# ModbusMaster

0.6

Generated by Doxygen 1.6.2

Thu Feb 4 21:10:54 2010

# Contents



# 1 Module Index

## 1.1 Modules

Here is a list of all modules:

# 2 Class Index

## 2.1 Class List

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



# 3 Module Documentation

## 3.1 ModbusMaster Object Instantiation/Initialization

### Functions

- ModbusMaster::ModbusMaster ()

  *Constructor.*

- ModbusMaster::ModbusMaster (uint8_t)
- ModbusMaster::ModbusMaster (uint8_t, uint8_t)
- void ModbusMaster::begin ()

  *Initialize class object.*

- void ModbusMaster::begin (uint16_t)

### 3.1.1 Function Documentation

#### 3.1.1.1 ModbusMaster::ModbusMaster (void) `[inherited]`

Constructor. Creates class object using default serial port 0, Modbus slave ID 1.

```
51 {
52   _u8SerialPort = 0;
53   _u8MBSlave = 1;
54 }
```

#### 3.1.1.2 ModbusMaster::ModbusMaster (uint8_t *u8MBSlave*) [inherited]

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Creates class object using default serial port 0, specified Modbus slave ID.

**Parameters:**

***u8MBSlave*** Modbus slave ID (1..255)

```
67 {
68   _u8SerialPort = 0;
69   _u8MBSlave = u8MBSlave;
70 }
```

#### 3.1.1.3 ModbusMaster::ModbusMaster (uint8_t *u8SerialPort*, uint8_t *u8MBSlave*) [inherited]

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Creates class object using specified serial port, Modbus slave ID.

**Parameters:**

***u8SerialPort*** serial port (0..3)

***u8MBSlave*** Modbus slave ID (1..255)

```
84 {
85   _u8SerialPort = (u8SerialPort > 3) ? 0 : u8SerialPort;
86   _u8MBSlave = u8MBSlave;
87 }
```

#### 3.1.1.4 void ModbusMaster::begin (void) [inherited]

Initialize class object. Sets up the serial port using default 19200 baud rate. Call once class has been instantiated, typically within setup().

```
99 {
100   begin(19200);
101 }
```

#### 3.1.1.5 void ModbusMaster::begin (uint16_t *u16BaudRate*) [inherited]

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Sets up the serial port using specified baud rate. Call once class has been instantiated, typically within setup().

**Parameters:**

***u16BaudRate*** baud rate, in standard increments (300..115200)

```
115 {
116   switch(_u8SerialPort)
117   {
118 #if defined(__AVR_ATmega1280__)
119     case 1:
120       MBSerial = Serial1;
121       break;
122
123     case 2:
124       MBSerial = Serial2;
125       break;
126
127     case 3:
128       MBSerial = Serial3;
129       break;
130 #endif
131
132     case 0:
133     default:
134       MBSerial = Serial;
135       break;
136   }
137
138   MBSerial.begin(u16BaudRate);
139 #if __MODBUSMASTER_DEBUG__
140   pinMode(4, OUTPUT);
141   pinMode(5, OUTPUT);
142 #endif
143 }
```

## 3.2 ModbusMaster Buffer Management

### Functions

- uint16_t ModbusMaster::GetResponseBuffer (uint8_t)

*Retrieve data from response buffer.*

- void ModbusMaster::ClearResponseBuffer ()
  *Clear Modbus response buffer.*
- uint8_t ModbusMaster::SetTransmitBuffer (uint8_t, uint16_t)
  *Place data in transmit buffer.*
- void ModbusMaster::ClearTransmitBuffer ()
  *Clear Modbus transmit buffer.*

### 3.2.1 Function Documentation

#### 3.2.1.1 uint16_t ModbusMaster::GetResponseBuffer (uint8_t *u8Index*) [inherited]

Retrieve data from response buffer.

**See also:**

ModbusMaster::ClearResponseBuffer()

**Parameters:**

***u8Index*** index of response buffer array (0x00..0x3F)

**Returns:**

value in position u8Index of response buffer (0x0000..0xFFFF)

```
155 {
156   if (u8Index < ku8MaxBufferSize)
157   {
158     return _u16ResponseBuffer[u8Index];
159   }
160   else
161   {
162     return 0xFFFF;
163   }
164 }
```

#### 3.2.1.2 void ModbusMaster::ClearResponseBuffer () [inherited]

Clear Modbus response buffer.

**See also:**

ModbusMaster::GetResponseBuffer(uint8_t u8Index)

```
174 {
175   uint8_t i;
176
177   for (i = 0; i < ku8MaxBufferSize; i++)
178   {
179     _u16ResponseBuffer[i] = 0;
180   }
181 }
```

#### 3.2.1.3 uint8_t ModbusMaster::SetTransmitBuffer (uint8_t *u8Index*, uint16_t *u16Value*) [inherited]

Place data in transmit buffer.

**See also:**

ModbusMaster::ClearTransmitBuffer()

**Parameters:**

***u8Index*** index of transmit buffer array (0x00..0x3F)

***u16Value*** value to place in position u8Index of transmit buffer (0x0000..0xFFFF)

**Returns:**

0 on success; exception number on failure

```
194 {
195   if (u8Index < ku8MaxBufferSize)
196   {
197     _u16TransmitBuffer[u8Index] = u16Value;
198     return ku8MBSuccess;
199   }
200   else
201   {
202     return ku8MBIllegalDataAddress;
203   }
204 }
```

#### 3.2.1.4 void ModbusMaster::ClearTransmitBuffer () [inherited]

Clear Modbus transmit buffer.

**See also:**

ModbusMaster::SetTransmitBuffer(uint8_t u8Index, uint16_t u16Value)

```
214 {
215   uint8_t i;
216
217   for (i = 0; i < ku8MaxBufferSize; i++)
218   {
219     _u16TransmitBuffer[i] = 0;
220   }
221 }
```

## 3.3 Modbus Function Codes for Discrete Coils/Inputs

### Functions

- uint8_t ModbusMaster::ReadCoils (uint16_t, uint16_t)

  *Modbus function 0x01 Read Coils.*

- uint8_t ModbusMaster::ReadDiscreteInputs (uint16_t, uint16_t)

  *Modbus function 0x02 Read Discrete Inputs.*

- uint8_t ModbusMaster::WriteSingleCoil (uint16_t, uint8_t)

  *Modbus function 0x05 Write Single Coil.*

- uint8_t ModbusMaster::WriteMultipleCoils (uint16_t, uint16_t)

  *Modbus function 0x0F Write Multiple Coils.*

### 3.3.1 Function Documentation

#### 3.3.1.1 uint8_t ModbusMaster::ReadCoils (uint16_t *u16ReadAddress*, uint16_t *u16BitQty*) [inherited]

Modbus function 0x01 Read Coils. This function code is used to read from 1 to 2000 contiguous status of coils in a remote device. The request specifies the starting address, i.e. the address of the first coil specified, and the number of coils. Coils are addressed starting at zero.

The coils in the response buffer are packed as one coil per bit of the data field. Status is indicated as 1=ON and 0=OFF. The LSB of the first data word contains the output addressed in the query. The other coils follow toward the high order end of this word and from low order to high order in subsequent words.

If the returned quantity is not a multiple of sixteen, the remaining bits in the final data word will be padded with zeros (toward the high order end of the word).

**Parameters:**

***u16ReadAddress*** address of first coil (0x0000..0xFFFF)

***u16BitQty*** quantity of coils to read (1..2000, enforced by remote device)

**Returns:**

0 on success; exception number on failure

```
248 {
249   _u16ReadAddress = u16ReadAddress;
250   _u16ReadQty = u16BitQty;
251   return ModbusMasterTransaction(ku8MBReadCoils);
252 }
```

#### 3.3.1.2 uint8_t ModbusMaster::ReadDiscreteInputs (uint16_t *u16ReadAddress*, uint16_t *u16BitQty*) `[inherited]`

Modbus function 0x02 Read Discrete Inputs. This function code is used to read from 1 to 2000 contiguous status of discrete inputs in a remote device. The request specifies the starting address, i.e. the address of the first input specified, and the number of inputs. Discrete inputs are addressed starting at zero.

The discrete inputs in the response buffer are packed as one input per bit of the data field. Status is indicated as 1=ON; 0=OFF. The LSB of the first data word contains the input addressed in the query. The other inputs follow toward the high order end of this word, and from low order to high order in subsequent words.

If the returned quantity is not a multiple of sixteen, the remaining bits in the final data word will be padded with zeros (toward the high order end of the word).

**Parameters:**

***u16ReadAddress*** address of first discrete input (0x0000..0xFFFF)

***u16BitQty*** quantity of discrete inputs to read (1..2000, enforced by remote device)

**Returns:**

0 on success; exception number on failure

```
280 {
281   _u16ReadAddress = u16ReadAddress;
282   _u16ReadQty = u16BitQty;
283   return ModbusMasterTransaction(ku8MBReadDiscreteInputs);
284 }
```

#### 3.3.1.3 uint8_t ModbusMaster::WriteSingleCoil (uint16_t *u16WriteAddress*, uint8_t *u8State*) [inherited]

Modbus function 0x05 Write Single Coil. This function code is used to write a single output to either ON or OFF in a remote device. The requested ON/OFF state is specified by a constant in the state field. A non-zero value requests the output to be ON and a value of 0 requests it to be OFF. The request specifies the address of the coil to be forced. Coils are addressed starting at zero.

**Parameters:**

***u16WriteAddress*** address of the coil (0x0000..0xFFFF)

***u8State*** 0=OFF, non-zero=ON (0x00..0xFF)

**Returns:**

0 on success; exception number on failure

```
352 {
353   _u16WriteAddress = u16WriteAddress;
354   _u16WriteQty = (u8State ? 0xFF00 : 0x0000);
355   return ModbusMasterTransaction(ku8MBWriteSingleCoil);
356 }
```

#### 3.3.1.4 uint8_t ModbusMaster::WriteMultipleCoils (uint16_t *u16WriteAddress*, uint16_t *u16BitQty*) [inherited]

Modbus function 0x0F Write Multiple Coils. This function code is used to force each coil in a sequence of coils to either ON or OFF in a remote device. The request specifies the coil references to be forced. Coils are addressed starting at zero.

The requested ON/OFF states are specified by contents of the transmit buffer. A logical '1' in a bit position of the buffer requests the corresponding output to be ON. A logical '0' requests it to be OFF.

**Parameters:**

***u16WriteAddress*** address of the first coil (0x0000..0xFFFF)

***u16BitQty*** quantity of coils to write (1..2000, enforced by remote device)

**Returns:**

0 on success; exception number on failure

```
399 {
400   _u16WriteAddress = u16WriteAddress;
401   _u16WriteQty = u16BitQty;
402   return ModbusMasterTransaction(ku8MBWriteMultipleCoils);
403 }
```

## 3.4 Modbus Function Codes for Holding/Input Registers

### Functions

- uint8_t ModbusMaster::ReadHoldingRegisters (uint16_t, uint16_t)

  *Modbus function 0x03 Read Holding Registers.*

- uint8_t ModbusMaster::ReadInputRegisters (uint16_t, uint8_t)

  *Modbus function 0x04 Read Input Registers.*

- uint8_t ModbusMaster::WriteSingleRegister (uint16_t, uint16_t)

  *Modbus function 0x06 Write Single Register.*

- uint8_t ModbusMaster::WriteMultipleRegisters (uint16_t, uint16_t)

  *Modbus function 0x10 Write Multiple Registers.*

- uint8_t ModbusMaster::MaskWriteRegister (uint16_t, uint16_t, uint16_t)

  *Modbus function 0x16 Mask Write Register.*

- uint8_t ModbusMaster::ReadWriteMultipleRegisters (uint16_t, uint16_t, uint16_t, uint16_t)

  *Modbus function 0x17 Read Write Multiple Registers.*

### 3.4.1 Function Documentation

#### 3.4.1.1 uint8_t ModbusMaster::ReadHoldingRegisters (uint16_t *u16ReadAddress*, uint16_t *u16ReadQty*) [inherited]

Modbus function 0x03 Read Holding Registers. This function code is used to read the contents of a contiguous block of holding registers in a remote device. The request specifies the starting register address and the number of registers. Registers are addressed starting at zero.

The register data in the response buffer is packed as one word per register.

**Parameters:**

***u16ReadAddress*** address of the first holding register (0x0000..0xFFFF)

***u16ReadQty*** quantity of holding registers to read (1..125, enforced by remote device)

**Returns:**

0 on success; exception number on failure

```
305 {
306   _u16ReadAddress = u16ReadAddress;
307   _u16ReadQty = u16ReadQty;
308   return ModbusMasterTransaction(ku8MBReadHoldingRegisters);
309 }
```

#### 3.4.1.2 uint8_t ModbusMaster::ReadInputRegisters (uint16_t *u16ReadAddress*, uint8_t *u16ReadQty*) [inherited]

Modbus function 0x04 Read Input Registers. This function code is used to read from 1 to 125 contiguous input registers in a remote device. The request specifies the starting register address and the number of registers. Registers are addressed starting at zero.

The register data in the response buffer is packed as one word per register.

**Parameters:**

***u16ReadAddress*** address of the first input register (0x0000..0xFFFF)

***u16ReadQty*** quantity of input registers to read (1..125, enforced by remote device)

**Returns:**

0 on success; exception number on failure

```
330 {
331   _u16ReadAddress = u16ReadAddress;
332   _u16ReadQty = u16ReadQty;
333   return ModbusMasterTransaction(ku8MBReadInputRegisters);
334 }
```

#### 3.4.1.3 uint8_t ModbusMaster::WriteSingleRegister (uint16_t *u16WriteAddress*, uint16_t *u16WriteValue*) [inherited]

Modbus function 0x06 Write Single Register. This function code is used to write a single holding register in a remote device. The request specifies the address of the register to be written. Registers are addressed starting at zero.

**Parameters:**

***u16WriteAddress*** address of the holding register (0x0000..0xFFFF)

***u16WriteValue*** value to be written to holding register (0x0000..0xFFFF)

**Returns:**

0 on success; exception number on failure

```
373 {
374   _u16WriteAddress = u16WriteAddress;
375   _u16WriteQty = 0;
376   _u16TransmitBuffer[0] = u16WriteValue;
377   return ModbusMasterTransaction(ku8MBWriteSingleRegister);
378 }
```

#### 3.4.1.4 uint8_t ModbusMaster::WriteMultipleRegisters (uint16_t *u16WriteAddress*, uint16_t *u16WriteQty*) [inherited]

Modbus function 0x10 Write Multiple Registers. This function code is used to write a block of contiguous registers (1 to 123 registers) in a remote device.

The requested written values are specified in the transmit buffer. Data is packed as one word per register.

**Parameters:**

***u16WriteAddress*** address of the holding register (0x0000..0xFFFF)

***u16WriteQty*** quantity of holding registers to write (1..123, enforced by remote device)

**Returns:**

0 on success; exception number on failure

```
422 {
423   _u16WriteAddress = u16WriteAddress;
424   _u16WriteQty = u16WriteQty;
425   return ModbusMasterTransaction(ku8MBWriteMultipleRegisters);
426 }
```

#### 3.4.1.5 uint8_t ModbusMaster::MaskWriteRegister (uint16_t *u16WriteAddress*, uint16_t *u16AndMask*, uint16_t *u16OrMask*) [inherited]

Modbus function 0x16 Mask Write Register. This function code 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 can be used to set or clear individual bits in the register.

The request specifies the holding register to be written, the data to be used as the AND mask, and the data to be used as the OR mask. Registers are addressed starting at zero.

The function's algorithm is:

Result = (Current Contents && And_Mask) || (Or_Mask && (~And_Mask))

**Parameters:**

***u16WriteAddress*** address of the holding register (0x0000..0xFFFF)

***u16AndMask*** AND mask (0x0000..0xFFFF)

***u16OrMask*** OR mask (0x0000..0xFFFF)

**Returns:**

0 on success; exception number on failure

```
453 {
454   _u16WriteAddress = u16WriteAddress;
455   _u16TransmitBuffer[0] = u16AndMask;
456   _u16TransmitBuffer[1] = u16OrMask;
457   return ModbusMasterTransaction(ku8MBMaskWriteRegister);
458 }
```

#### 3.4.1.6 uint8_t ModbusMaster::ReadWriteMultipleRegisters (uint16_t *u16ReadAddress*, uint16_t *u16ReadQty*, uint16_t *u16WriteAddress*, uint16_t *u16WriteQty*) [inherited]

Modbus function 0x17 Read Write Multiple Registers. This function code performs a combination of one read operation and one write operation in a single MODBUS transaction. The write operation is performed before the read. Holding registers are addressed starting at zero.

The request specifies the starting address and number of holding registers to be read as well as the starting address, and the number of holding registers. The data to be written is specified in the transmit buffer.

**Parameters:**

***u16ReadAddress*** address of the first holding register (0x0000..0xFFFF)

***u16ReadQty*** quantity of holding registers to read (1..125, enforced by remote device)

***u16WriteAddress*** address of the first holding register (0x0000..0xFFFF)

***u16WriteQty*** quantity of holding registers to write (1..121, enforced by remote device)

**Returns:**

0 on success; exception number on failure

```
483 {
484   _u16ReadAddress = u16ReadAddress;
485   _u16ReadQty = u16ReadQty;
486   _u16WriteAddress = u16WriteAddress;
487   _u16WriteQty = u16WriteQty;
488   return ModbusMasterTransaction(ku8MBReadWriteMultipleRegisters);
489 }
```

## 3.5 Modbus Function Codes, Exception Codes

### Variables

- static const uint8_t ModbusMaster::ku8MBIllegalFunction = 0x01

  *Modbus protocol illegal function exception.*

- static const uint8_t ModbusMaster::ku8MBIllegalDataAddress = 0x02

  *Modbus protocol illegal data address exception.*

- static const uint8_t ModbusMaster::ku8MBIllegalDataValue = 0x03

  *Modbus protocol illegal data value exception.*

- static const uint8_t ModbusMaster::ku8MBSlaveDeviceFailure = 0x04

  *Modbus protocol slave device failure exception.*

- static const uint8_t ModbusMaster::ku8MBSuccess = 0x00

  *ModbusMaster success.*

- static const uint8_t ModbusMaster::ku8MBInvalidSlaveID = 0xE0

  *ModbusMaster invalid response slave ID exception.*

- static const uint8_t ModbusMaster::ku8MBInvalidFunction = 0xE1

  *ModbusMaster invalid response function exception.*

- static const uint8_t ModbusMaster::ku8MBResponseTimedOut = 0xE2

  *ModbusMaster response timed out exception.*

- static const uint8_t ModbusMaster::ku8MBInvalidCRC = 0xE3

  *ModbusMaster invalid response CRC exception.*

### 3.5.1 Variable Documentation

#### 3.5.1.1 const uint8_t ModbusMaster::ku8MBIllegalFunction = 0x01 [static, inherited]

Modbus protocol illegal function exception. The function code received in the query is not an allowable action for the server (or slave). This may be because the function code is only applicable to newer devices, and was not implemented in the unit selected. It could also indicate that the server (or slave) is in the wrong state to process a request of this type, for example because it is unconfigured and is being asked to return register values.

#### 3.5.1.2 const uint8_t ModbusMaster::ku8MBIllegalDataAddress = 0x02 [static, inherited]

Modbus protocol illegal data address exception. The data address received in the query is not an allowable address for the server (or slave). More specifically, the combination of reference number and transfer length is invalid. For a controller with 100 registers, the ADU addresses the first register as 0, and the last one as 99. If a request is submitted with a starting register address of 96 and a quantity of registers of 4, then this request will successfully operate (address-wise at least) on registers 96, 97, 98, 99. If a request is submitted with a starting register address of 96 and a quantity of registers of 5, then this request will fail with Exception Code 0x02 "Illegal Data Address" since it attempts to operate on registers 96, 97, 98, 99 and 100, and there is no register with address 100.

#### 3.5.1.3 const uint8_t ModbusMaster::ku8MBIllegalDataValue = 0x03 [static, inherited]

Modbus protocol illegal data value exception. A value contained in the query data field is not an allowable value for server (or slave). This indicates a fault in the structure

of the remainder of a complex request, such as that the implied length is incorrect. It specifically does NOT mean that a data item submitted for storage in a register has a value outside the expectation of the application program, since the MODBUS protocol is unaware of the significance of any particular value of any particular register.

#### 3.5.1.4 const uint8_t ModbusMaster::ku8MBSlaveDeviceFailure = 0x04 [static, inherited]

Modbus protocol slave device failure exception. An unrecoverable error occurred while the server (or slave) was attempting to perform the requested action.

#### 3.5.1.5 const uint8_t ModbusMaster::ku8MBSuccess = 0x00 [static, inherited]

ModbusMaster success. Modbus transaction was successful; the following checks were valid:

- slave ID
- function code
- response code
- data
- CRC

#### 3.5.1.6 const uint8_t ModbusMaster::ku8MBInvalidSlaveID = 0xE0 [static, inherited]

ModbusMaster invalid response slave ID exception. The slave ID in the response does not match that of the request.

#### 3.5.1.7 const uint8_t ModbusMaster::ku8MBInvalidFunction = 0xE1 [static, inherited]

ModbusMaster invalid response function exception. The function code in the response does not match that of the request.

#### 3.5.1.8 const uint8_t ModbusMaster::ku8MBResponseTimedOut = 0xE2 [static, inherited]

ModbusMaster response timed out exception. The entire response was not received within the timeout period, ModbusMaster::ku8MBResponseTimeout.

#### 3.5.1.9 const uint8_t ModbusMaster::ku8MBInvalidCRC = 0xE3 [static, inherited]

ModbusMaster invalid response CRC exception. The CRC in the response does not match the one calculated.

# 4 Class Documentation

## 4.1 ModbusMaster Class Reference

Arduino class library for communicating with Modbus slaves over RS232/485 (via RTU protocol).

```
#include <ModbusMaster.h>
```

### Public Member Functions

- ModbusMaster ()

  *Constructor.*

- ModbusMaster (uint8_t)
- ModbusMaster (uint8_t, uint8_t)
- void begin ()

  *Initialize class object.*

- void begin (uint16_t)
- uint16_t GetResponseBuffer (uint8_t)

  *Retrieve data from response buffer.*

- void ClearResponseBuffer ()

  *Clear Modbus response buffer.*

- uint8_t SetTransmitBuffer (uint8_t, uint16_t)

*Place data in transmit buffer.*

- void ClearTransmitBuffer ()
  *Clear Modbus transmit buffer.*
- uint8_t ReadCoils (uint16_t, uint16_t)
  *Modbus function 0x01 Read Coils.*
- uint8_t ReadDiscreteInputs (uint16_t, uint16_t)
  *Modbus function 0x02 Read Discrete Inputs.*
- uint8_t ReadHoldingRegisters (uint16_t, uint16_t)
  *Modbus function 0x03 Read Holding Registers.*
- uint8_t ReadInputRegisters (uint16_t, uint8_t)
  *Modbus function 0x04 Read Input Registers.*
- uint8_t WriteSingleCoil (uint16_t, uint8_t)
  *Modbus function 0x05 Write Single Coil.*
- uint8_t WriteSingleRegister (uint16_t, uint16_t)
  *Modbus function 0x06 Write Single Register.*
- uint8_t WriteMultipleCoils (uint16_t, uint16_t)
  *Modbus function 0x0F Write Multiple Coils.*
- uint8_t WriteMultipleRegisters (uint16_t, uint16_t)
  *Modbus function 0x10 Write Multiple Registers.*
- uint8_t MaskWriteRegister (uint16_t, uint16_t, uint16_t)
  *Modbus function 0x16 Mask Write Register.*
- uint8_t ReadWriteMultipleRegisters (uint16_t, uint16_t, uint16_t, uint16_t)
  *Modbus function 0x17 Read Write Multiple Registers.*

### Static Public Attributes

- static const uint8_t ku8MBIllegalFunction = 0x01
  *Modbus protocol illegal function exception.*
- static const uint8_t ku8MBIllegalDataAddress = 0x02

*Modbus protocol illegal data address exception.*

- static const uint8_t ku8MBIllegalDataValue = 0x03
  *Modbus protocol illegal data value exception.*
- static const uint8_t ku8MBSlaveDeviceFailure = 0x04
  *Modbus protocol slave device failure exception.*
- static const uint8_t ku8MBSuccess = 0x00
  *ModbusMaster success.*
- static const uint8_t ku8MBInvalidSlaveID = 0xE0
  *ModbusMaster invalid response slave ID exception.*
- static const uint8_t ku8MBInvalidFunction = 0xE1
  *ModbusMaster invalid response function exception.*
- static const uint8_t ku8MBResponseTimedOut = 0xE2
  *ModbusMaster response timed out exception.*
- static const uint8_t ku8MBInvalidCRC = 0xE3
  *ModbusMaster invalid response CRC exception.*

### Private Member Functions

- uint8_t ModbusMasterTransaction (uint8_t u8MBFunction)
  *Modbus transaction engine.*

### Private Attributes

- uint8_t _u8SerialPort
  *serial port (0..3) initialized in constructor*
- uint8_t _u8MBSlave
  *Modbus slave (1..255) initialized in constructor.*
- uint16_t _u16BaudRate
  *baud rate (300..115200) initialized in begin()*
- uint16_t _u16ReadAddress

*slave register from which to read*

- uint16_t _u16ReadQty
  *quantity of words to read*
- uint16_t _u16ResponseBuffer [ku8MaxBufferSize]
  *buffer to store Modbus slave response; read via GetResponseBuffer()*
- uint16_t _u16WriteAddress
  *slave register to which to write*
- uint16_t _u16WriteQty
  *quantity of words to write*
- uint16_t _u16TransmitBuffer [ku8MaxBufferSize]
  *buffer containing data to transmit to Modbus slave; set via SetTransmitBuffer()*

### Static Private Attributes

- static const uint8_t ku8MaxBufferSize = 64
  *size of response/transmit buffers*
- static const uint8_t ku8MBReadCoils = 0x01
  *Modbus function 0x01 Read Coils.*
- static const uint8_t ku8MBReadDiscreteInputs = 0x02
  *Modbus function 0x02 Read Discrete Inputs.*
- static const uint8_t ku8MBWriteSingleCoil = 0x05
  *Modbus function 0x05 Write Single Coil.*
- static const uint8_t ku8MBWriteMultipleCoils = 0x0F
  *Modbus function 0x0F Write Multiple Coils.*
- static const uint8_t ku8MBReadHoldingRegisters = 0x03
  *Modbus function 0x03 Read Holding Registers.*
- static const uint8_t ku8MBReadInputRegisters = 0x04
  *Modbus function 0x04 Read Input Registers.*
- static const uint8_t ku8MBWriteSingleRegister = 0x06

*Modbus function 0x06 Write Single Register.*

- static const uint8_t ku8MBWriteMultipleRegisters = 0x10

  *Modbus function 0x10 Write Multiple Registers.*

- static const uint8_t ku8MBMaskWriteRegister = 0x16

  *Modbus function 0x16 Mask Write Register.*

- static const uint8_t ku8MBReadWriteMultipleRegisters = 0x17

  *Modbus function 0x17 Read Write Multiple Registers.*

- static const uint8_t ku8MBResponseTimeout = 200

  *Modbus timeout [milliseconds].*

### 4.1.1 Detailed Description

Arduino class library for communicating with Modbus slaves over RS232/485 (via RTU protocol).

**Examples:**

examples/Basic/Basic.pde, and examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde.

### 4.1.2 Member Function Documentation

#### 4.1.2.1 uint8_t ModbusMaster::ModbusMasterTransaction (uint8_t *u8MBFunction*) `[private]`

Modbus transaction engine. Sequence:

- assemble Modbus Request Application Data Unit (ADU), based on particular function called
- transmit request over selected serial port
- wait for/retrieve response
- evaluate/disassemble response
- return status (success/exception)

**Parameters:**

***u8MBFunction*** Modbus function (0x01..0xFF)

**Returns:**

0 on success; exception number on failure

```
507 {
508   uint8_t u8ModbusADU[256];
509   uint8_t u8ModbusADUSize = 0;
510   uint8_t i, u8Qty;
511   uint16_t u16CRC;
512   uint8_t u8TimeLeft = ku8MBResponseTimeout;
513   uint8_t u8BytesLeft = 8;
514   uint8_t u8MBStatus = ku8MBSuccess;
515 
516   // assemble Modbus Request Application Data Unit
517   u8ModbusADU[u8ModbusADUSize++] = _u8MBSlave;
518   u8ModbusADU[u8ModbusADUSize++] = u8MBFunction;
519 
520   switch(u8MBFunction)
521   {
522     case ku8MBReadCoils:
523     case ku8MBReadDiscreteInputs:
524     case ku8MBReadInputRegisters:
525     case ku8MBReadHoldingRegisters:
526     case ku8MBReadWriteMultipleRegisters:
527       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16ReadAddress);
528       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16ReadAddress);
529       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16ReadQty);
530       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16ReadQty);
531       break;
532   }
533 
534   switch(u8MBFunction)
535   {
536     case ku8MBWriteSingleCoil:
537     case ku8MBMaskWriteRegister:
538     case ku8MBWriteMultipleCoils:
539     case ku8MBWriteSingleRegister:
540     case ku8MBWriteMultipleRegisters:
541     case ku8MBReadWriteMultipleRegisters:
542       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16WriteAddress);
543       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteAddress);
544       break;
545   }
546 
547   switch(u8MBFunction)
548   {
549     case ku8MBWriteSingleCoil:
550       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16WriteQty);
551       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteQty);
552       break;
553 
554     case ku8MBWriteSingleRegister:
555       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[0]);
```

```
556       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[0]);
557       break;
558
559     case ku8MBWriteMultipleCoils:
560       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16WriteQty);
561       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteQty);
562       u8Qty = (_u16WriteQty % 8) ? ((_u16WriteQty >> 3) + 1) : (_u16WriteQty >> 3
    );
563       u8ModbusADU[u8ModbusADUSize++] = u8Qty;
564       for (i = 0; i < u8Qty; i++)
565       {
566         switch(i % 2)
567         {
568           case 0: // i is even
569             u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[i >> 1]);

570             break;
571
572           case 1: // i is odd
573             u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[i >> 1])
    ;
574             break;
575         }
576       }
577       break;
578
579     case ku8MBWriteMultipleRegisters:
580     case ku8MBReadWriteMultipleRegisters:
581       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16WriteQty);
582       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteQty);
583       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteQty << 1);
584
585       for (i = 0; i < lowByte(_u16WriteQty); i++)
586       {
587         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[i]);
588         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[i]);
589       }
590       break;
591
592     case ku8MBMaskWriteRegister:
593       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[0]);
594       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[0]);
595       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[1]);
596       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[1]);
597       break;
598   }
599
600
601   // append CRC
602   u16CRC = 0xFFFF;
603   for (i = 0; i < u8ModbusADUSize; i++)
604   {
605     u16CRC = _crc16_update(u16CRC, u8ModbusADU[i]);
606   }
607   u8ModbusADU[u8ModbusADUSize++] = lowByte(u16CRC);
608   u8ModbusADU[u8ModbusADUSize++] = highByte(u16CRC);
609   u8ModbusADU[u8ModbusADUSize] = 0;
```

```
610
611   // transmit request
612   for (i = 0; i < u8ModbusADUSize; i++)
613   {
614     MBSerial.print(u8ModbusADU[i], BYTE);
615   }
616
617   u8ModbusADUSize = 0;
618   MBSerial.flush();
619
620   // loop until we run out of time or bytes, or an error occurs
621   while (u8TimeLeft && u8BytesLeft && !u8MBStatus)
622   {
623     if (MBSerial.available())
624     {
625 #if __MODBUSMASTER_DEBUG__
626       digitalWrite(4, true);
627 #endif
628       u8ModbusADU[u8ModbusADUSize++] = MBSerial.read();
629       u8BytesLeft--;
630 #if __MODBUSMASTER_DEBUG__
631       digitalWrite(4, false);
632 #endif
633     }
634     else
635     {
636 #if __MODBUSMASTER_DEBUG__
637       digitalWrite(5, true);
638 #endif
639       delayMicroseconds(1000);
640       u8TimeLeft--;
641 #if __MODBUSMASTER_DEBUG__
642       digitalWrite(5, false);
643 #endif
644     }
645
646     // evaluate slave ID, function code once enough bytes have been read
647     if (u8ModbusADUSize == 5)
648     {
649       // verify response is for correct Modbus slave
650       if (u8ModbusADU[0] != _u8MBSlave)
651       {
652         u8MBStatus = ku8MBInvalidSlaveID;
653         break;
654       }
655
656       // verify response is for correct Modbus function code (mask exception bit
      7)
657       if ((u8ModbusADU[1] & 0x7F) != u8MBFunction)
658       {
659         u8MBStatus = ku8MBInvalidFunction;
660         break;
661       }
662
663       // check whether Modbus exception occurred; return Modbus Exception Code
664       if (bitRead(u8ModbusADU[1], 7))
665       {
```

```
666         u8MBStatus = u8ModbusADU[2];
667         break;
668       }
669
670       // evaluate returned Modbus function code
671       switch(u8ModbusADU[1])
672       {
673         case ku8MBReadCoils:
674         case ku8MBReadDiscreteInputs:
675         case ku8MBReadInputRegisters:
676         case ku8MBReadHoldingRegisters:
677         case ku8MBReadWriteMultipleRegisters:
678           u8BytesLeft = u8ModbusADU[2];
679           break;
680
681         case ku8MBWriteSingleCoil:
682         case ku8MBWriteMultipleCoils:
683         case ku8MBWriteSingleRegister:
684           u8BytesLeft = 3;
685           break;
686
687         case ku8MBMaskWriteRegister:
688           u8BytesLeft = 5;
689           break;
690       }
691     }
692
693     if (u8ModbusADUSize == 6)
694     {
695       switch(u8ModbusADU[1])
696       {
697         case ku8MBWriteMultipleRegisters:
698           u8BytesLeft = u8ModbusADU[5];
699           break;
700       }
701     }
702   }
703
704   // verify response is large enough to inspect further
705   if (!u8MBStatus && (u8TimeLeft == 0 || u8ModbusADUSize < 5))
706   {
707     u8MBStatus = ku8MBResponseTimedOut;
708   }
709
710   // calculate CRC
711   u16CRC = 0xFFFF;
712   for (i = 0; i < (u8ModbusADUSize - 2); i++)
713   {
714     u16CRC = _crc16_update(u16CRC, u8ModbusADU[i]);
715   }
716
717   // verify CRC
718   if (!u8MBStatus && (lowByte(u16CRC) != u8ModbusADU[u8ModbusADUSize - 2] ||
719     highByte(u16CRC) != u8ModbusADU[u8ModbusADUSize - 1]))
720   {
721     u8MBStatus = ku8MBInvalidCRC;
722   }
```

```
723
724   // disassemble ADU into words
725   if (!u8MBStatus)
726   {
727     // evaluate returned Modbus function code
728     switch(u8ModbusADU[1])
729     {
730       case ku8MBReadCoils:
731       case ku8MBReadDiscreteInputs:
732         // load bytes into word; response bytes are ordered L, H, L, H, ...
733         for (i = 0; i < (u8ModbusADU[2] >> 1); i++)
734         {
735           if (i < ku8MaxBufferSize)
736           {
737             _u16ResponseBuffer[i] = word(u8ModbusADU[2 * i + 4], u8ModbusADU[2 *
      i + 3]);
738           }
739         }
740
741         // in the event of an odd number of bytes, load last byte into zero-padde
      d word
742         if (u8ModbusADU[2] % 2)
743         {
744           if (i < ku8MaxBufferSize)
745           {
746             _u16ResponseBuffer[i] = word(0, u8ModbusADU[2 * i + 3]);
747           }
748         }
749         break;
750
751       case ku8MBReadInputRegisters:
752       case ku8MBReadHoldingRegisters:
753       case ku8MBReadWriteMultipleRegisters:
754         // load bytes into word; response bytes are ordered H, L, H, L, ...
755         for (i = 0; i < (u8ModbusADU[2] >> 1); i++)
756         {
757           if (i < ku8MaxBufferSize)
758           {
759             _u16ResponseBuffer[i] = word(u8ModbusADU[2 * i + 3], u8ModbusADU[2 *
      i + 4]);
760           }
761         }
762         break;
763     }
764   }
765
766   return u8MBStatus;
767 }
```

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

- ModbusMaster.h
- ModbusMaster.cpp

# 5 Example Documentation

## 5.1 examples/Basic/Basic.pde

```
/*

  Basic.pde - example using ModbusMaster library

  This file is part of ModbusMaster.

  ModbusMaster is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  ModbusMaster is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with ModbusMaster.  If not, see <http://www.gnu.org/licenses/>.

  Written by Doc Walker (Rx)
  Copyright  2009, 2010 Doc Walker <dfwmountaineers at gmail dot com>
  $Id: Basic.pde 29 2010-02-05 03:02:19Z dfwmountaineers $

*/

#include <ModbusMaster.h>


// instantiate ModbusMaster object as slave ID 2
// defaults to serial port 0 since no port was specified
ModbusMaster node(2);


void setup()
{
  // initialize Modbus communication baud rate
  node.begin(19200);
}


void loop()
{
  static uint32_t i;
  uint8_t j, result;
  uint16_t data[6];

  i++;

  // set word 0 of TX buffer to least-significant word of counter (bits 15..0)
  node.SetTransmitBuffer(0, lowWord(i));

  // set word 1 of TX buffer to most-significant word of counter (bits 31..16)
```

```
  node.SetTransmitBuffer(1, highWord(i));

  // slave 1: write TX buffer to (2) 16-bit registers starting at register 0
  result = node.WriteMultipleRegisters(0, 2);

  // slave 1: read (6) 16-bit registers starting at register 2 to RX buffer
  result = node.ReadHoldingRegisters(2, 6);

  // do something with data if read is successful
  if (result == node.ku8MBSuccess)
  {
    for (j = 0; j < 6; j++)
    {
      data[j] = node.GetResponseBuffer(j);
    }
  }
}
```

## 5.2 examples/PhoenixContact_nanoLC/PhoenixContact_-nanoLC.pde

```
/*

  PhoenixContact_nanoLC.pde - example using ModbusMaster library
  to communicate with PHOENIX CONTACT nanoLine controller.

  This file is part of ModbusMaster.

  ModbusMaster is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  ModbusMaster is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with ModbusMaster.  If not, see <http://www.gnu.org/licenses/>.

  Written by Doc Walker (Rx)
  Copyright  2009, 2010 Doc Walker <dfwmountaineers at gmail dot com>
  $Id: PhoenixContact_nanoLC.pde 29 2010-02-05 03:02:19Z dfwmountaineers $

*/

#include <ModbusMaster.h>

// discrete coils
#define NANO_DO(n)   (0x0000 + n)
#define NANO_FLAG(n) (0x1000 + n)

// discrete inputs
#define NANO_DI(n)   (0x0000 + n)
```

```
// analog holding registers
#define NANO_REG(n)  (0x0000 + 2 * n)
#define NANO_AO(n)   (0x1000 + 2 * n)
#define NANO_TCP(n)  (0x2000 + 2 * n)
#define NANO_OTP(n)  (0x3000 + 2 * n)
#define NANO_HSP(n)  (0x4000 + 2 * n)
#define NANO_TCA(n)  (0x5000 + 2 * n)
#define NANO_OTA(n)  (0x6000 + 2 * n)
#define NANO_HSA(n)  (0x7000 + 2 * n)

// analog input registers
#define NANO_AI(n)   (0x0000 + 2 * n)


// instantiate ModbusMaster object, serial port 0, Modbus slave ID 1
ModbusMaster nanoLC(0, 1);


void setup()
{
  // initialize Modbus communication baud rate
  nanoLC.begin(19200);
}


void loop()
{
  static uint32_t u32ShiftRegister;
  static uint32_t i;
  uint8_t u8Status;

  u32ShiftRegister = ((u32ShiftRegister < 0x01000000) ? (u32ShiftRegister << 4) :
       1);
  if (u32ShiftRegister == 0) u32ShiftRegister = 1;
  i++;

  // set word 0 of TX buffer to least-significant word of u32ShiftRegister (bits
      15..0)
  nanoLC.SetTransmitBuffer(0, lowWord(u32ShiftRegister));

  // set word 1 of TX buffer to most-significant word of u32ShiftRegister (bits 3
      1..16)
  nanoLC.SetTransmitBuffer(1, highWord(u32ShiftRegister));

  // set word 2 of TX buffer to least-significant word of i (bits 15..0)
  nanoLC.SetTransmitBuffer(2, lowWord(i));

  // set word 3 of TX buffer to most-significant word of i (bits 31..16)
  nanoLC.SetTransmitBuffer(3, highWord(i));

  // write TX buffer to (4) 16-bit registers starting at NANO_REG(1)
  // read (4) 16-bit registers starting at NANO_REG(0) to RX buffer
  // data is available via nanoLC.GetResponseBuffer(0..3)
  nanoLC.ReadWriteMultipleRegisters(NANO_REG(0), 4, NANO_REG(1), 4);

  // write lowWord(u32ShiftRegister) to single 16-bit register starting at NANO_R
```

```
    EG(3)
  nanoLC.WriteSingleRegister(NANO_REG(3), lowWord(u32ShiftRegister));

  // write highWord(u32ShiftRegister) to single 16-bit register starting at NANO_
    REG(3) + 1
  nanoLC.WriteSingleRegister(NANO_REG(3) + 1, highWord(u32ShiftRegister));

  // set word 0 of TX buffer to nanoLC.GetResponseBuffer(0) (bits 15..0)
  nanoLC.SetTransmitBuffer(0, nanoLC.GetResponseBuffer(0));

  // set word 1 of TX buffer to nanoLC.GetResponseBuffer(1) (bits 31..16)
  nanoLC.SetTransmitBuffer(1, nanoLC.GetResponseBuffer(1));

  // write TX buffer to (2) 16-bit registers starting at NANO_REG(4)
  nanoLC.WriteMultipleRegisters(NANO_REG(4), 2);

  // read 17 coils starting at NANO_FLAG(0) to RX buffer
  // bits 15..0 are available via nanoLC.GetResponseBuffer(0)
  // bit 16 is available via zero-padded nanoLC.GetResponseBuffer(1)
  nanoLC.ReadCoils(NANO_FLAG(0), 17);

  // read (66) 16-bit registers starting at NANO_REG(0) to RX buffer
  // generates Modbus exception ku8MBIllegalDataAddress (0x02)
  u8Status = nanoLC.ReadHoldingRegisters(NANO_REG(0), 66);
  if (u8Status == nanoLC.ku8MBIllegalDataAddress)
  {
    // read (64) 16-bit registers starting at NANO_REG(0) to RX buffer
    // data is available via nanoLC.GetResponseBuffer(0..63)
    u8Status = nanoLC.ReadHoldingRegisters(NANO_REG(0), 64);
  }

  // read (8) 16-bit registers starting at NANO_AO(0) to RX buffer
  // data is available via nanoLC.GetResponseBuffer(0..7)
  nanoLC.ReadHoldingRegisters(NANO_AO(0), 8);

  // read (64) 16-bit registers starting at NANO_TCP(0) to RX buffer
  // data is available via nanoLC.GetResponseBuffer(0..63)
  nanoLC.ReadHoldingRegisters(NANO_TCP(0), 64);

  // read (64) 16-bit registers starting at NANO_OTP(0) to RX buffer
  // data is available via nanoLC.GetResponseBuffer(0..63)
  nanoLC.ReadHoldingRegisters(NANO_OTP(0), 64);

  // read (64) 16-bit registers starting at NANO_TCA(0) to RX buffer
  // data is available via nanoLC.GetResponseBuffer(0..63)
  nanoLC.ReadHoldingRegisters(NANO_TCA(0), 64);

  // read (64) 16-bit registers starting at NANO_OTA(0) to RX buffer
  // data is available via nanoLC.GetResponseBuffer(0..63)
  nanoLC.ReadHoldingRegisters(NANO_OTA(0), 64);

  // read (8) 16-bit registers starting at NANO_AI(0) to RX buffer
  // data is available via nanoLC.GetResponseBuffer(0..7)
  nanoLC.ReadInputRegisters(NANO_AI(0), 8);
}
```