MODBUS - Code Documentation

Info

This is a document explaining some aspects of the MODBUS code library. In theory this library can be applied to other microcontrollers/dev boards. To get a better understanding it is advised to also look into the PI_MBUS_300.pdf while reading this.

Version 1.0, 24.10.2023.

Acronyms and Definitions

- **CRC** - Cyclical Redundancy Check
- **MSB** - Most Significant Bit
- **LSB** - Least Significant Bit
- **Holding Register** - 16 bit registers may, be read or written. Can be used for inputs, outputs, configuration data, or any requirements for "holding" data.
- **Input Register** - 16 bit registers used for input and may only be read.
- **Coil(s)** - 1 bit registers, used to control discrete outputs and may be read or written to.
- **Discrete Input Registers** - 1 bit registers used as inputs and may only be read.

Microcontroller set up

Important

To function properly a interrupt should be enabled for the USART port used.

MODBUS Function Codes

- **0x01** - Read Coil(s)
- **0x02** - Read Input Coil(s), this array is read only
- **0x03** - Read Holding Register(s)
- **0x04** - Read Input Register(s), this array is read only
- **0x05** - Write Single Coil
- **0x06** - Write Single Holding Register
- **0x0F** - Write Multiple Coils
- **0x16** - Write Multiple Holding Registers

Note: There are more functions per the MODBUS standard, but are just not implemented. The main functionality can be accomplished with just the mentioned functions

Almost ail of the functions consist of 2 parts:

- Extracting the wanted register addresses
- Preparing/extracting this data from the slaves register databases by putting them in the TxData array.

Send Data - Function

```
void sendData (uint8_t *data, int size)
{
  uint16_t crc = crc16(data, size);
  data[size] = crc&0xFF;    // CRC LOW
  data[size+1] = (crc>>8)&0xFF;  // CRC HIGH

  HAL_UART_Transmit(&huart1, data, size+2, 1000);
}
```

The function prepares the data that needs to be sent back to the master by calculating the CRC and attaching it to the TX data array and then transmitting it through UART back to the master device.

This function is called inside respective MODBUS functions.

MODBUS Exception Codes

> **Quote**
>
> "If the slave receives the query without a communication error, but cannot handle it, the slave will return an exception response informing the master of the nature of the error"

- **0x01 - Illegal function**, the function code received in the query is not recognized by the slave or is not allowed by the slave.
- **0x02 - Illegal Data Address**, the data address received in the query is not an allowed address for the slave, i.e., the register does not exist. If multiple registers were requested, at least one was not permitted.
- **0x03 - Illegal Data Value**, the value contained in the query's data field is nit acceptable to the slave.
- **0x04 - Slave Device Failure**, an unrecoverable error occurred.
- **0x06 - Slave Device Busy**, the slave is engaged in processing a long-duration command. The master should try again later.
- **0x0A - Gateway path unavailable**, gateway could not establish communication with target device.
- **0x0B - Gateway Target Device Failed to Respond**, specialized use in conjunction with gateways, indicates no response was received from the target device.
- **0x11 - Gateway Target Device Failed to Respond**, No response from slave, request timed out.

In case of an exception the response message consists of:

- **Function Code Field** - the slave sets the MSB of the function code to 1. This makes the function code value in an exception response exactly 80 hexadecimal higher than the value would be for a normal response.
- **Data Field** - the slave returns the exception code in the data field.

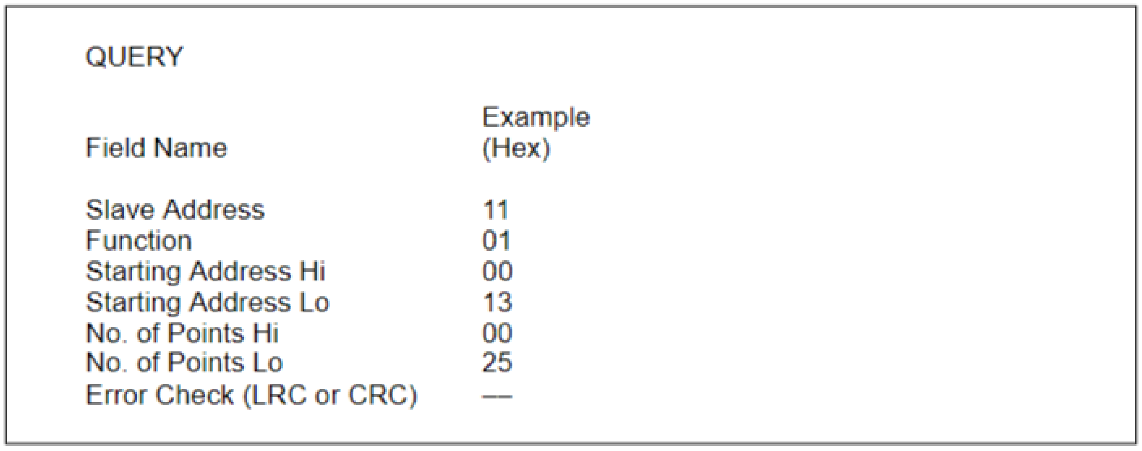
C Code

```
void modbusException (uint8_t exceptioncode)
{
        //| SLAVE_ID | FUNCTION_CODE | Exception code | CRC     |
        //| 1 BYTE   |  1 BYTE       |    1 BYTE      | 2 BYTES |

        TxData[0] = RxData[0];       // slave ID
        TxData[1] = RxData[1]|0x80;  // adding 1 to the MSB of the function code
        TxData[2] = exceptioncode;   // Load the Exception code
        sendData(TxData, 3);
}
```

Reading Coil(s) - 0x01

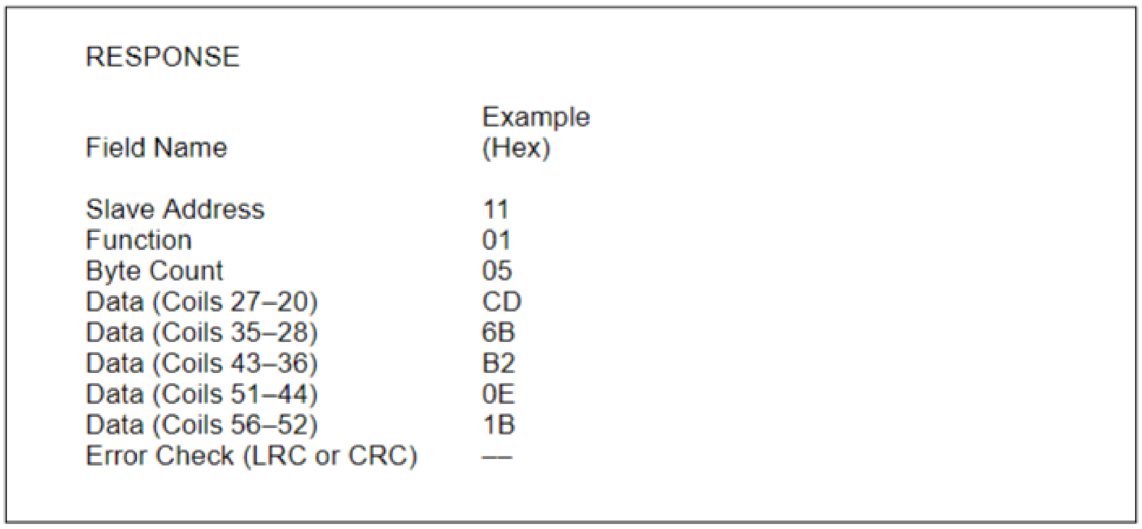
Info

This function allows the master device to read a maximum number of 2000 coils *(1 bit registers)* stored in the slave device. The first part of the function extracts the number of registers requested and the start address for the 1st register. This part is almost identical for every function hence it will only be mentioned if it has significant differences.



QUERY

Field Name	Example (Hex)
Slave Address	11
Function	01
Starting Address Hi	00
Starting Address Lo	13
No. of Points Hi	00
No. of Points Lo	25
Error Check (LRC or CRC)	––



RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	01
Byte Count	05
Data (Coils 27–20)	CD
Data (Coils 35–28)	6B
Data (Coils 43–36)	B2
Data (Coils 51–44)	0E
Data (Coils 56–52)	1B
Error Check (LRC or CRC)	––

Extracting Register Addresses

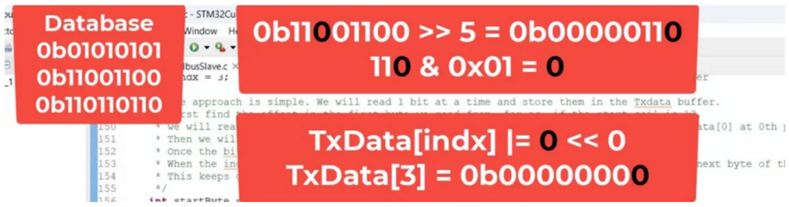
```
uint8_t readCoils (void)
{
        uint16_t startAddr = ((RxData[2]<<8)|RxData[3]);  // start Coil Address

        uint16_t numCoils = ((RxData[4]<<8)|RxData[5]);   // number to coils master has requested
        if ((numCoils<1)||(numCoils>2000))  // maximum no. of coils as per the MODBUS standard
        {
                modbusException (ILLEGAL_DATA_VALUE);  // send an exception
                return 0;
        }

        uint16_t endAddr = startAddr+numCoils-1;  // Last coils address
        if (endAddr>199)  // this number depends on the amount of registers the slave address has
        {
                modbusException(ILLEGAL_DATA_ADDRESS);   // send an exception
                return 0;
        }
```

The function also includes checking if the number of registers requested is legal *(no more than 2000)* and seeing if the register address exists.

Preparing the Response - TxData

```
        //reset TxData buffer
        memset (TxData, '\0', 256);

        // Prepare TxData buffer

        TxData[0] = SLAVE_ID;  // slave ID
        TxData[1] = RxData[1];  // function code
        TxData[2] = (numCoils/8) + ((numCoils%8)>0 ? 1:0);  // Byte count
        int indx = 3;  // we need to keep track of how many bytes has been stored in TxData Buffer

        int startByte = startAddr/8;  // which byte we have to start extracting the data from
        uint16_t bitPosition = startAddr%8;  // The shift position in the first byte
        int indxPosition = 0;  // The shift position in the current indx of the TxData buffer

        // Load the actual data into TxData buffer
        for (int i=0; i<numCoils; i++)
        {
                TxData[indx] |= ((Coils_Database[startByte] >> bitPosition) &0x01) << indxPosition;
                indxPosition++; bitPosition++;
                if (indxPosition>7)  // if the indxposition exceeds 7, we have to copy the data
 into the next byte position
                {
                        indxPosition = 0;
                        indx++;
                }
                if (bitPosition>7)  // if the bitposition exceeds 7, we have to increment the
 startbyte
                {
                        bitPosition=0;
                        startByte++;
                }
        }

        if (numCoils%8 != 0)indx++;  // increment the indx variable, only if the numcoils is not a
 multiple of 8
        sendData(TxData, indx);
        return 1;   // success
}
```

- It should be noted that the coils are stored as **uint8_t** on the slave, this means extra bitwise operations need to be made when finding in which byte the requested coil is in.
- The third byte in the TxData array is the byte count, so how many bytes is the incoming data distributed over. This is calculated by knwoing the number of coils the master requested.
- The below image explains how the wanted bit is extracted from the database array



```
Database
0b01010101
0b11001100
0b110110110

0b11001100 >> 5 = 0b00000110
110 & 0x01 = 0

TxData[indx] |= 0 << 0
TxData[3] = 0b00000000
```

Reading Input Coils - 0x02

Attention

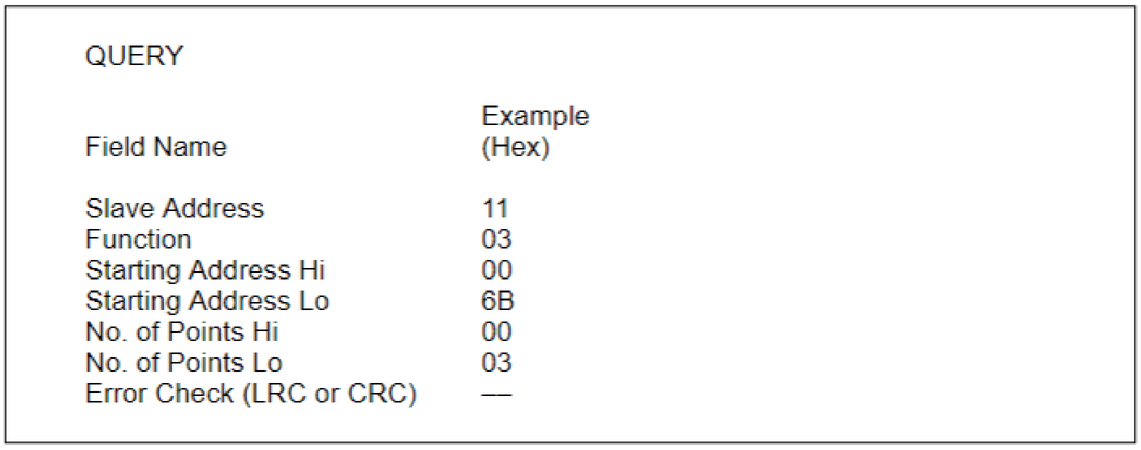
This function is **identical to the "Reading Coils function"** except the database array is different. This function uses the *Input_Database* which is defined as a constant so the master device cant write to it.

Reading Holding Registers - 0x03

Info

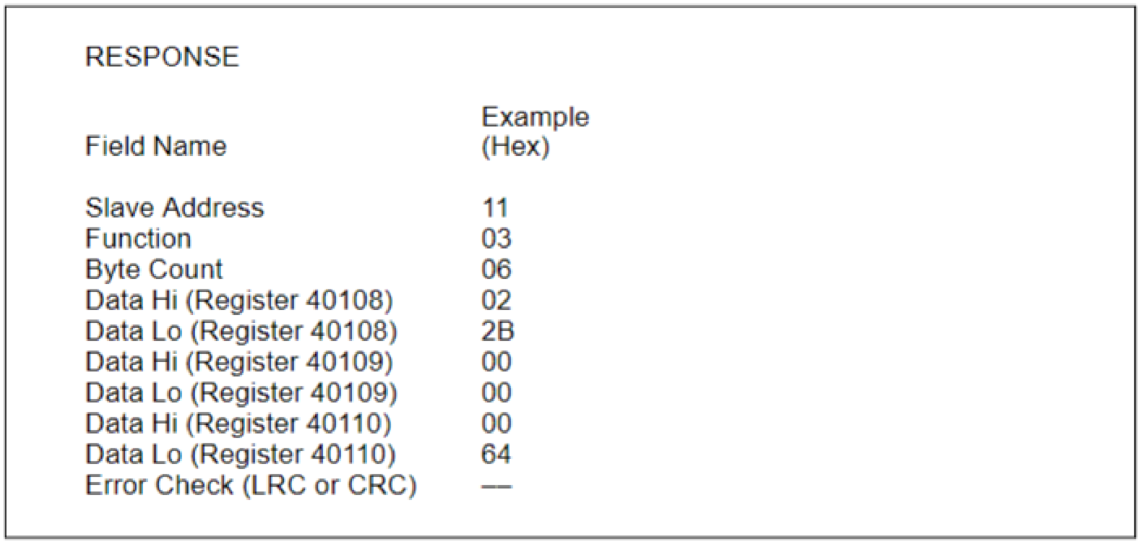
This function allows the master device to read 16bit register values from the Holding Register Database array *(Holding_Register_Database)* from the slave device.

The first part of the function is almost the same as **Reading Coils**, so it will not be covered.



QUERY

Field Name	Example (Hex)
Slave Address	11
Function	03
Starting Address Hi	00
Starting Address Lo	6B
No. of Points Hi	00
No. of Points Lo	03
Error Check (LRC or CRC)	––



RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	03
Byte Count	06
Data Hi (Register 40108)	02
Data Lo (Register 40108)	2B
Data Hi (Register 40109)	00
Data Lo (Register 40109)	00
Data Hi (Register 40110)	00
Data Lo (Register 40110)	64
Error Check (LRC or CRC)	––

Preparing Response - TxData

```
 // Prepare TxData buffer

 //| SLAVE_ID | FUNCTION_CODE | BYTE COUNT | DATA      | CRC     |
 //| 1 BYTE   |  1 BYTE       |  1 BYTE    | N*2 BYTES | 2 BYTES |

 TxData[0] = SLAVE_ID;  // slave ID
 TxData[1] = RxData[1];  // function code
 TxData[2] = numRegs*2;  // Byte count
 int indx = 3;  //to keep track of how many bytes has been stored in TxData Buffer

 for (int i=0; i<numRegs; i++)   // Load the actual data into TxData buffer
 {
   TxData[indx++] = (Holding_Registers_Database[startAddr]>>8)&0xFF;  // extract the higher byte
   TxData[indx++] = (Holding_Registers_Database[startAddr])&0xFF;   // extract the lower byte
   startAddr++;  // increment the register address
 }

 sendData(TxData, indx);
 return 1;   // success
}
```

- The third byte is the number of bytes the slave is sending, this is equal to the number of registers times 2, because the register is 16bits.
- The 16bit registers are converted into 2 bytes,, with the higher byte being extracted first.
  - The 16bit Register value is shifted to the right by 8 places to get the the higher 8bits
  - The register value is ANDed with 0xFF to get the lower 8bits.

Reading Input Registers - 0x04

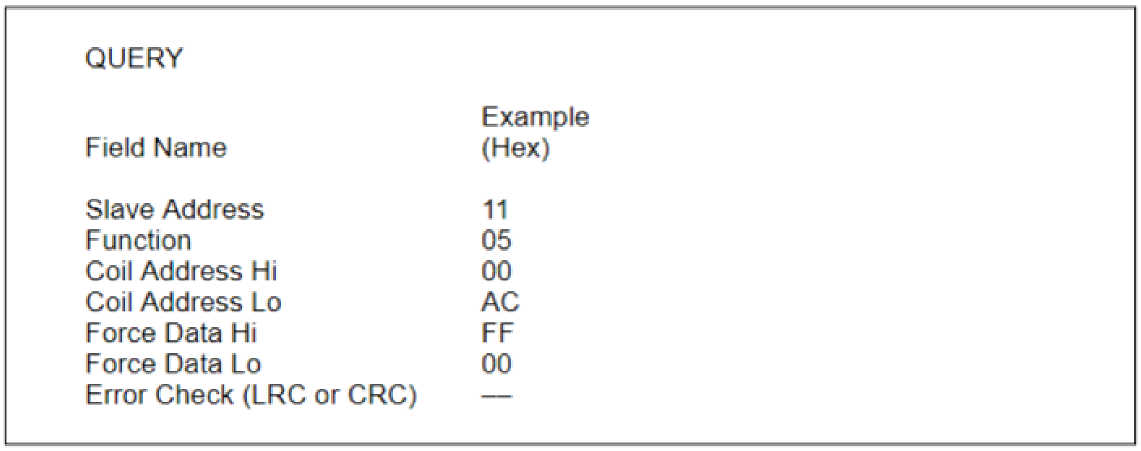
Attention

This function is **identical to the "Reading Holding Registers function"** except the database array is different. This function uses the *Input_Register_Database* which is defined as a constant so the master device cant write to it.

Write Single Coil - 0x05

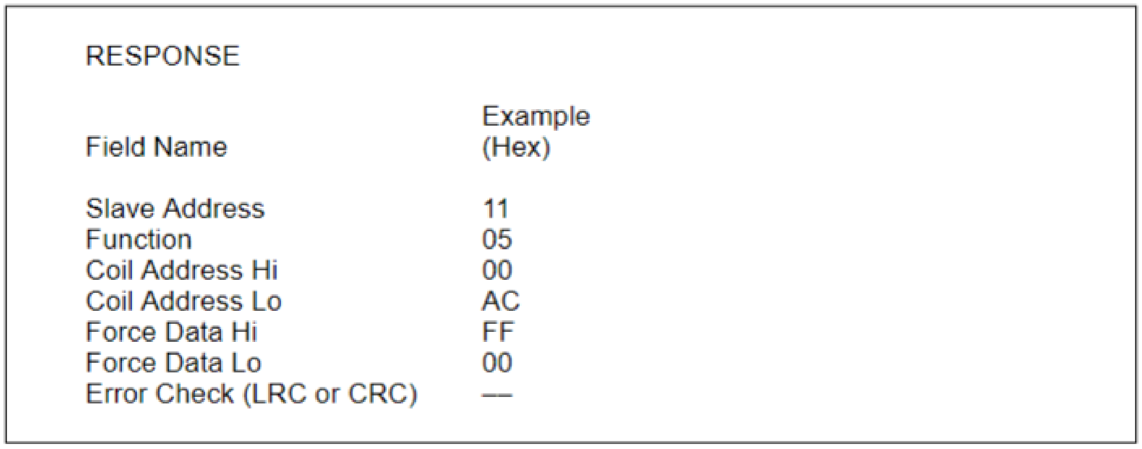
Info

This function allows the master device to write to a single 1bit register on the slave device.



QUERY

Field Name	Example (Hex)
Slave Address	11
Function	05
Coil Address Hi	00
Coil Address Lo	AC
Force Data Hi	FF
Force Data Lo	00
Error Check (LRC or CRC)	––



RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	05
Coil Address Hi	00
Coil Address Lo	AC
Force Data Hi	FF
Force Data Lo	00
Error Check (LRC or CRC)	––

Extracting the start address and calculating the bit position

```
uint8_t writeSingleCoil (void)
{
	uint16_t startAddr = ((RxData[2]<<8)|RxData[3]);  // start Coil Address

	if (startAddr>199)  // The Coil Address can not be more than 199 as we only have record of 200 Coils in total
	{
		modbusException(ILLEGAL_DATA_ADDRESS);   // send an exception
		return 0;
	}
```

```
        /* Calculation for the bit in the database, where the modification will be done */
        int startByte = startAddr/8;  // which byte we have to start writing the data into
        uint16_t bitPosition = startAddr%8;  // The shift position in the first byte
```

- **startbyte** is the byte in the database array where the bit is being written to
- **bitposition** is the position of the bit in said byte

  *Example: If the coil address is 15, the startByte = 14/8 = 1 and bitposition = 14%8 = 6. So here we are going to modify the **6th bit** in the **1st Byte** of the database.*

Writing to the coil

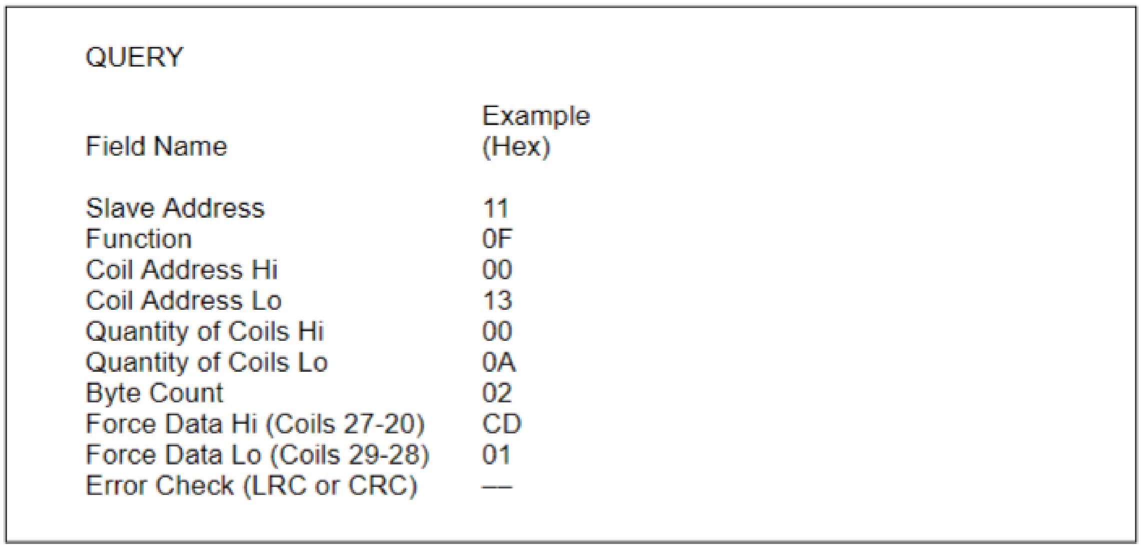
- If the 2 data bytes are **FF 00**, the coil will turn **ON**
- If the 2 data bytes are **00 00**, the coil will turn **OFF**

```
if ((RxData[4] == 0xFF) && (RxData[5] == 0x00))
        {
                Coils_Database[startByte] |= 1<<bitPosition; // Replace that bit with 1
        }

        else if ((RxData[4] == 0x00) && (RxData[5] == 0x00))
        {
                Coils_Database[startByte] &= ~(1<<bitPosition); // Replace that bit with 0
        }
```

Writing to Multiple Coils

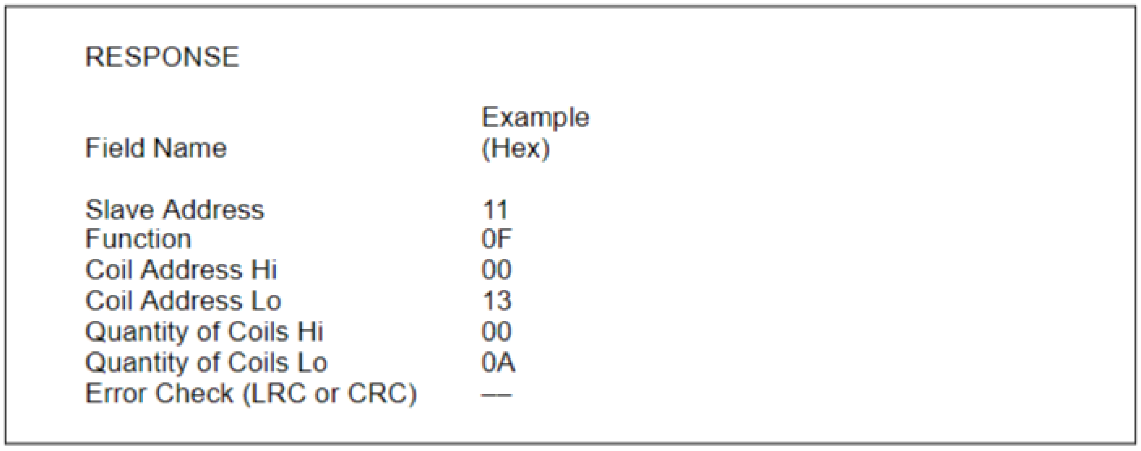
Info

This function allows the master to write to up to 1968 coils in one transmission to the coil database on the slave device.



QUERY

Field Name	Example (Hex)
Slave Address	11
Function	0F
Coil Address Hi	00
Coil Address Lo	13
Quantity of Coils Hi	00
Quantity of Coils Lo	0A
Byte Count	02
Force Data Hi (Coils 27-20)	CD
Force Data Lo (Coils 29-28)	01
Error Check (LRC or CRC)	––



RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	0F
Coil Address Hi	00
Coil Address Lo	13
Quantity of Coils Hi	00
Quantity of Coils Lo	0A
Error Check (LRC or CRC)	––

Extracting the start address and number of coils

```
uint8_t writeMultiCoils (void)
{
        uint16_t startAddr = ((RxData[2]<<8)|RxData[3]);  // start Coil Address

        uint16_t numCoils = ((RxData[4]<<8)|RxData[5]);   // number to coils master has requested
        if ((numCoils<1)||(numCoils>1968))  // maximum no. of coils as per the PDF
        {
                modbusException (ILLEGAL_DATA_VALUE);  // send an exception
                return 0;
        }

        uint16_t endAddr = startAddr+numCoils-1;  // Last coils address
        if (endAddr>199)  // the number depends on the size of the array
        {
                modbusException(ILLEGAL_DATA_ADDRESS);   // send an exception
                return 0;
        }
```

Writing to the coils

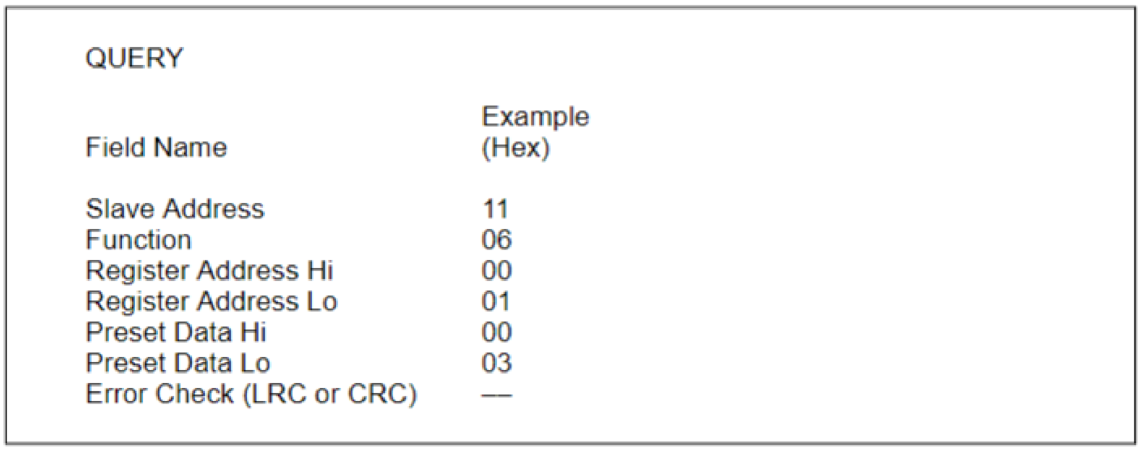
Attention

This process is very similar to writing to a single coil just used in a for loop

Write to Single Holding Register - 0x06

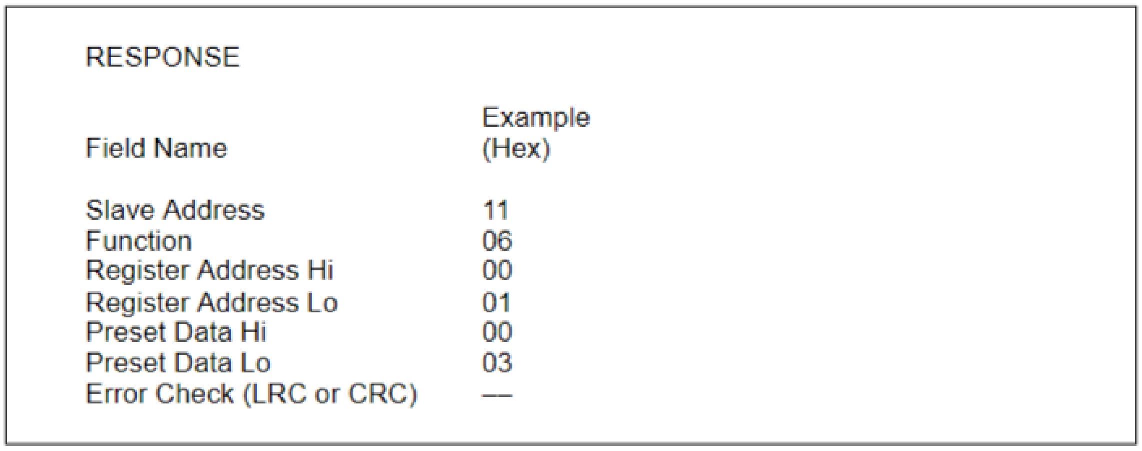
Info

This function allows the master device to write to a single 16bit register. To read from the database where this register is stored use function "Read Holding Register - 0x03"



QUERY

Field Name	Example (Hex)
Slave Address	11
Function	06
Register Address Hi	00
Register Address Lo	01
Preset Data Hi	00
Preset Data Lo	03
Error Check (LRC or CRC)	––



RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	06
Register Address Hi	00
Register Address Lo	01
Preset Data Hi	00
Preset Data Lo	03
Error Check (LRC or CRC)	––

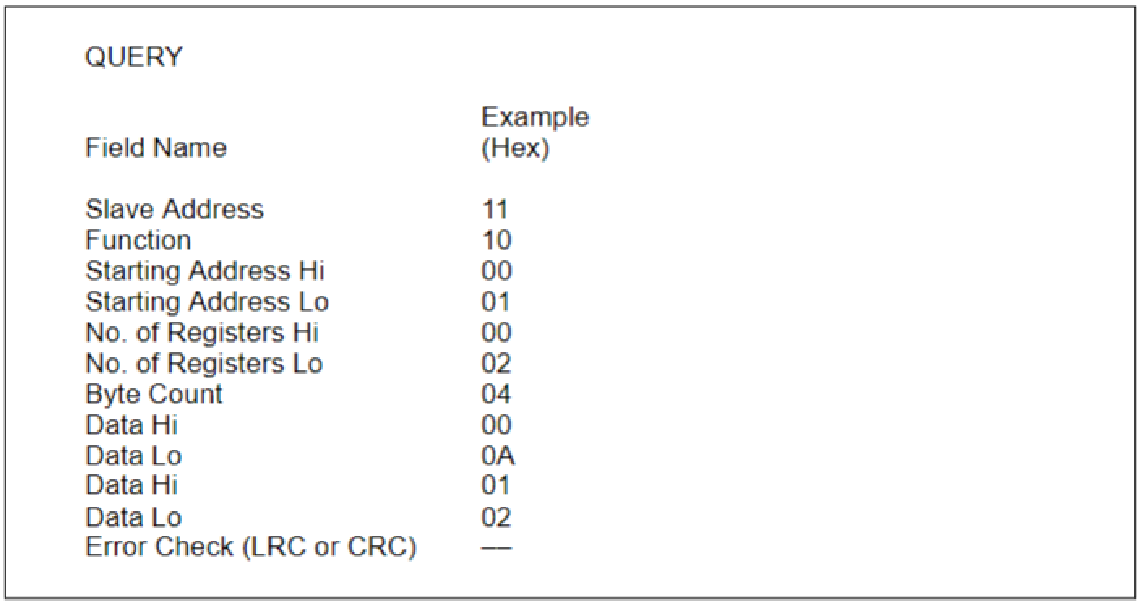
```
uint8_t writeSingleReg (void)
{
        uint16_t startAddr = ((RxData[2]<<8)|RxData[3]);  // start Register Address

        if (startAddr>49)  // the number depends on the size of the array the registers are stored in
        {
                modbusException(ILLEGAL_DATA_ADDRESS);   // send an exception
                return 0;
        }

        Holding_Registers_Database[startAddr] = (RxData[4]<<8)|RxData[5];

        // Prepare Response

        TxData[0] = SLAVE_ID;    // slave ID
        TxData[1] = RxData[1];   // function code
        TxData[2] = RxData[2];   // Start Addr HIGH Byte
        TxData[3] = RxData[3];   // Start Addr LOW Byte
        TxData[4] = RxData[4];   // Reg Data HIGH Byte
        TxData[5] = RxData[5];   // Reg Data LOW  Byte

        sendData(TxData, 6);
        return 1;   // success
}
```

- In order to write to a register the master simply sends the address of the register it wants to write to and the data it wants to write to it.
- Since one register is 16bits, byte 4 and 5 from the RxData array need to be combined.
- The response from the slave is exactly the same as the request.

Write to Multiple Holding Registers - 0x16

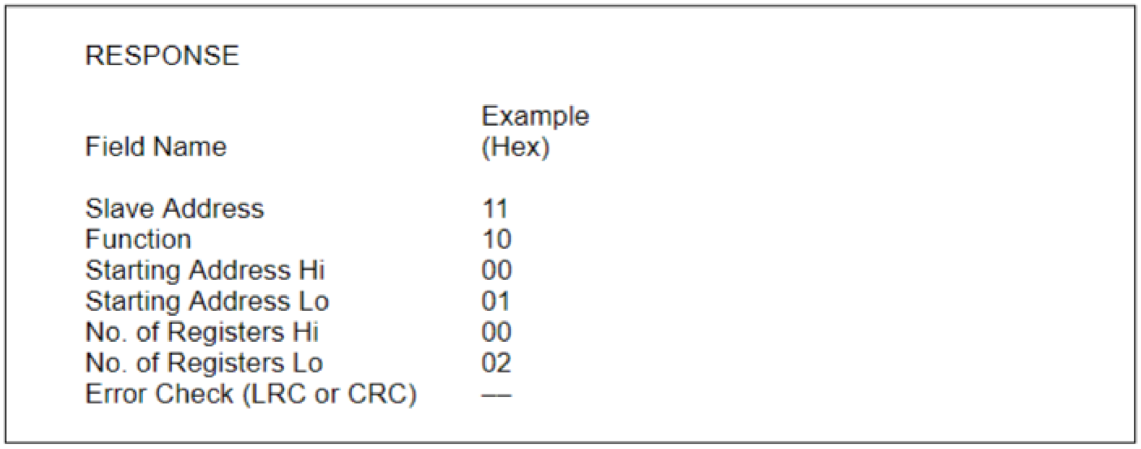
Info

This function allows the master device to write data to multiple 16bit registers in the slave device.



QUERY

Field Name	Example (Hex)
Slave Address	11
Function	10
Starting Address Hi	00
Starting Address Lo	01
No. of Registers Hi	00
No. of Registers Lo	02
Byte Count	04
Data Hi	00
Data Lo	0A
Data Hi	01
Data Lo	02
Error Check (LRC or CRC)	––



RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	10
Starting Address Hi	00
Starting Address Lo	01
No. of Registers Hi	00
No. of Registers Lo	02
Error Check (LRC or CRC)	––

The start of this function is almost the same as "Write to Single Holding Register", hence it wont be repeated.

Extracting the start address and number of registers

```
uint8_t writeHoldingRegs (void)
{
        uint16_t startAddr = ((RxData[2]<<8)|RxData[3]);  // start Register Address

        uint16_t numRegs = ((RxData[4]<<8)|RxData[5]);   // number to registers master has
requested
```

```
	if ((numRegs<1)||(numRegs>123))  // maximum no. of Registers as per the MODBUS standardw
	{
		modbusException (ILLEGAL_DATA_VALUE);  // send an exception
		return 0;
	}

	uint16_t endAddr = startAddr+numRegs-1;  // end Register
	if (endAddr>49)  // number depends on the size of the array the registers are in
	{
		modbusException(ILLEGAL_DATA_ADDRESS);   // send an exception
		return 0;
	}
```

Writing to the registers and preparing the response

```
	int indx = 7;  // to keep track of index in RxData
	for (int i=0; i<numRegs; i++)
	{
		Holding_Registers_Database[startAddr++] = (RxData[indx++]<<8)|RxData[indx++];
	}

	// Prepare Response

	//| SLAVE_ID | FUNCTION_CODE | Start Addr | num of Regs    | CRC     |
	//| 1 BYTE   |  1 BYTE       |  2 BYTE    | 2 BYTES      | 2 BYTES |

	TxData[0] = SLAVE_ID;    // slave ID
	TxData[1] = RxData[1];   // function code
	TxData[2] = RxData[2];   // Start Addr HIGH Byte
	TxData[3] = RxData[3];   // Start Addr LOW Byte
	TxData[4] = RxData[4];   // num of Regs HIGH Byte
	TxData[5] = RxData[5];   // num of Regs LOW Byte

	sendData(TxData, 6);
	return 1;   // success
}
```

- Since one register is 16bits, 2 bytes from the RxData array need to be combined.