

● lib-modbus

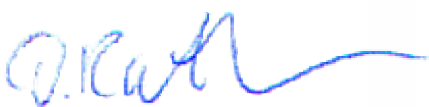
MODBUS/TCP Client- and Server-Library Programmers Manual

IBV-MODBUS-0001

Issue 1.0

06.03.2008

Author



Dominic Rath

Reviewed



Lutz Vollbracht

Released



Lutz Vollbracht

Change Log

Issue	Date	Chapter	Description	Author
1.0	06-Mar-2008	All	First Issue	Dominic Rath

Table of Contents

1. About this Document	4
1.1. Purpose	4
1.2. Terminology.....	4
1.3. Acronyms.....	4
1.4. References	4
2. Installation.....	5
2.1. Preconditions.....	5
2.2. Installation Procedure.....	5
3. Library Description.....	6
3.1. Header Files	6
3.2. Library Initialization.....	6
3.3. Server Interface	7
3.3.1. Backend Interface	7
3.4. Client Interface	9
3.4.1. Bit Access Functions.....	10
3.4.2. Register Access Functions	10
3.4.3. Miscellaneous Functions.....	11

1. About this Document

1.1. Purpose

This document functions as manual for the software module

"lib-modbus" - Release 1.0.

The intended readership is:

- ◆ Project Manager for review and release
- ◆ Software Engineer as input for the implementation and maintenance phase
- ◆ Software Engineer as input for the design and implementation of other modules
- ◆ Technical personnel as Installation Manual
- ◆ End User as User's Manual

This document is compliant with the IBV documentation standards.

Familiarity with the C programming language, the QNX Momentics development environment, and the MODBUS protocol and data model is assumed.

1.2. Terminology

MODBUS

Modbus is a communications protocol existing in multiple variants for communication over serial lines (MODBUS/ASCII and MODBUS/RTU) and over TCP/IP networks (MODBUS/TCP).

1.3. Acronyms

IBV	<u>IBV</u> - Echtzeit- und Embedded GmbH & Co. KG
TCP	Transmission Control Protocol
RTU	Remote Terminal Unit

1.4. References

- [MBAP] MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b
- [MBTCPIP] MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE V1.0b
- [MBSER] MODBUS over Serial Line Specification and Implementation Guide V1.02

2. Installation

The software is released for

- ◆ Operating System QNX 6.3.0 with Service Pack 3

2.1. Preconditions

- ◆ QNX 6 running on the target platform.
- ◆ Ethernet link to the target platform.

2.2. Installation Procedure

Copy the file "liblib-modbus.so" to the desired target directory (usually "/lib"), renaming it to "liblib-modbus.so.1".

The header files from the "common/" subdirectory have to be made available to the user application, either by copying them to a system wide location or by adding the directory to the include paths searched by the compiler.

The library files "liblib-modbus.so" and "liblib-modbus_g.so" (debug variant) from the "binaries/" subdirectory have to be made available to the user application, either by copying them to a system wide location or by adding the directory to the library paths searched by the linker.

3. Library Description

3.1. Header Files

The following headers have to be included by a user application:

- ◆ `modbus-common.h` Defines and declarations used by client and server
- ◆ `modbus-client.h` Application interface to client functionality
- ◆ `modbus-server.h` Application interface to server functionality

All necessary header files can be found in the "`source/common/`" subdirectory. A user application may decide to include only one of "`modbus-client.h`" and "`modbus-server.h`", if client and server functionality isn't required at the same time.

3.2. Library Initialization

Prior to calling any other library function, the libraries initialization function should be called:

```
int modbus_init();
```

The function takes no arguments and returns `MODBUS_ERR_OK` on success.

3.3. Server Interface

The server interface consists of two functions:

```
struct __server *modbus_start_server(int type, void *server_params,
    _backend_interface *backend);

int modbus_stop_server(struct __server *server);
```

To start a server, the user application has to call `modbus_start_server()`, specifying the connection type (only `MODBUS_CONNECTION_TCP` is implemented), a set of parameters specific to the connection type, and the backend interface used by the library to retrieve data for client requests.

The parameter set for TCP connections is a pointer to a structure of type `_modbus_tcp_params` as declared in `modbus-common.h`. It consists of a pointer to a C-String specifying the IP address the server should bind itself to, and a short integer specifying the TCP port the connection should listen at. The IP address pointer may be `NULL`, in which case the server will listen on all addresses assigned to the system.

```
typedef struct __modbus_tcp_params
{
    char *ip;
    short port;
} _tcp_params;
```

The `modbus_start_server()` function returns a pointer to the newly created server on success, or `NULL` in case of an error.

3.3.1. Backend Interface

The backend interface is declared in `modbus-server.h`. The callback functions supplied in the `_backend_interface` structure have to be implemented by the user application. They will be called by the server to retrieve data for client requests.

```
typedef struct __backend_interface
{
    int (*read_coils)(uint16_t start_ref, uint8_t *bit_array,
        uint16_t ref_count);
    int (*write_coils)(uint16_t start_ref, uint8_t *bit_array,
        uint16_t ref_count);
    int (*read_input_discretes)(uint16_t start_ref, uint8_t *bit_array,
        uint16_t ref_count);

    int (*read_input_registers)(uint16_t start_ref, uint16_t *reg_array,
        uint16_t ref_count);

    int (*read_holding_registers)(uint16_t start_ref, uint16_t *reg_array,
        uint16_t ref_count);
    int (*write_holding_registers)(uint16_t start_ref, uint16_t *reg_array,
        uint16_t ref_count);

    int (*read_exception_status)(uint8_t *exception_status);
```

```
int (*process_pdu)(uint8_t *request, uint8_t request_length,
                  uint8_t **reply,  uint8_t *reply_length);

int (*lock)();
int (*unlock)();
} _backend_interface;
```

The backend interface functions closely follow the MODBUS data model according to [MBAP]. References and reference counts are expressed as integer numbers between 0 and 65535 (zero based).

Functions accessing coils or input discretes use arrays of `uint8_t` where each array element corresponds to one coil or input discrete.

Functions operating on input or holdings registers use arrays of `uint16_t`, where each array element corresponds to one input or holding register.

Functions for reading data (`read_coils()`, `read_input_discretes()` and `read_input_registers()`) are supposed to fill the array elements with the appropriate application specific values. The first array element (index [0]) corresponds to the reference supplied as `start_ref`. The last array element (index [`ref_count`]) corresponds to reference `start_ref + ref_count - 1`.

Functions for writing data (`write_coils()`, `write_input_discretes` and `write_input_registers`) are supposed to use the values from the array elements to write application specific data.

The user application should return `MODBUS_ERR_ILLEGAL_DATA_ADDRESS` if addresses outside of the application specific data model are requested. If no errors occurred while processing the backend callback function, `MODBUS_ERR_OK` has to be returned to cause `lib-modbus` to send a positive reply.

When `read_exception_status()` gets called, the current status of the exception register (8 bit) should be returned to the MODBUS server library using the `uint8_t *exception_status` pointer.

The function `process_pdu()` will be called for all MODBUS function codes not handled by the server library. This allows the user application to handle additional application-specific function codes. In case the application doesn't want to handle a certain function code it should return `MODBUS_ERR_ILLEGAL_FUNCTION`.

The `request` and `request_length` arguments hold the MODBUS PDU as received by the server, while `reply` and `reply_length` are pointers to the location where the user application should store any results produced. The memory for the reply buffer has to be allocated by the user application. It will be freed by the server library once the reply has been sent. The length of the reply buffer has to be returned using the `reply_length` pointer.

The `lock()` and `unlock()` functions are called to ensure data consistency for the Read/Write Multiple Registers and Mask Write Register function codes. It is up to the user application to ensure that the data is held consistent between a `lock()` and a subsequent `unlock()` call.

In order to stop a previously started MODBUS server, the user application has to call the `modbus_stop_server()` function using the server pointer returned by the `modbus_start_server()` function as an argument. The pointer will be invalid once the `modbus_stop_server()` function returned.

3.4. Client Interface

Two functions exist that allow the user application to open and close connections to clients. In case of a MODBUS/TCP connection, an individual connection has to be opened for every client that should be accessed. If MODBUS/RTU support was to be implemented, a single connection could be used to access multiple clients.

```
struct __client_connection *modbus_open(int type, void *connection_params,
int timeout);

int modbus_close(struct __client_connection *connection);
```

The parameters required by the `modbus_open()` function are the same as those required by the `modbus_start_server()` function, see 3.3 “Server Interface” for more information. The additional `timeout` argument specifies the timeout for client requests in milliseconds. The function will return a pointer to the newly created client connection on success, or `NULL` in case an error occurred. The returned pointer has to be used in all subsequent calls as a handle to identify the particular connection.

A connection should be closed using the `modbus_close()` function if the client connection isn't required anymore. The connection pointer is invalid once the `modbus_close()` function returned, and must not be used in subsequent calls to client request functions.

All functions for sending MODBUS requests to a MODBUS server take a pointer to the connection and a slave ID as an argument. For MODBUS/TCP requests, the slave parameter is ignored, and the corresponding field in the request is filled with `0xFF`.

Possible return values from client request functions are

- ◆ `MODBUS_ERR_OK`
- ◆ `MODBUS_ERR_CONNECTION_CLOSED`
- ◆ `MODBUS_ERR_COMMUNICATION_FAILURE`
- ◆ One of the MODBUS protocol error codes

In case of a connection or communication error the corresponding client connection should be closed and reopened. No automatic retransmission is performed.

3.4.1. Bit Access Functions

The following functions may be called by the user application to send MODBUS requests accessing bit quantities:

```
int modbus_read_coils(struct __client_connection *connection, uint8_t
slave, uint16_t start_ref, uint8_t *bit_array, uint16_t ref_count);
```

```
int modbus_read_input_discretes(struct __client_connection *connection,
uint8_t slave, uint16_t start_ref, uint8_t *bit_array, uint16_t
ref_count);
```

```
int modbus_write_coil(struct __client_connection *connection, uint8_t
slave, uint16_t bit_ref, uint8_t bit_value);
```

```
int modbus_write_multiple_coils(struct __client_connection *connection,
uint8_t slave, uint16_t start_ref, uint8_t *bit_array, uint16_t
ref_count);
```

Arrays of `uint8_t` are used to transfer data from the user application to the MODBUS library. Each coil or input discrete corresponds to one array element, with index 0 referencing the item specified by the `start_ref` parameter. The `modbus_write_coil()` function operates on a single coil, whose value is supplied as a single `uint8_t` argument.

3.4.2. Register Access Functions

The following functions may be called by the user application to send MODBUS requests accessing registers:

```
int modbus_read_multiple_registers(struct __client_connection *connection,
int8_t slave, uint16_t start_ref, uint16_t *reg_array, uint16_t
ref_count);
```

```
int modbus_read_input_registers(struct __client_connection *connection,
uint8_t slave, uint16_t start_ref, uint16_t *reg_array, uint16_t
ref_count);
```

```
int modbus_write_single_register(struct __client_connection*, uint8_t
slave, uint16_t reg_ref, uint16_t reg_value);
```

```
int modbus_write_multiple_registers(struct __client_connection
*connection, uint8_t slave, uint16_t start_ref, uint16_t *reg_array,
uint16_t ref_count);
```

```
int modbus_mask_write_register(struct __client_connection *connection,
uint8_t slave, uint16_t reg_ref, uint16_t and_mask, uint16_t or_mask);
```

```
int modbus_read_write_registers(struct __client_connection *connection,
uint8_t slave, uint16_t read_ref, uint16_t *read_array, uint16_t
read_count, uint16_t write_ref, uint16_t *write_array, uint16_t
write_count);
```

Arrays of `uint16_t` are used to transfer data from the user application to the MODBUS library. Each input register or holding register corresponds to one array element, with index 0 referencing the item specified by the `start_ref` parameter.

The `modbus_write_single_register()` function operates on a single holding register, whose value is supplied as a single `uint16_t` argument.

The `modbus_mask_write_register()` function operates on a single holding register, too, but takes an `and_mask` and an `or_mask` as specified by the MODBUS standard.

The `modbus_read_write_registers()` function uses two arrays, one to hold the values that should be written, and one holding the values read from the server. Two different count parameters exist to allow reading and writing arbitrary numbers of registers.

3.4.3. Miscellaneous Functions

```
int modbus_read_exception_status(struct __client_connection *connection,
uint8_t slave, uint8_t *status);
```

The `modbus_read_exception_status()` function allows a client to read a servers exception status register. The exception register value is returned using the `uint8_t` Pointer `status`.

```
int modbus_send_pdu(struct __client_connection *connection, uint8_t slave,
uint8_t *request, uint8_t request_length, uint8_t **reply, uint8_t
*reply_length);
```

The `modbus_read_exception_status()` function provides direct access to the MODBUS PDU interface implemented by `lib-modbus`. The request is transmitted as is, and any data received is returned using the `reply` and `reply_length` pointers. The buffer space for the reply is allocated by the library, and has to be freed by the user application once the reply has been processed. In case of an error (either communication or protocol error), no reply is returned, and the pointers are set to `NULL`.