

ModbusMaster

v0.10.2

Generated by Doxygen 1.8.9.1

Thu May 21 2015 21:51:53

Contents

1	Module Index	1
1.1	Modules	1
2	Class Index	3
2.1	Class List	3
3	File Index	5
3.1	File List	5
4	Module Documentation	7
4.1	ModbusMaster Object Instantiation/Initialization	7
4.1.1	Detailed Description	7
4.1.2	Function Documentation	7
4.1.2.1	ModbusMaster	7
4.1.2.2	ModbusMaster	7
4.1.2.3	ModbusMaster	8
4.1.2.4	begin	8
4.1.2.5	begin	8
4.2	ModbusMaster Buffer Management	10
4.2.1	Detailed Description	10
4.2.2	Function Documentation	10
4.2.2.1	getResponseBuffer	10
4.2.2.2	clearResponseBuffer	10
4.2.2.3	setTransmitBuffer	11
4.2.2.4	clearTransmitBuffer	11
4.3	Modbus Function Codes for Discrete Coils/Inputs	12
4.3.1	Detailed Description	12
4.3.2	Function Documentation	12
4.3.2.1	readCoils	12

4.3.2.2	readDiscreteInputs	12
4.3.2.3	writeSingleCoil	13
4.3.2.4	writeMultipleCoils	13
4.4	Modbus Function Codes for Holding/Input Registers	15
4.4.1	Detailed Description	15
4.4.2	Function Documentation	15
4.4.2.1	readHoldingRegisters	15
4.4.2.2	readInputRegisters	16
4.4.2.3	writeSingleRegister	17
4.4.2.4	writeMultipleRegisters	17
4.4.2.5	maskWriteRegister	18
4.4.2.6	readWriteMultipleRegisters	18
4.5	Modbus Function Codes, Exception Codes	20
4.5.1	Detailed Description	20
4.5.2	Variable Documentation	20
4.5.2.1	ku8MBIllegalFunction	20
4.5.2.2	ku8MBIllegalDataAddress	20
4.5.2.3	ku8MBIllegalDataValue	21
4.5.2.4	ku8MBSlaveDeviceFailure	21
4.5.2.5	ku8MBSuccess	21
4.5.2.6	ku8MBInvalidSlaveID	21
4.5.2.7	ku8MBInvalidFunction	21
4.5.2.8	ku8MBResponseTimedOut	22
4.5.2.9	ku8MBInvalidCRC	22
4.6	<util/crc16.h>: CRC Computations	23
4.6.1	Detailed Description	23
5	Class Documentation	25
5.1	ModbusMaster Class Reference	25
5.1.1	Detailed Description	28
5.1.2	Member Function Documentation	28
5.1.2.1	idle	28
5.1.2.2	ModbusMasterTransaction	28
6	File Documentation	33
6.1	crc16.h File Reference	33
6.2	ModbusMaster.cpp File Reference	33
6.2.1	Detailed Description	33

6.3	ModbusMaster.h File Reference	33
6.3.1	Detailed Description	34
6.3.2	Macro Definition Documentation	34
6.3.2.1	__MODBUSMASTER_DEBUG__	34
7	Example Documentation	35
7.1	examples/Basic/Basic.pde	35
7.2	examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde	36
	Index	39

Chapter 1

Module Index

1.1 Modules

Here is a list of all modules:

ModbusMaster Object Instantiation/Initialization	7
ModbusMaster Buffer Management	10
Modbus Function Codes for Discrete Coils/Inputs	12
Modbus Function Codes for Holding/Input Registers	15
Modbus Function Codes, Exception Codes	20
<util/crc16.h>: CRC Computations	23

Chapter 2

Class Index

2.1 Class List

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

[ModbusMaster](#)

Arduino class library for communicating with Modbus slaves over RS232/485 (via RTU protocol) . . . [25](#)

Chapter 3

File Index

3.1 File List

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

crc16.h	33
ModbusMaster.cpp Arduino library for communicating with Modbus slaves over RS232/485 (via RTU protocol)	33
ModbusMaster.h Arduino library for communicating with Modbus slaves over RS232/485 (via RTU protocol)	33

Chapter 4

Module Documentation

4.1 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 \(uint16_t\)](#)

4.1.1 Detailed Description

4.1.2 Function Documentation

4.1.2.1 [ModbusMaster::ModbusMaster \(void \)](#)

Constructor.

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

```
54 {  
55   _u8SerialPort = 0;  
56   _u8MBSlave = 1;  
57 }
```

4.1.2.2 [ModbusMaster::ModbusMaster \(uint8_t u8MBSlave \)](#)

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

<i>u8MBSlave</i>	Modbus slave ID (1..255)
------------------	--------------------------

```

70 {
71   _u8SerialPort = 0;
72   _u8MBSlave = u8MBSlave;
73 }

```

4.1.2.3 ModbusMaster::ModbusMaster (uint8_t u8SerialPort, uint8_t u8MBSlave)

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

<i>u8SerialPort</i>	serial port (Serial, Serial1..Serial3)
<i>u8MBSlave</i>	Modbus slave ID (1..255)

```

87 {
88   _u8SerialPort = (u8SerialPort > 3) ? 0 : u8SerialPort;
89   _u8MBSlave = u8MBSlave;
90 }

```

4.1.2.4 void ModbusMaster::begin (void)

Initialize class object.

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

Examples:

[examples/Basic/Basic.pde](#), and [examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde](#).

```

102 {
103   begin(19200);
104 }

```

4.1.2.5 void ModbusMaster::begin (uint16_t u16BaudRate)

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

<i>u16BaudRate</i>	baud rate, in standard increments (300..115200)
--------------------	---

```

118 {
119   // txBuffer = (uint16_t*) calloc(ku8MaxBufferSize, sizeof(uint16_t));
120   _u8TransmitBufferIndex = 0;
121   u16TransmitBufferLength = 0;
122
123   switch(_u8SerialPort)

```

```
124 {
125 #if defined(UBRR1H)
126     case 1:
127         MBSerial = &Serial1;
128         break;
129 #endif
130
131 #if defined(UBRR2H)
132     case 2:
133         MBSerial = &Serial2;
134         break;
135 #endif
136
137 #if defined(UBRR3H)
138     case 3:
139         MBSerial = &Serial3;
140         break;
141 #endif
142
143     case 0:
144     default:
145         MBSerial = &Serial;
146         break;
147 }
148
149 MBSerial->begin(u16BaudRate);
150 #if __MODBUSMASTER_DEBUG__
151     pinMode(4, OUTPUT);
152     pinMode(5, OUTPUT);
153 #endif
154 }
```

4.2 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.

4.2.1 Detailed Description

4.2.2 Function Documentation

4.2.2.1 `uint16_t ModbusMaster::getResponseBuffer (uint8_t u8Index)`

Retrieve data from response buffer.

See also

[ModbusMaster::clearResponseBuffer\(\)](#)

Parameters

<i>u8Index</i>	index of response buffer array (0x00..0x3F)
----------------	---

Returns

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

Examples:

[examples/Basic/Basic.pde](#), and [examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde](#).

```

277 {
278   if (u8Index < ku8MaxBufferSize)
279   {
280     return _ul6ResponseBuffer[u8Index];
281   }
282   else
283   {
284     return 0xFFFF;
285   }
286 }
```

4.2.2.2 `void ModbusMaster::clearResponseBuffer ()`

Clear Modbus response buffer.

See also

[ModbusMaster::getResponseBuffer\(uint8_t u8Index\)](#)

```

296 {
297     uint8_t i;
298     for (i = 0; i < ku8MaxBufferSize; i++)
299     {
300         _ul6ResponseBuffer[i] = 0;
301     }
302 }
303 }
```

4.2.2.3 uint8_t ModbusMaster::setTransmitBuffer (uint8_t u8Index, uint16_t u16Value)

Place data in transmit buffer.

See also

[ModbusMaster::clearTransmitBuffer\(\)](#)

Parameters

<i>u8Index</i>	index of transmit buffer array (0x00..0x3F)
<i>u16Value</i>	value to place in position u8Index of transmit buffer (0x0000..0xFFFF)

Returns

0 on success; exception number on failure

Examples:

[examples/Basic/Basic.pde](#), and [examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde](#).

```

316 {
317     if (u8Index < ku8MaxBufferSize)
318     {
319         _ul6TransmitBuffer[u8Index] = u16Value;
320         return ku8MBSuccess;
321     }
322     else
323     {
324         return ku8MBIllegalDataAddress;
325     }
326 }
```

4.2.2.4 void ModbusMaster::clearTransmitBuffer ()

Clear Modbus transmit buffer.

See also

[ModbusMaster::setTransmitBuffer\(uint8_t u8Index, uint16_t u16Value\)](#)

```

336 {
337     uint8_t i;
338     for (i = 0; i < ku8MaxBufferSize; i++)
339     {
340         _ul6TransmitBuffer[i] = 0;
341     }
342 }
343 }
```

4.3 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.

4.3.1 Detailed Description

4.3.2 Function Documentation

4.3.2.1 `uint8_t ModbusMaster::readCoils (uint16_t u16ReadAddress, uint16_t u16BitQty)`

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

<i>u16ReadAddress</i>	address of first coil (0x0000..0xFFFF)
<i>u16BitQty</i>	quantity of coils to read (1..2000, enforced by remote device)

Returns

0 on success; exception number on failure

Examples:

[examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde.](#)

```

370 {
371   _u16ReadAddress = u16ReadAddress;
372   _u16ReadQty = u16BitQty;
373   return ModbusMasterTransaction(ku8MBReadCoils);
374 }
```

4.3.2.2 `uint8_t ModbusMaster::readDiscreteInputs (uint16_t u16ReadAddress, uint16_t u16BitQty)`

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

<i>u16ReadAddress</i>	address of first discrete input (0x0000..0xFFFF)
<i>u16BitQty</i>	quantity of discrete inputs to read (1..2000, enforced by remote device)

Returns

0 on success; exception number on failure

```

402 {
403     _u16ReadAddress = u16ReadAddress;
404     _u16ReadQty = u16BitQty;
405     return ModbusMasterTransaction(ku8MBReadDiscreteInputs);
406 }
```

4.3.2.3 uint8_t ModbusMaster::writeSingleCoil (uint16_t u16WriteAddress, uint8_t u8State)

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

<i>u16WriteAddress</i>	address of the coil (0x0000..0xFFFF)
<i>u8State</i>	0=OFF, non-zero=ON (0x00..0xFF)

Returns

0 on success; exception number on failure

```

474 {
475     _u16WriteAddress = u16WriteAddress;
476     _u16WriteQty = (u8State ? 0xFF00 : 0x0000);
477     return ModbusMasterTransaction(ku8MBWriteSingleCoil);
478 }
```

4.3.2.4 uint8_t ModbusMaster::writeMultipleCoils (uint16_t u16WriteAddress, uint16_t u16BitQty)

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

<i>u16WriteAddress</i>	address of the first coil (0x0000..0xFFFF)
<i>u16BitQty</i>	quantity of coils to write (1..2000, enforced by remote device)

Returns

0 on success; exception number on failure

```
521 {  
522     _u16WriteAddress = u16WriteAddress;  
523     _u16WriteQty = u16BitQty;  
524     return ModbusMasterTransaction(ku8MBWriteMultipleCoils);  
525 }
```

4.4 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.

4.4.1 Detailed Description

4.4.2 Function Documentation

4.4.2.1 `uint8_t ModbusMaster::readHoldingRegisters (uint16_t u16ReadAddress, uint16_t u16ReadQty)`

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

<code>u16ReadAddress</code>	address of the first holding register (0x0000..0xFFFF)
<code>u16ReadQty</code>	quantity of holding registers to read (1..125, enforced by remote device)

Returns

0 on success; exception number on failure

Examples:

[examples/Basic/Basic.pde](#), and [examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde](#).

```

427 {
428     _u16ReadAddress = u16ReadAddress;
429     _u16ReadQty = u16ReadQty;
430     return ModbusMasterTransaction(
431         ku8MBReadHoldingRegisters);
431 }
```

4.4.2.2 `uint8_t ModbusMaster::readInputRegisters (uint16_t u16ReadAddress, uint8_t u16ReadQty)`

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

<i>u16ReadAddress</i>	address of the first input register (0x0000..0xFFFF)
<i>u16ReadQty</i>	quantity of input registers to read (1..125, enforced by remote device)

Returns

0 on success; exception number on failure

Examples:

[examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde.](#)

```
452 {  
453   _u16ReadAddress = u16ReadAddress;  
454   _u16ReadQty = u16ReadQty;  
455   return ModbusMasterTransaction(ku8MBReadInputRegisters);  
456 }
```

4.4.2.3 uint8_t ModbusMaster::writeSingleRegister (uint16_t u16WriteAddress, uint16_t u16WriteValue)

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

<i>u16WriteAddress</i>	address of the holding register (0x0000..0xFFFF)
<i>u16WriteValue</i>	value to be written to holding register (0x0000..0xFFFF)

Returns

0 on success; exception number on failure

Examples:

[examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde.](#)

```
495 {  
496   _u16WriteAddress = u16WriteAddress;  
497   _u16WriteQty = 0;  
498   _u16TransmitBuffer[0] = u16WriteValue;  
499   return ModbusMasterTransaction(ku8MBWriteSingleRegister);  
500 }
```

4.4.2.4 uint8_t ModbusMaster::writeMultipleRegisters (uint16_t u16WriteAddress, uint16_t u16WriteQty)

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

<i>u16WriteAddress</i>	address of the holding register (0x0000..0xFFFF)
<i>u16WriteQty</i>	quantity of holding registers to write (1..123, enforced by remote device)

Returns

0 on success; exception number on failure

Examples:

[examples/Basic/Basic.pde](#), and [examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde](#).

```

549 {
550     _u16WriteAddress = u16WriteAddress;
551     _u16WriteQty = u16WriteQty;
552     return ModbusMasterTransaction(
553         ku8MBWriteMultipleRegisters);
553 }
```

4.4.2.5 uint8_t ModbusMaster::maskWriteRegister (uint16_t u16WriteAddress, uint16_t u16AndMask, uint16_t u16OrMask)

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

<i>u16WriteAddress</i>	address of the holding register (0x0000..0xFFFF)
<i>u16AndMask</i>	AND mask (0x0000..0xFFFF)
<i>u16OrMask</i>	OR mask (0x0000..0xFFFF)

Returns

0 on success; exception number on failure

```

587 {
588     _u16WriteAddress = u16WriteAddress;
589     _u16TransmitBuffer[0] = u16AndMask;
590     _u16TransmitBuffer[1] = u16OrMask;
591     return ModbusMasterTransaction(ku8MBMaskWriteRegister);
592 }
```

4.4.2.6 uint8_t ModbusMaster::readWriteMultipleRegisters (uint16_t u16ReadAddress, uint16_t u16ReadQty, uint16_t u16WriteAddress, uint16_t u16WriteQty)

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

<i>u16ReadAddress</i>	address of the first holding register (0x0000..0xFFFF)
<i>u16ReadQty</i>	quantity of holding registers to read (1..125, enforced by remote device)
<i>u16WriteAddress</i>	address of the first holding register (0x0000..0xFFFF)
<i>u16WriteQty</i>	quantity of holding registers to write (1..121, enforced by remote device)

Returns

0 on success; exception number on failure

Examples:

[examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde.](#)

```
617 {  
618   _u16ReadAddress = u16ReadAddress;  
619   _u16ReadQty = u16ReadQty;  
620   _u16WriteAddress = u16WriteAddress;  
621   _u16WriteQty = u16WriteQty;  
622   return ModbusMasterTransaction(  
        ku8MBReadWriteMultipleRegisters);  
623 }
```

4.5 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.

4.5.1 Detailed Description

4.5.2 Variable Documentation

4.5.2.1 const uint8_t ModbusMaster::ku8MBIllegalFunction = 0x01 [static]

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.

4.5.2.2 const uint8_t ModbusMaster::ku8MBIllegalDataAddress = 0x02 [static]

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

Examples:

[examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde](#).

4.5.2.3 `const uint8_t ModbusMaster::ku8MBIllegalDataValue = 0x03` `[static]`

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.

4.5.2.4 `const uint8_t ModbusMaster::ku8MBSlaveDeviceFailure = 0x04` `[static]`

Modbus protocol slave device failure exception.

An unrecoverable error occurred while the server (or slave) was attempting to perform the requested action.

4.5.2.5 `const uint8_t ModbusMaster::ku8MBSuccess = 0x00` `[static]`

[ModbusMaster](#) success.

Modbus transaction was successful; the following checks were valid:

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

Examples:

[examples/Basic/Basic.pde](#).

4.5.2.6 `const uint8_t ModbusMaster::ku8MBInvalidSlaveID = 0xE0` `[static]`

[ModbusMaster](#) invalid response slave ID exception.

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

4.5.2.7 `const uint8_t ModbusMaster::ku8MBInvalidFunction = 0xE1` `[static]`

[ModbusMaster](#) invalid response function exception.

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

4.5.2.8 `const uint8_t ModbusMaster::ku8MBResponseTimedOut = 0xE2` `[static]`

[ModbusMaster](#) response timed out exception.

The entire response was not received within the timeout period, [ModbusMaster::ku8MBResponseTimeout](#).

4.5.2.9 `const uint8_t ModbusMaster::ku8MBInvalidCRC = 0xE3` `[static]`

[ModbusMaster](#) invalid response CRC exception.

The CRC in the response does not match the one calculated.

4.6 <util/crc16.h>: CRC Computations

4.6.1 Detailed Description

```
#include <util/crc16.h>
```

This header file provides functions for calculating cyclic redundancy checks (CRC) using common polynomials. Modified by Doc Walker to be processor-independent (removed inline assembler to allow it to compile on SAM3X8E processors).

References:

Jack Crenshaw's "Implementing CRCs" article in the January 1992 issue of *Embedded Systems Programming*. This may be difficult to find, but it explains CRC's in very clear and concise terms. Well worth the effort to obtain a copy.

Chapter 5

Class Documentation

5.1 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)
- void [idle](#) (void(*)())
Set idle time callback function (cooperative multitasking).
- 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.
- void [beginTransaction](#) (uint16_t)
- uint8_t [requestFrom](#) (uint16_t, uint16_t)
- void [sendBit](#) (bool)
- void [send](#) (uint8_t)
- void [send](#) (uint16_t)
- void [send](#) (uint32_t)
- uint8_t [available](#) (void)
- uint16_t [receive](#) (void)
- 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 **writeMultipleCoils** ()
- uint8_t [writeMultipleRegisters](#) (uint16_t, uint16_t)
- Modbus function 0x10 Write Multiple Registers.*
- uint8_t **writeMultipleRegisters** ()
- 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.*
- uint8_t **readWriteMultipleRegisters** (uint16_t, uint16_t)

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()`
- `uint16_t * txBuffer`
- `uint8_t _u8TransmitBufferIndex`
- `uint16_t _u16TransmitBufferLength`
- `uint16_t * rxBuffer`
- `uint8_t _u8ResponseBufferIndex`
- `uint8_t _u8ResponseBufferLength`
- `void(* _idle)()`

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].

5.1.1 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](#).

5.1.2 Member Function Documentation

5.1.2.1 void ModbusMaster::idle (void(*)() *idle*)

Set idle time callback function (cooperative multitasking).

This function gets called in the idle time between transmission of data and response from slave. Do not call functions that read from the serial buffer that is used by [ModbusMaster](#). Use of i2c/TWI, 1-Wire, other serial ports, etc. is permitted within callback function.

See also

[ModbusMaster::ModbusMasterTransaction\(\)](#)

```
263 {
264   _idle = idle;
265 }
```

5.1.2.2 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

<i>u8MBFunction</i>	Modbus function (0x01..0xFF)
---------------------	------------------------------

Returns

0 on success; exception number on failure

```

649 {
650     uint8_t u8ModbusADU[256];
651     uint8_t u8ModbusADUSize = 0;
652     uint8_t i, u8Qty;
653     uint16_t u16CRC;
654     uint32_t u32StartTime;
655     uint8_t u8BytesLeft = 8;
656     uint8_t u8MBStatus = ku8MBSuccess;
657
658     // assemble Modbus Request Application Data Unit
659     u8ModbusADU[u8ModbusADUSize++] = _u8MBSlave;
660     u8ModbusADU[u8ModbusADUSize++] = u8MBFunction;
661
662     switch (u8MBFunction)
663     {
664     case ku8MBReadCoils:
665     case ku8MBReadDiscreteInputs:
666     case ku8MBReadInputRegisters:
667     case ku8MBReadHoldingRegisters:
668     case ku8MBReadWriteMultipleRegisters:
669         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16ReadAddress);
670         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16ReadAddress);
671         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16ReadQty);
672         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16ReadQty);
673         break;
674     }
675
676     switch (u8MBFunction)
677     {
678     case ku8MBWriteSingleCoil:
679     case ku8MBMaskWriteRegister:
680     case ku8MBWriteMultipleCoils:
681     case ku8MBWriteSingleRegister:
682     case ku8MBWriteMultipleRegisters:
683     case ku8MBReadWriteMultipleRegisters:
684         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16WriteAddress);
685         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteAddress);
686         break;
687     }
688
689     switch (u8MBFunction)
690     {
691     case ku8MBWriteSingleCoil:
692         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16WriteQty);
693         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteQty);
694         break;
695
696     case ku8MBWriteSingleRegister:
697         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[0]);
698         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[0]);
699         break;
700
701     case ku8MBWriteMultipleCoils:
702         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16WriteQty);
703         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteQty);
704         u8Qty = (_u16WriteQty % 8) ? ((_u16WriteQty >> 3) + 1) : (
        _u16WriteQty >> 3);
705         u8ModbusADU[u8ModbusADUSize++] = u8Qty;
706         for (i = 0; i < u8Qty; i++)
707         {
708             switch (i % 2)
709             {
710             case 0: // i is even
711                 u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[i >> 1]);
712                 break;
713
714             case 1: // i is odd
715                 u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[i >> 1]);
716                 break;
717             }

```

```

718     }
719     break;
720
721     case ku8MBWriteMultipleRegisters:
722     case ku8MBReadWriteMultipleRegisters:
723         u8ModbusADU[u8ModbusADUSize++] = highByte(_ul6WriteQty);
724         u8ModbusADU[u8ModbusADUSize++] = lowByte(_ul6WriteQty);
725         u8ModbusADU[u8ModbusADUSize++] = lowByte(_ul6WriteQty << 1);
726
727         for (i = 0; i < lowByte(_ul6WriteQty); i++)
728         {
729             u8ModbusADU[u8ModbusADUSize++] = highByte(_ul6TransmitBuffer[i]);
730             u8ModbusADU[u8ModbusADUSize++] = lowByte(_ul6TransmitBuffer[i]);
731         }
732         break;
733
734     case ku8MBMaskWriteRegister:
735         u8ModbusADU[u8ModbusADUSize++] = highByte(_ul6TransmitBuffer[0]);
736         u8ModbusADU[u8ModbusADUSize++] = lowByte(_ul6TransmitBuffer[0]);
737         u8ModbusADU[u8ModbusADUSize++] = highByte(_ul6TransmitBuffer[1]);
738         u8ModbusADU[u8ModbusADUSize++] = lowByte(_ul6TransmitBuffer[1]);
739         break;
740     }
741
742     // append CRC
743     u16CRC = 0xFFFF;
744     for (i = 0; i < u8ModbusADUSize; i++)
745     {
746         u16CRC = crc16_update(u16CRC, u8ModbusADU[i]);
747     }
748     u8ModbusADU[u8ModbusADUSize++] = lowByte(u16CRC);
749     u8ModbusADU[u8ModbusADUSize++] = highByte(u16CRC);
750     u8ModbusADU[u8ModbusADUSize] = 0;
751
752     // transmit request
753     for (i = 0; i < u8ModbusADUSize; i++)
754     {
755         #if defined(ARDUINO) && ARDUINO >= 100
756             MBSerial->write(u8ModbusADU[i]);
757         #else
758             MBSerial->print(u8ModbusADU[i], BYTE);
759         #endif
760     }
761
762     u8ModbusADUSize = 0;
763     MBSerial->flush();
764
765     // loop until we run out of time or bytes, or an error occurs
766     u32StartTime = millis();
767     while (u8BytesLeft && !u8MBStatus)
768     {
769         if (MBSerial->available())
770         {
771             #if __MODBUSMASTER_DEBUG__
772                 digitalWrite(4, true);
773             #endif
774             u8ModbusADU[u8ModbusADUSize++] = MBSerial->read();
775             u8BytesLeft--;
776             #if __MODBUSMASTER_DEBUG__
777                 digitalWrite(4, false);
778             #endif
779         }
780         else
781         {
782             #if __MODBUSMASTER_DEBUG__
783                 digitalWrite(5, true);
784             #endif
785             if (_idle)
786             {
787                 _idle();
788             }
789             #if __MODBUSMASTER_DEBUG__
790                 digitalWrite(5, false);
791             #endif
792         }
793
794         // evaluate slave ID, function code once enough bytes have been read
795         if (u8ModbusADUSize == 5)
796         {
797             // verify response is for correct Modbus slave
798             if (u8ModbusADU[0] != _u8MBSlave)

```

```
799     {
800         u8MBStatus = ku8MBInvalidSlaveID;
801         break;
802     }
803
804     // verify response is for correct Modbus function code (mask exception bit 7)
805     if ((u8ModbusADU[1] & 0x7F) != u8MBFunction)
806     {
807         u8MBStatus = ku8MBInvalidFunction;
808         break;
809     }
810
811     // check whether Modbus exception occurred; return Modbus Exception Code
812     if (bitRead(u8ModbusADU[1], 7))
813     {
814         u8MBStatus = u8ModbusADU[2];
815         break;
816     }
817
818     // evaluate returned Modbus function code
819     switch(u8ModbusADU[1])
820     {
821         case ku8MBReadCoils:
822         case ku8MBReadDiscreteInputs:
823         case ku8MBReadInputRegisters:
824         case ku8MBReadHoldingRegisters:
825         case ku8MBReadWriteMultipleRegisters:
826             u8BytesLeft = u8ModbusADU[2];
827             break;
828
829         case ku8MBWriteSingleCoil:
830         case ku8MBWriteMultipleCoils:
831         case ku8MBWriteSingleRegister:
832         case ku8MBWriteMultipleRegisters:
833             u8BytesLeft = 3;
834             break;
835
836         case ku8MBMaskWriteRegister:
837             u8BytesLeft = 5;
838             break;
839     }
840 }
841
842 if (millis() > (u32StartTime + ku8MBResponseTimeout))
843 {
844     u8MBStatus = ku8MBResponseTimedOut;
845 }
846
847 // verify response is large enough to inspect further
848 if (!u8MBStatus && u8ModbusADUSize >= 5)
849 {
850     // calculate CRC
851     u16CRC = 0xFFFF;
852     for (i = 0; i < (u8ModbusADUSize - 2); i++)
853     {
854         u16CRC = crc16_update(u16CRC, u8ModbusADU[i]);
855     }
856
857     // verify CRC
858     if (!u8MBStatus && (lowByte(u16CRC) != u8ModbusADU[u8ModbusADUSize - 2] ||
859         highByte(u16CRC) != u8ModbusADU[u8ModbusADUSize - 1]))
860     {
861         u8MBStatus = ku8MBInvalidCRC;
862     }
863 }
864
865 // disassemble ADU into words
866 if (!u8MBStatus)
867 {
868     // evaluate returned Modbus function code
869     switch(u8ModbusADU[1])
870     {
871         case ku8MBReadCoils:
872         case ku8MBReadDiscreteInputs:
873             // load bytes into word; response bytes are ordered L, H, L, H, ...
874             for (i = 0; i < (u8ModbusADU[2] >> 1); i++)
875             {
876                 if (i < ku8MaxBufferSize)
877                 {
878                     _u16ResponseBuffer[i] = word(u8ModbusADU[2 * i + 4], u8ModbusADU[2 * i + 3]);
879                 }
880             }
881         }
882     }
```

```
880
881     _u8ResponseBufferLength = i;
882 }
883
884 // in the event of an odd number of bytes, load last byte into zero-padded word
885 if (u8ModbusADU[2] % 2)
886 {
887     if (i < ku8MaxBufferSize)
888     {
889         _u16ResponseBuffer[i] = word(0, u8ModbusADU[2 * i + 3]);
890     }
891
892     _u8ResponseBufferLength = i + 1;
893 }
894 break;
895
896 case ku8MBReadInputRegisters:
897 case ku8MBReadHoldingRegisters:
898 case ku8MBReadWriteMultipleRegisters:
899     // load bytes into word; response bytes are ordered H, L, H, L, ...
900     for (i = 0; i < (u8ModbusADU[2] >> 1); i++)
901     {
902         if (i < ku8MaxBufferSize)
903         {
904             _u16ResponseBuffer[i] = word(u8ModbusADU[2 * i + 3], u8ModbusADU[2 * i + 4]);
905         }
906
907         _u8ResponseBufferLength = i;
908     }
909     break;
910 }
911 }
912
913 _u8TransmitBufferIndex = 0;
914 u16TransmitBufferLength = 0;
915 _u8ResponseBufferIndex = 0;
916 return u8MBStatus;
917 }
```

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

- [ModbusMaster.h](#)
- [ModbusMaster.cpp](#)

Chapter 6

File Documentation

6.1 crc16.h File Reference

6.2 ModbusMaster.cpp File Reference

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

```
#include "ModbusMaster.h"
```

6.2.1 Detailed Description

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

6.3 ModbusMaster.h File Reference

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

```
#include "WProgram.h"
#include "util/crc16.h"
```

Classes

- class [ModbusMaster](#)

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

Macros

- #define [__MODBUSMASTER_DEBUG__](#) (1)
Set to 1 to enable debugging features within class:
- #define [lowWord](#)(ww) ((uint16_t) ((ww) & 0xFFFF))

Macro to return low word of a 32-bit integer.

- `#define highWord(ww) ((uint16_t) ((ww) >> 16))`
Macro to return high word of a 32-bit integer.
- `#define LONG(hi, lo) ((uint32_t) ((hi) << 16 | (lo)))`
Macro to generate 32-bit integer from (2) 16-bit words.

6.3.1 Detailed Description

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

6.3.2 Macro Definition Documentation

6.3.2.1 `#define __MODBUSMASTER_DEBUG__ (1)`

Set to 1 to enable debugging features within class:

- pin 4 cycles for each byte read in the Modbus response
- pin 5 cycles for each millisecond timeout during the Modbus response

Chapter 7

Example Documentation

7.1 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 © 2009-2013 Doc Walker <4-20ma at wvfans dot net>

*/

#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: write TX buffer to (2) 16-bit registers starting at register 0
result = node.writeMultipleRegisters(0, 2);

// slave: 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);
    }
}
}

```

7.2 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 © 2009-2013 Doc Walker <4-20ma at wvfans dot net>

*/

#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 31..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_REG(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);
}

```

```
}
```

Index

<util/crc16.h>: CRC Computations, [23](#)

__MODBUSMASTER_DEBUG__

ModbusMaster.h, [34](#)

begin

ModbusMaster Object Instantiation/Initialization, [8](#)

clearResponseBuffer

ModbusMaster Buffer Management, [10](#)

clearTransmitBuffer

ModbusMaster Buffer Management, [11](#)

crc16.h, [33](#)

getResponseBuffer

ModbusMaster Buffer Management, [10](#)

idle

ModbusMaster, [28](#)

ku8MBIllegalDataAddress

Modbus Function Codes, Exception Codes, [20](#)

ku8MBIllegalDataValue

Modbus Function Codes, Exception Codes, [21](#)

ku8MBIllegalFunction

Modbus Function Codes, Exception Codes, [20](#)

ku8MBInvalidCRC

Modbus Function Codes, Exception Codes, [22](#)

ku8MBInvalidFunction

Modbus Function Codes, Exception Codes, [21](#)

ku8MBInvalidSlaveID

Modbus Function Codes, Exception Codes, [21](#)

ku8MBResponseTimedOut

Modbus Function Codes, Exception Codes, [21](#)

ku8MBSlaveDeviceFailure

Modbus Function Codes, Exception Codes, [21](#)

ku8MBSuccess

Modbus Function Codes, Exception Codes, [21](#)

maskWriteRegister

Modbus Function Codes for Holding/Input Registers,
[18](#)

Modbus Function Codes for Discrete Coils/Inputs, [12](#)

readCoils, [12](#)

readDiscreteInputs, [12](#)

writeMultipleCoils, [13](#)

writeSingleCoil, [13](#)

Modbus Function Codes for Holding/Input Registers, [15](#)

maskWriteRegister, [18](#)

readHoldingRegisters, [15](#)

readInputRegisters, [15](#)

readWriteMultipleRegisters, [18](#)

writeMultipleRegisters, [17](#)

writeSingleRegister, [17](#)

Modbus Function Codes, Exception Codes, [20](#)

ku8MBIllegalDataAddress, [20](#)

ku8MBIllegalDataValue, [21](#)

ku8MBIllegalFunction, [20](#)

ku8MBInvalidCRC, [22](#)

ku8MBInvalidFunction, [21](#)

ku8MBInvalidSlaveID, [21](#)

ku8MBResponseTimedOut, [21](#)

ku8MBSlaveDeviceFailure, [21](#)

ku8MBSuccess, [21](#)

ModbusMaster, [25](#)

idle, [28](#)

ModbusMaster Object Instantiation/Initialization, [7, 8](#)

ModbusMasterTransaction, [28](#)

ModbusMaster Buffer Management, [10](#)

clearResponseBuffer, [10](#)

clearTransmitBuffer, [11](#)

getResponseBuffer, [10](#)

setTransmitBuffer, [11](#)

ModbusMaster Object Instantiation/Initialization, [7](#)

begin, [8](#)

ModbusMaster, [7, 8](#)

ModbusMaster.cpp, [33](#)

ModbusMaster.h, [33](#)

__MODBUSMASTER_DEBUG__, [34](#)

ModbusMasterTransaction

ModbusMaster, [28](#)

readCoils

Modbus Function Codes for Discrete Coils/Inputs, [12](#)

readDiscreteInputs

Modbus Function Codes for Discrete Coils/Inputs, [12](#)

readHoldingRegisters

Modbus Function Codes for Holding/Input Registers,
[15](#)

readInputRegisters

Modbus Function Codes for Holding/Input Registers,
[15](#)

readWriteMultipleRegisters

Modbus Function Codes for Holding/Input Registers,
[18](#)

setTransmitBuffer

ModbusMaster Buffer Management, [11](#)

writeMultipleCoils

Modbus Function Codes for Discrete Coils/Inputs, [13](#)

writeMultipleRegisters

Modbus Function Codes for Holding/Input Registers,
[17](#)

writeSingleCoil

Modbus Function Codes for Discrete Coils/Inputs, [13](#)

writeSingleRegister

Modbus Function Codes for Holding/Input Registers,
[17](#)