUS_EXPORT const Char * cCpoGetObjectName (cModbusClientPort clientPort)
- MODBUS_EXPORT void cCpoSetObjectName (cModbusClientPort clientPort, const Char *name)
- MODBUS_EXPORT ProtocolType cCpoGetType (cModbusClientPort clientPort)
- MODBUS_EXPORT bool cCpoIsOpen (cModbusClientPort clientPort)
- MODBUS_EXPORT bool cCpoClose (cModbusClientPort clientPort)
- MODBUS_EXPORT uint32_t cCpoGetRepeatCount (cModbusClientPort clientPort)
- MODBUS_EXPORT void cCpoSetRepeatCount (cModbusClientPort clientPort, uint32_t count)
- MODBUS_EXPORT StatusCode cCpoReadCoils (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, uint16_t count, void *values)
- MODBUS_EXPORT StatusCode cCpoReadDiscreteInputs (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, uint16_t count, void *values)
- MODBUS_EXPORT StatusCode cCpoReadHoldingRegisters (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, uint16_t count, uint16_t *values)

- MODBUS_EXPORT StatusCode cCpoReadInputRegisters (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, uint16_t count, uint16_t *values)
- MODBUS_EXPORT StatusCode cCpoWriteSingleCoil (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, bool value)
- MODBUS_EXPORT StatusCode cCpoWriteSingleRegister (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, uint16_t value)
- MODBUS_EXPORT StatusCode cCpoReadExceptionStatus (cModbusClientPort clientPort, uint8_t unit, uint8_t *value)
- MODBUS_EXPORT StatusCode cCpoDiagnostics (cModbusClientPort clientPort, uint8_t unit, uint16_t subfunc, uint8_t insize, const uint8_t *indata, uint8_t *outsize, uint8_t *outdata)
- MODBUS_EXPORT StatusCode cCpoGetCommEventCounter (cModbusClientPort clientPort, uint8_t unit, uint16_t *status, uint16_t *eventCount)
- MODBUS_EXPORT StatusCode cCpoGetCommEventLog (cModbusClientPort clientPort, uint8_t unit, uint16_t *status, uint16_t *eventCount, uint16_t *messageCount, uint8_t *eventBuffSize, uint8_t *eventBuff)
- MODBUS_EXPORT StatusCode cCpoWriteMultipleCoils (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, uint16_t count, const void *values)
- MODBUS_EXPORT StatusCode cCpoWriteMultipleRegisters (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, uint16_t count, const uint16_t *values)
- MODBUS_EXPORT StatusCode cCpoReportServerID (cModbusClientPort clientPort, uint8_t unit, uint8_t *count, uint8_t *data)
- MODBUS_EXPORT StatusCode cCpoMaskWriteRegister (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, uint16_t andMask, uint16_t orMask)
- MODBUS_EXPORT StatusCode cCpoReadWriteMultipleRegisters (cModbusClientPort clientPort, uint8_t unit, uint16_t readOffset, uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t *writeValues)
- MODBUS_EXPORT StatusCode cCpoReadFIFOQueue (cModbusClientPort clientPort, uint8_t unit, uint16_t fifoadr, uint16_t *count, uint16_t *values)
- MODBUS_EXPORT StatusCode cCpoReadCoilsAsBoolArray (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, uint16_t count, bool *values)
- MODBUS_EXPORT StatusCode cCpoReadDiscreteInputsAsBoolArray (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, uint16_t count, bool *values)
- MODBUS_EXPORT StatusCode cCpoWriteMultipleCoilsAsBoolArray (cModbusClientPort clientPort, uint8_t unit, uint16_t offset, uint16_t count, const bool *values)
- MODBUS_EXPORT StatusCode cCpoGetLastStatus (cModbusClientPort clientPort)
- MODBUS_EXPORT StatusCode cCpoGetLastErrorStatus (cModbusClientPort clientPort)
- MODBUS_EXPORT const Char * cCpoGetLastErrorText (cModbusClientPort clientPort)
- MODBUS_EXPORT void cCpoConnectOpened (cModbusClientPort clientPort, pfSlotOpened funcPtr)
- MODBUS_EXPORT void cCpoConnectClosed (cModbusClientPort clientPort, pfSlotClosed funcPtr)
- MODBUS_EXPORT void cCpoConnectTx (cModbusClientPort clientPort, pfSlotTx funcPtr)
- MODBUS_EXPORT void cCpoConnectRx (cModbusClientPort clientPort, pfSlotRx funcPtr)
- MODBUS_EXPORT void cCpoConnectError (cModbusClientPort clientPort, pfSlotError funcPtr)
- MODBUS_EXPORT void cCpoDisconnectFunc (cModbusClientPort clientPort, void *funcPtr)
- MODBUS_EXPORT cModbusClient cCliCreate (uint8_t unit, ProtocolType type, const void *settings, bool blocking)
- MODBUS_EXPORT cModbusClient cCliCreateForClientPort (uint8_t unit, cModbusClientPort clientPort)
- MODBUS_EXPORT void cCliDelete (cModbusClient client)
- MODBUS_EXPORT const Char * cCliGetObjectName (cModbusClient client)
- MODBUS_EXPORT void cCliSetObjectName (cModbusClient client, const Char *name)
- MODBUS_EXPORT ProtocolType cCliGetType (cModbusClient client)
- MODBUS_EXPORT uint8_t cCliGetUnit (cModbusClient client)
- MODBUS_EXPORT void cCliSetUnit (cModbusClient client, uint8_t unit)
- MODBUS_EXPORT bool cCliIsOpen (cModbusClient client)
- MODBUS_EXPORT cModbusClientPort cCliGetPort (cModbusClient client)
- MODBUS_EXPORT StatusCode cReadCoils (cModbusClient client, uint16_t offset, uint16_t count, void *values)

- MODBUS_EXPORT StatusCode cReadDiscreteInputs (cModbusClient client, uint16_t offset, uint16_t count, void *values)
- MODBUS_EXPORT StatusCode cReadHoldingRegisters (cModbusClient client, uint16_t offset, uint16_t count, uint16_t *values)
- MODBUS_EXPORT StatusCode cReadInputRegisters (cModbusClient client, uint16_t offset, uint16_t count, uint16_t *values)
- MODBUS_EXPORT StatusCode cWriteSingleCoil (cModbusClient client, uint16_t offset, bool value)
- MODBUS_EXPORT StatusCode cWriteSingleRegister (cModbusClient client, uint16_t offset, uint16_t value)
- MODBUS_EXPORT StatusCode cReadExceptionStatus (cModbusClient client, uint8_t *value)
- MODBUS_EXPORT StatusCode cWriteMultipleCoils (cModbusClient client, uint16_t offset, uint16_t count, const void *values)
- MODBUS_EXPORT StatusCode cWriteMultipleRegisters (cModbusClient client, uint16_t offset, uint16_t count, const uint16_t *values)
- MODBUS_EXPORT StatusCode cMaskWriteRegister (cModbusClient client, uint16_t offset, uint16_t andMask, uint16_t orMask)
- MODBUS_EXPORT StatusCode cReadWriteMultipleRegisters (cModbusClient client, uint16_t readOffset, uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t *writeValues)
- MODBUS_EXPORT StatusCode cReadCoilsAsBoolArray (cModbusClient client, uint16_t offset, uint16_t count, bool *values)
- MODBUS_EXPORT StatusCode cReadDiscreteInputsAsBoolArray (cModbusClient client, uint16_t offset, uint16_t count, bool *values)
- MODBUS_EXPORT StatusCode cWriteMultipleCoilsAsBoolArray (cModbusClient client, uint16_t offset, uint16_t count, const bool *values)
- MODBUS_EXPORT StatusCode cCliGetLastPortStatus (cModbusClient client)
- MODBUS_EXPORT StatusCode cCliGetLastPortErrorStatus (cModbusClient client)
- MODBUS_EXPORT const Char * cCliGetLastPortErrorText (cModbusClient client)
- MODBUS_EXPORT cModbusServerPort cSpoCreate (cModbusInterface device, ProtocolType type, const void *settings, bool blocking)
- MODBUS_EXPORT void cSpoDelete (cModbusServerPort serverPort)
- MODBUS_EXPORT const Char * cSpoGetObjectName (cModbusServerPort serverPort)
- MODBUS_EXPORT void cSpoSetObjectName (cModbusServerPort serverPort, const Char *name)
- MODBUS_EXPORT ProtocolType cSpoGetType (cModbusServerPort serverPort)
- MODBUS_EXPORT bool cSpoIsTcpServer (cModbusServerPort serverPort)
- MODBUS_EXPORT cModbusInterface cSpoGetDevice (cModbusServerPort serverPort)
- MODBUS_EXPORT bool cSpoIsOpen (cModbusServerPort serverPort)
- MODBUS_EXPORT StatusCode cSpoOpen (cModbusServerPort serverPort)
- MODBUS_EXPORT StatusCode cSpoClose (cModbusServerPort serverPort)
- MODBUS_EXPORT StatusCode cSpoProcess (cModbusServerPort serverPort)
- MODBUS_EXPORT void cSpoConnectOpened (cModbusServerPort serverPort, pfSlotOpened funcPtr)
- MODBUS_EXPORT void cSpoConnectClosed (cModbusServerPort serverPort, pfSlotClosed funcPtr)
- MODBUS_EXPORT void cSpoConnectTx (cModbusServerPort serverPort, pfSlotTx funcPtr)
- MODBUS_EXPORT void cSpoConnectRx (cModbusServerPort serverPort, pfSlotRx funcPtr)
- MODBUS_EXPORT void cSpoConnectError (cModbusServerPort serverPort, pfSlotError funcPtr)
- MODBUS_EXPORT void cSpoConnectNewConnection (cModbusServerPort serverPort, pfSlotNewConnection funcPtr)
- MODBUS_EXPORT void cSpoConnectCloseConnection (cModbusServerPort serverPort, pfSlotCloseConnection funcPtr)
- MODBUS_EXPORT void cSpoDisconnectFunc (cModbusServerPort serverPort, void *funcPtr)

### 8.1.1 Detailed Description

Contains library interface for C language.

**Author**

serhmarch

**Date**

May 2024

### 8.1.2 Typedef Documentation

#### 8.1.2.1 pfDiagnostics

```
typedef StatusCode(* pfDiagnostics) (cModbusDevice dev, uint8_t unit, uint16_t subfunc, uint8↩
_t insize, const uint8_t *indata, uint8_t *outsize, uint8_t *outdata)
```

Pointer to C function for diagnostics. `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::diagnostics

#### 8.1.2.2 pfGetCommEventCounter

```
typedef StatusCode(* pfGetCommEventCounter) (cModbusDevice dev, uint8_t unit, uint16_t *status,
uint16_t *eventCount)
```

Pointer to C function for get communication event counter. `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::getCommEventCounter

#### 8.1.2.3 pfGetCommEventLog

```
typedef StatusCode(* pfGetCommEventLog) (cModbusDevice dev, uint8_t unit, uint16_t *status,
uint16_t *eventCount, uint16_t *messageCount, uint8_t *eventBuffSize, uint8_t *eventBuff)
```

Pointer to C function for get communication event logs. `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::getCommEventLog

#### 8.1.2.4 pfMaskWriteRegister

```
typedef StatusCode(* pfMaskWriteRegister) (cModbusDevice dev, uint8_t unit, uint16_t offset,
uint16_t andMask, uint16_t orMask)
```

Pointer to C function for mask write registers (4x). `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::maskWriteRegister

#### 8.1.2.5 pfReadCoils

```
typedef StatusCode(* pfReadCoils) (cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t
count, void *values)
```

Pointer to C function for read coils (0x). `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::readCoils

#### 8.1.2.6 pfReadDiscreteInputs

```
typedef StatusCode(* pfReadDiscreteInputs) (cModbusDevice dev, uint8_t unit, uint16_t offset,
uint16_t count, void *values)
```

Pointer to C function for read discrete inputs (1x). `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::readDiscreteInputs

#### 8.1.2.7 pfReadExceptionStatus

```
typedef StatusCode(* pfReadExceptionStatus) (cModbusDevice dev, uint8_t unit, uint8_t *status)
```

Pointer to C function for read exception status bits. `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::readExceptionStatus

#### 8.1.2.8 pfReadFIFOQueue

```
typedef StatusCode(* pfReadFIFOQueue) (cModbusDevice dev, uint8_t unit, uint16_t fifoadr,
uint16_t *count, uint16_t *values)
```

Pointer to C function for read FIFO queue. `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::readFIFOQueue

#### 8.1.2.9 pfReadHoldingRegisters

```
typedef StatusCode(* pfReadHoldingRegisters) (cModbusDevice dev, uint8_t unit, uint16_t offset,
uint16_t count, uint16_t *values)
```

Pointer to C function for read holding registers (4x). `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::readHoldingRegisters

#### 8.1.2.10 pfReadInputRegisters

```
typedef StatusCode(* pfReadInputRegisters) (cModbusDevice dev, uint8_t unit, uint16_t offset,
uint16_t count, uint16_t *values)
```

Pointer to C function for read input registers (3x). `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::readInputRegisters

#### 8.1.2.11 pfReadWriteMultipleRegisters

```
typedef StatusCode(* pfReadWriteMultipleRegisters) (cModbusDevice dev, uint8_t unit, uint16_t
readOffset, uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t write↩
Count, const uint16_t *writeValues)
```

Pointer to C function for write registers (4x). `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::writeMultipleRegisters

#### 8.1.2.12 pfReportServerID

```
typedef StatusCode(* pfReportServerID) (cModbusDevice dev, uint8_t unit, uint8_t *count, uint8↩
_t *data)
```

Pointer to C function for report server id. `dev` - pointer to any struct that can hold memory data.

**See also**

`ModbusInterface::reportServerID`

#### 8.1.2.13 pfSlotCloseConnection

```
typedef void(* pfSlotCloseConnection) (const Char *source)
```

Pointer to C callback function. `dev` - pointer to any struct that can hold memory data.

**See also**

`ModbusTcpServer::signalCloseConnection`

#### 8.1.2.14 pfSlotClosed

```
typedef void(* pfSlotClosed) (const Char *source)
```

Pointer to C callback function. `dev` - pointer to any struct that can hold memory data.

**See also**

`ModbusClientPort::signalClosed` and `ModbusServerPort::signalClosed`

#### 8.1.2.15 pfSlotError

```
typedef void(* pfSlotError) (const Char *source, StatusCode status, const Char *text)
```

Pointer to C callback function. `dev` - pointer to any struct that can hold memory data.

**See also**

`ModbusClientPort::signalError` and `ModbusServerPort::signalError`

#### 8.1.2.16 pfSlotNewConnection

```
typedef void(* pfSlotNewConnection) (const Char *source)
```

Pointer to C callback function. `dev` - pointer to any struct that can hold memory data.

**See also**

`ModbusTcpServer::signalNewConnection`

#### 8.1.2.17 pfSlotOpened

```
typedef void(* pfSlotOpened) (const Char *source)
```

Pointer to C callback function. `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusClientPort::signalOpened and ModbusServerPort::signalOpened

#### 8.1.2.18 pfSlotRx

```
typedef void(* pfSlotRx) (const Char *source, const uint8_t *buff, uint16_t size)
```

Pointer to C callback function. `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusClientPort::signalRx and ModbusServerPort::signalRx

#### 8.1.2.19 pfSlotTx

```
typedef void(* pfSlotTx) (const Char *source, const uint8_t *buff, uint16_t size)
```

Pointer to C callback function. `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusClientPort::signalTx and ModbusServerPort::signalTx

#### 8.1.2.20 pfWriteMultipleCoils

```
typedef StatusCode(* pfWriteMultipleCoils) (cModbusDevice dev, uint8_t unit, uint16_t offset,
uint16_t count, const void *values)
```

Pointer to C function for write coils (0x). `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::writeMultipleCoils

#### 8.1.2.21 pfWriteMultipleRegisters

```
typedef StatusCode(* pfWriteMultipleRegisters) (cModbusDevice dev, uint8_t unit, uint16_↩
t offset, uint16_t count, const uint16_t *values)
```

Pointer to C function for write registers (4x). `dev` - pointer to any struct that can hold memory data.

**See also**

ModbusInterface::writeMultipleRegisters

#### 8.1.2.22 pfWriteSingleCoil

```
typedef StatusCode(* pfWriteSingleCoil) (cModbusDevice dev, uint8_t unit, uint16_t offset,
bool value)
```

Pointer to C function for write single coil (0x). `dev` - pointer to any struct that can hold memory data.

**See also**

`ModbusInterface::writeSingleCoil`

#### 8.1.2.23 pfWriteSingleRegister

```
typedef StatusCode(* pfWriteSingleRegister) (cModbusDevice dev, uint8_t unit, uint16_t offset,
uint16_t value)
```

Pointer to C function for write single register (4x). `dev` - pointer to any struct that can hold memory data.

**See also**

`ModbusInterface::writeSingleRegister`

### 8.1.3 Function Documentation

#### 8.1.3.1 cCliCreate()

```
MODBUS_EXPORT cModbusClient cCliCreate (
            uint8_t unit,
            ProtocolType type,
            const void * settings,
            bool blocking)
```

Creates `ModbusClient` object and returns handle to it.

**See also**

`Modbus::createClient`

#### 8.1.3.2 cCliCreateForClientPort()

```
MODBUS_EXPORT cModbusClient cCliCreateForClientPort (
            uint8_t unit,
            cModbusClientPort clientPort)
```

Creates `ModbusClient` object with `unit` for port `clientPort` and returns handle to it.

#### 8.1.3.3 cCliDelete()

```
MODBUS_EXPORT void cCliDelete (
            cModbusClient client)
```

Deletes previously created ModbusClient object represented by client handle

#### 8.1.3.4 cCliGetLastPortErrorStatus()

```
MODBUS_EXPORT StatusCode cCliGetLastPortErrorStatus (
            cModbusClient client)
```

Wrapper for ModbusClient::lastPortErrorStatus

#### 8.1.3.5 cCliGetLastPortErrorText()

```
MODBUS_EXPORT const Char * cCliGetLastPortErrorText (
            cModbusClient client)
```

Wrapper for ModbusClient::lastPortErrorText

#### 8.1.3.6 cCliGetLastPortStatus()

```
MODBUS_EXPORT StatusCode cCliGetLastPortStatus (
            cModbusClient client)
```

Wrapper for ModbusClient::lastPortStatus

#### 8.1.3.7 cCliGetObjectName()

```
MODBUS_EXPORT const Char * cCliGetObjectName (
            cModbusClient client)
```

Wrapper for ModbusClient::objectName

#### 8.1.3.8 cCliGetPort()

```
MODBUS_EXPORT cModbusClientPort cCliGetPort (
            cModbusClient client)
```

Wrapper for ModbusClient::port

#### 8.1.3.9 cCliGetType()

```
MODBUS_EXPORT ProtocolType cCliGetType (
            cModbusClient client)
```

Wrapper for ModbusClient::type

#### 8.1.3.10 cCliGetUnit()

```
MODBUS_EXPORT uint8_t cCliGetUnit (
            cModbusClient client)
```

Wrapper for ModbusClient::unit

#### 8.1.3.11 cCliIsOpen()

```
MODBUS_EXPORT bool cCliIsOpen (
            cModbusClient client)
```

Wrapper for ModbusClient::isOpen

#### 8.1.3.12 cCliSetObjectName()

```
MODBUS_EXPORT void cCliSetObjectName (
            cModbusClient client,
            const Char * name)
```

Wrapper for ModbusClient::setObjectName

#### 8.1.3.13 cCliSetUnit()

```
MODBUS_EXPORT void cCliSetUnit (
            cModbusClient client,
            uint8_t unit)
```

Wrapper for ModbusClient::setUnit

#### 8.1.3.14 cCpoClose()

```
MODBUS_EXPORT bool cCpoClose (
            cModbusClientPort clientPort)
```

Wrapper for ModbusClientPort::close

#### 8.1.3.15 cCpoConnectClosed()

```
MODBUS_EXPORT void cCpoConnectClosed (
            cModbusClientPort clientPort,
            pfSlotClosed funcPtr)
```

Connects funcPtr-function to ModbusClientPort::signalClosed signal

#### 8.1.3.16 cCpoConnectError()

```
MODBUS_EXPORT void cCpoConnectError (
            cModbusClientPort clientPort,
            pfSlotError funcPtr)
```

Connects `funcPtr`-function to `ModbusClientPort::signalError` signal

#### 8.1.3.17 cCpoConnectOpened()

```
MODBUS_EXPORT void cCpoConnectOpened (
            cModbusClientPort clientPort,
            pfSlotOpened funcPtr)
```

Connects `funcPtr`-function to `ModbusClientPort::signalOpened` signal

#### 8.1.3.18 cCpoConnectRx()

```
MODBUS_EXPORT void cCpoConnectRx (
            cModbusClientPort clientPort,
            pfSlotRx funcPtr)
```

Connects `funcPtr`-function to `ModbusClientPort::signalRx` signal

#### 8.1.3.19 cCpoConnectTx()

```
MODBUS_EXPORT void cCpoConnectTx (
            cModbusClientPort clientPort,
            pfSlotTx funcPtr)
```

Connects `funcPtr`-function to `ModbusClientPort::signalTx` signal

#### 8.1.3.20 cCpoCreate()

```
MODBUS_EXPORT cModbusClientPort cCpoCreate (
            ProtocolType type,
            const void * settings,
            bool blocking)
```

Creates `ModbusClientPort` object and returns handle to it.

**See also**

`Modbus::createClientPort`

#### 8.1.3.21 cCpoCreateForPort()

```
MODBUS_EXPORT cModbusClientPort cCpoCreateForPort (
            cModbusPort port)
```

Creates `ModbusClientPort` object and returns handle to it.

#### 8.1.3.22 cCpoDelete()

```
MODBUS_EXPORT void cCpoDelete (
            cModbusClientPort clientPort)
```

Deletes previously created ModbusClientPort object represented by port handle

#### 8.1.3.23 cCpoDiagnostics()

```
MODBUS_EXPORT StatusCode cCpoDiagnostics (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t subfunc,
            uint8_t insize,
            const uint8_t * indata,
            uint8_t * outsize,
            uint8_t * outdata)
```

Wrapper for ModbusClientPort::diagnostics

#### 8.1.3.24 cCpoDisconnectFunc()

```
MODBUS_EXPORT void cCpoDisconnectFunc (
            cModbusClientPort clientPort,
            void * funcPtr)
```

Disconnects funcPtr-function from ModbusClientPort

#### 8.1.3.25 cCpoGetCommEventCounter()

```
MODBUS_EXPORT StatusCode cCpoGetCommEventCounter (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t * status,
            uint16_t * eventCount)
```

Wrapper for ModbusClientPort::getCommEventCounter

#### 8.1.3.26 cCpoGetCommEventLog()

```
MODBUS_EXPORT StatusCode cCpoGetCommEventLog (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t * status,
            uint16_t * eventCount,
            uint16_t * messageCount,
            uint8_t * eventBuffSize,
            uint8_t * eventBuff)
```

Wrapper for ModbusClientPort::getCommEventLog

#### 8.1.3.27 cCpoGetLastErrorStatus()

```
MODBUS_EXPORT StatusCode cCpoGetLastErrorStatus (
            cModbusClientPort clientPort)
```

Wrapper for ModbusClientPort::getLastErrorStatus

#### 8.1.3.28 cCpoGetLastErrorText()

```
MODBUS_EXPORT const Char * cCpoGetLastErrorText (
            cModbusClientPort clientPort)
```

Wrapper for ModbusClientPort::getLastErrorText

#### 8.1.3.29 cCpoGetLastStatus()

```
MODBUS_EXPORT StatusCode cCpoGetLastStatus (
            cModbusClientPort clientPort)
```

Wrapper for ModbusClientPort::getLastStatus

#### 8.1.3.30 cCpoGetObjectName()

```
MODBUS_EXPORT const Char * cCpoGetObjectName (
            cModbusClientPort clientPort)
```

Wrapper for ModbusClientPort::objectName

#### 8.1.3.31 cCpoGetRepeatCount()

```
MODBUS_EXPORT uint32_t cCpoGetRepeatCount (
            cModbusClientPort clientPort)
```

Wrapper for ModbusClientPort::repeatCount

#### 8.1.3.32 cCpoGetType()

```
MODBUS_EXPORT ProtocolType cCpoGetType (
            cModbusClientPort clientPort)
```

Wrapper for ModbusClientPort::type

#### 8.1.3.33 cCpoIsOpen()

```
MODBUS_EXPORT bool cCpoIsOpen (
            cModbusClientPort clientPort)
```

Wrapper for ModbusClientPort::isOpen

#### 8.1.3.34 cCpoMaskWriteRegister()

```
MODBUS_EXPORT StatusCode cCpoMaskWriteRegister (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            uint16_t andMask,
            uint16_t orMask)
```

Wrapper for ModbusClientPort::maskWriteRegister

#### 8.1.3.35 cCpoReadCoils()

```
MODBUS_EXPORT StatusCode cCpoReadCoils (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            void * values)
```

Wrapper for ModbusClientPort::readCoils

#### 8.1.3.36 cCpoReadCoilsAsBoolArray()

```
MODBUS_EXPORT StatusCode cCpoReadCoilsAsBoolArray (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            bool * values)
```

Wrapper for ModbusClientPort::readCoilsAsBoolArray

#### 8.1.3.37 cCpoReadDiscreteInputs()

```
MODBUS_EXPORT StatusCode cCpoReadDiscreteInputs (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            void * values)
```

Wrapper for ModbusClientPort::readDiscreteInputs

#### 8.1.3.38 cCpoReadDiscreteInputsAsBoolArray()

```
MODBUS_EXPORT StatusCode cCpoReadDiscreteInputsAsBoolArray (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            bool * values)
```

Wrapper for ModbusClientPort::readDiscreteInputsAsBoolArray

#### 8.1.3.39 cCpoReadExceptionStatus()

```
MODBUS_EXPORT StatusCode cCpoReadExceptionStatus (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint8_t * value)
```

Wrapper for ModbusClientPort::readExceptionStatus

#### 8.1.3.40 cCpoReadFIFOQueue()

```
MODBUS_EXPORT StatusCode cCpoReadFIFOQueue (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t fifoadr,
            uint16_t * count,
            uint16_t * values)
```

Wrapper for ModbusClientPort::readFIFOQueue

#### 8.1.3.41 cCpoReadHoldingRegisters()

```
MODBUS_EXPORT StatusCode cCpoReadHoldingRegisters (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            uint16_t * values)
```

Wrapper for ModbusClientPort::readHoldingRegisters

#### 8.1.3.42 cCpoReadInputRegisters()

```
MODBUS_EXPORT StatusCode cCpoReadInputRegisters (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            uint16_t * values)
```

Wrapper for ModbusClientPort::readInputRegisters

#### 8.1.3.43 cCpoReadWriteMultipleRegisters()

```
MODBUS_EXPORT StatusCode cCpoReadWriteMultipleRegisters (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t readOffset,
            uint16_t readCount,
            uint16_t * readValues,
            uint16_t writeOffset,
            uint16_t writeCount,
            const uint16_t * writeValues)
```

Wrapper for ModbusClientPort::readWriteMultipleRegisters

#### 8.1.3.44 cCpoReportServerID()

```
MODBUS_EXPORT StatusCode cCpoReportServerID (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint8_t * count,
            uint8_t * data)
```

Wrapper for ModbusClientPort::reportServerID

#### 8.1.3.45 cCpoSetObjectName()

```
MODBUS_EXPORT void cCpoSetObjectName (
            cModbusClientPort clientPort,
            const Char * name)
```

Wrapper for ModbusClientPort::setObjectName

#### 8.1.3.46 cCpoSetRepeatCount()

```
MODBUS_EXPORT void cCpoSetRepeatCount (
            cModbusClientPort clientPort,
            uint32_t count)
```

Wrapper for ModbusClientPort::setRepeatCount

#### 8.1.3.47 cCpoWriteMultipleCoils()

```
MODBUS_EXPORT StatusCode cCpoWriteMultipleCoils (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            const void * values)
```

Wrapper for ModbusClientPort::writeMultipleCoils

#### 8.1.3.48 cCpoWriteMultipleCoilsAsBoolArray()

```
MODBUS_EXPORT StatusCode cCpoWriteMultipleCoilsAsBoolArray (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            const bool * values)
```

Wrapper for ModbusClientPort::writeMultipleCoilsAsBoolArray

#### 8.1.3.49 cCpoWriteMultipleRegisters()

```
MODBUS_EXPORT StatusCode cCpoWriteMultipleRegisters (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            uint16_t count,
            const uint16_t * values)
```

Wrapper for ModbusClientPort::writeMultipleRegisters

#### 8.1.3.50 cCpoWriteSingleCoil()

```
MODBUS_EXPORT StatusCode cCpoWriteSingleCoil (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            bool value)
```

Wrapper for ModbusClientPort::writeSingleCoil

#### 8.1.3.51 cCpoWriteSingleRegister()

```
MODBUS_EXPORT StatusCode cCpoWriteSingleRegister (
            cModbusClientPort clientPort,
            uint8_t unit,
            uint16_t offset,
            uint16_t value)
```

Wrapper for ModbusClientPort::writeSingleRegister

#### 8.1.3.52 cCreateModbusDevice()

```
MODBUS_EXPORT cModbusInterface cCreateModbusDevice (
            cModbusDevice device,
            pfReadCoils readCoils,
            pfReadDiscreteInputs readDiscreteInputs,
            pfReadHoldingRegisters readHoldingRegisters,
            pfReadInputRegisters readInputRegisters,
            pfWriteSingleCoil writeSingleCoil,
            pfWriteSingleRegister writeSingleRegister,
            pfReadExceptionStatus readExceptionStatus,
            pfDiagnostics diagnostics,
            pfGetCommEventCounter getCommEventCounter,
            pfGetCommEventLog getCommEventLog,
            pfWriteMultipleCoils writeMultipleCoils,
            pfWriteMultipleRegisters writeMultipleRegisters,
            pfReportServerID reportServerID,
            pfMaskWriteRegister maskWriteRegister,
            pfReadWriteMultipleRegisters readWriteMultipleRegisters,
            pfReadFIFOQueue readFIFOQueue)
```

Function create ModbusInterface object and returns pointer to it for server. dev - pointer to any struct that can hold memory data. readCoils, readDiscreteInputs, readHoldingRegisters, readInputRegisters, write↩ SingleCoil, writeSingleRegister, readExceptionStatus, diagnostics, getCommEventCounter, getCommEventLog, writeMultipleCoils writeMultipleRegisters reportServerID, maskWriteRegister, readWriteMultipleRegisters, read↩ FIFOQueue - pointers to corresponding Modbus functions to process data. Any pointer can have NULL value. In this case server will return Status_BadIllegalFunction.

#### 8.1.3.53 cDeleteModbusDevice()

```
MODBUS_EXPORT void cDeleteModbusDevice (
            cModbusInterface dev)
```

Deletes previously created ModbusInterface object represented by dev handle

#### 8.1.3.54 cMaskWriteRegister()

```
MODBUS_EXPORT StatusCode cMaskWriteRegister (
            cModbusClient client,
            uint16_t offset,
            uint16_t andMask,
            uint16_t orMask)
```

Wrapper for ModbusClient::maskWriteRegister

#### 8.1.3.55 cPortCreate()

```
MODBUS_EXPORT cModbusPort cPortCreate (
            ProtocolType type,
            const void * settings,
            bool blocking)
```

Creates ModbusPort object and returns handle to it.

**See also**

Modbus::createPort

#### 8.1.3.56 cPortDelete()

```
MODBUS_EXPORT void cPortDelete (
            cModbusPort port)
```

Deletes previously created ModbusPort object represented by port handle

#### 8.1.3.57 cReadCoils()

```
MODBUS_EXPORT StatusCode cReadCoils (
            cModbusClient client,
            uint16_t offset,
            uint16_t count,
            void * values)
```

Wrapper for ModbusClient::readCoils

#### 8.1.3.58 cReadCoilsAsBoolArray()

```
MODBUS_EXPORT StatusCode cReadCoilsAsBoolArray (
            cModbusClient client,
            uint16_t offset,
            uint16_t count,
            bool * values)
```

Wrapper for ModbusClient::readCoilsAsBoolArray

#### 8.1.3.59 cReadDiscreteInputs()

```
MODBUS_EXPORT StatusCode cReadDiscreteInputs (
            cModbusClient client,
            uint16_t offset,
            uint16_t count,
            void * values)
```

Wrapper for ModbusClient::readDiscreteInputs

#### 8.1.3.60 cReadDiscreteInputsAsBoolArray()

```
MODBUS_EXPORT StatusCode cReadDiscreteInputsAsBoolArray (
            cModbusClient client,
            uint16_t offset,
            uint16_t count,
            bool * values)
```

Wrapper for ModbusClient::readDiscreteInputsAsBoolArray

#### 8.1.3.61 cReadExceptionStatus()

```
MODBUS_EXPORT StatusCode cReadExceptionStatus (
            cModbusClient client,
            uint8_t * value)
```

Wrapper for ModbusClient::readExceptionStatus

#### 8.1.3.62 cReadHoldingRegisters()

```
MODBUS_EXPORT StatusCode cReadHoldingRegisters (
            cModbusClient client,
            uint16_t offset,
            uint16_t count,
            uint16_t * values)
```

Wrapper for ModbusClient::readHoldingRegisters

#### 8.1.3.63 cReadInputRegisters()

```
MODBUS_EXPORT StatusCode cReadInputRegisters (
            cModbusClient client,
            uint16_t offset,
            uint16_t count,
            uint16_t * values)
```

Wrapper for ModbusClient::readInputRegisters

#### 8.1.3.64 cReadWriteMultipleRegisters()

```
MODBUS_EXPORT StatusCode cReadWriteMultipleRegisters (
            cModbusClient client,
            uint16_t readOffset,
            uint16_t readCount,
            uint16_t * readValues,
            uint16_t writeOffset,
            uint16_t writeCount,
            const uint16_t * writeValues)
```

Wrapper for ModbusClient::readWriteMultipleRegisters

#### 8.1.3.65 cSpoClose()

```
MODBUS_EXPORT StatusCode cSpoClose (
            cModbusServerPort serverPort)
```

Wrapper for ModbusServerPort::close

#### 8.1.3.66 cSpoConnectCloseConnection()

```
MODBUS_EXPORT void cSpoConnectCloseConnection (
            cModbusServerPort serverPort,
            pfSlotCloseConnection funcPtr)
```

Connects funcPtr-function to ModbusServerPort::signalCloseConnection signal

#### 8.1.3.67 cSpoConnectClosed()

```
MODBUS_EXPORT void cSpoConnectClosed (
            cModbusServerPort serverPort,
            pfSlotClosed funcPtr)
```

Connects funcPtr-function to ModbusServerPort::signalClosed signal

#### 8.1.3.68 cSpoConnectError()

```
MODBUS_EXPORT void cSpoConnectError (
            cModbusServerPort serverPort,
            pfSlotError funcPtr)
```

Connects `funcPtr`-function to `ModbusServerPort::signalError` signal

#### 8.1.3.69 cSpoConnectNewConnection()

```
MODBUS_EXPORT void cSpoConnectNewConnection (
            cModbusServerPort serverPort,
            pfSlotNewConnection funcPtr)
```

Connects `funcPtr`-function to `ModbusServerPort::signalNewConnection` signal

#### 8.1.3.70 cSpoConnectOpened()

```
MODBUS_EXPORT void cSpoConnectOpened (
            cModbusServerPort serverPort,
            pfSlotOpened funcPtr)
```

Connects `funcPtr`-function to `ModbusServerPort::signalOpened` signal

#### 8.1.3.71 cSpoConnectRx()

```
MODBUS_EXPORT void cSpoConnectRx (
            cModbusServerPort serverPort,
            pfSlotRx funcPtr)
```

Connects `funcPtr`-function to `ModbusServerPort::signalRx` signal

#### 8.1.3.72 cSpoConnectTx()

```
MODBUS_EXPORT void cSpoConnectTx (
            cModbusServerPort serverPort,
            pfSlotTx funcPtr)
```

Connects `funcPtr`-function to `ModbusServerPort::signalTx` signal

#### 8.1.3.73 cSpoCreate()

```
MODBUS_EXPORT cModbusServerPort cSpoCreate (
            cModbusInterface device,
            ProtocolType type,
            const void * settings,
            bool blocking)
```

Creates `ModbusServerPort` object and returns handle to it.

**See also**

`Modbus::createServerPort`

#### 8.1.3.74 cSpoDelete()

```
MODBUS_EXPORT void cSpoDelete (
            cModbusServerPort serverPort)
```

Deletes previously created ModbusServerPort object represented by serverPort handle

#### 8.1.3.75 cSpoDisconnectFunc()

```
MODBUS_EXPORT void cSpoDisconnectFunc (
            cModbusServerPort serverPort,
            void * funcPtr)
```

Disconnects funcPtr-function from ModbusServerPort

#### 8.1.3.76 cSpoGetDevice()

```
MODBUS_EXPORT cModbusInterface cSpoGetDevice (
            cModbusServerPort serverPort)
```

Wrapper for ModbusServerPort::device

#### 8.1.3.77 cSpoGetObjectName()

```
MODBUS_EXPORT const Char * cSpoGetObjectName (
            cModbusServerPort serverPort)
```

Wrapper for ModbusServerPort::objectName

#### 8.1.3.78 cSpoGetType()

```
MODBUS_EXPORT ProtocolType cSpoGetType (
            cModbusServerPort serverPort)
```

Wrapper for ModbusServerPort::type

#### 8.1.3.79 cSpoIsOpen()

```
MODBUS_EXPORT bool cSpoIsOpen (
            cModbusServerPort serverPort)
```

Wrapper for ModbusServerPort::isOpen

#### 8.1.3.80 cSpoIsTcpServer()

```
MODBUS_EXPORT bool cSpoIsTcpServer (
            cModbusServerPort serverPort)
```

Wrapper for ModbusServerPort::isTcpServer

#### 8.1.3.81 cSpoOpen()

```
MODBUS_EXPORT StatusCode cSpoOpen (
            cModbusServerPort serverPort)
```

Wrapper for ModbusServerPort::open

#### 8.1.3.82 cSpoProcess()

```
MODBUS_EXPORT StatusCode cSpoProcess (
            cModbusServerPort serverPort)
```

Wrapper for ModbusServerPort::process

#### 8.1.3.83 cSpoSetObjectName()

```
MODBUS_EXPORT void cSpoSetObjectName (
            cModbusServerPort serverPort,
            const Char * name)
```

Wrapper for ModbusServerPort::setObjectName

#### 8.1.3.84 cWriteMultipleCoils()

```
MODBUS_EXPORT StatusCode cWriteMultipleCoils (
            cModbusClient client,
            uint16_t offset,
            uint16_t count,
            const void * values)
```

Wrapper for ModbusClient::writeMultipleCoils

#### 8.1.3.85 cWriteMultipleCoilsAsBoolArray()

```
MODBUS_EXPORT StatusCode cWriteMultipleCoilsAsBoolArray (
            cModbusClient client,
            uint16_t offset,
            uint16_t count,
            const bool * values)
```

Wrapper for ModbusClient::lastPortStatus

#### 8.1.3.86 cWriteMultipleRegisters()

```
MODBUS_EXPORT StatusCode cWriteMultipleRegisters (
            cModbusClient client,
            uint16_t offset,
            uint16_t count,
            const uint16_t * values)
```

Wrapper for ModbusClient::writeMultipleRegisters

#### 8.1.3.87 cWriteSingleCoil()

```
MODBUS_EXPORT StatusCode cWriteSingleCoil (
            cModbusClient client,
            uint16_t offset,
            bool value)
```

Wrapper for ModbusClient::writeSingleCoil

#### 8.1.3.88 cWriteSingleRegister()

```
MODBUS_EXPORT StatusCode cWriteSingleRegister (
            cModbusClient client,
            uint16_t offset,
            uint16_t value)
```

Wrapper for ModbusClient::writeSingleRegister

## 8.2 cModbus.h

Go to the documentation of this file.

```
00001
00008 #ifndef CMODBUS_H
00009 #define CMODBUS_H
00010
00011 #include <stdbool.h>
00012 #include "ModbusGlobal.h"
00013
00014 #ifdef __cplusplus
00015 using namespace Modbus;
00016 extern "C" {
00017 #endif
00018
00019 #ifdef __cplusplus
00020 class ModbusPort      ;
00021 class ModbusInterface ;
00022 class ModbusClientPort;
00023 class ModbusClient    ;
00024 class ModbusServerPort;
00025
00026 #else
00027 typedef struct ModbusPort       ModbusPort      ;
00028 typedef struct ModbusInterface  ModbusInterface ;
00029 typedef struct ModbusClientPort ModbusClientPort;
00030 typedef struct ModbusClient     ModbusClient    ;
00031 typedef struct ModbusServerPort ModbusServerPort;
00032 #endif
00033
00035 typedef ModbusPort* cModbusPort;
00036
00038 typedef ModbusClientPort* cModbusClientPort;
00039
00041 typedef ModbusClient* cModbusClient;
00042
00044 typedef ModbusServerPort* cModbusServerPort;
00045
00047 typedef ModbusInterface* cModbusInterface;
00048
00050 typedef void* cModbusDevice;
00051
00052 #ifndef MBF_READ_COILS_DISABLE
00054 typedef StatusCode (*pfReadCoils)(cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t count,
      void *values);
00055 #endif // MBF_READ_COILS_DISABLE
00056
00057 #ifndef MBF_READ_DISCRETE_INPUTS_DISABLE
00059 typedef StatusCode (*pfReadDiscreteInputs)(cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t
      count, void *values);
00060 #endif // MBF_READ_DISCRETE_INPUTS_DISABLE
00061
```

```
00062 #ifndef MBF_READ_HOLDING_REGISTERS_DISABLE
00064 typedef StatusCode (*pfReadHoldingRegisters)(cModbusDevice dev, uint8_t unit, uint16_t offset,
      uint16_t count, uint16_t *values);
00065 #endif // MBF_READ_HOLDING_REGISTERS_DISABLE
00066
00067 #ifndef MBF_READ_INPUT_REGISTERS_DISABLE
00069 typedef StatusCode (*pfReadInputRegisters)(cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t
      count, uint16_t *values);
00070 #endif // MBF_READ_INPUT_REGISTERS_DISABLE
00071
00072 #ifndef MBF_WRITE_SINGLE_COIL_DISABLE
00074 typedef StatusCode (*pfWriteSingleCoil)(cModbusDevice dev, uint8_t unit, uint16_t offset, bool value);
00075 #endif // MBF_WRITE_SINGLE_COIL_DISABLE
00076
00077 #ifndef MBF_WRITE_SINGLE_REGISTER_DISABLE
00079 typedef StatusCode (*pfWriteSingleRegister)(cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t
      value);
00080 #endif // MBF_WRITE_SINGLE_REGISTER_DISABLE
00081
00082 #ifndef MBF_READ_EXCEPTION_STATUS_DISABLE
00084 typedef StatusCode (*pfReadExceptionStatus)(cModbusDevice dev, uint8_t unit, uint8_t *status);
00085 #endif // MBF_READ_EXCEPTION_STATUS_DISABLE
00086
00087 #ifndef MBF_DIAGNOSTICS_DISABLE
00089 typedef StatusCode (*pfDiagnostics)(cModbusDevice dev, uint8_t unit, uint16_t subfunc, uint8_t insize,
      const uint8_t *indata, uint8_t *outsize, uint8_t *outdata);
00090 #endif // MBF_DIAGNOSTICS_DISABLE
00091
00092 #ifndef MBF_GET_COMM_EVENT_COUNTER_DISABLE
00094 typedef StatusCode (*pfGetCommEventCounter)(cModbusDevice dev, uint8_t unit, uint16_t *status,
      uint16_t *eventCount);
00095 #endif // MBF_GET_COMM_EVENT_COUNTER_DISABLE
00096
00097 #ifndef MBF_GET_COMM_EVENT_LOG_DISABLE
00099 typedef StatusCode (*pfGetCommEventLog)(cModbusDevice dev, uint8_t unit, uint16_t *status, uint16_t
      *eventCount, uint16_t *messageCount, uint8_t *eventBuffSize, uint8_t *eventBuff);
00100 #endif // MBF_GET_COMM_EVENT_LOG_DISABLE
00101
00102 #ifndef MBF_WRITE_MULTIPLE_COILS_DISABLE
00104 typedef StatusCode (*pfWriteMultipleCoils)(cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t
      count, const void *values);
00105 #endif // MBF_WRITE_MULTIPLE_COILS_DISABLE
00106
00107 #ifndef MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00109 typedef StatusCode (*pfWriteMultipleRegisters)(cModbusDevice dev, uint8_t unit, uint16_t offset,
      uint16_t count, const uint16_t *values);
00110 #endif // MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00111
00112 #ifndef MBF_REPORT_SERVER_ID_DISABLE
00114 typedef StatusCode (*pfReportServerID)(cModbusDevice dev, uint8_t unit, uint8_t *count, uint8_t
      *data);
00115 #endif // MBF_REPORT_SERVER_ID_DISABLE
00116
00117 #ifndef MBF_MASK_WRITE_REGISTER_DISABLE
00119 typedef StatusCode (*pfMaskWriteRegister)(cModbusDevice dev, uint8_t unit, uint16_t offset, uint16_t
      andMask, uint16_t orMask);
00120 #endif // MBF_MASK_WRITE_REGISTER_DISABLE
00121
00122 #ifndef MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00124 typedef StatusCode (*pfReadWriteMultipleRegisters)(cModbusDevice dev, uint8_t unit, uint16_t
      readOffset, uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const
      uint16_t *writeValues);
00125 #endif // MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00126
00127 #ifndef MBF_READ_FIFO_QUEUE_DISABLE
00129 typedef StatusCode (*pfReadFIFOQueue)(cModbusDevice dev, uint8_t unit, uint16_t fifoadr, uint16_t
      *count, uint16_t *values);
00130 #endif // MBF_READ_FIFO_QUEUE_DISABLE
00131
00133 typedef void (*pfSlotOpened)(const Char *source);
00134
00136 typedef void (*pfSlotClosed)(const Char *source);
00137
00139 typedef void (*pfSlotTx)(const Char *source, const uint8_t* buff, uint16_t size);
00140
00142 typedef void (*pfSlotRx)(const Char *source, const uint8_t* buff, uint16_t size);
00143
00145 typedef void (*pfSlotError)(const Char *source, StatusCode status, const Char *text);
00146
00148 typedef void (*pfSlotNewConnection)(const Char *source);
00149
00151 typedef void (*pfSlotCloseConnection)(const Char *source);
00152
00172 MODBUS_EXPORT cModbusInterface cCreateModbusDevice(cModbusDevice                device
00173 #ifndef MBF_READ_COILS_DISABLE
00174                                                    , pfReadCoils                 readCoils
00175 #endif // MBF_READ_COILS_DISABLE
```

```
00176 #ifndef MBF_READ_DISCRETE_INPUTS_DISABLE
00177                                                 , pfReadDiscreteInputs          readDiscreteInputs
00178 #endif // MBF_READ_DISCRETE_INPUTS_DISABLE
00179 #ifndef MBF_READ_HOLDING_REGISTERS_DISABLE
00180                                                 , pfReadHoldingRegisters        readHoldingRegisters
00181 #endif // MBF_READ_HOLDING_REGISTERS_DISABLE
00182 #ifndef MBF_READ_INPUT_REGISTERS_DISABLE
00183                                                 , pfReadInputRegisters          readInputRegisters
00184 #endif // MBF_READ_INPUT_REGISTERS_DISABLE
00185 #ifndef MBF_WRITE_SINGLE_COIL_DISABLE
00186                                                 , pfWriteSingleCoil             writeSingleCoil
00187 #endif // MBF_WRITE_SINGLE_COIL_DISABLE
00188 #ifndef MBF_WRITE_SINGLE_REGISTER_DISABLE
00189                                                 , pfWriteSingleRegister         writeSingleRegister
00190 #endif // MBF_WRITE_SINGLE_REGISTER_DISABLE
00191 #ifndef MBF_READ_EXCEPTION_STATUS_DISABLE
00192                                                 , pfReadExceptionStatus         readExceptionStatus
00193 #endif // MBF_READ_EXCEPTION_STATUS_DISABLE
00194 #ifndef MBF_DIAGNOSTICS_DISABLE
00195                                                 , pfDiagnostics                 diagnostics
00196 #endif // MBF_DIAGNOSTICS_DISABLE
00197 #ifndef MBF_GET_COMM_EVENT_COUNTER_DISABLE
00198                                                 , pfGetCommEventCounter         getCommEventCounter
00199 #endif // MBF_GET_COMM_EVENT_COUNTER_DISABLE
00200 #ifndef MBF_GET_COMM_EVENT_LOG_DISABLE
00201                                                 , pfGetCommEventLog             getCommEventLog
00202 #endif // MBF_GET_COMM_EVENT_LOG_DISABLE
00203 #ifndef MBF_WRITE_MULTIPLE_COILS_DISABLE
00204                                                 , pfWriteMultipleCoils          writeMultipleCoils
00205 #endif // MBF_WRITE_MULTIPLE_COILS_DISABLE
00206 #ifndef MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00207                                                 , pfWriteMultipleRegisters      writeMultipleRegisters
00208 #endif // MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00209 #ifndef MBF_REPORT_SERVER_ID_DISABLE
00210                                                 , pfReportServerID              reportServerID
00211 #endif // MBF_REPORT_SERVER_ID_DISABLE
00212 #ifndef MBF_MASK_WRITE_REGISTER_DISABLE
00213                                                 , pfMaskWriteRegister           maskWriteRegister
00214 #endif // MBF_MASK_WRITE_REGISTER_DISABLE
00215 #ifndef MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00216                                                 , pfReadWriteMultipleRegisters
      readWriteMultipleRegisters
00217 #endif // MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00218 #ifndef MBF_READ_FIFO_QUEUE_DISABLE
00219                                                 , pfReadFIFOQueue               readFIFOQueue
00220 #endif // MBF_READ_FIFO_QUEUE_DISABLE
00221                                                 );
00222
00223
00225 MODBUS_EXPORT void cDeleteModbusDevice(cModbusInterface dev);
00226
00227 //
      ------------------------------------------------------------------------------------------------------------
00228 // ---------------------------------------------- ModbusPort
      ----------------------------------------------
00229 //
      ------------------------------------------------------------------------------------------------------------
00230
00232 MODBUS_EXPORT cModbusPort cPortCreate(ProtocolType type, const void *settings, bool blocking);
00233
00235 MODBUS_EXPORT void cPortDelete(cModbusPort port);
00236
00237
00238 //
      ------------------------------------------------------------------------------------------------------------
00239 // ------------------------------------------- ModbusClientPort
      -------------------------------------------
00240 //
      ------------------------------------------------------------------------------------------------------------
00241 #ifndef MB_CLIENT_DISABLE
00242
00244 MODBUS_EXPORT cModbusClientPort cCpoCreate(ProtocolType type, const void *settings, bool blocking);
00245
00247 MODBUS_EXPORT cModbusClientPort cCpoCreateForPort(cModbusPort port);
00248
00250 MODBUS_EXPORT void cCpoDelete(cModbusClientPort clientPort);
00251
00253 MODBUS_EXPORT const Char *cCpoGetObjectName(cModbusClientPort clientPort);
00254
00256 MODBUS_EXPORT void cCpoSetObjectName(cModbusClientPort clientPort, const Char *name);
00257
00259 MODBUS_EXPORT ProtocolType cCpoGetType(cModbusClientPort clientPort);
00260
00262 MODBUS_EXPORT bool cCpoIsOpen(cModbusClientPort clientPort);
00263
00265 MODBUS_EXPORT bool cCpoClose(cModbusClientPort clientPort);
00266
```

```
00268 MODBUS_EXPORT uint32_t cCpoGetRepeatCount(cModbusClientPort clientPort);
00269
00271 MODBUS_EXPORT void cCpoSetRepeatCount(cModbusClientPort clientPort, uint32_t count);
00272
00273 #ifndef MBF_READ_COILS_DISABLE
00275 MODBUS_EXPORT StatusCode cCpoReadCoils(cModbusClientPort clientPort, uint8_t unit, uint16_t offset,
      uint16_t count, void *values);
00276 #endif // MBF_READ_COILS_DISABLE
00277
00278 #ifndef MBF_READ_DISCRETE_INPUTS_DISABLE
00280 MODBUS_EXPORT StatusCode cCpoReadDiscreteInputs(cModbusClientPort clientPort, uint8_t unit, uint16_t
      offset, uint16_t count, void *values);
00281 #endif // MBF_READ_DISCRETE_INPUTS_DISABLE
00282
00283 #ifndef MBF_READ_HOLDING_REGISTERS_DISABLE
00285 MODBUS_EXPORT StatusCode cCpoReadHoldingRegisters(cModbusClientPort clientPort, uint8_t unit, uint16_t
      offset, uint16_t count, uint16_t *values);
00286 #endif // MBF_READ_HOLDING_REGISTERS_DISABLE
00287
00288 #ifndef MBF_READ_INPUT_REGISTERS_DISABLE
00290 MODBUS_EXPORT StatusCode cCpoReadInputRegisters(cModbusClientPort clientPort, uint8_t unit, uint16_t
      offset, uint16_t count, uint16_t *values);
00291 #endif // MBF_READ_INPUT_REGISTERS_DISABLE
00292
00293 #ifndef MBF_WRITE_SINGLE_COIL_DISABLE
00295 MODBUS_EXPORT StatusCode cCpoWriteSingleCoil(cModbusClientPort clientPort, uint8_t unit, uint16_t
      offset, bool value);
00296 #endif // MBF_WRITE_SINGLE_COIL_DISABLE
00297
00298 #ifndef MBF_WRITE_SINGLE_REGISTER_DISABLE
00300 MODBUS_EXPORT StatusCode cCpoWriteSingleRegister(cModbusClientPort clientPort, uint8_t unit, uint16_t
      offset, uint16_t value);
00301 #endif // MBF_READ_EXCEPTION_STATUS_DISABLE
00302
00303 #ifndef MBF_DIAGNOSTICS_DISABLE
00305 MODBUS_EXPORT StatusCode cCpoReadExceptionStatus(cModbusClientPort clientPort, uint8_t unit, uint8_t
      *value);
00306 #endif // MBF_DIAGNOSTICS_DISABLE
00307
00308 #ifndef MBF_DIAGNOSTICS_DISABLE
00310 MODBUS_EXPORT StatusCode cCpoDiagnostics(cModbusClientPort clientPort, uint8_t unit, uint16_t subfunc,
      uint8_t insize, const uint8_t *indata, uint8_t *outsize, uint8_t *outdata);
00311 #endif // MBF_DIAGNOSTICS_DISABLE
00312
00313 #ifndef MBF_GET_COMM_EVENT_COUNTER_DISABLE
00315 MODBUS_EXPORT StatusCode cCpoGetCommEventCounter(cModbusClientPort clientPort, uint8_t unit, uint16_t
      *status, uint16_t *eventCount);
00316 #endif // MBF_GET_COMM_EVENT_COUNTER_DISABLE
00317
00318 #ifndef MBF_GET_COMM_EVENT_LOG_DISABLE
00320 MODBUS_EXPORT StatusCode cCpoGetCommEventLog(cModbusClientPort clientPort, uint8_t unit, uint16_t
      *status, uint16_t *eventCount, uint16_t *messageCount, uint8_t *eventBuffSize, uint8_t *eventBuff);
00321 #endif // MBF_GET_COMM_EVENT_LOG_DISABLE
00322
00323 #ifndef MBF_WRITE_MULTIPLE_COILS_DISABLE
00325 MODBUS_EXPORT StatusCode cCpoWriteMultipleCoils(cModbusClientPort clientPort, uint8_t unit, uint16_t
      offset, uint16_t count, const void *values);
00326 #endif // MBF_WRITE_MULTIPLE_COILS_DISABLE
00327
00328 #ifndef MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00330 MODBUS_EXPORT StatusCode cCpoWriteMultipleRegisters(cModbusClientPort clientPort, uint8_t unit,
      uint16_t offset, uint16_t count, const uint16_t *values);
00331 #endif // MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00332
00333 #ifndef MBF_REPORT_SERVER_ID_DISABLE
00335 MODBUS_EXPORT StatusCode cCpoReportServerID(cModbusClientPort clientPort, uint8_t unit, uint8_t
      *count, uint8_t *data);
00336 #endif // MBF_REPORT_SERVER_ID_DISABLE
00337
00338 #ifndef MBF_MASK_WRITE_REGISTER_DISABLE
00340 MODBUS_EXPORT StatusCode cCpoMaskWriteRegister(cModbusClientPort clientPort, uint8_t unit, uint16_t
      offset, uint16_t andMask, uint16_t orMask);
00341 #endif // MBF_MASK_WRITE_REGISTER_DISABLE
00342
00343 #ifndef MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00345 MODBUS_EXPORT StatusCode cCpoReadWriteMultipleRegisters(cModbusClientPort clientPort, uint8_t unit,
      uint16_t readOffset, uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t
      writeCount, const uint16_t *writeValues);
00346 #endif // MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00347
00348 #ifndef MBF_READ_FIFO_QUEUE_DISABLE
00350 MODBUS_EXPORT StatusCode cCpoReadFIFOQueue(cModbusClientPort clientPort, uint8_t unit, uint16_t
      fifoadr, uint16_t *count, uint16_t *values);
00351 #endif // MBF_READ_FIFO_QUEUE_DISABLE
00352
00353 #ifndef MBF_READ_COILS_DISABLE
00355 MODBUS_EXPORT StatusCode cCpoReadCoilsAsBoolArray(cModbusClientPort clientPort, uint8_t unit, uint16_t
```

```
    offset, uint16_t count, bool *values);
00356 #endif // MBF_READ_COILS_DISABLE
00357
00358 #ifndef MBF_READ_DISCRETE_INPUTS_DISABLE
00360 MODBUS_EXPORT StatusCode cCpoReadDiscreteInputsAsBoolArray(cModbusClientPort clientPort, uint8_t unit,
    uint16_t offset, uint16_t count, bool *values);
00361 #endif // MBF_READ_DISCRETE_INPUTS_DISABLE
00362
00363 #ifndef MBF_WRITE_MULTIPLE_COILS_DISABLE
00365 MODBUS_EXPORT StatusCode cCpoWriteMultipleCoilsAsBoolArray(cModbusClientPort clientPort, uint8_t unit,
    uint16_t offset, uint16_t count, const bool *values);
00366 #endif // MBF_WRITE_MULTIPLE_COILS_DISABLE
00367
00369 MODBUS_EXPORT StatusCode cCpoGetLastStatus(cModbusClientPort clientPort);
00370
00372 MODBUS_EXPORT StatusCode cCpoGetLastErrorStatus(cModbusClientPort clientPort);
00373
00375 MODBUS_EXPORT const Char *cCpoGetLastErrorText(cModbusClientPort clientPort);
00376
00378 MODBUS_EXPORT void cCpoConnectOpened(cModbusClientPort clientPort, pfSlotOpened funcPtr);
00379
00381 MODBUS_EXPORT void cCpoConnectClosed(cModbusClientPort clientPort, pfSlotClosed funcPtr);
00382
00384 MODBUS_EXPORT void cCpoConnectTx(cModbusClientPort clientPort, pfSlotTx funcPtr);
00385
00387 MODBUS_EXPORT void cCpoConnectRx(cModbusClientPort clientPort, pfSlotRx funcPtr);
00388
00390 MODBUS_EXPORT void cCpoConnectError(cModbusClientPort clientPort, pfSlotError funcPtr);
00391
00393 MODBUS_EXPORT void cCpoDisconnectFunc(cModbusClientPort clientPort, void *funcPtr);
00394
00395
00396 //
    ------------------------------------------------------------------------------------------------------------
00397 // ------------------------------------------ ModbusClient
    ------------------------------------------
00398 //
    ------------------------------------------------------------------------------------------------------------
00399
00401 MODBUS_EXPORT cModbusClient cCliCreate(uint8_t unit, ProtocolType type, const void *settings, bool
    blocking);
00402
00404 MODBUS_EXPORT cModbusClient cCliCreateForClientPort(uint8_t unit, cModbusClientPort clientPort);
00405
00407 MODBUS_EXPORT void cCliDelete(cModbusClient client);
00408
00410 MODBUS_EXPORT const Char *cCliGetObjectName(cModbusClient client);
00411
00413 MODBUS_EXPORT void cCliSetObjectName(cModbusClient client, const Char *name);
00414
00416 MODBUS_EXPORT ProtocolType cCliGetType(cModbusClient client);
00417
00419 MODBUS_EXPORT uint8_t cCliGetUnit(cModbusClient client);
00420
00422 MODBUS_EXPORT void cCliSetUnit(cModbusClient client, uint8_t unit);
00423
00425 MODBUS_EXPORT bool cCliIsOpen(cModbusClient client);
00426
00428 MODBUS_EXPORT cModbusClientPort cCliGetPort(cModbusClient client);
00429
00431 MODBUS_EXPORT StatusCode cReadCoils(cModbusClient client, uint16_t offset, uint16_t count, void
    *values);
00432
00434 MODBUS_EXPORT StatusCode cReadDiscreteInputs(cModbusClient client, uint16_t offset, uint16_t count,
    void *values);
00435
00437 MODBUS_EXPORT StatusCode cReadHoldingRegisters(cModbusClient client, uint16_t offset, uint16_t count,
    uint16_t *values);
00438
00440 MODBUS_EXPORT StatusCode cReadInputRegisters(cModbusClient client, uint16_t offset, uint16_t count,
    uint16_t *values);
00441
00443 MODBUS_EXPORT StatusCode cWriteSingleCoil(cModbusClient client, uint16_t offset, bool value);
00444
00446 MODBUS_EXPORT StatusCode cWriteSingleRegister(cModbusClient client, uint16_t offset, uint16_t value);
00447
00449 MODBUS_EXPORT StatusCode cReadExceptionStatus(cModbusClient client, uint8_t *value);
00450
00452 MODBUS_EXPORT StatusCode cWriteMultipleCoils(cModbusClient client, uint16_t offset, uint16_t count,
    const void *values);
00453
00455 MODBUS_EXPORT StatusCode cWriteMultipleRegisters(cModbusClient client, uint16_t offset, uint16_t
    count, const uint16_t *values);
00456
00458 MODBUS_EXPORT StatusCode cMaskWriteRegister(cModbusClient client, uint16_t offset, uint16_t andMask,
    uint16_t orMask);
00459
```

```
00461 MODBUS_EXPORT StatusCode cReadWriteMultipleRegisters(cModbusClient client, uint16_t readOffset,
      uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t
      *writeValues);
00462
00464 MODBUS_EXPORT StatusCode cReadCoilsAsBoolArray(cModbusClient client, uint16_t offset, uint16_t count,
      bool *values);
00465
00467 MODBUS_EXPORT StatusCode cReadDiscreteInputsAsBoolArray(cModbusClient client, uint16_t offset,
      uint16_t count, bool *values);
00468
00470 MODBUS_EXPORT StatusCode cWriteMultipleCoilsAsBoolArray(cModbusClient client, uint16_t offset,
      uint16_t count, const bool *values);
00471
00473 MODBUS_EXPORT StatusCode cCliGetLastPortStatus(cModbusClient client);
00474
00476 MODBUS_EXPORT StatusCode cCliGetLastPortErrorStatus(cModbusClient client);
00477
00479 MODBUS_EXPORT const Char *cCliGetLastPortErrorText(cModbusClient client);
00480
00481 #endif // MB_CLIENT_DISABLE
00482
00483 //
      ------------------------------------------------------------------------------------------------------------
00484 // ----------------------------------------- ModbusServerPort
      -----------------------------------------
00485 //
      ------------------------------------------------------------------------------------------------------------
00486
00487 #ifndef MB_SERVER_DISABLE
00488
00490 MODBUS_EXPORT cModbusServerPort cSpoCreate(cModbusInterface device, ProtocolType type, const void
      *settings, bool blocking);
00491
00493 MODBUS_EXPORT void cSpoDelete(cModbusServerPort serverPort);
00494
00496 MODBUS_EXPORT const Char *cSpoGetObjectName(cModbusServerPort serverPort);
00497
00499 MODBUS_EXPORT void cSpoSetObjectName(cModbusServerPort serverPort, const Char *name);
00500
00502 MODBUS_EXPORT ProtocolType cSpoGetType(cModbusServerPort serverPort);
00503
00505 MODBUS_EXPORT bool cSpoIsTcpServer(cModbusServerPort serverPort);
00506
00508 MODBUS_EXPORT cModbusInterface cSpoGetDevice(cModbusServerPort serverPort);
00509
00511 MODBUS_EXPORT bool cSpoIsOpen(cModbusServerPort serverPort);
00512
00514 MODBUS_EXPORT StatusCode cSpoOpen(cModbusServerPort serverPort);
00515
00517 MODBUS_EXPORT StatusCode cSpoClose(cModbusServerPort serverPort);
00518
00520 MODBUS_EXPORT StatusCode cSpoProcess(cModbusServerPort serverPort);
00521
00523 MODBUS_EXPORT void cSpoConnectOpened(cModbusServerPort serverPort, pfSlotOpened funcPtr);
00524
00526 MODBUS_EXPORT void cSpoConnectClosed(cModbusServerPort serverPort, pfSlotClosed funcPtr);
00527
00529 MODBUS_EXPORT void cSpoConnectTx(cModbusServerPort serverPort, pfSlotTx funcPtr);
00530
00532 MODBUS_EXPORT void cSpoConnectRx(cModbusServerPort serverPort, pfSlotRx funcPtr);
00533
00535 MODBUS_EXPORT void cSpoConnectError(cModbusServerPort serverPort, pfSlotError funcPtr);
00536
00538 MODBUS_EXPORT void cSpoConnectNewConnection(cModbusServerPort serverPort, pfSlotNewConnection
      funcPtr);
00539
00541 MODBUS_EXPORT void cSpoConnectCloseConnection(cModbusServerPort serverPort, pfSlotCloseConnection
      funcPtr);
00542
00544 MODBUS_EXPORT void cSpoDisconnectFunc(cModbusServerPort serverPort, void *funcPtr);
00545
00546 #endif // MB_SERVER_DISABLE
00547
00548 #ifdef __cplusplus
00549 } // extern "C"
00550 #endif
00551
00552 #endif // CMODBUS_H
```

## 8.3 c:/Users/march/Dropbox/PRJ/ModbusLib/src/Modbus.h File Reference

Contains general definitions of the Modbus protocol.

```
#include <string>
#include <list>
#include "ModbusGlobal.h"
```

### Classes

- class ModbusInterface

  *Main interface of Modbus communication protocol.*
- class Modbus::Address

  *Modbus Data Address class. Represents Modbus Data Address.*

### Namespaces

- namespace Modbus

  *Main Modbus namespace. Contains classes, functions and constants to work with Modbus-protocol.*

### Macros

- #define **sIEC61131Prefix0x** "%Q"
- #define **sIEC61131Prefix1x** "%I"
- #define **sIEC61131Prefix3x** "%IW"
- #define **sIEC61131Prefix4x** "%MW"
- #define **cIEC61131SuffixHex** 'h'

### Typedefs

- typedef std::string **Modbus::String**

  *Modbus::String class for strings.*
- template<class T >
  using **Modbus::List** = std::list<T>

  *Modbus::List template class.*

### Functions

- MODBUS_EXPORT String Modbus::getLastErrorText ()
- MODBUS_EXPORT String Modbus::trim (const String &str)
- template<class StringT , class T >
  StringT Modbus::toBinString (T value)
- template<class StringT , class T >
  StringT Modbus::toOctString (T value)
- template<class StringT , class T >
  StringT Modbus::toHexString (T value)
- template<class StringT , class T >
  StringT Modbus::toDecString (T value)
- template<class StringT , class T >
  StringT Modbus::toDecString (T value, int c, char fillChar='0')
- template<typename StringT >
  bool Modbus::startsWith (const StringT &s, const char *prefix)
- int Modbus::decDigitValue (int ch)
- int Modbus::hexDigitValue (int ch)
- String Modbus::toModbusString (int val)
- MODBUS_EXPORT String Modbus::bytesToString (const uint8_t *buff, uint32_t count)
- MODBUS_EXPORT String Modbus::asciiToString (const uint8_t *buff, uint32_t count)
- MODBUS_EXPORT List< String > Modbus::availableSerialPorts ()
- MODBUS_EXPORT List< int32_t > Modbus::availableBaudRate ()
- MODBUS_EXPORT List< int8_t > Modbus::availableDataBits ()
- MODBUS_EXPORT List< Parity > Modbus::availableParity ()
- MODBUS_EXPORT List< StopBits > Modbus::availableStopBits ()
- MODBUS_EXPORT List< FlowControl > Modbus::availableFlowControl ()
- MODBUS_EXPORT ModbusPort * Modbus::createPort (ProtocolType type, const void *settings, bool blocking)
- MODBUS_EXPORT ModbusClientPort * Modbus::createClientPort (ProtocolType type, const void *settings, bool blocking)
- MODBUS_EXPORT ModbusServerPort * Modbus::createServerPort (ModbusInterface *device, ProtocolType type, const void *settings, bool blocking)
- StatusCode Modbus::readMemRegs (uint32_t offset, uint32_t count, void *values, const void *memBuff, uint32_t memRegCount)
- StatusCode Modbus::writeMemRegs (uint32_t offset, uint32_t count, const void *values, void *memBuff, uint32_t memRegCount)
- StatusCode Modbus::readMemBits (uint32_t offset, uint32_t count, void *values, const void *memBuff, uint32_t memBitCount)
- StatusCode Modbus::writeMemBits (uint32_t offset, uint32_t count, const void *values, void *memBuff, uint32_t memBitCount)

## 8.3.1 Detailed Description

Contains general definitions of the Modbus protocol.

**Author**

serhmarch

**Date**

May 2024

## 8.4 Modbus.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUS_H
00009 #define MODBUS_H
00010
00011 #include <string>
00012 #include <list>
00013
00014 #include "ModbusGlobal.h"
00015
00016 class ModbusPort;
00017 class ModbusClientPort;
00018 class ModbusServerPort;
00019
00020 //
      ------------------------------------------------------------------------------------------------------------
00021 // ------------------------------------------ Modbus interface
      ------------------------------------------
00022 //
      ------------------------------------------------------------------------------------------------------------
00023
00047 class MODBUS_EXPORT ModbusInterface
00048 {
00049 public:
00050
00051 #ifndef MBF_READ_COILS_DISABLE
00058     virtual Modbus::StatusCode readCoils(uint8_t unit, uint16_t offset, uint16_t count, void *values);
00059 #endif // MBF_READ_COILS_DISABLE
00060
00061 #ifndef MBF_READ_DISCRETE_INPUTS_DISABLE
00068     virtual Modbus::StatusCode readDiscreteInputs(uint8_t unit, uint16_t offset, uint16_t count, void
      *values);
00069 #endif // MBF_READ_DISCRETE_INPUTS_DISABLE
00070
00071 #ifndef MBF_READ_HOLDING_REGISTERS_DISABLE
00078     virtual Modbus::StatusCode readHoldingRegisters(uint8_t unit, uint16_t offset, uint16_t count,
      uint16_t *values);
00079 #endif // MBF_READ_HOLDING_REGISTERS_DISABLE
00080
00081 #ifndef MBF_READ_INPUT_REGISTERS_DISABLE
00088     virtual Modbus::StatusCode readInputRegisters(uint8_t unit, uint16_t offset, uint16_t count,
      uint16_t *values);
00089 #endif // MBF_READ_INPUT_REGISTERS_DISABLE
00090
00091 #ifndef MBF_WRITE_SINGLE_COIL_DISABLE
00097     virtual Modbus::StatusCode writeSingleCoil(uint8_t unit, uint16_t offset, bool value);
00098 #endif // MBF_WRITE_SINGLE_COIL_DISABLE
00099
00100 #ifndef MBF_WRITE_SINGLE_REGISTER_DISABLE
00106     virtual Modbus::StatusCode writeSingleRegister(uint8_t unit, uint16_t offset, uint16_t value);
00107 #endif // MBF_WRITE_SINGLE_REGISTER_DISABLE
00108
00109 #ifndef MBF_READ_EXCEPTION_STATUS_DISABLE
00114     virtual Modbus::StatusCode readExceptionStatus(uint8_t unit, uint8_t *status);
00115 #endif // MBF_READ_EXCEPTION_STATUS_DISABLE
00116
00117 #ifndef MBF_DIAGNOSTICS_DISABLE
00127     virtual Modbus::StatusCode diagnostics(uint8_t unit, uint16_t subfunc, uint8_t insize, const
      uint8_t *indata, uint8_t *outsize, uint8_t *outdata);
00128 #endif // MBF_DIAGNOSTICS_DISABLE
00129
00130 #ifndef MBF_GET_COMM_EVENT_COUNTER_DISABLE
00136     virtual Modbus::StatusCode getCommEventCounter(uint8_t unit, uint16_t *status, uint16_t
      *eventCount);
00137 #endif // MBF_GET_COMM_EVENT_COUNTER_DISABLE
00138
00139 #ifndef MBF_GET_COMM_EVENT_LOG_DISABLE
00148     virtual Modbus::StatusCode getCommEventLog(uint8_t unit, uint16_t *status, uint16_t *eventCount,
      uint16_t *messageCount, uint8_t *eventBuffSize, uint8_t *eventBuff);
00149 #endif // MBF_GET_COMM_EVENT_LOG_DISABLE
00150
00151 #ifndef MBF_WRITE_MULTIPLE_COILS_DISABLE
00159     virtual Modbus::StatusCode writeMultipleCoils(uint8_t unit, uint16_t offset, uint16_t count, const
      void *values);
00160 #endif // MBF_WRITE_MULTIPLE_COILS_DISABLE
00161
00162 #ifndef MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00169     virtual Modbus::StatusCode writeMultipleRegisters(uint8_t unit, uint16_t offset, uint16_t count,
      const uint16_t *values);
00170 #endif // MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00171
00172 #ifndef MBF_REPORT_SERVER_ID_DISABLE
00178     virtual Modbus::StatusCode reportServerID(uint8_t unit, uint8_t *count, uint8_t *data);
```

```
00179 #endif // MBF_REPORT_SERVER_ID_DISABLE
00180
00181 #ifndef MBF_MASK_WRITE_REGISTER_DISABLE
00191     virtual Modbus::StatusCode maskWriteRegister(uint8_t unit, uint16_t offset, uint16_t andMask,
      uint16_t orMask);
00192 #endif // MBF_MASK_WRITE_REGISTER_DISABLE
00193
00194 #ifndef MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00204     virtual Modbus::StatusCode readWriteMultipleRegisters(uint8_t unit, uint16_t readOffset, uint16_t
      readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t
      *writeValues);
00205 #endif // MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00206
00207 #ifndef MBF_READ_FIFO_QUEUE_DISABLE
00214     virtual Modbus::StatusCode readFIFOQueue(uint8_t unit, uint16_t fifoadr, uint16_t *count, uint16_t
      *values);
00215 #endif // MBF_READ_FIFO_QUEUE_DISABLE
00216
00217 };
00218
00219 //
      ------------------------------------------------------------------------------------------------------------------------
00220 // ------------------------------------------ Modbus namespace
      ------------------------------------------
00221 //
      ------------------------------------------------------------------------------------------------------------------------

00222
00224 namespace Modbus {
00225
00227 typedef std::string String;
00228
00230 template <class T>
00231 using List = std::list<T>;
00232
00234 MODBUS_EXPORT String getLastErrorText();
00235
00237 MODBUS_EXPORT String trim(const String &str);
00238
00240 template<class StringT, class T>
00241 StringT toBinString(T value)
00242 {
00243     int c = sizeof(value) * MB_BYTE_SZ_BITES;
00244     StringT res(c, '0');
00245     while (value)
00246     {
00247         res[--c] = '0' + static_cast<char>(value & 1);
00248         value >>= 1;
00249     }
00250     return res;
00251 }
00252
00254 template<class StringT, class T>
00255 StringT toOctString(T value)
00256 {
00257     int c = (sizeof(value) * MB_BYTE_SZ_BITES + 2) / 3;
00258     StringT res(c, '0');
00259     while (value)
00260     {
00261         res[--c] = '0' + static_cast<char>(value & 7);
00262         value >>= 3;
00263     }
00264     return res;
00265 }
00266
00268 template<class StringT, class T>
00269 StringT toHexString(T value)
00270 {
00271     int c = sizeof(value) * 2;
00272     StringT res(c, '0');
00273     T v;
00274     while (value)
00275     {
00276         v = value & 0xF;
00277         if (v < 10)
00278             res[--c] = '0' + static_cast<char>(v);
00279         else
00280             res[--c] = 'A' - 10 + static_cast<char>(v);
00281         value >>= 4;
00282     }
00283     return res;
00284 }
00285
00287 template<class StringT, class T>
00288 StringT toDecString(T value)
00289 {
00290     using CharT = typename StringT::value_type;
```

```
00291     const size_t sz = sizeof(T)*3+1;
00292     CharT buffer[sz];
00293     buffer[sz-1] = '\0';
00294     CharT *ptr = &buffer[sz-1];
00295     do
00296     {
00297         --ptr;
00298         T v = value % 10;
00299         ptr[0] = ('0' + static_cast<char>(v));
00300         value /= 10;
00301     }
00302     while (value);
00303     return StringT(ptr);
00304 }
00305
00308 template<class StringT, class T>
00309 StringT toDecString(T value, int c, char fillChar = '0')
00310 {
00311     StringT res(c, fillChar);
00312     do
00313     {
00314         T v = value % 10;
00315         res[--c] = ('0' + static_cast<char>(v));
00316         value /= 10;
00317     }
00318     while (value && c);
00319     return res;
00320 }
00321
00323 template <typename StringT>
00324 bool startsWith(const StringT& s, const char* prefix)
00325 {
00326     if (!prefix)
00327         return false;
00328
00329     using CharT = typename StringT::value_type;
00330
00331     size_t prefixLen = std::char_traits<char>::length(prefix);
00332     if (prefixLen > static_cast<size_t>(s.size()))
00333         return false;
00334
00335     auto it = s.begin();
00336     for (size_t i = 0; i < prefixLen; ++i, ++it)
00337     {
00338         if (static_cast<CharT>(prefix[i]) != *it)
00339             return false;
00340     }
00341     return true;
00342 }
00343
00346 inline int decDigitValue(int ch)
00347 {
00348     switch (ch)
00349     {
00350     case '0':case '1':case '2':case '3':case '4':case '5':case '6':case '7':case '8':case '9':
00351         return ch-'0';
00352     default:
00353         return -1;
00354     }
00355 }
00356
00359 inline int hexDigitValue(int ch)
00360 {
00361     switch (ch)
00362     {
00363     case '0':case '1':case '2':case '3':case '4':case '5':case '6':case '7':case '8':case '9':
00364         return ch-'0';
00365     case 'A':case 'B':case 'C':case 'D':case 'E':case 'F':
00366         return ch-'A'+10;
00367     case 'a':case 'b':case 'c':case 'd':case 'e':case 'f':
00368         return ch-'a'+10;
00369     default:
00370         return -1;
00371     }
00372 }
00373
00374 #ifdef QT_CORE_LIB
00375
00377 inline int decDigitValue(QChar ch) { return decDigitValue(ch.toLatin1()); }
00378
00380 inline int hexDigitValue(QChar ch) { return hexDigitValue(ch.toLatin1()); }
00381
00382 #endif // QT_CORE_LIB
00383
00386 inline String toModbusString(int val) { return std::to_string(val); }
00387
00389 MODBUS_EXPORT String bytesToString(const uint8_t* buff, uint32_t count);
```

```
00390
00392 MODBUS_EXPORT String asciiToString(const uint8_t* buff, uint32_t count);
00393
00395 MODBUS_EXPORT List<String> availableSerialPorts();
00396
00398 MODBUS_EXPORT List<int32_t> availableBaudRate();
00399
00401 MODBUS_EXPORT List<int8_t> availableDataBits();
00402
00404 MODBUS_EXPORT List<Parity> availableParity();
00405
00407 MODBUS_EXPORT List<StopBits> availableStopBits();
00408
00410 MODBUS_EXPORT List<FlowControl> availableFlowControl();
00411
00416 MODBUS_EXPORT ModbusPort *createPort(ProtocolType type, const void *settings, bool blocking);
00417
00418 #ifndef MB_CLIENT_DISABLE
00423 MODBUS_EXPORT ModbusClientPort *createClientPort(ProtocolType type, const void *settings, bool
      blocking);
00424 #endif // MB_CLIENT_DISABLE
00425
00426 #ifndef MB_SERVER_DISABLE
00432 MODBUS_EXPORT ModbusServerPort *createServerPort(ModbusInterface *device, ProtocolType type, const
      void *settings, bool blocking);
00433 #endif // MB_SERVER_DISABLE
00434
00436 inline StatusCode readMemRegs(uint32_t offset, uint32_t count, void *values, const void *memBuff,
      uint32_t memRegCount)
00437 {
00438     return readMemRegs(offset, count , values, memBuff, memRegCount, nullptr);
00439 }
00440
00442 inline StatusCode writeMemRegs(uint32_t offset, uint32_t count, const void *values, void *memBuff,
      uint32_t memRegCount)
00443 {
00444     return writeMemRegs(offset, count , values, memBuff, memRegCount, nullptr);
00445 }
00446
00448 inline StatusCode readMemBits(uint32_t offset, uint32_t count, void *values, const void *memBuff,
      uint32_t memBitCount)
00449 {
00450     return readMemBits(offset, count , values, memBuff, memBitCount, nullptr);
00451 }
00452
00454 inline StatusCode writeMemBits(uint32_t offset, uint32_t count, const void *values, void *memBuff,
      uint32_t memBitCount)
00455 {
00456     return writeMemBits(offset, count , values, memBuff, memBitCount, nullptr);
00457 }
00458
00459
00460 #ifndef MB_ADDRESS_CLASS_DISABLE
00461
00462 #define sIEC61131Prefix0x "%Q"
00463 #define sIEC61131Prefix1x "%I"
00464 #define sIEC61131Prefix3x "%IW"
00465 #define sIEC61131Prefix4x "%MW"
00466 #define cIEC61131SuffixHex 'h'
00467
00469
00476 class Address
00477 {
00478 public:
00480     enum Notation
00481     {
00482         Notation_Default    ,
00483         Notation_Modbus     ,
00484         Notation_IEC61131   ,
00485         Notation_IEC61131Hex
00486     };
00487
00488 public:
00490     Address() : m_type(Modbus::Memory_Unknown), m_offset(0) {}
00491
00493     Address(Modbus::MemoryType type, uint16_t offset) : m_type(type), m_offset(offset) {}
00494
00497     Address(uint32_t adr) { this->operator=(adr); }
00498
00499 public:
00501     inline bool isValid() const { return m_type != Modbus::Memory_Unknown; }
00502
00504     inline Modbus::MemoryType type() const { return static_cast<Modbus::MemoryType>(m_type); }
00505
00507     inline uint16_t offset() const { return m_offset; }
00508
00510     inline void setOffset(uint16_t offset) { m_offset = offset; }
```

```
00511
00513     inline uint32_t number() const { return m_offset+1; }
00514
00516     inline void setNumber(uint16_t number) { m_offset = number-1; }
00517
00520     inline int toInt() const { return (m_type*100000) + number();  }
00521
00524     inline operator uint32_t () const { return (m_type*100000) + number();  }
00525
00527     inline Address &operator=(uint32_t v)
00528     {
00529         uint32_t number = v % 100000;
00530         if ((number < 1) || (number > 65536))
00531         {
00532             m_type = Modbus::Memory_Unknown;
00533             m_offset = 0;
00534             return *this;
00535         }
00536         uint16_t type = static_cast<uint16_t>(v/100000);
00537         switch(type)
00538         {
00539         case Modbus::Memory_0x:
00540         case Modbus::Memory_1x:
00541         case Modbus::Memory_3x:
00542         case Modbus::Memory_4x:
00543             m_type = type;
00544             m_offset = static_cast<uint16_t>(number-1);
00545             break;
00546         default:
00547             m_type = Modbus::Memory_Unknown;
00548             m_offset = 0;
00549             break;
00550         }
00551         return *this;
00552     }
00553
00555     inline Address& operator+= (uint16_t c) { m_offset += c; return *this; }
00556
00558     template<class StringT>
00559     static Address fromString(const StringT &s)
00560     {
00561         if (s.size() && s.at(0) == '%')
00562         {
00563             Address adr;
00564             decltype(s.size()) i;
00565             // Note: 3x (%IW) handled before 1x (%I)
00566             if (startsWith(s, sIEC61131Prefix3x)) // Check if string starts with sIEC61131Prefix3x
00567             {
00568                 adr.m_type = Modbus::Memory_3x;
00569                 i = sizeof(sIEC61131Prefix3x)-1;
00570             }
00571             else if (startsWith(s, sIEC61131Prefix4x)) // Check if string starts with
    sIEC61131Prefix4x
00572             {
00573                 adr.m_type = Modbus::Memory_4x;
00574                 i = sizeof(sIEC61131Prefix4x)-1;
00575             }
00576             else if (startsWith(s, sIEC61131Prefix0x)) // Check if string starts with
    sIEC61131Prefix0x
00577             {
00578                 adr.m_type = Modbus::Memory_0x;
00579                 i = sizeof(sIEC61131Prefix0x)-1;
00580             }
00581             else if (startsWith(s, sIEC61131Prefix1x)) // Check if string starts with
    sIEC61131Prefix1x
00582             {
00583                 adr.m_type = Modbus::Memory_1x;
00584                 i = sizeof(sIEC61131Prefix1x)-1;
00585             }
00586             else
00587                 return Address();
00588
00589             adr.m_offset = 0;
00590             auto suffix = s.back();
00591             if (suffix == cIEC61131SuffixHex)
00592             {
00593                 for (; i < s.size()-1; i++)
00594                 {
00595                     adr.m_offset *= 16;
00596                     int d = hexDigitValue(s.at(i));
00597                     if (d < 0)
00598                         return Address();
00599                     adr.m_offset += static_cast<uint16_t>(d);
00600                 }
00601             }
00602             else
00603             {
```

```
00604                 for (; i < s.size(); i++)
00605                 {
00606                     adr.m_offset *= 10;
00607                     int d = decDigitValue(s.at(i));
00608                     if (d < 0)
00609                         return Address();
00610                     adr.m_offset += static_cast<uint16_t>(d);
00611                 }
00612             }
00613             return adr;
00614         }
00615         uint32_t acc = 0;
00616         for (decltype(s.size()) i = 0; i < s.size(); i++)
00617         {
00618             acc *= 10;
00619             int d = decDigitValue(s.at(i));
00620             if (d < 0)
00621                 return Address();
00622             acc += static_cast<uint16_t>(d);
00623         }
00624         return Address(acc);
00625     }
00626
00629     template<class StringT>
00630     StringT toString(Notation notation) const
00631     {
00632         if (isValid())
00633         {
00634             switch (notation)
00635             {
00636             case Notation_IEC61131:
00637                 switch (m_type)
00638                 {
00639                 case Modbus::Memory_0x:
00640                     return StringT(sIEC61131Prefix0x) + toDecString<StringT>(offset());
00641                 case Modbus::Memory_1x:
00642                     return StringT(sIEC61131Prefix1x) + toDecString<StringT>(offset());
00643                 case Modbus::Memory_3x:
00644                     return StringT(sIEC61131Prefix3x) + toDecString<StringT>(offset());
00645                 case Modbus::Memory_4x:
00646                     return StringT(sIEC61131Prefix4x) + toDecString<StringT>(offset());
00647                 default:
00648                     return StringT();;
00649                 }
00650                 break;
00651             case Notation_IEC61131Hex:
00652             {
00653                 switch (m_type)
00654                 {
00655                 case Modbus::Memory_0x:
00656                     return StringT(sIEC61131Prefix0x) + toHexString<StringT>(offset()) +
      cIEC61131SuffixHex;
00657                 case Modbus::Memory_1x:
00658                     return StringT(sIEC61131Prefix1x) + toHexString<StringT>(offset()) +
      cIEC61131SuffixHex;
00659                 case Modbus::Memory_3x:
00660                     return StringT(sIEC61131Prefix3x) + toHexString<StringT>(offset()) +
      cIEC61131SuffixHex;
00661                 case Modbus::Memory_4x:
00662                     return StringT(sIEC61131Prefix4x) + toHexString<StringT>(offset()) +
      cIEC61131SuffixHex;
00663                 default:
00664                     return StringT();
00665                 }
00666             }
00667                 break;
00668             default:
00669                 return toDecString<StringT>(toInt(), 6);
00670             }
00671         }
00672         else
00673             return StringT();
00674     }
00675
00676 private:
00677     uint16_t m_type;
00678     uint16_t m_offset;
00679 };
00680
00681 #endif // MB_ADDRESS_CLASS_DISABLE
00682
00683 } //namespace Modbus
00684
00685 #endif // MODBUS_H
```

## 8.5 Modbus_config.h

```
00001 #ifndef MODBUS_CONFIG_H
00002 #define MODBUS_CONFIG_H
00003
00004 #define MODBUSLIB_VERSION_MAJOR 0
00005 #define MODBUSLIB_VERSION_MINOR 4
00006 #define MODBUSLIB_VERSION_PATCH 4
00007
00008 #define MB_DYNAMIC_LINKING
00009
00010 /* #undef MB_CLIENT_DISABLE */
00011 /* #undef MB_SERVER_DISABLE */
00012 /* #undef MB_C_SUPPORT_DISABLE */
00013
00014 /* #undef MBF_READ_COILS_DISABLE */
00015 /* #undef MBF_READ_DISCRETE_INPUTS_DISABLE */
00016 /* #undef MBF_READ_HOLDING_REGISTERS_DISABLE */
00017 /* #undef MBF_READ_INPUT_REGISTERS_DISABLE */
00018 /* #undef MBF_WRITE_SINGLE_COIL_DISABLE */
00019 /* #undef MBF_WRITE_SINGLE_REGISTER_DISABLE */
00020 /* #undef MBF_READ_EXCEPTION_STATUS_DISABLE */
00021 /* #undef MBF_DIAGNOSTICS_DISABLE */
00022 /* #undef MBF_GET_COMM_EVENT_COUNTER_DISABLE */
00023 /* #undef MBF_GET_COMM_EVENT_LOG_DISABLE */
00024 /* #undef MBF_WRITE_MULTIPLE_COILS_DISABLE */
00025 /* #undef MBF_WRITE_MULTIPLE_REGISTERS_DISABLE */
00026 /* #undef MBF_REPORT_SERVER_ID_DISABLE */
00027 /* #undef MBF_READ_FILE_RECORD_DISABLE */
00028 /* #undef MBF_WRITE_FILE_RECORD */
00029 /* #undef MBF_MASK_WRITE_REGISTER_DISABLE */
00030 /* #undef MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE */
00031 /* #undef MBF_READ_FIFO_QUEUE_DISABLE */
00032
00033 /* #undef MB_ADDRESS_CLASS_DISABLE */
00034
00035 #endif // MODBUS_CONFIG_H
```

## 8.6 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusAscPort.h File Reference

Contains definition of ASCII serial port class.

```
#include "ModbusSerialPort.h"
```

### Classes

- class ModbusAscPort

  *Implements ASCII version of the Modbus communication protocol.*

### 8.6.1 Detailed Description

Contains definition of ASCII serial port class.

Contains definition of base server side port class.

**Author**

serhmarch

**Date**

May 2024

## 8.7 ModbusAscPort.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSASCPORT_H
00009 #define MODBUSASCPORT_H
00010
00011 #include "ModbusSerialPort.h"
00012
00019 class MODBUS_EXPORT ModbusAscPort : public ModbusSerialPort
00020 {
00021 public:
00023     ModbusAscPort(bool blocking = false);
00024
00026     ~ModbusAscPort();
00027
00028 public:
00030     Modbus::ProtocolType type() const override { return Modbus::ASC; }
00031
00032 protected:
00033     Modbus::StatusCode writeBuffer(uint8_t unit, uint8_t func, uint8_t *buff, uint16_t szInBuff)
      override;
00034     Modbus::StatusCode readBuffer(uint8_t &unit, uint8_t &func, uint8_t *buff, uint16_t maxSzBuff,
      uint16_t *szOutBuff) override;
00035
00036 protected:
00037     using ModbusSerialPort::ModbusSerialPort;
00038 };
00039
00040 #endif // MODBUSASCPORT_H
```

## 8.8 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusClient.h File Reference

Header file of Modbus client.

```
#include "ModbusObject.h"
```

### Classes

- class ModbusClient

  *The `ModbusClient` class implements the interface of the client part of the Modbus protocol.*

### 8.8.1 Detailed Description

Header file of Modbus client.

**Author**

serhmarch

**Date**

May 2024

# 8.9 ModbusClient.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSCLIENT_H
00009 #define MODBUSCLIENT_H
00010
00011 #include "ModbusObject.h"
00012
00013 class ModbusClientPort;
00014
00024 class MODBUS_EXPORT ModbusClient : public ModbusObject
00025 {
00026 public:
00030     ModbusClient(uint8_t unit, ModbusClientPort *port);
00031
00032 public:
00034     Modbus::ProtocolType type() const;
00035
00037     uint8_t unit() const;
00038
00040     void setUnit(uint8_t unit);
00041
00043     bool isOpen() const;
00044
00046     ModbusClientPort *port() const;
00047
00048 public:
00049
00050 #ifndef MBF_READ_COILS_DISABLE
00052     Modbus::StatusCode readCoils(uint16_t offset, uint16_t count, void *values);
00053 #endif // MBF_READ_COILS_DISABLE
00054
00055 #ifndef MBF_READ_DISCRETE_INPUTS_DISABLE
00057     Modbus::StatusCode readDiscreteInputs(uint16_t offset, uint16_t count, void *values);
00058 #endif // MBF_READ_DISCRETE_INPUTS_DISABLE
00059
00060 #ifndef MBF_READ_HOLDING_REGISTERS_DISABLE
00062     Modbus::StatusCode readHoldingRegisters(uint16_t offset, uint16_t count, uint16_t *values);
00063 #endif // MBF_READ_HOLDING_REGISTERS_DISABLE
00064
00065 #ifndef MBF_READ_INPUT_REGISTERS_DISABLE
00067     Modbus::StatusCode readInputRegisters(uint16_t offset, uint16_t count, uint16_t *values);
00068 #endif // MBF_READ_INPUT_REGISTERS_DISABLE
00069
00070 #ifndef MBF_WRITE_SINGLE_COIL_DISABLE
00072     Modbus::StatusCode writeSingleCoil(uint16_t offset, bool value);
00073 #endif // MBF_WRITE_SINGLE_COIL_DISABLE
00074
00075 #ifndef MBF_WRITE_SINGLE_REGISTER_DISABLE
00077     Modbus::StatusCode writeSingleRegister(uint16_t offset, uint16_t value);
00078 #endif // MBF_WRITE_SINGLE_REGISTER_DISABLE
00079
00080 #ifndef MBF_READ_EXCEPTION_STATUS_DISABLE
00082     Modbus::StatusCode readExceptionStatus(uint8_t *value);
00083 #endif // MBF_READ_EXCEPTION_STATUS_DISABLE
00084
00085 #ifndef MBF_DIAGNOSTICS_DISABLE
00087     Modbus::StatusCode diagnostics(uint16_t subfunc, uint8_t insize, const uint8_t *indata, uint8_t
      *outsize, uint8_t *outdata);
00088 #endif // MBF_DIAGNOSTICS_DISABLE
00089
00090 #ifndef MBF_GET_COMM_EVENT_COUNTER_DISABLE
00092     Modbus::StatusCode getCommEventCounter(uint16_t *status, uint16_t *eventCount);
00093 #endif // MBF_GET_COMM_EVENT_COUNTER_DISABLE
00094
00095 #ifndef MBF_GET_COMM_EVENT_LOG_DISABLE
00097     Modbus::StatusCode getCommEventLog(uint16_t *status, uint16_t *eventCount, uint16_t *messageCount,
      uint8_t *eventBuffSize, uint8_t *eventBuff);
00098 #endif // MBF_GET_COMM_EVENT_LOG_DISABLE
00099
00100 #ifndef MBF_WRITE_MULTIPLE_COILS_DISABLE
00102     Modbus::StatusCode writeMultipleCoils(uint16_t offset, uint16_t count, const void *values);
00103 #endif // MBF_WRITE_MULTIPLE_COILS_DISABLE
00104
00105 #ifndef MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00107     Modbus::StatusCode writeMultipleRegisters(uint16_t offset, uint16_t count, const uint16_t
      *values);
00108 #endif // MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00109
00110 #ifndef MBF_REPORT_SERVER_ID_DISABLE
00112     Modbus::StatusCode reportServerID(uint8_t *count, uint8_t *data);
00113 #endif // MBF_REPORT_SERVER_ID_DISABLE
00114
00115 #ifndef MBF_MASK_WRITE_REGISTER_DISABLE
```

```
00117     Modbus::StatusCode maskWriteRegister(uint16_t offset, uint16_t andMask, uint16_t orMask);
00118 #endif // MBF_MASK_WRITE_REGISTER_DISABLE
00119
00120 #ifndef MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00122     Modbus::StatusCode readWriteMultipleRegisters(uint16_t readOffset, uint16_t readCount, uint16_t
      *readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t *writeValues);
00123 #endif // MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00124
00125 #ifndef MBF_READ_FIFO_QUEUE_DISABLE
00127     Modbus::StatusCode readFIFOQueue(uint16_t fifoadr, uint16_t *count, uint16_t *values);
00128 #endif // MBF_READ_FIFO_QUEUE_DISABLE
00129
00130 #ifndef MBF_READ_COILS_DISABLE
00132     Modbus::StatusCode readCoilsAsBoolArray(uint16_t offset, uint16_t count, bool *values);
00133 #endif // MBF_READ_COILS_DISABLE
00134
00135 #ifndef MBF_READ_DISCRETE_INPUTS_DISABLE
00137     Modbus::StatusCode readDiscreteInputsAsBoolArray(uint16_t offset, uint16_t count, bool *values);
00138 #endif // MBF_READ_DISCRETE_INPUTS_DISABLE
00139
00140 #ifndef MBF_WRITE_MULTIPLE_COILS_DISABLE
00142     Modbus::StatusCode writeMultipleCoilsAsBoolArray(uint16_t offset, uint16_t count, const bool
      *values);
00143 #endif // MBF_WRITE_MULTIPLE_COILS_DISABLE
00144
00145 public:
00147     Modbus::StatusCode lastPortStatus() const;
00148
00150     Modbus::StatusCode lastPortErrorStatus() const;
00151
00153     const Modbus::Char *lastPortErrorText() const;
00154
00155 protected:
00157     using ModbusObject::ModbusObject;
00159 };
00160
00161 #endif // MODBUSCLIENT_H
```

## 8.10 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusClientPort.h File Reference

General file of the algorithm of the client part of the Modbus protocol port.

```
#include "ModbusObject.h"
```

### Classes

- class ModbusClientPort

  *The `ModbusClientPort` class implements the algorithm of the client part of the Modbus communication protocol port.*

### 8.10.1 Detailed Description

General file of the algorithm of the client part of the Modbus protocol port.

**Author**

serhmarch

**Date**

May 2024

## 8.11 ModbusClientPort.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSCLIENTPORT_H
00009 #define MODBUSCLIENTPORT_H
00010
00011 #include "ModbusObject.h"
00012
00013 class ModbusPort;
00014
00054 class MODBUS_EXPORT ModbusClientPort : public ModbusObject, public ModbusInterface
00055 {
00056 public:
00059     enum RequestStatus
00060     {
00061         Enable,
00062         Disable,
00063         Process
00064     };
00065
00066 public:
00070     ModbusClientPort(ModbusPort *port);
00071
00072 public:
00074     Modbus::ProtocolType type() const;
00075
00077     ModbusPort *port() const;
00078
00080     void setPort(ModbusPort *port);
00081
00083     Modbus::StatusCode close();
00084
00086     bool isOpen() const;
00087
00089     uint32_t tries() const;
00090
00092     void setTries(uint32_t v);
00093
00095     inline uint32_t repeatCount() const { return tries(); }
00096
00098     inline void setRepeatCount(uint32_t v) { setTries(v); }
00099
00102     bool isBroadcastEnabled() const;
00103
00106     void setBroadcastEnabled(bool enable);
00107
00108 public: // Main interface
00109
00110 #ifndef MBF_READ_COILS_DISABLE
00112     Modbus::StatusCode readCoils(ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t count,
      void *values);
00113 #endif // MBF_READ_COILS_DISABLE
00114
00115 #ifndef MBF_READ_DISCRETE_INPUTS_DISABLE
00117     Modbus::StatusCode readDiscreteInputs(ModbusObject *client, uint8_t unit, uint16_t offset,
      uint16_t count, void *values);
00118 #endif // MBF_READ_DISCRETE_INPUTS_DISABLE
00119
00120 #ifndef MBF_READ_HOLDING_REGISTERS_DISABLE
00122     Modbus::StatusCode readHoldingRegisters(ModbusObject *client, uint8_t unit, uint16_t offset,
      uint16_t count, uint16_t *values);
00123 #endif // MBF_READ_HOLDING_REGISTERS_DISABLE
00124
00125 #ifndef MBF_READ_INPUT_REGISTERS_DISABLE
00127     Modbus::StatusCode readInputRegisters(ModbusObject *client, uint8_t unit, uint16_t offset,
      uint16_t count, uint16_t *values);
00128 #endif // MBF_READ_INPUT_REGISTERS_DISABLE
00129
00130 #ifndef MBF_WRITE_SINGLE_COIL_DISABLE
00132     Modbus::StatusCode writeSingleCoil(ModbusObject *client, uint8_t unit, uint16_t offset, bool
      value);
00133 #endif // MBF_WRITE_SINGLE_COIL_DISABLE
00134
00135 #ifndef MBF_WRITE_SINGLE_REGISTER_DISABLE
00137     Modbus::StatusCode writeSingleRegister(ModbusObject *client, uint8_t unit, uint16_t offset,
      uint16_t value);
00138 #endif // MBF_WRITE_SINGLE_REGISTER_DISABLE
00139
00140 #ifndef MBF_READ_EXCEPTION_STATUS_DISABLE
00142     Modbus::StatusCode readExceptionStatus(ModbusObject *client, uint8_t unit, uint8_t *value);
00143 #endif // MBF_READ_EXCEPTION_STATUS_DISABLE
00144
00145 #ifndef MBF_DIAGNOSTICS_DISABLE
00147     Modbus::StatusCode diagnostics(ModbusObject *client, uint8_t unit, uint16_t subfunc, uint8_t
      insize, const uint8_t *indata, uint8_t *outsize, uint8_t *outdata);
```

```
00148 #endif // MBF_DIAGNOSTICS_DISABLE
00149
00150 #ifndef MBF_GET_COMM_EVENT_COUNTER_DISABLE
00152     Modbus::StatusCode getCommEventCounter(ModbusObject *client, uint8_t unit, uint16_t *status,
      uint16_t *eventCount);
00153 #endif // MBF_GET_COMM_EVENT_COUNTER_DISABLE
00154
00155 #ifndef MBF_GET_COMM_EVENT_LOG_DISABLE
00157     Modbus::StatusCode getCommEventLog(ModbusObject *client, uint8_t unit, uint16_t *status, uint16_t
      *eventCount, uint16_t *messageCount, uint8_t *eventBuffSize, uint8_t *eventBuff);
00158 #endif // MBF_GET_COMM_EVENT_LOG_DISABLE
00159
00160 #ifndef MBF_WRITE_MULTIPLE_COILS_DISABLE
00162     Modbus::StatusCode writeMultipleCoils(ModbusObject *client, uint8_t unit, uint16_t offset,
      uint16_t count, const void *values);
00163 #endif // MBF_WRITE_MULTIPLE_COILS_DISABLE
00164
00165 #ifndef MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00167     Modbus::StatusCode writeMultipleRegisters(ModbusObject *client, uint8_t unit, uint16_t offset,
      uint16_t count, const uint16_t *values);
00168 #endif // MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00169
00170 #ifndef MBF_REPORT_SERVER_ID_DISABLE
00172     Modbus::StatusCode reportServerID(ModbusObject *client, uint8_t unit, uint8_t *count, uint8_t
      *data);
00173 #endif // MBF_REPORT_SERVER_ID_DISABLE
00174
00175 #ifndef MBF_MASK_WRITE_REGISTER_DISABLE
00177     Modbus::StatusCode maskWriteRegister(ModbusObject *client, uint8_t unit, uint16_t offset, uint16_t
      andMask, uint16_t orMask);
00178 #endif // MBF_MASK_WRITE_REGISTER_DISABLE
00179
00180 #ifndef MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00182     Modbus::StatusCode readWriteMultipleRegisters(ModbusObject *client, uint8_t unit, uint16_t
      readOffset, uint16_t readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const
      uint16_t *writeValues);
00183 #endif // MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00184
00185 #ifndef MBF_READ_FIFO_QUEUE_DISABLE
00187     Modbus::StatusCode readFIFOQueue(ModbusObject *client, uint8_t unit, uint16_t fifoadr, uint16_t
      *count, uint16_t *values);
00188 #endif // MBF_READ_FIFO_QUEUE_DISABLE
00189
00190 #ifndef MBF_READ_COILS_DISABLE
00192     Modbus::StatusCode readCoilsAsBoolArray(ModbusObject *client, uint8_t unit, uint16_t offset,
      uint16_t count, bool *values);
00193 #endif // MBF_READ_COILS_DISABLE
00194
00195 #ifndef MBF_READ_DISCRETE_INPUTS_DISABLE
00197     Modbus::StatusCode readDiscreteInputsAsBoolArray(ModbusObject *client, uint8_t unit, uint16_t
      offset, uint16_t count, bool *values);
00198 #endif // MBF_READ_DISCRETE_INPUTS_DISABLE
00199
00200 #ifndef MBF_WRITE_MULTIPLE_COILS_DISABLE
00202     Modbus::StatusCode writeMultipleCoilsAsBoolArray(ModbusObject *client, uint8_t unit, uint16_t
      offset, uint16_t count, const bool *values);
00203 #endif // MBF_WRITE_MULTIPLE_COILS_DISABLE
00204
00205 public: // Modbus Interface
00206
00207 #ifndef MBF_READ_COILS_DISABLE
00208     Modbus::StatusCode readCoils(uint8_t unit, uint16_t offset, uint16_t count, void *values)
      override;
00209 #endif // MBF_READ_COILS_DISABLE
00210
00211 #ifndef MBF_READ_DISCRETE_INPUTS_DISABLE
00212     Modbus::StatusCode readDiscreteInputs(uint8_t unit, uint16_t offset, uint16_t count, void *values)
      override;
00213 #endif // MBF_READ_DISCRETE_INPUTS_DISABLE
00214
00215 #ifndef MBF_READ_HOLDING_REGISTERS_DISABLE
00216     Modbus::StatusCode readHoldingRegisters(uint8_t unit, uint16_t offset, uint16_t count, uint16_t
      *values) override;
00217 #endif // MBF_READ_HOLDING_REGISTERS_DISABLE
00218
00219 #ifndef MBF_READ_INPUT_REGISTERS_DISABLE
00220     Modbus::StatusCode readInputRegisters(uint8_t unit, uint16_t offset, uint16_t count, uint16_t
      *values) override;
00221 #endif // MBF_READ_INPUT_REGISTERS_DISABLE
00222
00223 #ifndef MBF_WRITE_SINGLE_COIL_DISABLE
00224     Modbus::StatusCode writeSingleCoil(uint8_t unit, uint16_t offset, bool value) override;
00225 #endif // MBF_WRITE_SINGLE_COIL_DISABLE
00226
00227 #ifndef MBF_WRITE_SINGLE_REGISTER_DISABLE
00228     Modbus::StatusCode writeSingleRegister(uint8_t unit, uint16_t offset, uint16_t value) override;
00229 #endif // MBF_WRITE_SINGLE_REGISTER_DISABLE
```

```
00230
00231 #ifndef MBF_READ_EXCEPTION_STATUS_DISABLE
00232     Modbus::StatusCode readExceptionStatus(uint8_t unit, uint8_t *value) override;
00233 #endif // MBF_READ_EXCEPTION_STATUS_DISABLE
00234
00235 #ifndef MBF_DIAGNOSTICS_DISABLE
00236     Modbus::StatusCode diagnostics(uint8_t unit, uint16_t subfunc, uint8_t insize, const uint8_t
      *indata, uint8_t *outsize, uint8_t *outdata) override;
00237 #endif // MBF_DIAGNOSTICS_DISABLE
00238
00239 #ifndef MBF_GET_COMM_EVENT_COUNTER_DISABLE
00240     Modbus::StatusCode getCommEventCounter(uint8_t unit, uint16_t *status, uint16_t *eventCount)
      override;
00241 #endif // MBF_GET_COMM_EVENT_COUNTER_DISABLE
00242
00243 #ifndef MBF_GET_COMM_EVENT_LOG_DISABLE
00244     Modbus::StatusCode getCommEventLog(uint8_t unit, uint16_t *status, uint16_t *eventCount, uint16_t
      *messageCount, uint8_t *eventBuffSize, uint8_t *eventBuff) override;
00245 #endif // MBF_GET_COMM_EVENT_LOG_DISABLE
00246
00247 #ifndef MBF_WRITE_MULTIPLE_COILS_DISABLE
00248     Modbus::StatusCode writeMultipleCoils(uint8_t unit, uint16_t offset, uint16_t count, const void
      *values) override;
00249 #endif // MBF_WRITE_MULTIPLE_COILS_DISABLE
00250
00251 #ifndef MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00252     Modbus::StatusCode writeMultipleRegisters(uint8_t unit, uint16_t offset, uint16_t count, const
      uint16_t *values) override;
00253 #endif // MBF_WRITE_MULTIPLE_REGISTERS_DISABLE
00254
00255 #ifndef MBF_REPORT_SERVER_ID_DISABLE
00256     Modbus::StatusCode reportServerID(uint8_t unit, uint8_t *count, uint8_t *data) override;
00257 #endif // MBF_REPORT_SERVER_ID_DISABLE
00258
00259 #ifndef MBF_MASK_WRITE_REGISTER_DISABLE
00260     Modbus::StatusCode maskWriteRegister(uint8_t unit, uint16_t offset, uint16_t andMask, uint16_t
      orMask) override;
00261 #endif // MBF_MASK_WRITE_REGISTER_DISABLE
00262
00263 #ifndef MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00264     Modbus::StatusCode readWriteMultipleRegisters(uint8_t unit, uint16_t readOffset, uint16_t
      readCount, uint16_t *readValues, uint16_t writeOffset, uint16_t writeCount, const uint16_t
      *writeValues) override;
00265 #endif // MBF_READ_WRITE_MULTIPLE_REGISTERS_DISABLE
00266
00267 #ifndef MBF_READ_FIFO_QUEUE_DISABLE
00268     Modbus::StatusCode readFIFOQueue(uint8_t unit, uint16_t fifoadr, uint16_t *count, uint16_t
      *values) override;
00269 #endif // MBF_READ_FIFO_QUEUE_DISABLE
00270
00271 #ifndef MBF_READ_COILS_DISABLE
00273     inline Modbus::StatusCode readCoilsAsBoolArray(uint8_t unit, uint16_t offset, uint16_t count, bool
      *values) { return readCoilsAsBoolArray(this, unit, offset, count, values); }
00274 #endif // MBF_READ_COILS_DISABLE
00275
00276 #ifndef MBF_READ_DISCRETE_INPUTS_DISABLE
00278     inline Modbus::StatusCode readDiscreteInputsAsBoolArray(uint8_t unit, uint16_t offset, uint16_t
      count, bool *values) { return readDiscreteInputsAsBoolArray(this, unit, offset, count, values); }
00279 #endif // MBF_READ_DISCRETE_INPUTS_DISABLE
00280
00281 #ifndef MBF_WRITE_MULTIPLE_COILS_DISABLE
00283     inline Modbus::StatusCode writeMultipleCoilsAsBoolArray(uint8_t unit, uint16_t offset, uint16_t
      count, const bool *values) { return writeMultipleCoilsAsBoolArray(this, unit, offset, count, values);
      }
00284 #endif // MBF_WRITE_MULTIPLE_COILS_DISABLE
00285
00286 public:
00288     Modbus::StatusCode lastStatus() const;
00289
00291     Modbus::Timestamp lastStatusTimestamp() const;
00292
00294     Modbus::StatusCode lastErrorStatus() const;
00295
00297     const Modbus::Char *lastErrorText() const;
00298
00300     uint32_t lastTries() const;
00301
00303     inline uint32_t lastRepeatCount() const { return lastTries(); }
00304
00305 public:
00307     const ModbusObject *currentClient() const;
00308
00314     RequestStatus getRequestStatus(ModbusObject *client);
00315
00317     void cancelRequest(ModbusObject *client);
00318
00319 public: // SIGNALS
```

```
00321     void signalOpened(const Modbus::Char *source);
00322
00324     void signalClosed(const Modbus::Char *source);
00325
00327     void signalTx(const Modbus::Char *source, const uint8_t* buff, uint16_t size);
00328
00330     void signalRx(const Modbus::Char *source, const uint8_t* buff, uint16_t size);
00331
00333     void signalError(const Modbus::Char *source, Modbus::StatusCode status, const Modbus::Char *text);
00334
00335 private:
00336     Modbus::StatusCode request(uint8_t unit, uint8_t func, uint8_t *buff, uint16_t szInBuff, uint16_t
      maxSzBuff, uint16_t *szOutBuff);
00337     Modbus::StatusCode process();
00338     friend class ModbusClient;
00339 };
00340
00341 #endif // MODBUSCLIENTPORT_H
```

# 8.12 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusGlobal.h File Reference

Contains general definitions of the Modbus libarary (for C++ and "pure" C).

```
#include <stdint.h>
#include <string.h>
#include "ModbusPlatform.h"
#include "Modbus_config.h"
```

### Classes

- struct Modbus::SerialSettings

  *Struct to define settings for Serial Port.*
- struct Modbus::TcpSettings

  *Struct to define settings for TCP connection.*

### Namespaces

- namespace Modbus

  *Main Modbus namespace. Contains classes, functions and constants to work with Modbus-protocol.*

### Macros

- #define **MODBUSLIB_VERSION** ((MODBUSLIB_VERSION_MAJOR<<16)|(MODBUSLIB_VERSION_↩ MINOR<<8)|(MODBUSLIB_VERSION_PATCH))

  *ModbusLib version value that defines as MODBUSLIB_VERSION = (major << 16) + (minor << 8) + patch.*
- #define **MODBUSLIB_VERSION_STR** MODBUSLIB_VERSION_STR_MAKE(MODBUSLIB_VERSION_↩ MAJOR,MODBUSLIB_VERSION_MINOR,MODBUSLIB_VERSION_PATCH)

  *ModbusLib version value that defines as MODBUSLIB_VERSION_STR "major.minor.patch".*
- #define **MODBUS_EXPORT** MB_DECL_IMPORT

  *MODBUS_EXPORT defines macro for import/export functions and classes.*
- #define StringLiteral(cstr)

  *Macro for creating string literal, must be used like: `StringLiteral("Some string")`*
- #define CharLiteral(cchar)

*Macro for creating char literal, must be used like: 'CharLiteral('A')`.*

- #define GET_BIT(bitBuff, bitNum)

  *Macro for get bit with number `bitNum` from array `bitBuff`.*
- #define SET_BIT(bitBuff, bitNum, value)

  *Macro for set bit `value` with number `bitNum` to array `bitBuff`.*
- #define GET_BITS(bitBuff, bitNum, bitCount, boolBuff)

  *Macro for get bits begins with number `bitNum` with `count` from input bit array `bitBuff` to output bool array `boolBuff`.*
- #define SET_BITS(bitBuff, bitNum, bitCount, boolBuff)

  *Macro for set bits begins with number `bitNum` with `count` from input bool array `boolBuff`to output bit array `bitBuff`.*
- #define **MB_UNITMAP_SIZE** 32
- #define MB_UNITMAP_GET_BIT(unitmap, unit)
- #define MB_UNITMAP_SET_BIT(unitmap, unit, value)
- #define **MB_BYTE_SZ_BITES** 8

  *8 = count bits in byte (byte size in bits)*
- #define **MB_REGE_SZ_BITES** 16

  *16 = count bits in 16 bit register (register size in bits)*
- #define **MB_REGE_SZ_BYTES** 2

  *2 = count bytes in 16 bit register (register size in bytes)*
- #define **MB_MAX_BYTES** 255

  *255 - count_of_bytes in function readHoldingRegisters, readCoils etc*
- #define **MB_MAX_REGISTERS** 127

  *127 = 255(count_of_bytes in function readHoldingRegisters etc) / 2 (register size in bytes)*
- #define **MB_MAX_DISCRETS** 2040

  *2040 = 255(count_of_bytes in function readCoils etc) ∗ 8 (bits in byte)*
- #define **MB_VALUE_BUFF_SZ** 255

  *Same as `MB_MAX_BYTES`*
- #define MB_RTU_IO_BUFF_SZ 264

  *Maximum func data size: WriteMultipleCoils 261 = 1 byte(function) + 2 bytes (starting offset) + 2 bytes (count) + 1 bytes (byte count) + 255 bytes(maximum data length)*
- #define **MB_ASC_IO_BUFF_SZ** 529

  *1 byte(start symbol ':')+(( 1 byte(unit) + 261 (max func data size: WriteMultipleCoils)) + 1 byte(LRC) ))∗2+2 bytes(CR+LF)*
- #define **MB_TCP_IO_BUFF_SZ** 268

  *6 bytes(tcp-prefix)+1 byte(unit)+261 (max func data size: WriteMultipleCoils)*
- #define **GET_COMM_EVENT_LOG_MAX** 64

  *Maximum events for `GetCommEventLog` function.*
- #define **READ_FIFO_QUEUE_MAX** 31

  *Maximum events for `GetCommEventLog` function.*

### Modbus Functions

*Modbus Function's codes.*

- #define **MBF_READ_COILS** 1
- #define **MBF_READ_DISCRETE_INPUTS** 2
- #define **MBF_READ_HOLDING_REGISTERS** 3
- #define **MBF_READ_INPUT_REGISTERS** 4
- #define **MBF_WRITE_SINGLE_COIL** 5
- #define **MBF_WRITE_SINGLE_REGISTER** 6
- #define **MBF_READ_EXCEPTION_STATUS** 7
- #define **MBF_DIAGNOSTICS** 8
- #define **MBF_GET_COMM_EVENT_COUNTER** 11
- #define **MBF_GET_COMM_EVENT_LOG** 12

- #define **MBF_WRITE_MULTIPLE_COILS** 15
- #define **MBF_WRITE_MULTIPLE_REGISTERS** 16
- #define **MBF_REPORT_SERVER_ID** 17
- #define **MBF_READ_FILE_RECORD** 20
- #define **MBF_WRITE_FILE_RECORD** 21
- #define **MBF_MASK_WRITE_REGISTER** 22
- #define **MBF_READ_WRITE_MULTIPLE_REGISTERS** 23
- #define **MBF_READ_FIFO_QUEUE** 24
- #define **MBF_ENCAPSULATED_INTERFACE_TRANSPORT** 43
- #define **MBF_ILLEGAL_FUNCTION** 73
- #define **MBF_EXCEPTION** 128

### Typedefs

- typedef void * **Modbus::Handle**

  *Handle type for native OS values.*
- typedef char **Modbus::Char**

  *Type for Modbus character.*
- typedef uint32_t **Modbus::Timer**

  *Type for Modbus timer.*
- typedef int64_t **Modbus::Timestamp**

  *Type for Modbus timestamp (in UNIX millisec format)*
- typedef enum Modbus::_MemoryType **Modbus::MemoryType**

  *Defines type of memory used in Modbus protocol.*
- typedef enum Modbus::_Color **Modbus::Color**

  *Enum of color (used for console text color).*

### Enumerations

- enum Modbus::Constants { Modbus::VALID_MODBUS_ADDRESS_BEGIN = 1 , Modbus::VALID_MODBUS_ADDRESS_END = 247 , Modbus::STANDARD_TCP_PORT = 502 }

  *Define list of contants of Modbus protocol.*
- enum Modbus::_MemoryType {
  Modbus::Memory_Unknown = 0xFFFF , Modbus::Memory_0x = 0 , Modbus::Memory_Coils = Memory_0x ,
  Modbus::Memory_1x = 1 ,
  Modbus::Memory_DiscreteInputs = Memory_1x , Modbus::Memory_3x = 3 , Modbus::Memory_InputRegisters = Memory_3x , Modbus::Memory_4x = 4 ,
  Modbus::Memory_HoldingRegisters = Memory_4x }

  *Defines type of memory used in Modbus protocol.*
- enum Modbus::_Color {
  **Color_Black** , **Color_Red** , **Color_Green** , **Color_Yellow** ,
  **Color_Blue** , **Color_Magenta** , **Color_Cyan** , **Color_White** ,
  **Color_Default** }

  *Enum of color (used for console text color).*
- enum Modbus::StatusCode {
  Modbus::Status_Processing = 0x80000000 , Modbus::Status_Good = 0x00000000 , Modbus::Status_Bad = 0x01000000 , Modbus::Status_Uncertain = 0x02000000 ,
  Modbus::Status_BadIllegalFunction = Status_Bad | 0x01 , Modbus::Status_BadIllegalDataAddress = Status_Bad | 0x02 , Modbus::Status_BadIllegalDataValue = Status_Bad | 0x03 , Modbus::Status_BadServerDeviceFailure = Status_Bad | 0x04 ,
  Modbus::Status_BadAcknowledge = Status_Bad | 0x05 , Modbus::Status_BadServerDeviceBusy = Status↩
  _Bad | 0x06 , Modbus::Status_BadNegativeAcknowledge = Status_Bad | 0x07 , Modbus::Status_BadMemoryParityError = Status_Bad | 0x08 ,
  Modbus::Status_BadGatewayPathUnavailable = Status_Bad | 0x0A , Modbus::Status_BadGatewayTargetDeviceFailedToRespo

= Status_Bad | 0x0B , Modbus::Status_BadEmptyResponse = Status_Bad | 0x101 , Modbus::Status_BadNotCorrectRequest , Modbus::Status_BadNotCorrectResponse , Modbus::Status_BadWriteBufferOverflow , Modbus::Status_BadReadBufferOverflo , Modbus::Status_BadSerialOpen = Status_Bad | 0x201 , Modbus::Status_BadSerialWrite , Modbus::Status_BadSerialRead , Modbus::Status_BadSerialReadTimeout , Modbus::Status_BadSerialWriteTimeout , Modbus::Status_BadAscMissColon = Status_Bad | 0x301 , Modbus::Status_BadAscMissCrLf , Modbus::Status_BadAscChar , Modbus::Status_BadLrc , Modbus::Status_BadCrc = Status_Bad | 0x401 , Modbus::Status_BadTcpCreate = Status_Bad | 0x501 , Modbus::Status_BadTcpConnect , Modbus::Status_BadTcpWrite , Modbus::Status_BadTcpRead , Modbus::Status_BadTcpBind , Modbus::Status_BadTcpListen , Modbus::Status_BadTcpAccep , Modbus::Status_BadTcpDisconnect }

*Defines status of executed Modbus functions.*

- enum Modbus::ProtocolType { Modbus::ASC , Modbus::RTU , Modbus::TCP }

  *Defines type of Modbus protocol.*
- enum Modbus::Parity { Modbus::NoParity , Modbus::EvenParity , Modbus::OddParity , Modbus::SpaceParity , Modbus::MarkParity }

  *Defines Parity for serial port.*
- enum Modbus::StopBits { Modbus::OneStop , Modbus::OneAndHalfStop , Modbus::TwoStop }

  *Defines Stop Bits for serial port.*
- enum Modbus::FlowControl { Modbus::NoFlowControl , Modbus::HardwareControl , Modbus::SoftwareControl }

  *FlowControl Parity for serial port.*

## Functions

- bool Modbus::StatusIsProcessing (StatusCode status)
- bool Modbus::StatusIsGood (StatusCode status)
- bool Modbus::StatusIsBad (StatusCode status)
- bool Modbus::StatusIsUncertain (StatusCode status)
- bool Modbus::StatusIsStandardError (StatusCode status)
- bool Modbus::getBit (const void *bitBuff, uint16_t bitNum)
- bool Modbus::getBitS (const void *bitBuff, uint16_t bitNum, uint16_t maxBitCount)
- void Modbus::setBit (void *bitBuff, uint16_t bitNum, bool value)
- void Modbus::setBitS (void *bitBuff, uint16_t bitNum, bool value, uint16_t maxBitCount)
- bool * Modbus::getBits (const void *bitBuff, uint16_t bitNum, uint16_t bitCount, bool *boolBuff)
- bool * Modbus::getBitsS (const void *bitBuff, uint16_t bitNum, uint16_t bitCount, bool *boolBuff, uint16_t maxBitCount)
- void * Modbus::setBits (void *bitBuff, uint16_t bitNum, uint16_t bitCount, const bool *boolBuff)
- void * Modbus::setBitsS (void *bitBuff, uint16_t bitNum, uint16_t bitCount, const bool *boolBuff, uint16_t maxBitCount)
- MODBUS_EXPORT uint32_t Modbus::modbusLibVersion ()
- MODBUS_EXPORT const Char * Modbus::modbusLibVersionStr ()
- uint16_t Modbus::toModbusOffset (uint32_t adr)
- MODBUS_EXPORT uint16_t Modbus::crc16 (const uint8_t *byteArr, uint32_t count)
- MODBUS_EXPORT uint8_t Modbus::lrc (const uint8_t *byteArr, uint32_t count)
- MODBUS_EXPORT StatusCode Modbus::readMemRegs (uint32_t offset, uint32_t count, void *values, const void *memBuff, uint32_t memRegCount, uint32_t *outCount)
- MODBUS_EXPORT StatusCode Modbus::writeMemRegs (uint32_t offset, uint32_t count, const void *values, void *memBuff, uint32_t memRegCount, uint32_t *outCount)
- MODBUS_EXPORT StatusCode Modbus::readMemBits (uint32_t offset, uint32_t count, void *values, const void *memBuff, uint32_t memBitCount, uint32_t *outCount)

- MODBUS_EXPORT StatusCode Modbus::writeMemBits (uint32_t offset, uint32_t count, const void *values, void *memBuff, uint32_t memBitCount, uint32_t *outCount)
- MODBUS_EXPORT uint32_t Modbus::bytesToAscii (const uint8_t *bytesBuff, uint8_t *asciiBuff, uint32_t count)
- MODBUS_EXPORT uint32_t Modbus::asciiToBytes (const uint8_t *asciiBuff, uint8_t *bytesBuff, uint32_t count)
- MODBUS_EXPORT Char * Modbus::sbytes (const uint8_t *buff, uint32_t count, Char *str, uint32_t strmaxlen)
- MODBUS_EXPORT Char * Modbus::sascii (const uint8_t *buff, uint32_t count, Char *str, uint32_t strmaxlen)
- MODBUS_EXPORT const Char * Modbus::sprotocolType (ProtocolType type)
- MODBUS_EXPORT ProtocolType Modbus::toprotocolType (const Char *s)
- MODBUS_EXPORT const Char * Modbus::sbaudRate (int32_t baudRate)
- MODBUS_EXPORT int32_t Modbus::tobaudRate (const Char *s)
- MODBUS_EXPORT const Char * Modbus::sdataBits (int8_t dataBits)
- MODBUS_EXPORT int8_t Modbus::todataBits (const Char *s)
- MODBUS_EXPORT const Char * Modbus::sparity (Parity parity)
- MODBUS_EXPORT Parity Modbus::toparity (const Char *s)
- MODBUS_EXPORT const Char * Modbus::sstopBits (StopBits stopBits)
- MODBUS_EXPORT StopBits Modbus::tostopBits (const Char *s)
- MODBUS_EXPORT const Char * Modbus::sflowControl (FlowControl flowControl)
- MODBUS_EXPORT FlowControl Modbus::toflowControl (const Char *s)
- MODBUS_EXPORT Timer Modbus::timer ()
- MODBUS_EXPORT Timestamp Modbus::currentTimestamp ()
- MODBUS_EXPORT void Modbus::setConsoleColor (Color color)
- MODBUS_EXPORT void Modbus::msleep (uint32_t msec)

### 8.12.1 Detailed Description

Contains general definitions of the Modbus libarary (for C++ and "pure" C).

**Author**

serhmarch

**Date**

May 2024

### 8.12.2 Macro Definition Documentation

#### 8.12.2.1 CharLiteral

```
#define CharLiteral(
             cchar)
```

**Value:**

```
cchar
```

Macro for creating char literal, must be used like: ‘CharLiteral('A')`.

#### 8.12.2.2 GET_BIT

```
#define GET_BIT(
                bitBuff,
                bitNum)
```

**Value:**
```
((((const uint8_t*)(bitBuff))[(bitNum)/8] & (1<<((bitNum)%8))) != 0)
```

Macro for get bit with number `bitNum` from array `bitBuff`.

#### 8.12.2.3 GET_BITS

```
#define GET_BITS(
                bitBuff,
                bitNum,
                bitCount,
                boolBuff)
```

**Value:**
```
    for (uint16_t __i__ = 0; __i__ < bitCount; __i__++)
      \
        boolBuff[__i__] = (((const uint8_t*)(bitBuff))[((bitNum)+__i__)/8] & (1<<(((bitNum)+__i__)%8))) != 0;
```

Macro for get bits begins with number `bitNum` with `count` from input bit array `bitBuff` to output bool array `boolBuff`.

#### 8.12.2.4 MB_RTU_IO_BUFF_SZ

```
#define MB_RTU_IO_BUFF_SZ 264
```

Maximum func data size: WriteMultipleCoils 261 = 1 byte(function) + 2 bytes (starting offset) + 2 bytes (count) + 1 bytes (byte count) + 255 bytes(maximum data length)

1 byte(unit) + 261 (max func data size: WriteMultipleCoils) + 2 bytes(CRC)

#### 8.12.2.5 MB_UNITMAP_GET_BIT

```
#define MB_UNITMAP_GET_BIT(
                unitmap,
                unit)
```

**Value:**
```
((((const uint8_t*)(unitmap))[(unit)/8] & (1<<((unit)%8))) != 0)
```

#### 8.12.2.6 MB_UNITMAP_SET_BIT

```
#define MB_UNITMAP_SET_BIT(
                unitmap,
                unit,
                value)
```

**Value:**
```
    if (value)
      \
        ((uint8_t*)(unitmap))[(unit)/8] |= (1<<((unit)%8));
      \
    else
      \
        ((uint8_t*)(unitmap))[(unit)/8] &= (~(1<<((unit)%8)));
```

#### 8.12.2.7 SET_BIT

```
#define SET_BIT(
                bitBuff,
                bitNum,
                value)
```

**Value:**

```
    if (value)
      \
        ((uint8_t*)(bitBuff))[(bitNum)/8] |= (1<<((bitNum)%8));
      \
    else
      \
        ((uint8_t*)(bitBuff))[(bitNum)/8] &= (~(1<<((bitNum)%8)));
```

Macro for set bit `value` with number `bitNum` to array `bitBuff`.

#### 8.12.2.8 SET_BITS

```
#define SET_BITS(
                bitBuff,
                bitNum,
                bitCount,
                boolBuff)
```

**Value:**

```
    for (uint16_t __i__ = 0; __i__ < bitCount; __i__++)
      \
        if (boolBuff[__i__])
      \
            ((uint8_t*)(bitBuff))[((bitNum)+__i__)/8] |= (1<<(((bitNum)+__i__)%8));
      \
        else
      \
            ((uint8_t*)(bitBuff))[((bitNum)+__i__)/8] &= (~(1<<(((bitNum)+__i__)%8)));
```

Macro for set bits begins with number `bitNum` with `count` from input bool array `boolBuff`to output bit array `bitBuff`.

#### 8.12.2.9 StringLiteral

```
#define StringLiteral(
                cstr)
```

**Value:**

```
cstr
```

Macro for creating string literal, must be used like: `StringLiteral("Some string")`

## 8.13 ModbusGlobal.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSGLOBAL_H
00009 #define MODBUSGLOBAL_H
00010
00011 #include <stdint.h>
00012 #include <string.h>
00013
00014 #ifdef QT_CORE_LIB
00015 #include <qobjectdefs.h>
00016 #include <QString>
00017 #endif
00018
00019 #include "ModbusPlatform.h"
00020 #include "Modbus_config.h"
00021
00023 #define MODBUSLIB_VERSION
      ((MODBUSLIB_VERSION_MAJOR<<16)|(MODBUSLIB_VERSION_MINOR<<8)|(MODBUSLIB_VERSION_PATCH))
00024
00026 #define MODBUSLIB_VERSION_STR_HELPER(major,minor,patch) #major"."#minor"."#patch
00027
00028 #define MODBUSLIB_VERSION_STR_MAKE(major,minor,patch) MODBUSLIB_VERSION_STR_HELPER(major,minor,patch)
00030
00032 #define MODBUSLIB_VERSION_STR
      MODBUSLIB_VERSION_STR_MAKE(MODBUSLIB_VERSION_MAJOR,MODBUSLIB_VERSION_MINOR,MODBUSLIB_VERSION_PATCH)
00033
00034
00035
00037 #ifdef MB_DYNAMIC_LINKING
00038
00039 #if defined(MODBUS_EXPORTS) && defined(MB_DECL_EXPORT)
00040 #define MODBUS_EXPORT MB_DECL_EXPORT
00041 #elif defined(MB_DECL_IMPORT)
00042 #define MODBUS_EXPORT MB_DECL_IMPORT
00043 #else
00044 #define MODBUS_EXPORT
00045 #endif
00046
00047 #else // MB_DYNAMIC_LINKING
00048
00049 #define MODBUS_EXPORT
00050
00051 #endif // MB_DYNAMIC_LINKING
00052
00053
00054
00056 #define StringLiteral(cstr) cstr
00057
00059 #define CharLiteral(cchar) cchar
00060
00061 //
      ------------------------------------------------------------------------------------------------------------
00062 // ------------------------------------------- Helper macros
      ---------------------------------------------
00063 //
      ------------------------------------------------------------------------------------------------------------
00064
00066 #define GET_BIT(bitBuff, bitNum) ((((const uint8_t*)(bitBuff))[(bitNum)/8] & (1<<((bitNum)%8))) != 0)
00067
00069 #define SET_BIT(bitBuff, bitNum, value)
      \
00070     if (value)
      \
00071         ((uint8_t*)(bitBuff))[(bitNum)/8] |= (1<<((bitNum)%8));
      \
00072     else
      \
00073         ((uint8_t*)(bitBuff))[(bitNum)/8] &= (~(1<<((bitNum)%8)));
00074
00076 #define GET_BITS(bitBuff, bitNum, bitCount, boolBuff)
      \
00077     for (uint16_t __i__ = 0; __i__ < bitCount; __i__++)
      \
00078         boolBuff[__i__] = (((const uint8_t*)(bitBuff))[((bitNum)+__i__)/8] & (1<<(((bitNum)+__i__)%8)))
      != 0;
00079
00081 #define SET_BITS(bitBuff, bitNum, bitCount, boolBuff)
      \
00082     for (uint16_t __i__ = 0; __i__ < bitCount; __i__++)
      \
00083         if (boolBuff[__i__])
      \
00084             ((uint8_t*)(bitBuff))[((bitNum)+__i__)/8] |= (1<<(((bitNum)+__i__)%8));
      \
```

```
00085         else
      \
00086             ((uint8_t*)(bitBuff))[((bitNum)+__i__)/8] &= (~(1«(((bitNum)+__i__)%8)));
00087
00089 #define MB_UNITMAP_SIZE 32
00090
00092 #define MB_UNITMAP_GET_BIT(unitmap, unit) ((((const uint8_t*)(unitmap))[(unit)/8] & (1«((unit)%8))) !=
      0)
00093
00095 #define MB_UNITMAP_SET_BIT(unitmap, unit, value)
      \
00096     if (value)
      \
00097         ((uint8_t*)(unitmap))[(unit)/8] |= (1«((unit)%8));
      \
00098     else
      \
00099         ((uint8_t*)(unitmap))[(unit)/8] &= (~(1«((unit)%8)));
00100
00101 //
      ------------------------------------------------------------------------------------------------------------
00102 // ---------------------------------------- Modbus function codes
      ----------------------------------------
00103 //
      ------------------------------------------------------------------------------------------------------------
00104
00108 #define MBF_READ_COILS                          1
00109 #define MBF_READ_DISCRETE_INPUTS                2
00110 #define MBF_READ_HOLDING_REGISTERS              3
00111 #define MBF_READ_INPUT_REGISTERS                4
00112 #define MBF_WRITE_SINGLE_COIL                   5
00113 #define MBF_WRITE_SINGLE_REGISTER               6
00114 #define MBF_READ_EXCEPTION_STATUS               7
00115 #define MBF_DIAGNOSTICS                         8
00116 #define MBF_GET_COMM_EVENT_COUNTER              11
00117 #define MBF_GET_COMM_EVENT_LOG                  12
00118 #define MBF_WRITE_MULTIPLE_COILS                15
00119 #define MBF_WRITE_MULTIPLE_REGISTERS            16
00120 #define MBF_REPORT_SERVER_ID                    17
00121 #define MBF_READ_FILE_RECORD                    20
00122 #define MBF_WRITE_FILE_RECORD                   21
00123 #define MBF_MASK_WRITE_REGISTER                 22
00124 #define MBF_READ_WRITE_MULTIPLE_REGISTERS       23
00125 #define MBF_READ_FIFO_QUEUE                     24
00126 #define MBF_ENCAPSULATED_INTERFACE_TRANSPORT    43
00127 #define MBF_ILLEGAL_FUNCTION                    73
00128 #define MBF_EXCEPTION                           128
00130
00131
00132 //
      ------------------------------------------------------------------------------------------------------------
00133 // ---------------------------------------- Modbus count constants
      ----------------------------------------
00134 //
      ------------------------------------------------------------------------------------------------------------
00135
00137 #define MB_BYTE_SZ_BITES 8
00138
00140 #define MB_REGE_SZ_BITES 16
00141
00143 #define MB_REGE_SZ_BYTES 2
00144
00146 #define MB_MAX_BYTES 255
00147
00149 #define MB_MAX_REGISTERS 127
00150
00152 #define MB_MAX_DISCRETS 2040
00153
00155 #define MB_VALUE_BUFF_SZ 255
00156
00159
00161 #define MB_RTU_IO_BUFF_SZ 264
00162
00164 #define MB_ASC_IO_BUFF_SZ 529
00165
00167 #define MB_TCP_IO_BUFF_SZ 268
00168
00170 #define GET_COMM_EVENT_LOG_MAX 64
00171
00173 #define READ_FIFO_QUEUE_MAX 31
00174
00175 #ifdef __cplusplus
00176
00177 namespace Modbus {
00178
00179 #ifdef QT_CORE_LIB
00180 Q_NAMESPACE
```

```
00181 #endif
00182
00183 #endif // __cplusplus
00184
00186 typedef void* Handle;
00187
00189 typedef char Char;
00190
00192 typedef uint32_t Timer;
00193
00195 typedef int64_t Timestamp;
00196
00198 enum Constants
00199 {
00200     VALID_MODBUS_ADDRESS_BEGIN = 1   ,
00201     VALID_MODBUS_ADDRESS_END   = 247,
00202     STANDARD_TCP_PORT          = 502
00203 };
00204
00205 //=========== Modbus protocol types ==============
00206
00208 typedef enum _MemoryType
00209 {
00210     Memory_Unknown = 0xFFFF,
00211     Memory_0x = 0,
00212     Memory_Coils = Memory_0x,
00213     Memory_1x = 1,
00214     Memory_DiscreteInputs = Memory_1x,
00215     Memory_3x = 3,
00216     Memory_InputRegisters = Memory_3x,
00217     Memory_4x = 4,
00218     Memory_HoldingRegisters = Memory_4x,
00219 } MemoryType;
00220
00222 typedef enum _Color
00223 {                   //   Unix Color    Windows Color
00224     Color_Black  , // 30 = Black     0 = Black
00225     Color_Red    , // 31 = Red       4 = Red
00226     Color_Green  , // 32 = Green     2 = Green
00227     Color_Yellow , // 33 = Yellow    6 = Yellow
00228     Color_Blue   , // 34 = Blue      1 = Blue
00229     Color_Magenta, // 35 = Magenta   13 = Light Purple
00230     Color_Cyan   , // 36 = Cyan      9 = Light Blue
00231     Color_White  , // 37 = White     7 = White (default)
00232     Color_Default
00233 } Color;
00234
00235
00237 #ifdef __cplusplus // Note: for Qt/moc support
00238 enum StatusCode
00239 #else
00240 typedef enum _StatusCode
00241 #endif
00242 {
00243     Status_Processing               = 0x80000000,
00244     Status_Good                     = 0x00000000,
00245     Status_Bad                      = 0x01000000,
00246     Status_Uncertain                = 0x02000000,
00247
00248     //------ Modbus standart errors begin -------
00249     // from 0 to 255
00250     Status_BadIllegalFunction                    = Status_Bad | 0x01,
00251     Status_BadIllegalDataAddress                 = Status_Bad | 0x02,
00252     Status_BadIllegalDataValue                   = Status_Bad | 0x03,
00253     Status_BadServerDeviceFailure                = Status_Bad | 0x04,
00254     Status_BadAcknowledge                        = Status_Bad | 0x05,
00255     Status_BadServerDeviceBusy                   = Status_Bad | 0x06,
00256     Status_BadNegativeAcknowledge                = Status_Bad | 0x07,
00257     Status_BadMemoryParityError                  = Status_Bad | 0x08,
00258     Status_BadGatewayPathUnavailable             = Status_Bad | 0x0A,
00259     Status_BadGatewayTargetDeviceFailedToRespond = Status_Bad | 0x0B,
00260     //------- Modbus standart errors end --------
00261
00262     //------- Modbus common errors begin --------
00263     Status_BadEmptyResponse         = Status_Bad | 0x101,
00264     Status_BadNotCorrectRequest     ,
00265     Status_BadNotCorrectResponse    ,
00266     Status_BadWriteBufferOverflow   ,
00267     Status_BadReadBufferOverflow    ,
00268
00269     //-------- Modbus common errors end ---------
00270
00271     //--_ Modbus serial specified errors begin --
00272     Status_BadSerialOpen            = Status_Bad | 0x201,
00273     Status_BadSerialWrite           ,
00274     Status_BadSerialRead            ,
00275     Status_BadSerialReadTimeout     ,
```

```
00276     Status_BadSerialWriteTimeout    ,
00277     //---_ Modbus serial specified errors end ---
00278
00279     //---- Modbus ASC specified errors begin ----
00280     Status_BadAscMissColon          = Status_Bad | 0x301,
00281     Status_BadAscMissCrLf           ,
00282     Status_BadAscChar               ,
00283     Status_BadLrc                   ,
00284     //---- Modbus ASC specified errors end ----
00285
00286     //---- Modbus RTU specified errors begin ----
00287     Status_BadCrc                   = Status_Bad | 0x401,
00288     //----- Modbus RTU specified errors end -----
00289
00290     //--_ Modbus TCP specified errors begin --
00291     Status_BadTcpCreate             = Status_Bad | 0x501,
00292     Status_BadTcpConnect,
00293     Status_BadTcpWrite,
00294     Status_BadTcpRead,
00295     Status_BadTcpBind,
00296     Status_BadTcpListen,
00297     Status_BadTcpAccept,
00298     Status_BadTcpDisconnect,
00299     //---_ Modbus TCP specified errors end ---
00300 }
00301 #ifdef __cplusplus
00302 ;
00303 #else
00304 StatusCode;
00305 #endif
00306
00308 #ifdef __cplusplus // Note: for Qt/moc support
00309 enum ProtocolType
00310 #else
00311 typedef enum _ProtocolType
00312 #endif
00313 {
00314     ASC,
00315     RTU,
00316     TCP
00317 }
00318 #ifdef __cplusplus
00319 ;
00320 #else
00321 ProtocolType;
00322 #endif
00323
00324
00326 #ifdef __cplusplus // Note: for Qt/moc support
00327 enum Parity
00328 #else
00329 typedef enum _Parity
00330 #endif
00331 {
00332     NoParity   ,
00333     EvenParity ,
00334     OddParity  ,
00335     SpaceParity,
00336     MarkParity
00337 }
00338 #ifdef __cplusplus
00339 ;
00340 #else
00341 Parity;
00342 #endif
00343
00344
00346 #ifdef __cplusplus // Note: for Qt/moc support
00347 enum StopBits
00348 #else
00349 typedef enum _StopBits
00350 #endif
00351 {
00352     OneStop       ,
00353     OneAndHalfStop,
00354     TwoStop
00355 }
00356 #ifdef __cplusplus
00357 ;
00358 #else
00359 StopBits;
00360 #endif
00361
00363 #ifdef __cplusplus // Note: for Qt/moc support
00364 enum FlowControl
00365 #else
00366 typedef enum _FlowControl
```

```
00367 #endif
00368 {
00369     NoFlowControl  ,
00370     HardwareControl,
00371     SoftwareControl
00372 }
00373 #ifdef __cplusplus
00374 ;
00375 #else
00376 FlowControl;
00377 #endif
00378
00379 #ifdef QT_CORE_LIB
00380 Q_ENUM_NS(StatusCode)
00381 Q_ENUM_NS(ProtocolType)
00382 Q_ENUM_NS(Parity)
00383 Q_ENUM_NS(StopBits)
00384 Q_ENUM_NS(FlowControl)
00385 #endif
00386
00388 typedef struct
00389 {
00390     const Char *portName        ;
00391     int32_t     baudRate        ;
00392     int8_t      dataBits        ;
00393     Parity      parity          ;
00394     StopBits    stopBits        ;
00395     FlowControl flowControl     ;
00396     uint32_t    timeoutFirstByte;
00397     uint32_t    timeoutInterByte;
00398 } SerialSettings;
00399
00401 typedef struct
00402 {
00403     const Char *host   ;
00404     uint16_t    port   ;
00405     uint32_t    timeout;
00406     uint32_t    maxconn;
00407 } TcpSettings;
00408
00409 #ifdef __cplusplus
00410 extern "C" {
00411 #endif
00412
00414 inline bool StatusIsProcessing(StatusCode status) { return status == Status_Processing; }
00415
00417 inline bool StatusIsGood(StatusCode status) { return (status & 0xFF000000) == Status_Good; }
00418
00420 inline bool StatusIsBad(StatusCode status) { return (status & Status_Bad) != 0; }
00421
00423 inline bool StatusIsUncertain(StatusCode status) { return (status & Status_Uncertain) != 0; }
00424
00426 inline bool StatusIsStandardError(StatusCode status) { return (status & Status_Bad) && ((status &
      0xFF00) == 0); }
00427
00429 inline bool getBit(const void *bitBuff, uint16_t bitNum) { return GET_BIT (bitBuff, bitNum); }
00430
00432 inline bool getBitS(const void *bitBuff, uint16_t bitNum, uint16_t maxBitCount) { return (bitNum <
      maxBitCount) ? getBit(bitBuff, bitNum) : false; }
00433
00435 inline void setBit(void *bitBuff, uint16_t bitNum, bool value) { SET_BIT (bitBuff, bitNum, value) }
00436
00438 inline void setBitS(void *bitBuff, uint16_t bitNum, bool value, uint16_t maxBitCount) { if (bitNum <
      maxBitCount) setBit(bitBuff, bitNum, value); }
00439
00443 inline bool *getBits(const void *bitBuff, uint16_t bitNum, uint16_t bitCount, bool *boolBuff) {
      GET_BITS(bitBuff, bitNum, bitCount, boolBuff) return boolBuff; }
00444
00447 inline bool *getBitsS(const void *bitBuff, uint16_t bitNum, uint16_t bitCount, bool *boolBuff,
      uint16_t maxBitCount) { if ((bitNum+bitCount) <= maxBitCount) getBits(bitBuff, bitNum, bitCount,
      boolBuff); return boolBuff; }
00448
00452 inline void *setBits(void *bitBuff, uint16_t bitNum, uint16_t bitCount, const bool *boolBuff) {
      SET_BITS(bitBuff, bitNum, bitCount, boolBuff) return bitBuff; }
00453
00456 inline void *setBitsS(void *bitBuff, uint16_t bitNum, uint16_t bitCount, const bool *boolBuff,
      uint16_t maxBitCount) { if ((bitNum + bitCount) <= maxBitCount) setBits(bitBuff, bitNum, bitCount,
      boolBuff); return bitBuff; }
00457
00459 MODBUS_EXPORT uint32_t modbusLibVersion();
00460
00462 MODBUS_EXPORT const Char* modbusLibVersionStr();
00463
00465 inline uint16_t toModbusOffset(uint32_t adr) { return (uint16_t)(adr - 1); }
00466
00469 MODBUS_EXPORT uint16_t crc16(const uint8_t *byteArr, uint32_t count);
00470
```

```
00473 MODBUS_EXPORT uint8_t lrc(const uint8_t *byteArr, uint32_t count);
00474
00483 MODBUS_EXPORT StatusCode readMemRegs(uint32_t offset, uint32_t count, void *values, const void
      *memBuff, uint32_t memRegCount, uint32_t *outCount);
00484
00493 MODBUS_EXPORT StatusCode writeMemRegs(uint32_t offset, uint32_t count, const void *values, void
      *memBuff, uint32_t memRegCount, uint32_t *outCount);
00494
00503 MODBUS_EXPORT StatusCode readMemBits(uint32_t offset, uint32_t count, void *values, const void
      *memBuff, uint32_t memBitCount, uint32_t *outCount);
00504
00513 MODBUS_EXPORT StatusCode writeMemBits(uint32_t offset, uint32_t count, const void *values, void
      *memBuff, uint32_t memBitCount, uint32_t *outCount);
00514
00522 MODBUS_EXPORT uint32_t bytesToAscii(const uint8_t* bytesBuff, uint8_t* asciiBuff, uint32_t count);
00523
00531 MODBUS_EXPORT uint32_t asciiToBytes(const uint8_t* asciiBuff, uint8_t* bytesBuff, uint32_t count);
00532
00534 MODBUS_EXPORT Char *sbytes(const uint8_t* buff, uint32_t count, Char *str, uint32_t strmaxlen);
00535
00537 MODBUS_EXPORT Char *sascii(const uint8_t* buff, uint32_t count, Char *str, uint32_t strmaxlen);
00538
00541 MODBUS_EXPORT const Char *sprotocolType(ProtocolType type);
00542
00545 MODBUS_EXPORT ProtocolType toprotocolType(const Char *s);
00546
00550 MODBUS_EXPORT const Char *sbaudRate(int32_t baudRate);
00551
00554 MODBUS_EXPORT int32_t tobaudRate(const Char *s);
00555
00559 MODBUS_EXPORT const Char *sdataBits(int8_t dataBits);
00560
00563 MODBUS_EXPORT int8_t todataBits(const Char *s);
00564
00568 MODBUS_EXPORT const Char *sparity(Parity parity);
00569
00572 MODBUS_EXPORT Parity toparity(const Char *s);
00573
00577 MODBUS_EXPORT const Char *sstopBits(StopBits stopBits);
00578
00581 MODBUS_EXPORT StopBits tostopBits(const Char *s);
00582
00586 MODBUS_EXPORT const Char *sflowControl(FlowControl flowControl);
00587
00590 MODBUS_EXPORT FlowControl toflowControl(const Char *s);
00591
00593 MODBUS_EXPORT Timer timer();
00594
00596 MODBUS_EXPORT Timestamp currentTimestamp();
00597
00599 MODBUS_EXPORT void setConsoleColor(Color color);
00600
00602 MODBUS_EXPORT void msleep(uint32_t msec);
00603
00604 #ifdef __cplusplus
00605 } //extern "C"
00606 #endif
00607
00608 #ifdef __cplusplus
00609 } //namespace Modbus
00610 #endif
00611
00612 #endif // MODBUSGLOBAL_H
```

## 8.14 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusObject.h File Reference

The header file defines the class templates used to create signal/slot-like mechanism.

```
#include "Modbus.h"
```

### Classes

- class ModbusSlotBase< ReturnType, Args >

`ModbusSlotBase` *base template for slot (method or function)*

- class ModbusSlotMethod< T, ReturnType, Args >

  `ModbusSlotMethod` *template class hold pointer to object and its method*

- class ModbusSlotFunction< ReturnType, Args >

  `ModbusSlotFunction` *template class hold pointer to slot function*

- class ModbusObject

  *The* `ModbusObject` *class is the base class for objects that use signal/slot mechanism.*

### Typedefs

- template<class T , class ReturnType , class ... Args>
  using **ModbusMethodPointer** = ReturnType(T::*)(Args...)

  `ModbusMethodPointer`*-pointer to class method template type*

- template<class ReturnType , class ... Args>
  using **ModbusFunctionPointer** = ReturnType (*)(Args...)

  `ModbusFunctionPointer` *pointer to function template type*

## 8.14.1 Detailed Description

The header file defines the class templates used to create signal/slot-like mechanism.

**Author**

serhmarch

**Date**

May 2024

# 8.15 ModbusObject.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSOBJECT_H
00009 #define MODBUSOBJECT_H
00010
00011 #include "Modbus.h"
00012
00014 template <class T, class ReturnType, class ... Args>
00015 using ModbusMethodPointer = ReturnType(T::*)(Args...);
00016
00018 template <class ReturnType, class ... Args>
00019 using ModbusFunctionPointer = ReturnType (*)(Args...);
00020
00022 template <class ReturnType, class ... Args>
00023 class ModbusSlotBase
00024 {
00025 public:
00027     virtual ~ModbusSlotBase() {}
00028
00031     virtual void *object() const { return nullptr; }
00032
00034     virtual void *methodOrFunction() const = 0;
00035
00037     virtual ReturnType exec(Args ...  args) = 0;
00038 };
00039
00040
00041
00043 template <class T, class ReturnType, class ... Args>
```

```
00044 class ModbusSlotMethod : public ModbusSlotBase<ReturnType, Args ...>
00045 {
00046 public:
00050     ModbusSlotMethod(T* object, ModbusMethodPointer<T, ReturnType, Args...> methodPtr) :
      m_object(object), m_methodPtr(methodPtr) {}
00051
00052 public:
00053     void *object() const override { return m_object; }
00054     void *methodOrFunction() const override { return reinterpret_cast<void*>(m_voidPtr); }
00055
00056     ReturnType exec(Args ...  args) override
00057     {
00058         return (m_object->*m_methodPtr)(args...);
00059     }
00060
00061 private:
00062     T* m_object;
00063     union
00064     {
00065         ModbusMethodPointer<T, ReturnType, Args...> m_methodPtr;
00066         void *m_voidPtr;
00067     };
00068 };
00069
00070
00072 template <class ReturnType, class ... Args>
00073 class ModbusSlotFunction : public ModbusSlotBase<ReturnType, Args ...>
00074 {
00075 public:
00078     ModbusSlotFunction(ModbusFunctionPointer<ReturnType, Args...> funcPtr) : m_funcPtr(funcPtr) {}
00079
00080 public:
00081     void *methodOrFunction() const override { return m_voidPtr; }
00082     ReturnType exec(Args ...  args) override
00083     {
00084         return m_funcPtr(args...);
00085     }
00086
00087 private:
00088     union
00089     {
00090         ModbusFunctionPointer<ReturnType, Args...> m_funcPtr;
00091         void *m_voidPtr;
00092     };
00093 };
00094
00095 class ModbusObjectPrivate;
00096
00114 class MODBUS_EXPORT ModbusObject
00115 {
00116 public:
00120     static ModbusObject *sender();
00121
00122 public:
00124     ModbusObject();
00125
00127     virtual ~ModbusObject();
00128
00129 public:
00131     const Modbus::Char *objectName() const;
00132
00134     void setObjectName(const Modbus::Char *name);
00135
00136 public:
00147     template <class SignalClass, class T, class ReturnType, class ... Args>
00148     void connect(ModbusMethodPointer<SignalClass, ReturnType, Args ...> signalMethodPtr, T *object,
      ModbusMethodPointer<T, ReturnType, Args ...> objectMethodPtr)
00149     {
00150         ModbusSlotMethod<T, ReturnType, Args ...> *slotMethod = new ModbusSlotMethod<T, ReturnType,
      Args ...>(object, objectMethodPtr);
00151         union {
00152             ModbusMethodPointer<SignalClass, ReturnType, Args ...> signalMethodPtr;
00153             void* voidPtr;
00154         } converter;
00155         converter.signalMethodPtr = signalMethodPtr;
00156         setSlot(converter.voidPtr, slotMethod);
00157     }
00158
00161     template <class SignalClass, class ReturnType, class ... Args>
00162     void connect(ModbusMethodPointer<SignalClass, ReturnType, Args ...> signalMethodPtr,
      ModbusFunctionPointer<ReturnType, Args ...> funcPtr)
00163     {
00164         ModbusSlotFunction<ReturnType, Args ...> *slotFunc = new ModbusSlotFunction<ReturnType, Args
      ...>(funcPtr);
00165         union {
00166             ModbusMethodPointer<SignalClass, ReturnType, Args ...> signalMethodPtr;
00167             void* voidPtr;
```

```
00168         } converter;
00169         converter.signalMethodPtr = signalMethodPtr;
00170         setSlot(converter.voidPtr, slotFunc);
00171     }
00172
00174     template <class ReturnType, class ... Args>
00175     inline void disconnect(ModbusFunctionPointer<ReturnType, Args ...> funcPtr)
00176     {
00177         disconnect(nullptr, funcPtr);
00178     }
00179
00181     inline void disconnectFunc(void *funcPtr)
00182     {
00183         disconnect(nullptr, funcPtr);
00184     }
00185
00187     template <class T, class ReturnType, class ... Args>
00188     inline void disconnect(T *object, ModbusMethodPointer<T, ReturnType, Args ...> objectMethodPtr)
00189     {
00190         union {
00191             ModbusMethodPointer<T, ReturnType, Args ...> objectMethodPtr;
00192             void* voidPtr;
00193         } converter;
00194         converter.objectMethodPtr = objectMethodPtr;
00195         disconnect(object, converter.voidPtr);
00196     }
00197
00199     template <class T>
00200     inline void disconnect(T *object)
00201     {
00202         disconnect(object, nullptr);
00203     }
00204
00205
00206 protected:
00208     template <class T, class ... Args>
00209     void emitSignal(const char *thisMethodId, ModbusMethodPointer<T, void, Args ...> thisMethod, Args
      ... args)
00210     {
00211         dummy = thisMethodId; // Note: present because of weird MSVC compiler optimization,
00212                               // when diff signals can have same address
00213         //printf("Emit signal: %s\n", thisMethodId);
00214         union {
00215             ModbusMethodPointer<T, void, Args ...> thisMethod;
00216             void* voidPtr;
00217         } converter;
00218         converter.thisMethod = thisMethod;
00219
00220         pushSender(this);
00221         int i = 0;
00222         while (void* itemSlot = slot(converter.voidPtr, i++))
00223         {
00224             ModbusSlotBase<void, Args...> *slotBase = reinterpret_cast<ModbusSlotBase<void, Args...>
      *>(itemSlot);
00225             slotBase->exec(args...);
00226         }
00227         popSender();
00228     }
00229
00230 private:
00231     void *slot(void *signalMethodPtr, int i) const;
00232     void setSlot(void *signalMethodPtr, void *slotPtr);
00233     void disconnect(void *object, void *methodOrFunc);
00234
00235 private:
00236     static void pushSender(ModbusObject *sender);
00237     static void popSender();
00238
00239 protected:
00241     static const char* dummy; // Note: prevent weird MSVC compiler optimization
00242     ModbusObjectPrivate *d_ptr;
00243     ModbusObject(ModbusObjectPrivate *d);
00245 };
00246
00247 #endif // MODBUSOBJECT_H
```

## 8.16 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusPlatform.h File Reference

Definition of platform specific macros.

### 8.16.1 Detailed Description

Definition of platform specific macros.

**Author**

serhmarch

**Date**

May 2024

## 8.17 ModbusPlatform.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSPLATFORM_H
00009 #define MODBUSPLATFORM_H
00010
00011 #if defined (_WIN32)|| defined(_WIN64)|| defined(__WIN32__) || defined(__WINDOWS__)
00012 #define MB_OS_WINDOWS
00013 #endif
00014
00015 // Linux, BSD and Solaris define "unix", OSX doesn't, even though it derives from BSD
00016 #if defined(unix) || defined(__unix__) || defined(__unix)
00017 #define MB_PLATFORM_UNIX
00018 #endif
00019
00020 #if BSD>=0
00021 #define MB_OS_BSD
00022 #endif
00023
00024 #if __APPLE__
00025 #define MB_OS_OSX
00026 #endif
00027
00028
00029 #ifdef _MSC_VER
00030
00031 #define MB_DECL_IMPORT __declspec (dllimport)
00032 #define MB_DECL_EXPORT __declspec (dllexport)
00033
00034 #else
00035
00036 #define MB_DECL_IMPORT
00037 #define MB_DECL_EXPORT
00038
00039 #endif
00040
00041 #endif // MODBUSPLATFORM_H
```

## 8.18 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusPort.h File Reference

Header file of abstract class `ModbusPort`.

```
#include <string>
#include <list>
#include "Modbus.h"
```

### Classes

- class ModbusPort

  *The abstract class `ModbusPort` is the base class for a specific implementation of the Modbus communication protocol.*

### 8.18.1 Detailed Description

Header file of abstract class `ModbusPort`.

**Author**

serhmarch

**Date**

May 2024

## 8.19 ModbusPort.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSPORT_H
00009 #define MODBUSPORT_H
00010
00011 #include <string>
00012 #include <list>
00013
00014 #include "Modbus.h"
00015
00016 class ModbusPortPrivate;
00017
00024 class MODBUS_EXPORT ModbusPort
00025 {
00026 public:
00028     virtual ~ModbusPort();
00029
00030 public:
00032     virtual Modbus::ProtocolType type() const = 0;
00033
00035     virtual Modbus::Handle handle() const = 0;
00036
00038     virtual Modbus::StatusCode open() = 0;
00039
00041     virtual Modbus::StatusCode close() = 0;
00042
00044     virtual bool isOpen() const = 0;
00045
00048     virtual void setNextRequestRepeated(bool v);
00049
00050 public:
00052     bool isChanged() const;
00053
00055     bool isServerMode() const;
00056
00058     virtual void setServerMode(bool mode);
00059
00061     bool isBlocking() const;
00062
00064     bool isNonBlocking() const;
00065
00067     uint32_t timeout() const;
00068
00070     void setTimeout(uint32_t timeout);
00071
00072 public: // errors
00074     Modbus::StatusCode lastErrorStatus() const;
00075
00077     const Modbus::Char *lastErrorText() const;
```

```
00078
00079 public:
00081     virtual Modbus::StatusCode writeBuffer(uint8_t unit, uint8_t func, uint8_t *buff, uint16_t
      szInBuff) = 0;
00082
00084     virtual Modbus::StatusCode readBuffer(uint8_t &unit, uint8_t &func, uint8_t *buff, uint16_t
      maxSzBuff, uint16_t *szOutBuff) = 0;
00085
00087     virtual Modbus::StatusCode write() = 0;
00088
00090     virtual Modbus::StatusCode read() = 0;
00091
00092 public: // buffer
00094     virtual const uint8_t *readBufferData() const = 0;
00095
00097     virtual uint16_t readBufferSize() const = 0;
00098
00100     virtual const uint8_t *writeBufferData() const = 0;
00101
00103     virtual uint16_t writeBufferSize() const = 0;
00104
00105 protected:
00107     Modbus::StatusCode setError(Modbus::StatusCode status, const Modbus::Char *text);
00108
00109 protected:
00111     ModbusPortPrivate *d_ptr;
00112     ModbusPort(ModbusPortPrivate *d);
00114 };
00115
00116 #endif // MODBUSPORT_H
```

# 8.20 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusQt.h File Reference

Qt support file for ModbusLib.

```
#include "Modbus.h"
#include <QMetaEnum>
#include <QHash>
#include <QVariant>
```

### Classes

- class Modbus::Strings

  *Sets constant key values for the map of settings.*
- class Modbus::Defaults

  *Holds the default values of the settings.*

### Namespaces

- namespace Modbus

  *Main Modbus namespace. Contains classes, functions and constants to work with Modbus-protocol.*

### Typedefs

- typedef QHash< QString, QVariant > **Modbus::Settings**

  *Map for settings of Modbus protocol where key has type `QString` and value is `QVariant`.*

### Functions

- MODBUS_EXPORT uint8_t Modbus::getSettingUnit (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT ProtocolType Modbus::getSettingType (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint32_t Modbus::getSettingTries (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT QString Modbus::getSettingHost (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint16_t Modbus::getSettingPort (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint32_t Modbus::getSettingTimeout (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint32_t Modbus::getSettingMaxconn (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT QString Modbus::getSettingSerialPortName (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT int32_t Modbus::getSettingBaudRate (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT int8_t Modbus::getSettingDataBits (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT Parity Modbus::getSettingParity (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT StopBits Modbus::getSettingStopBits (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT FlowControl Modbus::getSettingFlowControl (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint32_t Modbus::getSettingTimeoutFirstByte (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT uint32_t Modbus::getSettingTimeoutInterByte (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT bool Modbus::getSettingBroadcastEnabled (const Settings &s, bool *ok=nullptr)
- MODBUS_EXPORT void Modbus::setSettingUnit (Settings &s, uint8_t v)
- MODBUS_EXPORT void Modbus::setSettingType (Settings &s, ProtocolType v)
- MODBUS_EXPORT void Modbus::setSettingTries (Settings &s, uint32_t)
- MODBUS_EXPORT void Modbus::setSettingHost (Settings &s, const QString &v)
- MODBUS_EXPORT void Modbus::setSettingPort (Settings &s, uint16_t v)
- MODBUS_EXPORT void Modbus::setSettingTimeout (Settings &s, uint32_t v)
- MODBUS_EXPORT void Modbus::setSettingMaxconn (Settings &s, uint32_t v)
- MODBUS_EXPORT void Modbus::setSettingSerialPortName (Settings &s, const QString &v)
- MODBUS_EXPORT void Modbus::setSettingBaudRate (Settings &s, int32_t v)
- MODBUS_EXPORT void Modbus::setSettingDataBits (Settings &s, int8_t v)
- MODBUS_EXPORT void Modbus::setSettingParity (Settings &s, Parity v)
- MODBUS_EXPORT void Modbus::setSettingStopBits (Settings &s, StopBits v)
- MODBUS_EXPORT void Modbus::setSettingFlowControl (Settings &s, FlowControl v)
- MODBUS_EXPORT void Modbus::setSettingTimeoutFirstByte (Settings &s, uint32_t v)
- MODBUS_EXPORT void Modbus::setSettingTimeoutInterByte (Settings &s, uint32_t v)
- MODBUS_EXPORT void Modbus::setSettingBroadcastEnabled (Settings &s, bool v)
- Address Modbus::addressFromQString (const QString &s)
- template<class EnumType >
  QString Modbus::enumKey (int value)
- template<class EnumType >
  QString Modbus::enumKey (EnumType value, const QString &byDef=QString())
- template<class EnumType >
  EnumType Modbus::enumValue (const QString &key, bool *ok=nullptr, EnumType defaultValue=static_cast< EnumType >(-1))
- template<class EnumType >
  EnumType Modbus::enumValue (const QVariant &value, bool *ok=nullptr, EnumType defaultValue=static_cast< EnumType >(-1))
- template<class EnumType >
  EnumType Modbus::enumValue (const QVariant &value, EnumType defaultValue)
- template<class EnumType >
  EnumType Modbus::enumValue (const QVariant &value)
- MODBUS_EXPORT ProtocolType Modbus::toProtocolType (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT ProtocolType Modbus::toProtocolType (const QVariant &v, bool *ok=nullptr)
- MODBUS_EXPORT int32_t Modbus::toBaudRate (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT int32_t Modbus::toBaudRate (const QVariant &v, bool *ok=nullptr)
- MODBUS_EXPORT int8_t Modbus::toDataBits (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT int8_t Modbus::toDataBits (const QVariant &v, bool *ok=nullptr)

- MODBUS_EXPORT Parity Modbus::toParity (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT Parity Modbus::toParity (const QVariant &v, bool *ok=nullptr)
- MODBUS_EXPORT StopBits Modbus::toStopBits (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT StopBits Modbus::toStopBits (const QVariant &v, bool *ok=nullptr)
- MODBUS_EXPORT FlowControl Modbus::toFlowControl (const QString &s, bool *ok=nullptr)
- MODBUS_EXPORT FlowControl Modbus::toFlowControl (const QVariant &v, bool *ok=nullptr)
- MODBUS_EXPORT QString Modbus::toString (StatusCode v)
- MODBUS_EXPORT QString Modbus::toString (ProtocolType v)
- MODBUS_EXPORT QString Modbus::toString (Parity v)
- MODBUS_EXPORT QString Modbus::toString (StopBits v)
- MODBUS_EXPORT QString Modbus::toString (FlowControl v)
- QString Modbus::bytesToString (const QByteArray &v)
- QString Modbus::asciiToString (const QByteArray &v)
- MODBUS_EXPORT QStringList Modbus::availableSerialPortList ()
- MODBUS_EXPORT ModbusPort * Modbus::createPort (const Settings &settings, bool blocking=false)
- MODBUS_EXPORT ModbusClientPort * Modbus::createClientPort (const Settings &settings, bool blocking=false)
- MODBUS_EXPORT ModbusServerPort * Modbus::createServerPort (ModbusInterface *device, const Settings &settings, bool blocking=false)

### 8.20.1 Detailed Description

Qt support file for ModbusLib.

**Author**

serhmarch

**Date**

May 2024

## 8.21 ModbusQt.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSQT_H
00009 #define MODBUSQT_H
00010
00011 #include "Modbus.h"
00012
00013 #include <QMetaEnum>
00014 #include <QHash>
00015 #include <QVariant>
00016
00017 namespace Modbus {
00018
00020 typedef QHash<QString, QVariant> Settings;
00021
00024 class MODBUS_EXPORT Strings
00025 {
00026 public:
00027     const QString unit              ;
00028     const QString type              ;
00029     const QString tries             ;
00030     const QString host              ;
00031     const QString port              ;
00032     const QString timeout           ;
00033     const QString maxconn           ;
00034     const QString serialPortName    ;
00035     const QString baudRate          ;
```

```
00036     const QString dataBits          ;
00037     const QString parity            ;
00038     const QString stopBits          ;
00039     const QString flowControl       ;
00040     const QString timeoutFirstByte  ;
00041     const QString timeoutInterByte  ;
00042     const QString isBroadcastEnabled;
00043
00044     const QString NoParity          ;
00045     const QString EvenParity        ;
00046     const QString OddParity         ;
00047     const QString SpaceParity       ;
00048     const QString MarkParity        ;
00049
00050     const QString OneStop           ;
00051     const QString OneAndHalfStop    ;
00052     const QString TwoStop           ;
00053
00054     const QString NoFlowControl     ;
00055     const QString HardwareControl   ;
00056     const QString SoftwareControl   ;
00057
00059     Strings();
00060
00062     static const Strings &instance();
00063 };
00064
00067 class MODBUS_EXPORT Defaults
00068 {
00069 public:
00070     const uint8_t      unit              ;
00071     const ProtocolType type              ;
00072     const uint32_t     tries             ;
00073     const QString      host              ;
00074     const uint16_t     port              ;
00075     const uint32_t     timeout           ;
00076     const uint32_t     maxconn           ;
00077     const QString      serialPortName    ;
00078     const int32_t      baudRate          ;
00079     const int8_t       dataBits          ;
00080     const Parity       parity            ;
00081     const StopBits     stopBits          ;
00082     const FlowControl  flowControl       ;
00083     const uint32_t     timeoutFirstByte  ;
00084     const uint32_t     timeoutInterByte  ;
00085     const bool         isBroadcastEnabled;
00086
00088     Defaults();
00089
00091     static const Defaults &instance();
00092 };
00093
00096 MODBUS_EXPORT uint8_t getSettingUnit(const Settings &s, bool *ok = nullptr);
00097
00100 MODBUS_EXPORT ProtocolType getSettingType(const Settings &s, bool *ok = nullptr);
00101
00104 MODBUS_EXPORT uint32_t getSettingTries(const Settings &s, bool *ok = nullptr);
00105
00108 MODBUS_EXPORT QString getSettingHost(const Settings &s, bool *ok = nullptr);
00109
00112 MODBUS_EXPORT uint16_t getSettingPort(const Settings &s, bool *ok = nullptr);
00113
00116 MODBUS_EXPORT uint32_t getSettingTimeout(const Settings &s, bool *ok = nullptr);
00117
00120 MODBUS_EXPORT uint32_t getSettingMaxconn(const Settings &s, bool *ok = nullptr);
00121
00124 MODBUS_EXPORT QString getSettingSerialPortName(const Settings &s, bool *ok = nullptr);
00125
00128 MODBUS_EXPORT int32_t getSettingBaudRate(const Settings &s, bool *ok = nullptr);
00129
00132 MODBUS_EXPORT int8_t getSettingDataBits(const Settings &s, bool *ok = nullptr);
00133
00136 MODBUS_EXPORT Parity getSettingParity(const Settings &s, bool *ok = nullptr);
00137
00140 MODBUS_EXPORT StopBits getSettingStopBits(const Settings &s, bool *ok = nullptr);
00141
00144 MODBUS_EXPORT FlowControl getSettingFlowControl(const Settings &s, bool *ok = nullptr);
00145
00148 MODBUS_EXPORT uint32_t getSettingTimeoutFirstByte(const Settings &s, bool *ok = nullptr);
00149
00152 MODBUS_EXPORT uint32_t getSettingTimeoutInterByte(const Settings &s, bool *ok = nullptr);
00153
00156 MODBUS_EXPORT bool getSettingBroadcastEnabled(const Settings &s, bool *ok = nullptr);
00157
00159 MODBUS_EXPORT void setSettingUnit(Settings &s, uint8_t v);
00160
00162 MODBUS_EXPORT void setSettingType(Settings &s, ProtocolType v);
```

```
00163
00165 MODBUS_EXPORT void setSettingTries(Settings &s, uint32_t);
00166
00168 MODBUS_EXPORT void setSettingHost(Settings &s, const QString &v);
00169
00171 MODBUS_EXPORT void setSettingPort(Settings &s, uint16_t v);
00172
00174 MODBUS_EXPORT void setSettingTimeout(Settings &s, uint32_t v);
00175
00177 MODBUS_EXPORT void setSettingMaxconn(Settings &s, uint32_t v);
00178
00180 MODBUS_EXPORT void setSettingSerialPortName(Settings &s, const QString&v);
00181
00183 MODBUS_EXPORT void setSettingBaudRate(Settings &s, int32_t v);
00184
00186 MODBUS_EXPORT void setSettingDataBits(Settings &s, int8_t v);
00187
00189 MODBUS_EXPORT void setSettingParity(Settings &s, Parity v);
00190
00192 MODBUS_EXPORT void setSettingStopBits(Settings &s, StopBits v);
00193
00195 MODBUS_EXPORT void setSettingFlowControl(Settings &s, FlowControl v);
00196
00198 MODBUS_EXPORT void setSettingTimeoutFirstByte(Settings &s, uint32_t v);
00199
00201 MODBUS_EXPORT void setSettingTimeoutInterByte(Settings &s, uint32_t v);
00202
00204 MODBUS_EXPORT void setSettingBroadcastEnabled(Settings &s, bool v);
00205
00207 inline Address addressFromQString(const QString &s) { return Address::fromString(s); }
00208
00210 template <class EnumType>
00211 inline QString enumKey(int value)
00212 {
00213     const QMetaEnum me = QMetaEnum::fromType<EnumType>();
00214     return QString(me.valueToKey(value));
00215 }
00216
00218 template <class EnumType>
00219 inline QString enumKey(EnumType value, const QString &byDef = QString())
00220 {
00221     const QMetaEnum me = QMetaEnum::fromType<EnumType>();
00222     const char *key = me.valueToKey(value);
00223     if (key)
00224         return QString(me.valueToKey(value));
00225     else
00226         return byDef;
00227 }
00228
00230 template <class EnumType>
00231 inline EnumType enumValue(const QString& key, bool* ok = nullptr, EnumType defaultValue =
      static_cast<EnumType>(-1))
00232 {
00233     bool okInner;
00234     const QMetaEnum me = QMetaEnum::fromType<EnumType>();
00235     EnumType v = static_cast<EnumType>(me.keyToValue(key.toLatin1().constData(), &okInner));
00236     if (ok)
00237         *ok = okInner;
00238     if (okInner)
00239         return v;
00240     return defaultValue;
00241 }
00242
00246 template <class EnumType>
00247 inline EnumType enumValue(const QVariant& value, bool *ok = nullptr, EnumType defaultValue =
      static_cast<EnumType>(-1))
00248 {
00249     bool okInner;
00250     int v = value.toInt(&okInner);
00251     if (okInner)
00252     {
00253         const QMetaEnum me = QMetaEnum::fromType<EnumType>();
00254         if (me.valueToKey(v)) // check value exists
00255         {
00256             if (ok)
00257                 *ok = true;
00258             return static_cast<EnumType>(v);
00259         }
00260         if (ok)
00261             *ok = false;
00262         return defaultValue;
00263     }
00264     return enumValue<EnumType>(value.toString(), ok, defaultValue);
00265 }
00266
00269 template <class EnumType>
00270 inline EnumType enumValue(const QVariant& value, EnumType defaultValue)
```

```
00271 {
00272     return enumValue<EnumType>(value, nullptr, defaultValue);
00273 }
00274
00276 template <class EnumType>
00277 inline EnumType enumValue(const QVariant& value)
00278 {
00279     return enumValue<EnumType>(value, nullptr);
00280 }
00281
00284 MODBUS_EXPORT ProtocolType toProtocolType(const QString &s, bool *ok = nullptr);
00285
00288 MODBUS_EXPORT ProtocolType toProtocolType(const QVariant &v, bool *ok = nullptr);
00289
00292 MODBUS_EXPORT int32_t toBaudRate(const QString &s, bool *ok = nullptr);
00293
00296 MODBUS_EXPORT int32_t toBaudRate(const QVariant &v, bool *ok = nullptr);
00297
00300 MODBUS_EXPORT int8_t toDataBits(const QString &s, bool *ok = nullptr);
00301
00304 MODBUS_EXPORT int8_t toDataBits(const QVariant &v, bool *ok = nullptr);
00305
00308 MODBUS_EXPORT Parity toParity(const QString &s, bool *ok = nullptr);
00309
00312 MODBUS_EXPORT Parity toParity(const QVariant &v, bool *ok = nullptr);
00313
00316 MODBUS_EXPORT StopBits toStopBits(const QString &s, bool *ok = nullptr);
00317
00320 MODBUS_EXPORT StopBits toStopBits(const QVariant &v, bool *ok = nullptr);
00321
00324 MODBUS_EXPORT FlowControl toFlowControl(const QString &s, bool *ok = nullptr);
00325
00328 MODBUS_EXPORT FlowControl toFlowControl(const QVariant &v, bool *ok = nullptr);
00329
00331 MODBUS_EXPORT QString toString(StatusCode v);
00332
00334 MODBUS_EXPORT QString toString(ProtocolType v);
00335
00337 MODBUS_EXPORT QString toString(Parity v);
00338
00340 MODBUS_EXPORT QString toString(StopBits v);
00341
00343 MODBUS_EXPORT QString toString(FlowControl v);
00344
00346 inline QString bytesToString(const QByteArray &v) { return bytesToString(reinterpret_cast<const
      uint8_t*>(v.constData()), v.size()).data(); }
00347
00349 inline QString asciiToString(const QByteArray &v) { return asciiToString(reinterpret_cast<const
      uint8_t*>(v.constData()), v.size()).data(); }
00350
00352 MODBUS_EXPORT QStringList availableSerialPortList();
00353
00356 MODBUS_EXPORT ModbusPort *createPort(const Settings &settings, bool blocking = false);
00357
00360 MODBUS_EXPORT ModbusClientPort *createClientPort(const Settings &settings, bool blocking = false);
00361
00364 MODBUS_EXPORT ModbusServerPort *createServerPort(ModbusInterface *device, const Settings &settings,
      bool blocking = false);
00365
00366 } // namespace Modbus
00367
00368 #endif // MODBUSQT_H
```

## 8.22 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusRtuPort.h File Reference

Contains definition of RTU serial port class.

```
#include "ModbusSerialPort.h"
```

### Classes

- class ModbusRtuPort

  *Implements RTU version of the Modbus communication protocol.*

### 8.22.1 Detailed Description

Contains definition of RTU serial port class.

**Author**

serhmarch

**Date**

May 2024

## 8.23 ModbusRtuPort.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSRTUPORT_H
00009 #define MODBUSRTUPORT_H
00010
00011 #include "ModbusSerialPort.h"
00012
00019 class MODBUS_EXPORT ModbusRtuPort : public ModbusSerialPort
00020 {
00021 public:
00023     ModbusRtuPort(bool blocking = false);
00024
00026     ~ModbusRtuPort();
00027
00028 public:
00030     Modbus::ProtocolType type() const override { return Modbus::RTU; }
00031
00032 protected:
00033     Modbus::StatusCode writeBuffer(uint8_t unit, uint8_t func, uint8_t *buff, uint16_t szInBuff)
      override;
00034     Modbus::StatusCode readBuffer(uint8_t &unit, uint8_t &func, uint8_t *buff, uint16_t maxSzBuff,
      uint16_t *szOutBuff) override;
00035
00036 protected:
00037     using ModbusSerialPort::ModbusSerialPort;
00038 };
00039
00040 #endif // MODBUSRTUPORT_H
```

## 8.24 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusSerialPort.h File Reference

Contains definition of base serial port class.

```
#include "ModbusPort.h"
```

**Classes**

- class ModbusSerialPort

  *The abstract class `ModbusSerialPort` is the base class serial port Modbus communications.*
- struct ModbusSerialPort::Defaults

  *Holds the default values of the settings.*

### 8.24.1 Detailed Description

Contains definition of base serial port class.

**Author**

serhmarch

**Date**

May 2024

## 8.25 ModbusSerialPort.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSSERIALPORT_H
00009 #define MODBUSSERIALPORT_H
00010
00011 #include "ModbusPort.h"
00012
00020 class MODBUS_EXPORT ModbusSerialPort : public ModbusPort
00021 {
00022 public:
00025     struct MODBUS_EXPORT Defaults
00026     {
00027         const Modbus::Char       *portName        ;
00028         const int32_t             baudRate        ;
00029         const int8_t              dataBits        ;
00030         const Modbus::Parity      parity          ;
00031         const Modbus::StopBits    stopBits        ;
00032         const Modbus::FlowControl flowControl     ;
00033         const uint32_t            timeoutFirstByte;
00034         const uint32_t            timeoutInterByte;
00035
00037         Defaults();
00038
00040         static const Defaults &instance();
00041     };
00042
00043 public:
00045     ~ModbusSerialPort();
00046
00047 public:
00049     Modbus::Handle handle() const override;
00050
00052     Modbus::StatusCode open() override;
00053
00055     Modbus::StatusCode close() override;
00056
00058     bool isOpen() const override;
00059
00060 public: // settings
00062     const Modbus::Char *portName() const;
00063
00065     void setPortName(const Modbus::Char *portName);
00066
00068     int32_t baudRate() const;
00069
00071     void setBaudRate(int32_t baudRate);
00072
00074     int8_t dataBits() const;
00075
00077     void setDataBits(int8_t dataBits);
00078
00080     Modbus::Parity parity() const;
00081
00083     void setParity(Modbus::Parity parity);
00084
00086     Modbus::StopBits stopBits() const;
00087
00089     void setStopBits(Modbus::StopBits stopBits);
00090
00092     Modbus::FlowControl flowControl() const;
```

```
00093
00095     void setFlowControl(Modbus::FlowControl flowControl);
00096
00098     inline uint32_t timeoutFirstByte() const { return timeout(); }
00099
00101     inline void setTimeoutFirstByte(uint32_t timeout) { setTimeout(timeout); }
00102
00104     uint32_t timeoutInterByte() const;
00105
00107     void setTimeoutInterByte(uint32_t timeout);
00108
00109 public:
00110     const uint8_t *readBufferData() const override;
00111     uint16_t readBufferSize() const override;
00112     const uint8_t *writeBufferData() const override;
00113     uint16_t writeBufferSize() const override;
00114
00115 protected:
00116     Modbus::StatusCode write() override;
00117     Modbus::StatusCode read() override;
00118
00119 protected:
00121     using ModbusPort::ModbusPort;
00123 };
00124
00125 #endif // MODBUSSERIALPORT_H
```

## 8.26 ModbusServerPort.h

```
00001
00008 #ifndef MODBUSSERVERPORT_H
00009 #define MODBUSSERVERPORT_H
00010
00011 #include "ModbusObject.h"
00012
00021 class MODBUS_EXPORT ModbusServerPort : public ModbusObject
00022 {
00023 public:
00026     ModbusInterface *device() const;
00027
00030     void setDevice(ModbusInterface *device);
00031
00032 public: // server port interface
00034     virtual Modbus::ProtocolType type() const = 0;
00035
00037     virtual bool isTcpServer() const;
00038
00041     virtual Modbus::StatusCode open() = 0;
00042
00044     virtual Modbus::StatusCode close() = 0;
00045
00047     virtual bool isOpen() const = 0;
00048
00051     bool isBroadcastEnabled() const;
00052
00055     virtual void setBroadcastEnabled(bool enable);
00056
00064     const void *unitMap() const;
00065
00068     virtual void setUnitMap(const void *unitmap);
00069
00071     void *context() const;
00072
00074     void setContext(void *context);
00075
00078     virtual Modbus::StatusCode process() = 0;
00079
00080 public:
00082     bool isStateClosed() const;
00083
00084 public: // SIGNALS
00086     void signalOpened(const Modbus::Char *source);
00087
00089     void signalClosed(const Modbus::Char *source);
00090
00093     void signalTx(const Modbus::Char *source, const uint8_t* buff, uint16_t size);
00094
00097     void signalRx(const Modbus::Char *source, const uint8_t* buff, uint16_t size);
00098
00100     void signalError(const Modbus::Char *source, Modbus::StatusCode status, const Modbus::Char *text);
00101
00102 protected:
00103     using ModbusObject::ModbusObject;
```

```
00104 };
00105
00106 #endif // MODBUSSERVERPORT_H
00107
```

## 8.27 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusServerResource.h File Reference

The header file defines the class that controls specific port.

```
#include "ModbusServerPort.h"
```

### Classes

- class ModbusServerResource

  *Implements direct control for `ModbusPort` derived classes (TCP or serial) for server side.*

### 8.27.1 Detailed Description

The header file defines the class that controls specific port.

**Author**

serhmarch

**Date**

May 2024

## 8.28 ModbusServerResource.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSSERVERRESOURCE_H
00009 #define MODBUSSERVERRESOURCE_H
00010
00011 #include "ModbusServerPort.h"
00012
00013 class ModbusPort;
00014
00024 class MODBUS_EXPORT ModbusServerResource : public ModbusServerPort
00025 {
00026 public:
00030     ModbusServerResource(ModbusPort *port, ModbusInterface *device);
00031
00032 public:
00034     ModbusPort *port() const;
00035
00036 public: // server port interface
00038     Modbus::ProtocolType type() const override;
00039
00040     Modbus::StatusCode open() override;
00041
00042     Modbus::StatusCode close() override;
00043
00044     bool isOpen() const override;
00045
```

```
00046     Modbus::StatusCode process() override;
00047
00048 protected:
00050     virtual Modbus::StatusCode processInputData(const uint8_t *buff, uint16_t sz);
00051
00053     virtual Modbus::StatusCode processDevice();
00054
00056     virtual Modbus::StatusCode processOutputData(uint8_t *buff, uint16_t &sz);
00057
00058 protected:
00059     using ModbusServerPort::ModbusServerPort;
00060 };
00061
00062 #endif // MODBUSSERVERRESOURCE_H
```

## 8.29 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusTcpPort.h File Reference

Header file of class ModbusTcpPort.

```
#include "ModbusPort.h"
```

### Classes

- class ModbusTcpPort

  *Class ModbusTcpPort implements TCP version of Modbus protocol.*
- struct ModbusTcpPort::Defaults

  *Defaults class contain default settings values for ModbusTcpPort.*

### 8.29.1 Detailed Description

Header file of class ModbusTcpPort.

**Author**

serhmarch

**Date**

April 2024

## 8.30 ModbusTcpPort.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSTCPPORT_H
00009 #define MODBUSTCPPORT_H
00010
00011 #include "ModbusPort.h"
00012
00013 class ModbusTcpSocket;
00014
00021 class MODBUS_EXPORT ModbusTcpPort : public ModbusPort
00022 {
00023 public:
00026     struct MODBUS_EXPORT Defaults
```

```
00027     {
00028         const Modbus::Char *host   ;
00029         const uint16_t      port   ;
00030         const uint32_t      timeout;
00031 
00033         Defaults();
00034 
00036         static const Defaults &instance();
00037     };
00038 
00039 public:
00041     ModbusTcpPort(ModbusTcpSocket *socket, bool blocking = false);
00042 
00044     ModbusTcpPort(bool blocking = false);
00045 
00047     ~ModbusTcpPort();
00048 
00049 public:
00051     Modbus::ProtocolType type() const override { return Modbus::TCP; }
00052 
00054     Modbus::Handle handle() const override;
00055 
00056     Modbus::StatusCode open() override;
00057     Modbus::StatusCode close() override;
00058     bool isOpen() const override;
00059 
00060 public:
00062     const Modbus::Char *host() const;
00063 
00065     void setHost(const Modbus::Char *host);
00066 
00068     uint16_t port() const;
00069 
00071     void setPort(uint16_t port);
00072 
00074     void setNextRequestRepeated(bool v) override;
00075 
00077     bool autoIncrement() const;
00078 
00079 public:
00080     const uint8_t *readBufferData() const override;
00081     uint16_t readBufferSize() const override;
00082     const uint8_t *writeBufferData() const override;
00083     uint16_t writeBufferSize() const override;
00084 
00085 protected:
00086     Modbus::StatusCode write() override;
00087     Modbus::StatusCode read() override;
00088     Modbus::StatusCode writeBuffer(uint8_t unit, uint8_t func, uint8_t *buff, uint16_t szInBuff)
      override;
00089     Modbus::StatusCode readBuffer(uint8_t &unit, uint8_t &func, uint8_t *buff, uint16_t maxSzBuff,
      uint16_t *szOutBuff) override;
00090 
00091 protected:
00092     using ModbusPort::ModbusPort;
00093 };
00094 
00095 #endif // MODBUSTCPPORT_H
```

## 8.31 c:/Users/march/Dropbox/PRJ/ModbusLib/src/ModbusTcpServer.h File Reference

Header file of Modbus TCP server.

```
#include "ModbusServerPort.h"
```

### Classes

- class ModbusTcpServer

  *The `ModbusTcpServer` class implements TCP server part of the Modbus protocol.*
- struct ModbusTcpServer::Defaults

  *`Defaults` class constain default settings values for `ModbusTcpServer`.*

### 8.31.1 Detailed Description

Header file of Modbus TCP server.

**Author**

serhmarch

**Date**

May 2024

## 8.32 ModbusTcpServer.h

Go to the documentation of this file.

```
00001
00008 #ifndef MODBUSSERVERTCP_H
00009 #define MODBUSSERVERTCP_H
00010
00011 #include "ModbusServerPort.h"
00012
00013 class ModbusTcpSocket;
00014
00021 class MODBUS_EXPORT ModbusTcpServer : public ModbusServerPort
00022 {
00023 public:
00026     struct MODBUS_EXPORT Defaults
00027     {
00028         const uint16_t port   ;
00029         const uint32_t timeout;
00030         const uint32_t maxconn;
00031
00033         Defaults();
00034
00036         static const Defaults &instance();
00037     };
00038
00039 public:
00041     ModbusTcpServer(ModbusInterface *device);
00042
00044     ~ModbusTcpServer();
00045
00046 public:
00048     uint16_t port() const;
00049
00051     void setPort(uint16_t port);
00052
00054     uint32_t timeout() const;
00055
00057     void setTimeout(uint32_t timeout);
00058
00060     uint32_t maxConnections() const;
00061
00063     void setMaxConnections(uint32_t maxconn);
00064
00065 public:
00067     Modbus::ProtocolType type() const override { return Modbus::TCP; }
00068
00070     bool isTcpServer() const override { return true; }
00071
00078     Modbus::StatusCode open() override;
00079
00083     Modbus::StatusCode close() override;
00084
00086     bool isOpen() const override;
00087
00090     void setBroadcastEnabled(bool enable) override;
00091
00094     void setUnitMap(const void *unitmap) override;
00095
00097     Modbus::StatusCode process() override;
00098
00099 public:
00102     virtual ModbusServerPort *createTcpPort(ModbusTcpSocket *socket);
00103
```

```
00106     virtual void deleteTcpPort(ModbusServerPort *port);
00107
00108 public: // SIGNALS
00110     void signalNewConnection(const Modbus::Char *source);
00111
00113     void signalCloseConnection(const Modbus::Char *source);
00114
00115 protected:
00117     ModbusTcpSocket *nextPendingConnection();
00118
00120     void clearConnections();
00121
00122 protected:
00123     using ModbusServerPort::ModbusServerPort;
00124 };
00125
00126 #endif // MODBUSSERVERTCP_H
```

# Index