ModbusMaster

v1.0.0

Generated by Doxygen 1.8.12

Contents

Chapter 1

Module Index

1.1 Modules

Here is a list of all modules:

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

Chapter 4

Module Documentation

4.1 ModbusMaster Object Instantiation/Initialization

Functions

- ModbusMaster::ModbusMaster ()

  Constructor.
- void ModbusMaster::begin (uint8_t, Stream &serial)

  Initialize class object.

4.1.1 Detailed Description

4.1.2 Function Documentation

4.1.2.1 ModbusMaster()

```
ModbusMaster::ModbusMaster (
            void  )
```

Constructor.

Creates class object; initialize it using ModbusMaster::begin().

```
45 {
46   _idle = 0;
47   _preTransmission = 0;
48   _postTransmission = 0;
49 }
```

4.1.2.2 begin()

```
void ModbusMaster::begin (
            uint8_t slave,
            Stream & serial )
```

Initialize class object.

Assigns the Modbus slave ID and serial port. Call once class has been instantiated, typically within setup().

Parameters

slave	Modbus slave ID (1..255)
&serial	reference to serial port object (Serial, Serial1, ... Serial3)

Examples:

examples/Basic/Basic.pde, examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde, and examples/RS485_HalfDuplex/RS485_HalfDuplex.ino.

```
62 {
63 //  txBuffer = (uint16_t*) calloc(ku8MaxBufferSize, sizeof(uint16_t));
64   _u8MBSlave = slave;
65   _serial = &serial;
66   _u8TransmitBufferIndex = 0;
67   u16TransmitBufferLength = 0;
68 
69 #if __MODBUSMASTER_DEBUG__
70   pinMode(__MODBUSMASTER_DEBUG_PIN_A__, OUTPUT);
71   pinMode(__MODBUSMASTER_DEBUG_PIN_B__, OUTPUT);
72 #endif
73 }
```

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 getResponseBuffer()

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

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)

Examples:

examples/Basic/Basic.pde, examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde, and examples/RS485_HalfDuplex/RS485_HalfDuplex.ino.

```
229 {
230   if (u8Index < ku8MaxBufferSize)
231   {
232     return _u16ResponseBuffer[u8Index];
233   }
234   else
235   {
236     return 0xFFFF;
237   }
238 }
```

4.2.2.2 clearResponseBuffer()

```
void ModbusMaster::clearResponseBuffer ( )
```

Clear Modbus response buffer.

See also

ModbusMaster::getResponseBuffer(uint8_t u8Index)

```
248 {
249   uint8_t i;
250
251   for (i = 0; i < ku8MaxBufferSize; i++)
252   {
253     _u16ResponseBuffer[i] = 0;
254   }
255 }
```

4.2.2.3 setTransmitBuffer()

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

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

Examples:

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

```
268 {
269   if (u8Index < ku8MaxBufferSize)
270   {
271     _u16TransmitBuffer[u8Index] = u16Value;
272     return ku8MBSuccess;
273   }
274   else
275   {
276     return ku8MBIllegalDataAddress;
277   }
278 }
```

4.2.2.4 clearTransmitBuffer()

```
void ModbusMaster::clearTransmitBuffer ( )
```

Clear Modbus transmit buffer.

See also

ModbusMaster::setTransmitBuffer(uint8_t u8Index, uint16_t u16Value)

```
288 {
289   uint8_t i;
290
291   for (i = 0; i < ku8MaxBufferSize; i++)
292   {
293     _u16TransmitBuffer[i] = 0;
294   }
295 }
```

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 readCoils()

```
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

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

Examples:

examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde.

```
322 {
323   _u16ReadAddress = u16ReadAddress;
324   _u16ReadQty = u16BitQty;
325   return ModbusMasterTransaction(ku8MBReadCoils);
326 }
```

4.3.2.2 readDiscreteInputs()

```
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

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

```
354 {
355   _u16ReadAddress = u16ReadAddress;
356   _u16ReadQty = u16BitQty;
357   return ModbusMasterTransaction(ku8MBReadDiscreteInputs);
358 }
```

4.3.2.3 writeSingleCoil()

```
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

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

Returns

0 on success; exception number on failure

Examples:

examples/RS485_HalfDuplex/RS485_HalfDuplex.ino.

```
426 {
427   _u16WriteAddress = u16WriteAddress;
428   _u16WriteQty = (u8State ? 0xFF00 : 0x0000);
429   return ModbusMasterTransaction(ku8MBWriteSingleCoil);
430 }
```

4.3.2.4 writeMultipleCoils()

```
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

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

```
473 {
474   _u16WriteAddress = u16WriteAddress;
475   _u16WriteQty = u16BitQty;
476   return ModbusMasterTransaction(ku8MBWriteMultipleCoils);
477 }
```

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 readHoldingRegisters()

```
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

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

Examples:

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

```
379 {
380   _u16ReadAddress = u16ReadAddress;
381   _u16ReadQty = u16ReadQty;
382   return ModbusMasterTransaction(
      ku8MBReadHoldingRegisters);
383 }
```

4.4.2.2 readInputRegisters()

```
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

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

Examples:

examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde, and examples/RS485_HalfDuplex/RS485_↩ HalfDuplex.ino.

```
404 {
405   _u16ReadAddress = u16ReadAddress;
406   _u16ReadQty = u16ReadQty;
407   return ModbusMasterTransaction(ku8MBReadInputRegisters);
408 }
```

4.4.2.3 writeSingleRegister()

```
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

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

Examples:

examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde.

```
447 {
448   _u16WriteAddress = u16WriteAddress;
449   _u16WriteQty = 0;
450   _u16TransmitBuffer[0] = u16WriteValue;
451   return ModbusMasterTransaction(ku8MBWriteSingleRegister);
452 }
```

4.4.2.4 writeMultipleRegisters()

```
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

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

Examples:

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

```
501 {
502   _u16WriteAddress = u16WriteAddress;
503   _u16WriteQty = u16WriteQty;
504   return ModbusMasterTransaction(
      ku8MBWriteMultipleRegisters);
505 }
```

4.4.2.5 maskWriteRegister()

```
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

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

```
539 {
540   _u16WriteAddress = u16WriteAddress;
541   _u16TransmitBuffer[0] = u16AndMask;
542   _u16TransmitBuffer[1] = u16OrMask;
543   return ModbusMasterTransaction(ku8MBMaskWriteRegister);
544 }
```

4.4.2.6 readWriteMultipleRegisters()

```
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

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

Examples:

examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde.

```
569 {
570   _u16ReadAddress = u16ReadAddress;
571   _u16ReadQty = u16ReadQty;
572   _u16WriteAddress = u16WriteAddress;
573   _u16WriteQty = u16WriteQty;
574   return ModbusMasterTransaction(
      ku8MBReadWriteMultipleRegisters);
575 }
```

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 ku8MBIllegalFunction

```
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 ku8MBIllegalDataAddress

```
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

1.

Examples:

examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde.

4.5.2.3 ku8MBIllegalDataValue

```
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 ku8MBSlaveDeviceFailure

```
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 ku8MBSuccess

```
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, and examples/RS485_HalfDuplex/RS485_HalfDuplex.ino.

4.5.2.6 ku8MBInvalidSlaveID

```
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 ku8MBInvalidFunction

```
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 ku8MBResponseTimedOut

```
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 ku8MBInvalidCRC

```
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

Functions

- static uint16_t crc16_update (uint16_t crc, uint8_t a)

  Processor-independent CRC-16 calculation.

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.

4.6.2 Function Documentation

4.6.2.1 crc16_update()

```
static uint16_t crc16_update (
            uint16_t crc,
            uint8_t a )  [static]
```

Processor-independent CRC-16 calculation.

Polynomial: x^16 + x^15 + x^2 + 1 (0xA001)
Initial value: 0xFFFF

This CRC is normally used in disk-drive controllers.

Parameters

uint16_t	crc (0x0000..0xFFFF)
uint8_t	a (0x00..0xFF)

Returns

calculated CRC (0x0000..0xFFFF)

```
72 {
73   int i;
74
75   crc ^= a;
76   for (i = 0; i < 8; ++i)
77   {
78     if (crc & 1)
79       crc = (crc >> 1) ^ 0xA001;
80     else
81       crc = (crc >> 1);
82   }
83
84   return crc;
85 }
```

4.7 "util/word.h": Utility Functions for Manipulating Words

Functions

- static uint16_t lowWord (uint32_t ww)

  Return low word of a 32-bit integer.
- static uint16_t highWord (uint32_t ww)

  Return high word of a 32-bit integer.

4.7.1 Detailed Description

```
#include "util/word.h"
```

This header file provides utility functions for manipulating words.

4.7.2 Function Documentation

4.7.2.1 lowWord()

```
static uint16_t lowWord (
            uint32_t ww )  [inline], [static]
```

Return low word of a 32-bit integer.

Parameters

uint32_t	ww (0x00000000..0xFFFFFFFF)

Returns

low word of input (0x0000..0xFFFF)

Examples:

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

```
47 {
48   return (uint16_t) ((ww) & 0xFFFF);
49 }
```

4.7.2.2 highWord()

```
static uint16_t highWord (
            uint32_t ww )  [inline], [static]
```

Return high word of a 32-bit integer.

Parameters

uint32_t	ww (0x00000000..0xFFFFFFFF)

Returns

high word of input (0x0000..0xFFFF)

Examples:

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

```
59 {
60   return (uint16_t) ((ww) >> 16);
61 }
```

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.
- void begin (uint8_t, Stream &serial)
  Initialize class object.
- void idle (void(∗)())
  Set idle time callback function (cooperative multitasking).
- void preTransmission (void(∗)())
  Set pre-transmission callback function.
- void postTransmission (void(∗)())
  Set post-transmission callback function.
- 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 **beginTransmission** (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

- Stream * _serial

  reference to serial port object
- uint8_t _u8MBSlave

  Modbus slave (1..255) 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**)()
- void(* **_preTransmission**)()
- void(* **_postTransmission**)()

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 uint16_t ku16MBResponseTimeout = 2000

  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, examples/PhoenixContact_nanoLC/PhoenixContact_nanoLC.pde, and examples/RS485_HalfDuplex/RS485_HalfDuplex.ino.

5.1.2 Member Function Documentation

5.1.2.1 idle()

```
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()

```
182 {
183   _idle = idle;
184 }
```

5.1.2.2 preTransmission()

```
void ModbusMaster::preTransmission (
             void(*)() preTransmission )
```

Set pre-transmission callback function.

This function gets called just before a Modbus message is sent over serial. Typical usage of this callback is to enable an RS485 transceiver's Driver Enable pin, and optionally disable its Receiver Enable pin.

See also

ModbusMaster::ModbusMasterTransaction()
ModbusMaster::postTransmission()

Examples:

examples/RS485_HalfDuplex/RS485_HalfDuplex.ino.

```
197 {
198   _preTransmission = preTransmission;
199 }
```

5.1.2.3 postTransmission()

```
void ModbusMaster::postTransmission (
             void(*)() postTransmission )
```

Set post-transmission callback function.

This function gets called after a Modbus message has finished sending (i.e. after all data has been physically transmitted onto the serial bus).

Typical usage of this callback is to enable an RS485 transceiver's Receiver Enable pin, and disable its Driver Enable pin.

See also

ModbusMaster::ModbusMasterTransaction()
ModbusMaster::preTransmission()

Examples:

examples/RS485_HalfDuplex/RS485_HalfDuplex.ino.

```
215 {
216   _postTransmission = postTransmission;
217 }
```

5.1.2.4 ModbusMasterTransaction()

```
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

```
601 {
602   uint8_t u8ModbusADU[256];
603   uint8_t u8ModbusADUSize = 0;
604   uint8_t i, u8Qty;
605   uint16_t u16CRC;
606   uint32_t u32StartTime;
607   uint8_t u8BytesLeft = 8;
608   uint8_t u8MBStatus = ku8MBSuccess;
609
610   // assemble Modbus Request Application Data Unit
611   u8ModbusADU[u8ModbusADUSize++] = _u8MBSlave;
612   u8ModbusADU[u8ModbusADUSize++] = u8MBFunction;
613
614   switch(u8MBFunction)
615   {
616     case ku8MBReadCoils:
617     case ku8MBReadDiscreteInputs:
618     case ku8MBReadInputRegisters:
619     case ku8MBReadHoldingRegisters:
620     case ku8MBReadWriteMultipleRegisters:
621       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16ReadAddress);
622       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16ReadAddress);
623       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16ReadQty);
624       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16ReadQty);
625       break;
626   }
627
628   switch(u8MBFunction)
629   {
630     case ku8MBWriteSingleCoil:
631     case ku8MBMaskWriteRegister:
632     case ku8MBWriteMultipleCoils:
633     case ku8MBWriteSingleRegister:
634     case ku8MBWriteMultipleRegisters:
635     case ku8MBReadWriteMultipleRegisters:
636       u8ModbusADU[u8ModbusADUSize++] = highByte(_u16WriteAddress);
637       u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteAddress);
```

```
638       break;
639     }
640
641     switch(u8MBFunction)
642     {
643       case ku8MBWriteSingleCoil:
644         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16WriteQty);
645         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteQty);
646         break;
647
648       case ku8MBWriteSingleRegister:
649         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[0]);
650         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[0]);
651         break;
652
653       case ku8MBWriteMultipleCoils:
654         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16WriteQty);
655         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteQty);
656         u8Qty = (_u16WriteQty % 8) ? ((_u16WriteQty >> 3) + 1) : (
    _u16WriteQty >> 3);
657         u8ModbusADU[u8ModbusADUSize++] = u8Qty;
658         for (i = 0; i < u8Qty; i++)
659         {
660           switch(i % 2)
661           {
662             case 0: // i is even
663               u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[i >> 1]);
664               break;
665
666             case 1: // i is odd
667               u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[i >> 1]);
668               break;
669           }
670         }
671         break;
672
673       case ku8MBWriteMultipleRegisters:
674       case ku8MBReadWriteMultipleRegisters:
675         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16WriteQty);
676         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteQty);
677         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16WriteQty << 1);
678
679         for (i = 0; i < lowByte(_u16WriteQty); i++)
680         {
681           u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[i]);
682           u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[i]);
683         }
684         break;
685
686       case ku8MBMaskWriteRegister:
687         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[0]);
688         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[0]);
689         u8ModbusADU[u8ModbusADUSize++] = highByte(_u16TransmitBuffer[1]);
690         u8ModbusADU[u8ModbusADUSize++] = lowByte(_u16TransmitBuffer[1]);
691         break;
692     }
693
694     // append CRC
695     u16CRC = 0xFFFF;
696     for (i = 0; i < u8ModbusADUSize; i++)
697     {
698       u16CRC = crc16_update(u16CRC, u8ModbusADU[i]);
699     }
700     u8ModbusADU[u8ModbusADUSize++] = lowByte(u16CRC);
701     u8ModbusADU[u8ModbusADUSize++] = highByte(u16CRC);
702     u8ModbusADU[u8ModbusADUSize] = 0;
703
704     // flush receive buffer before transmitting request
705     while (_serial->read() != -1);
706
707     // transmit request
708     if (_preTransmission)
709     {
710       _preTransmission();
711     }
712     for (i = 0; i < u8ModbusADUSize; i++)
713     {
714 #if defined(ARDUINO) && ARDUINO >= 100
715       _serial->write(u8ModbusADU[i]);
716 #else
717       _serial->print(u8ModbusADU[i], BYTE);
```

```
718 #endif
719   }
720
721   u8ModbusADUSize = 0;
722   _serial->flush();    // flush transmit buffer
723   if (_postTransmission)
724   {
725     _postTransmission();
726   }
727
728   // loop until we run out of time or bytes, or an error occurs
729   u32StartTime = millis();
730   while (u8BytesLeft && !u8MBStatus)
731   {
732     if (_serial->available())
733     {
734 #if __MODBUSMASTER_DEBUG__
735       digitalWrite(__MODBUSMASTER_DEBUG_PIN_A__, true);
736 #endif
737       u8ModbusADU[u8ModbusADUSize++] = _serial->read();
738       u8BytesLeft--;
739 #if __MODBUSMASTER_DEBUG__
740       digitalWrite(__MODBUSMASTER_DEBUG_PIN_A__, false);
741 #endif
742     }
743     else
744     {
745 #if __MODBUSMASTER_DEBUG__
746       digitalWrite(__MODBUSMASTER_DEBUG_PIN_B__, true);
747 #endif
748       if (_idle)
749       {
750         _idle();
751       }
752 #if __MODBUSMASTER_DEBUG__
753       digitalWrite(__MODBUSMASTER_DEBUG_PIN_B__, false);
754 #endif
755     }
756
757     // evaluate slave ID, function code once enough bytes have been read
758     if (u8ModbusADUSize == 5)
759     {
760       // verify response is for correct Modbus slave
761       if (u8ModbusADU[0] != _u8MBSlave)
762       {
763         u8MBStatus = ku8MBInvalidSlaveID;
764         break;
765       }
766
767       // verify response is for correct Modbus function code (mask exception bit 7)
768       if ((u8ModbusADU[1] & 0x7F) != u8MBFunction)
769       {
770         u8MBStatus = ku8MBInvalidFunction;
771         break;
772       }
773
774       // check whether Modbus exception occurred; return Modbus Exception Code
775       if (bitRead(u8ModbusADU[1], 7))
776       {
777         u8MBStatus = u8ModbusADU[2];
778         break;
779       }
780
781       // evaluate returned Modbus function code
782       switch(u8ModbusADU[1])
783       {
784         case ku8MBReadCoils:
785         case ku8MBReadDiscreteInputs:
786         case ku8MBReadInputRegisters:
787         case ku8MBReadHoldingRegisters:
788         case ku8MBReadWriteMultipleRegisters:
789           u8BytesLeft = u8ModbusADU[2];
790           break;
791
792         case ku8MBWriteSingleCoil:
793         case ku8MBWriteMultipleCoils:
794         case ku8MBWriteSingleRegister:
795         case ku8MBWriteMultipleRegisters:
796           u8BytesLeft = 3;
797           break;
798
```

```
799         case ku8MBMaskWriteRegister:
800           u8BytesLeft = 5;
801           break;
802       }
803     }
804     if ((millis() - u32StartTime) > ku16MBResponseTimeout)
805     {
806       u8MBStatus = ku8MBResponseTimedOut;
807     }
808   }
809
810   // verify response is large enough to inspect further
811   if (!u8MBStatus && u8ModbusADUSize >= 5)
812   {
813     // calculate CRC
814     u16CRC = 0xFFFF;
815     for (i = 0; i < (u8ModbusADUSize - 2); i++)
816     {
817       u16CRC = crc16_update(u16CRC, u8ModbusADU[i]);
818     }
819
820     // verify CRC
821     if (!u8MBStatus && (lowByte(u16CRC) != u8ModbusADU[u8ModbusADUSize - 2] ||
822       highByte(u16CRC) != u8ModbusADU[u8ModbusADUSize - 1]))
823     {
824       u8MBStatus = ku8MBInvalidCRC;
825     }
826   }
827
828   // disassemble ADU into words
829   if (!u8MBStatus)
830   {
831     // evaluate returned Modbus function code
832     switch(u8ModbusADU[1])
833     {
834       case ku8MBReadCoils:
835       case ku8MBReadDiscreteInputs:
836         // load bytes into word; response bytes are ordered L, H, L, H, ...
837         for (i = 0; i < (u8ModbusADU[2] >> 1); i++)
838         {
839           if (i < ku8MaxBufferSize)
840           {
841             _u16ResponseBuffer[i] = word(u8ModbusADU[2 * i + 4], u8ModbusADU[2 * i + 3]);
842           }
843
844           _u8ResponseBufferLength = i;
845         }
846
847         // in the event of an odd number of bytes, load last byte into zero-padded word
848         if (u8ModbusADU[2] % 2)
849         {
850           if (i < ku8MaxBufferSize)
851           {
852             _u16ResponseBuffer[i] = word(0, u8ModbusADU[2 * i + 3]);
853           }
854
855           _u8ResponseBufferLength = i + 1;
856         }
857         break;
858
859       case ku8MBReadInputRegisters:
860       case ku8MBReadHoldingRegisters:
861       case ku8MBReadWriteMultipleRegisters:
862         // load bytes into word; response bytes are ordered H, L, H, L, ...
863         for (i = 0; i < (u8ModbusADU[2] >> 1); i++)
864         {
865           if (i < ku8MaxBufferSize)
866           {
867             _u16ResponseBuffer[i] = word(u8ModbusADU[2 * i + 3], u8ModbusADU[2 * i + 4]);
868           }
869
870           _u8ResponseBufferLength = i;
871         }
872         break;
873     }
874   }
875
876   _u8TransmitBufferIndex = 0;
877   u16TransmitBufferLength = 0;
878   _u8ResponseBufferIndex = 0;
879   return u8MBStatus;
```

```
880 }
```

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

CRC Computations.

Functions

- static uint16_t crc16_update (uint16_t crc, uint8_t a)
  Processor-independent CRC-16 calculation.

6.1.1 Detailed Description

CRC Computations.

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"
#include "util/word.h"
```

Classes

- class ModbusMaster

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

Macros

- #define __MODBUSMASTER_DEBUG__ (0)

  Set to 1 to enable debugging features within class:
- #define **__MODBUSMASTER_DEBUG_PIN_A__** 4
- #define **__MODBUSMASTER_DEBUG_PIN_B__** 5

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 __MODBUSMASTER_DEBUG__

```
#define __MODBUSMASTER_DEBUG__ (0)
```

Set to 1 to enable debugging features within class:

- PIN A cycles for each byte read in the Modbus response
- PIN B cycles for each millisecond timeout during the Modbus response

6.4 word.h File Reference

Utility Functions for Manipulating Words.

Functions

- static uint16_t lowWord (uint32_t ww)

  Return low word of a 32-bit integer.
- static uint16_t highWord (uint32_t ww)

  Return high word of a 32-bit integer.

6.4.1 Detailed Description

Utility Functions for Manipulating Words.

Chapter 7

Example Documentation

7.1 examples/Basic/Basic.pde

```
/*

  Basic.pde - example using ModbusMaster library

  Library:: ModbusMaster
  Author:: Doc Walker <4-20ma@wvfans.net>

  Copyright:: 2009-2016 Doc Walker

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.

*/

#include <ModbusMaster.h>


// instantiate ModbusMaster object
ModbusMaster node;


void setup()
{
  // use Serial (port 0); initialize Modbus communication baud rate
  Serial.begin(19200);

  // communicate with Modbus slave ID 2 over Serial (port 0)
  node.begin(2, Serial);
}


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.

  Library:: ModbusMaster
  Author:: Doc Walker <4-20ma@wvfans.net>

  Copyright:: 2009-2016 Doc Walker

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.

*/

#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
ModbusMaster nanoLC;


void setup()
{
```

```
  // use Serial (port 0); initialize Modbus communication baud rate
  Serial.begin(19200);

  // communicate with Modbus slave ID 1 over Serial (port 0)
  nanoLC.begin(1, Serial);
}


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);
}
```

7.3 examples/RS485_HalfDuplex/RS485_HalfDuplex.ino

```
/*

  RS485_HalfDuplex.pde - example using ModbusMaster library to communicate
  with EPSolar LS2024B controller using a half-duplex RS485 transceiver.

  This example is tested against an EPSolar LS2024B solar charge controller.
  See here for protocol specs:
  http://www.solar-elektro.cz/data/dokumenty/1733_modbus_protocol.pdf

  Library:: ModbusMaster
  Author:: Marius Kintel <marius at kintel dot net>

  Copyright:: 2009-2016 Doc Walker

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.

*/

#include <ModbusMaster.h>

#define MAX485_DE      3
#define MAX485_RE_NEG  2

// instantiate ModbusMaster object
ModbusMaster node;

void preTransmission()
{
  digitalWrite(MAX485_RE_NEG, 1);
  digitalWrite(MAX485_DE, 1);
}

void postTransmission()
{
  digitalWrite(MAX485_RE_NEG, 0);
  digitalWrite(MAX485_DE, 0);
}

void setup()
{
  pinMode(MAX485_RE_NEG, OUTPUT);
  pinMode(MAX485_DE, OUTPUT);
  // Init in receive mode
  digitalWrite(MAX485_RE_NEG, 0);
  digitalWrite(MAX485_DE, 0);

  // Modbus communication runs at 115200 baud
  Serial.begin(115200);

  // Modbus slave ID 1
  node.begin(1, Serial);
  // Callbacks allow us to configure the RS485 transceiver correctly
  node.preTransmission(preTransmission);
```

```
  node.postTransmission(postTransmission);
}

bool state = true;

void loop()
{
  uint8_t result;
  uint16_t data[6];

  // Toggle the coil at address 0x0002 (Manual Load Control)
  result = node.writeSingleCoil(0x0002, state);
  state = !state;

  // Read 16 registers starting at 0x3100)
  result = node.readInputRegisters(0x3100, 16);
  if (result == node.ku8MBSuccess)
  {
    Serial.print("Vbatt: ");
    Serial.println(node.getResponseBuffer(0x04)/100.0f);
    Serial.print("Vload: ");
    Serial.println(node.getResponseBuffer(0xC0)/100.0f);
    Serial.print("Pload: ");
    Serial.println((node.getResponseBuffer(0x0D) +
                    node.getResponseBuffer(0x0E) << 16)/100.0f);
  }

  delay(1000);
}
```

Index