User manual for liblightmodbus v1.4

Jacek Wieczorek
and all the contributors

July 2, 2018

Contents

1 About liblightmodbus

Liblightmodbus is a lightweight cross-platform Modbus RTU library. Its main advantages are possibility of extensive configuration and modular structure, allowing user to pick only needed features when building.
Liblightmodbus is developed with embedded targets as well as personal computers in mind.

2 Building liblightmodbus

The library can be built for various systems in many different configurations using two build systems: CMake (recommended) or plain Makefile+Bash scripts. See sections **??** and **??**.

2.1 CMake standard build

> ☞ In order to build library, CMake 3.3 or higher is required.

In order to start CMake build, create a new directory where building will happen (`mkdir build`) and enter it (`cd build`). For simple, default configuration build for PC, you can call CMake straight away (`cmake ..`). For custom build configuration see section **??** This will result in creation of `Makefile` - you can use `make` to build the library. The resulting static library file will be located in the same directory you're in.

2.2 Customized CMake build

Only difference between standard and customized CMake builds are the arguments passed to it. CMake variables can be set by adding arguments in form `-D<VARIABLE>="<VALUE>"` to its invocation. In following subsections you'll find information about build variables that affect library configuration.

2.2.1 Managing library modules

`MODULES` CMake variable determines what library modules and functionalities are going to be included during the build process. It should take value of desired module names list, separated with semicolons (and no spaces!). See below for list of available modules:

- `SLAVE_BASE` - slave basic feature set (required by all other slave-related modules)
- `F01S` - slave support for Modbus function 01
- `F02S` - slave support for Modbus function 02
- `F03S` - slave support for Modbus function 03
- `F04S` - slave support for Modbus function 04
- `F05S` - slave support for Modbus function 05
- `F06S` - slave support for Modbus function 06
- `F15S` - slave support for Modbus function 15
- `F16S` - slave support for Modbus function 16
- `F22S` - slave support for Modbus function 22
- `MASTER_BASE` - master basic feature set (required by all other master-related modules)
- `F01M` - master support for Modbus function 01
- `F02M` - master support for Modbus function 02
- `F03M` - master support for Modbus function 03
- `F04M` - master support for Modbus function 04

- F05M - master support for Modbus function 05
- F06M - master support for Modbus function 06
- F15M - master support for Modbus function 15
- F16M - master support for Modbus function 16
- F22M - master support for Modbus function 22
- SLAVE_USER_FUNCTIONS - support for user-defined Modbus functions on slave side (??)
- MASTER_USER_FUNCTIONS - support for user-defined Modbus functions on master side (??)
- REGISTER_CALLBACK - slave register callback function (??)
- COIL_CALLBACK - slave coil callback function (??)
- NO_MASTER_DATA_BUFFER - disable exclusive master's processed data buffer (??)
- MASTER_INVASIVE_PARSING - allow master to modify received frame (use with NO_MASTER_DATA_BUFFER) (??)
- EXPERIMENTAL - enable experimental features
- ADDON_EXAMINE - enable standalone modbusExamine (??) function add-on

The default MODULES variable value is: ,,SLAVE_BASE;F01S;F02S;F03S;F04S;F05S;F06S;F15S;F16S;F22S; SLAVE_USER_FUNCTIONS;MASTER_BASE;F01M;F02M;F03M;F04M;F05M;F06M;F15M;F16M;F22M; MASTER_USER_FUNCTIONS;ADDON_EXAMINE"

If you only want to specify which modules should be added when building, you can use the ADD_MODULES variable. The MODULES variable will become concatenation of default modules list and ADD_MODULES.

2.2.2 Disabling dynamic memory allocation

Liblightmodbus also has feature to disable dynamic memory allocation in order to make it more suitable for embedded systems. Use following CMake options to enable this feature for desired library components:

- -DSTATIC_MEM_SLAVE_REQUEST=<size> - set slave request buffer desired size in bytes
- -DSTATIC_MEM_SLAVE_RESPONSE=<size> - set slave response buffer desired size in bytes
- -DSTATIC_MEM_MASTER_REQUEST=<size> - set master request buffer desired size in bytes
- -DSTATIC_MEM_MASTER_RESPONSE=<size> - set master response buffer desired size in bytes
- -DSTATIC_MEM_MASTER_DATA=<size> - set master processed data buffer desired size in bytes

2.2.3 Endianness configuration

Modbus is strictly big-endian protocol. That means you have to specify target system endianness when building. By default, CMake will check your system endainness and apply that setting, but in order to override it, use -DENDIANNES="big/little" argument.

2.2.4 Specifying compiler and linker

To change used compiler and/or linker, use following arguments: -DCMAKE_C_COMPILER="<compiler>", -DCMAKE_LINKER="<linker>".

2.2.5 Building for AVR

Building the library for AVR microcontrollers is made easy thank to the `AVR` CMake variable. If you want to build for AVR, just add `-DAVR=<part>` argument to the CMake invocation, where `<part>` is the target microcontroller type (for instance `atmega328`).

2.2.6 Debug/release builds

In order to specify build type - debug or release, use `-DCMAKE_BUILD_TYPE=<type>` where `<type>` is either ,,`Debug`" or ,,`Release`".

2.2.7 Coverage testing build

Even though it's probably not what you want to do, you can enable coverage test build with `-DCOVERAGE_TEST="1"`.

2.3 Building with deprecated, proprietary build system

> ☞ The old build system use is deprecated and it will not be supported in versions newer than v1.4.

Firstly, you will need to copy all files from `old-build-system` directory to the project root. Then you will be able to use `./genconf.sh` script to generate build configurations (details on usage in its help message: `./genconf.sh --help`). The available options more or less correspond to the ones described in section **??**. After the configuration is generated, you can use `make` command to trigger the build. In order to build for AVR, use dedicated makeifile - `make -f makefile-avr`.

2.4 Library configuration file

During the build process, a special library configuration header file - `include/lightmodbus/libconf.h` is created. It contains macros specifying current library configuration. Names of these macros correspond to CMake configuration variable names.
If you know, what you are doing you can edit this file manually, however keep in mind that library will need rebuilding after making any changes.

3 Library core

3.1 Data types

This section describes data types declared in the library main header file: `lightmodbus/lightmodbus.h`.

3.1.1 Error type – `ModbusError`

3.1.2 Modbus exception type – `ModbusExceptionCode`

`ModbusExceptionCode` is a special enum type meant for storing Modbus protocol exception codes. It is defined as follows:

```
1 typedef enum modbusExceptionCode
2 {
3     MODBUS_EXCEP_ILLEGAL_FUNCTION = 1,
4     MODBUS_EXCEP_ILLEGAL_ADDRESS = 2,
5     MODBUS_EXCEP_ILLEGAL_VALUE = 3,
6     MODBUS_EXCEP_SLAVE_FAILURE = 4,
7     MODBUS_EXCEP_ACK = 5,
8     MODBUS_EXCEP_NACK = 7
9 } ModbusExceptionCode;
```

The integer values correspond to actual Modbus exception codes of the same meaning.

This type has its use in `modbusBuildException` (**??**) slave function, `ModbusMaster` (**??**) `exception` member and `ModbusSlave` (**??**) `lastException` member.

3.1.3 Modbus data type type – `ModbusDataType`

`ModbusDataType` is an enum type describing format of the data user is dealing with. It is defined as:

```
1 typedef enum modbusDataType
2 {
3     MODBUS_HOLDING_REGISTER = 1,
4     MODBUS_INPUT_REGISTER = 2,
5     MODBUS_COIL = 4,
6     MODBUS_DISCRETE_INPUT = 8
7 } ModbusDataType;
```

Enumerated values correspond to Modbus protocol data types: holding register, input register, coil and discrete input.

3.1.4 Modbus parsing helper type – `ModbusParser`

3.2 Functions

The core part of the library declares few functions commonly used in the library code, that might also be useful for the user. Prototypes of all functions described in this section can be found in the `include/lightmodbus.h` header file.

3.2.1 Bit masks operation functions – `modbusMaskRead`, `modbusMaskWrite`

```
uint8_t modbusMaskRead( const uint8_t *mask, uint16_t maskLength, uint16_t bit )
uint8_t modbusMaskWrite( uint8_t *mask, uint16_t maskLength, uint16_t bit, uint8_t
    ↪ value )
```

`modbusMaskRead` and `modbusMaskWrite` are bit mask operation functions.

`modbusMaskRead` returns `bit`-th bit from byte array `mask` of length `maskLength` bytes.

`modbusMaskWrite` writes `value` to `bit`-th bit in byte array `mask` of length `maskLength` bytes. If no error occurs, the bit value is returned.

☞ If the desired bit lies outside the array, these functions return `MODBUS_ERROR_OTHER`.

3.2.2 Modbus 16-bit CRC calculation function – `modbusCRC`

```
uint16_t modbusCRC( const uint8_t *data, uint16_t length )
```

`modbusCRC` function calculates and returns 16-bit Modbus cyclic redundancy checksum for memory area consisting of `length` bytes, starting at `data`.

3.2.3 Modbus endianness functions – `modbusSwapEndian`, `modbusMatchEndian`

```
1  static inline uint16_t modbusSwapEndian( uint16_t data )
2      { return ( data << 8 ) | ( data >> 8 ); }
3
4  #ifdef LIGHTMODBUS_BIG_ENDIAN
5      static inline uint16_t modbusMatchEndian( uint16_t data )
6          { return data; }
7  #else
8      static inline uint16_t modbusMatchEndian( uint16_t data )
9          { return modbusSwapEndian( data ); }
10 #endif
```

`modbusSwapEndian` and `modbusMatchEndian` are functions meant for changing 16-bit data portions endianness.

`modbusSwapEndian` works unconditionally and always returns 16-bit word of data with endianness altered.

`modbusSwapEndian` works only when the system library is working on is not big-endian. In other words, the function matches endiannes of the given data to with Modbus protocol endianness.

4 Slave device functionality

4.1 Slave-related data types

4.1.1 Slave device data container – `ModbusSlave`

4.1.2 Register query type – `ModbusRegisterQuery`

`ModbusRegisterQuery` is special enumeration type used when either slave register or coil callback function is enabled. That implies this type can only be used if `LIGHTMODBUS_REGISTER_CALLBACK` or `LIGHTMODBUS_COIL_CALLBACK` is defined. It's defined as:

```
1 typedef enum modbusRegisterQuery
2 {
3     MODBUS_REGQ_R,
4     MODBUS_REGQ_W,
5     MODBUS_REGQ_R_CHECK,
6     MODBUS_REGQ_W_CHECK
7 } ModbusRegisterQuery;
```

Each value corresponds to different type of register query:

- `MODBUS_REGQ_R` - read register/coil
- `MODBUS_REGQ_W` - write register/coil
- `MODBUS_REGQ_R_CHECK` - check if register/coil can be read
- `MODBUS_REGQ_W_CHECK` - check if register/coil can be written

Value of this type is passed to user defined register/coil callback functions in order to access data or check if it can be read or written. For more information on register callbacks, see `ModbusRegisterCallbackFunction` (**??**).

> ☞ This is experimental feature only available if `REGISTER_CALLBACK` or `COIL_CALLBACK` library module was included during compilation. See section **??** for more information.

4.1.3 User-defined Modbus function type – `ModbusSlaveUserFunction`

```
1 #ifdef LIGHTMODBUS_SLAVE_USER_FUNCTIONS
2     struct modbusSlave;
3     typedef struct modbusSlaveUserFunction
4     {
5         uint8_t function;
6         ModbusError ( *handler )( struct modbusSlave *status, ModbusParser *parser
              ↪ );
7     } ModbusSlaveUserFunction;
8 #endif
```

`ModbusSlaveUserFunction` type associates Modbus function code with dedicated parsing function defined by user. In other words, it informs the library that Modbus frame using function of code

`function` should be passed to user function `handler` for parsing.

For more information please see section **??** on slave side user defined functions.

> ☞ This feature is only available if `SLAVE_USER_FUNCTIONS` library module was included during compilation. See section **??** for more information.

4.1.4 Register callback function type – `ModbusRegisterCallbackFunction`

```
typedef uint16_t ( *ModbusRegisterCallbackFunction )( ModbusRegisterQuery query,
    ↪ ModbusDataType datatype, uint16_t index, uint16_t value )
```

`ModbusRegisterCallbackFunction` type is a pointer to a function that can serve as register/coil callback function. Such function accepts different kinds of queries (`ModbusRegisterQuery` (**??**)), determines access rights to different registers and mediates in register reads and writes. For more information, see section **??**

> ☞ This is experimental feature only available if `REGISTER_CALLBACK` or `COIL_CALLBACK` library module was included during compilation. See section **??** for more information.

4.2 Setup and cleanup

The library provides two functions for setting up and cleaning up the `ModbusSlave` (**??**) structure. They are described in following subsections.

4.2.1 Modbus slave device initialization function – `modbusSlaveInit`

```
ModbusError modbusSlaveInit( ModbusSlave *status )
```

`modbusSlaveInit` function initializes `ModbusSlave` (**??**) structure pointed by `status` pointer for use. It sets up internal buffers and default values and verifies values already written to the structure by the user.

On success `MODBUS_OK` error value is returned.

new errors?

4.2.2 Modbus slave device destructor function - `modbusSlaveEnd`

```
ModbusError modbusSlaveEnd( ModbusSlave *status )
```

`modbusSlaveEnd` function cleans up data stored in the `ModbusSlave` (**??**) structure pointed by `status` pointer. This function must be called before freeing memory allocated for the `ModbusSlave` structure.
On success, the function returns `MODBUS_OK` error value.

errors

4.3 Request processing

4.3.1 Universal request parser function – `modbusParseRequest`

```
ModbusError modbusParseRequest( ModbusSlave *status )
```

`modbusParseRequest` function processes request for slave device currently loaded into `ModbusSlave` (**??**) structure pointed by `status`.

The function processes request frame loaded in `status->request.frame` of `status->request.length` bytes length. The frame is interpreted using dedicated `modbusParseRequest**` function or one defined by user if provided (see section **??** for more information).

If function execution succeeds, a response frame of `status->response.length` bytes length is written to `status->response.frame`.

> ☞ Upon calling, the function automatically attempts to free memory allocated[a] for response frame (`status->response.frame`). If, for some reason, you decided to free it by yourself, make sure you set this pointer to `NULL` before calling this function.
>
> [a] If dynamic memory allocation is enabled

On successful exit, error value of `MODBUS_OK` is returned. If Modbus exception frame is returned in the response buffer, the function returns `MODBUS_ERROR_EXCEPTION`[1]. `MODBUS_ERROR_CRC` indicates that frame CRC was incorrect and that due to this fact, parsing was not attempted. Other error values mean erroneous execution - see `ModbusError` (**??**).

errors

4.3.2 Modbus exception builder function – `modbusBuildException`

4.3.3 User-defined Modbus functions

> ☞ This feature is only available if `SLAVE_USER_FUNCTIONS` library module was included during compilation. See section **??** for more information.

Since liblightmodbus version 1.4, users can define their own Modbus functions and provide parsing functions for the library to deal with them. If `LIGHTMODBUS_SLAVE_USER_FUNCTIONS` macro is defined, `ModbusSlave` (**??**) structure contains pointer to array of `ModbusSlaveUserFunction` (**??**) structures.

In order to provide library hint which function should be treated differently, user should set up an array of `ModbusSlaveUserFunction` structures and provide pointer to it in `ModbusSlave.userFunctions`. The number of user-defined functions in the array should be written to `ModbusSlave.userFunctionCount`. See the example below:

```
1 static ModbusSlaveUserFunction userf[2] =
2 {
```

[1] This implies successful execution

```
    {77, foo},
    {16, NULL},
};
slave.userFunctions = userf;
slave.userFunctionCount = 2;
```

This code fragment makes library pass frames with function code 77 to some function `foo` for parsing. Please also note, that user-defined functions array can be used to disable support for some functions defined by default or override them.

> ☞ User-defined function override standard parsing functions defined by liblightmodbus.

If `modbusParseRequest` doesn't find matching function code in the user function array it checks the function code against standard Modbus function codes and then attempts parsing. If it finds the function code in the user function array, but function pointer is `NULL` or function code is handled neither by user nor by standard Modbus protocol, it builds exception frame letting master device know, that the function is unsupported.

For example of user parsing function, please see below:

```
ModbusError foo( ModbusSlave *status, ModbusParser *parser )
{
    //Throw exception if slave address is divisible by 2
    if ( parser->base.address % 2 == 0 )
        return modbusBuilException( status, parser->base.function,
            ↪ MODBUS_EXCEPTION_SLAVE_FAILURE );

    //Return empty frame in response
    //Assumes static slave response buffer is disabled
    status->response.frame = calloc( 16, 1 );
    status->response.length = 16;

    //Successful exit
    return MODBUS_OK;
}
```

This function, `foo`, makes exception responses whenever slave address is divisible by two. This is just a silly example, but below are some real guidelines for your own parsing functions:

- When building an exception frame with `modbusBuildException` (**??**), make sure you return the value it gave you when you return from the parsing function.
- Be aware of big-endian data - `modbusMatchEndian` (**??**) is there to help.
- Make sure you do not respond when request frame is broadcasted.
- Always know if you built library with dynamic memory allocation enabled. This affects the way you write data to the response buffer.

It's also worth to keep in mind that:

- Liblightmodbus guarantees that you can access data in `base` structure of the `ModbusParser` (**??**) - that means you know the function code and slave address.

- If dynamic response buffer allocation is enabled, the buffer will be automatically freed by `modbusParseRequest` (**??**)

If that is not enough of information for you, you can always look at the Modbus functions implemented in the library in the `src/slave` directory.

4.3.4 User-defined register/coil callback functions

> ☞ This is experimental feature only available if `REGISTER_CALLBACK` or `COIL_CALLBACK` library module was included during compilation. See section **??** for more information.

Prior to liblightmodbus version 1.4 arrays of registers and coils declared by user had to be continuous areas of memory. That means huge waste of memory when, for example, you want to have just a few control register with higher indices. Release v1.4 brings register and coil callback functions to solve this problem. There is, however, a drawback - the array based register/coil and callback systems cannot coexist for one data type[2].

The register/coil callback is an user-defined function determining if certain register or coil can be accessed and performing virtual register reads and writes. The below can be an example of register callback function simply mapping register accesses to an array:

```
1  uint16_t regs[16], iregs[16]; //Holding registers and input registers arrays
2  uint8_t writeacc[16]; //Some write locks
3
4  uint16_t rcallback( ModbusRegisterQuery query, ModbusDataType datatype, uint16_t
       ↪ index, uint16_t value )
5  {
6      //All can be read
7      if ( query == MODBUS_REGQ_R_CHECK ) return 1;
8
9      //writeacc determines if holding register can be written
10     if ( query == MODBUS_REGQ_W_CHECK ) return writeacc[index];
11
12     //Read
13     if ( query == MODBUS_REGQ_R )
14     {
15         if ( datatype == MODBUS_HOLDING_REGISTER ) return regs[index];
16         if ( datatype == MODBUS_INPUT_REGISTER ) return iregs[index];
17     }
18
19     //Write
20     if ( query == MODBUS_REGQ_W && datatype == MODBUS_HOLDING_REGISTER )
21         iregs[index] = value;
22
23     return 0;
24 }
```

The first function argument of type`ModbusRegisterQuery` (**??**) determines query type. Two types of register queries are distinguished - read/write requests (`MODBUS_REGQ_R`, `MODBUS_REGQ_W`) and

[2] 16-bit data types (input and holding registers) and 1-bit data types (coils and discrete inputs)

read/write access inquires (`MODBUS_REGQ_R_CHECK`, `MODBUS_REGQ_W_CHECK`).

Upon reception of access inquiry, the function should return 1 if certain kind of access is granted to register of type determined by second argument of type `ModbusDataType` (**??**) and index determined by third argument. Otherwise 0 should be returned. If the callback function denies access to certain register, this results in slave device returning `MODBUS_EXCEPTION_SLAVE_FAILURE` exception frame. When callback function receives a read request it should return 16-bit unsigned integer value of the register. If it receives a write request, the register should be written with value from the fourth parameter.

Liblightmodbus guarantees that no write requests will be ever made for discrete input and input register types. Neither will it request to read/write some register after the callback function denied certain kind of access for it. Access rights for registers will always be checked before reading/writing data. It's also guaranteed that the index argument will always be less than count of certain type registers set in `ModbusSlave`.

> ☞ Even though it's clearly stated what kind of queries will never be made by liblightmodbus internal Modbus functions, beware of your own functions.

`ModbusSlave` (**??**) structure has two pointers to user callbacks: `registerCallback` for holding and input registers and `coilCallback` for coils and discrete inputs. They may point to the same function if you want. After setting up the pointer, you only need to setup the virtual register count, like you normally would with the array based register/coil system. After such initialization, the slave device is ready for normal use.

5 Master device functionality

5.1 Master-related data types

5.1.1 Master device data container – `ModbusMaster`

5.2 Setup and cleanup

5.2.1 Modbus master device initialization function – `modbusMasterInit`

```
ModbusError modbusMasterInit( ModbusMaster *status )
```

`modbusMasterInit` function initializes `ModbusMaster` (**??**) structure pointed by `status` pointer for use. It sets up internal buffers and default values and verifies values already written to the structure by the user.

On success `MODBUS_OK` error value is returned.

new errors?

5.2.2 Modbus master device destructor function – `modbusMasterEnd`

```
ModbusError modbusMasterEnd( ModbusMaster *status )
```

`modbusMasterEnd` function cleans up data stored in the `ModbusMaster` (**??**) structure pointed by `status` pointer. This function must be called before freeing memory allocated for the `ModbusMaster` structure.
On success, the function returns `MODBUS_OK` error value.

errors

5.3 Making requests

5.3.1 Read multiple discrete inputs/coils – `modbusBuildRequest0102`

```
ModbusError modbusParseResponse0102( ModbusMaster *status, ModbusParser *parser,
    ↪ ModbusParser *requestParser )
```