Modbus Server Stack

User Manual

Version 1.0

THIS PAGE IS INTENTIONALLY LEFT BLANK

Table of Contents

1. Introduction

This document is the User Manual for the RAPIDSEA Modbus Server Stack. It provides a complete reference covering all API functions, mandatory application callbacks, data structures, configuration parameters, error codes, and integration guidelines for deploying a Modbus server (slave) on RTU, ASCII, or TCP/IP transport layers.

Document structure:

- Section 2 - Modbus Protocol Overview
- Section 3 - Data Structures and Macros
- Section 4 - API Reference (rs_ functions)
- Section 5 - Callback Reference (rcb_ functions)
- Section 6 - Error Codes

2. Modbus Protocol Overview

2.1 Protocol Background

Modbus is a serial communication protocol published by Modicon in 1979. It has become a de facto standard in industrial automation because it is simple, robust, openly documented, and royalty-free. A single Modbus bus supports one server (slave) device and up to 247 uniquely addressed devices on a multi-drop RS-485 network. The TCP/IP variant supports any number of IP-addressable server devices.

2.2 Server Role

The Modbus server (slave) responds to requests issued by one or more clients (masters). It maintains a register map divided into four spaces and exposes the data through callbacks that the application implements. The RAPIDSEA server stack handles all framing, CRC/LRC computation, and function code dispatch transparently.

2.3 Register Model

Register Type	Access	Width	Typical Use
Coil (0x)	Read / Write	1 bit	Digital outputs — relays, LEDs, actuators
Discrete Input (1x)	Read only	1 bit	Digital inputs — switches, sensor flags
Input Register (3x)	Read only	16 bit	Analog inputs — ADC channels
Holding Register (4x)	Read / Write	16 bit	Analog outputs, configuration words

2.4 Supported Function Codes

FC	Name	Register Space	Direction
01 (0x01)	Read Coils	Coil (0x)	Client reads
02 (0x02)	Read Discrete Inputs	Discrete Input (1x)	Client reads
03 (0x03)	Read Holding Registers	Holding Register (4x)	Client reads
04 (0x04)	Read Input Registers	Input Register (3x)	Client reads
05 (0x05)	Write Single Coil	Coil (0x)	Client writes
06 (0x06)	Write Single Register	Holding Register (4x)	Client writes
15 (0x0F)	Write Multiple Coils	Coil (0x)	Client writes
16 (0x10)	Write Multiple Registers	Holding Register (4x)	Client writes

2.5 Transport Layer Variants

2.5.1 RTU Mode

Binary encoding with 8 data bits per byte. Frames are delimited by silent gaps of at least 3.5 character times. A 16-bit CRC is appended to every frame. RTU is the most compact and widely deployed Modbus variant.

2.5.2 ASCII Mode

Each byte is encoded as two ASCII hexadecimal characters. Frames start with a colon ':' and end with CR/LF. Error detection uses a Longitudinal Redundancy Check (LRC). Frame size is approximately double that of RTU.

2.5.3 TCP/IP Mode

Modbus PDUs are wrapped in a 6-byte MBAP header and carried over standard TCP sockets (default port 502). The RAPIDSEA server supports up to RS_MODBUS_MAX_NUM_OF_CLIENT (5) simultaneous client connections.

2.6 Stack Characteristics

Attribute	Value
Supported transport layers	RTU, ASCII, TCP/IP
Baud rates (serial)	9600 – 1 000 000 bps
Max concurrent TCP clients	5 (RS_MODBUS_MAX_NUM_OF_CLIENT)
ROM size (server stack only)	≈ 22 588 bytes
Custom function code support	Yes — rcb_mb_handle_custom_request callback
OS / bare-metal compatibility	Linux, Windows, FreeRTOS, bare-metal

3. Data Structures and Macros

3.1 Communication Type Constants

Macro	Value	Description
RS_MODBUS_COMM_TYPE_RTU	0	Serial RTU binary framing
RS_MODBUS_COMM_TYPE_ASCII	1	Serial ASCII hex framing
RS_MODBUS_COMM_TYPE_TCP_IP	2	TCP/IP socket transport

3.2 Error Code Constants

Callbacks return 0 on success or one of the following negated values to signal a Modbus exception to the requesting client.

Macro	Value	Modbus Exception Code	Meaning
RS_MODBUS_ERROR_INVALID_FUNCTION	1	01	Illegal Function — FC not supported
RS_MODBUS_ERROR_INVALID_DATA_ADDR	2	02	Illegal Data Address — register out of range
RS_MODBUS_ERROR_INVALID_VALUE	3	03	Illegal Data Value — value not acceptable
RS_MODBUS_ERROR_SLAVE_DEVICE_FAILURE	4	04	Server Failure — unrecoverable internal error
RS_MODBUS_ERROR_TIMEOUT	5	05	Gateway Timeout
RS_MODBUS_ERROR_SLAVE_DEVICE_BC	6	06	Server Busy
RS_MODBUS_ERROR_PARITY	8	08	Memory Parity Error

3.3 Buffer Size Constants

Macro	Value	Description
RS_MODBUS_RTU_BUF_MAX_LEN	256	RX/TX buffer size for RTU mode
RS_MODBUS_ASCII_BUF_MAX_LEN	513	RX/TX buffer size for ASCII mode
RS_MODBUS_TCP_BUF_MAX_LEN	260	RX/TX buffer size per TCP connection
RS_MODBUS_RTU_FIFO_BUF_MAX_LEN	256	FIFO buffer size for RTU mode
RS_MODBUS_ASCII_FIFO_BUF_MAX_LEN	513	FIFO buffer size for ASCII mode
RS_MODBUS_MAX_NUM_OF_CLIENT	5	Maximum simultaneous TCP client connections
RS_TRANSPORT_INTERFACE_ADDR_LEN	32	Maximum address string length (IP or port name)

3.4 rs_modbus_server_config_t

Holds all parameters required to open a Modbus server channel. Populate fully before calling rs_modbus_server_open.

```
typedef struct tag_rs_modbus_server_config {
    uint8_t     server_id;       // Slave address 1–247
    uint8_t     comm_type;        // RTU / ASCII / TCP_IP
    uint8_t     addr[32];         // Port name or IP address
    uint8_t     parity;           // Serial parity
    uint8_t     flow_ctrl;        // Flow control
    uint8_t     data_length;      // Data bits
    uint8_t     stop_bits;        // Stop bits
    uint16_t    port;             // TCP port (default 502)
    uint32_t    baud_rate;        // Baud rate (serial)
    uint16_t    num_conn_accept;  // Max TCP clients
    rs_handle_t comm_handle;      // Transport handle
    rs_handle_t accept_handle;    // TCP accept handle
} rs_modbus_server_config_t;
```

Field	Type	Relevant For	Description
server_id	uint8_t	RTU, ASCII, TCP	Modbus slave address (1–247)
comm_type	uint8_t	All	RS_MODBUS_COMM_TYPE_RTU / ASCII / TCP_IP
addr[32]	uint8_t[]	All	Serial port name (e.g. /dev/ttyUSB0, COM3) or IP address string (e.g. 192.168.1.10)
parity	uint8_t	RTU, ASCII	RS_SERIAL_PORT_PARITY_DISABLED=0 (none), RS_SERIAL_PORT_PARITY_ODD=1 (odd), RS_SERIAL_PORT_PARITY_EVEN=2 (even)
flow_ctrl	uint8_t	RTU, ASCII	RS_SERIAL_PORT_FLOW_CTRL_DISABLED or hardware flow control constant
data_length	uint8_t	RTU, ASCII	RS_SERIAL_PORT_DATA_LENGTH_8BITS (standard) or 7-bit for ASCII
stop_bits	uint8_t	RTU, ASCII	RS_SERIAL_PORT_STOP_BITS_ONE / TWO

port	uint16_t	TCP	TCP port number (standard Modbus port: 502)
baud_rate	uint32_t	RTU, ASCII	Baud rate in bps. Use RS_SERIAL_PORT_BAUD_* constants
num_conn_accept	uint16_t	TCP	Maximum simultaneous client connections (max RS_MODBUS_MAX_NUM_OF_CLIENT = 5)
comm_handle	rs_handle_t	All	Transport interface handle. Leave 0; set by stack on open
accept_handle	rs_handle_t	TCP	Socket accept handle. Leave 0; managed by stack

3.5 Instance Structures

The application allocates one of the following transport-specific instance structures as a global or static variable. Pass a pointer cast to rs_modbus_server_instance_t to rs_modbus_server_open. The first member of every instance is the base rs_modbus_server_instance_t, ensuring safe type-punning.

Structure	Transport
rs_modbus_rtu_server_instance_t	RTU
rs_modbus_ascii_server_instance_t	ASCII
rs_modbus_tcp_server_instance_t	TCP

4. API Reference

4.1 rs_modbus_server_open

```
rs_handle_t  rs_modbus_server_open(
    rs_modbus_server_instance_t *ptr_instance,
    rs_modbus_server_config_t   *ptr_config );
```

Initializes the Modbus Server channel for communication.

Opens the underlying communication channel (serial port or TCP socket) and prepares the server instance for receiving client requests. The application must allocate the appropriate transport-specific instance structure (rs_modbus_rtu_server_instance_t, rs_modbus_ascii_server_instance_t, or rs_modbus_tcp_server_instance_t) and pass its address cast to rs_modbus_server_instance_t*. The returned handle must be stored and passed to all subsequent server API calls.

Parameters

Parameter	Direction	Description
ptr_instance	[in]	Pointer to the caller-allocated Modbus server instance structure
ptr_config	[in]	Pointer to a fully populated rs_modbus_server_config_t

Return value: Positive handle value on success. Negative error code on failure.

4.2 rs_modbus_server_run

```
rs_ret_val_t  rs_modbus_server_run(
    rs_handle_t  handle,
    uint32_t     u32_run );
```

Starts or stops the Modbus server instance.

When u32_run = 1, the server transitions to the active state and will process incoming Modbus frames on each call to rs_modbus_server_process. When u32_run = 0, the server stops accepting new requests and enters an idle state. The instance and its handle remain valid; the server can be restarted by calling this function again with u32_run = 1.

Parameters

Parameter	Direction	Description
handle	[in]	Handle returned by rs_modbus_server_open
u32_run	[in]	1 = start the server, 0 = stop the server

Return value: 0 on success. Negative error code on failure.

4.3 rs_modbus_server_process

```
rs_ret_val_t  rs_modbus_server_process(
    rs_handle_t  handle );
```

Performs the Modbus server state-machine maintenance step.

This function must be called periodically from the application's main loop or a dedicated task. It reads incoming data from the transport layer, decodes Modbus frames, dispatches function codes to the appropriate rcb_ callbacks, and transmits response frames. For RTU and ASCII transports, the call frequency determines the inter-character timeout resolution.

Parameters

Parameter	Direction	Description
handle	[in]	Handle returned by rs_modbus_server_open

Return value: 0 on success. Negative error code on failure.

NOTE:
For TCP mode, also call rs_modbus_server_tcp_comm_handler periodically to handle client connect / disconnect events.

4.4 rs_modbus_server_tcp_comm_handler

```
rs_ret_val_t  rs_modbus_server_tcp_comm_handler(
    rs_handle_t  handle );
```

Manages TCP client connections and per-connection communication.

This function handles TCP-specific tasks: accepting new client connections, closing disconnected clients, and processing per-connection receive/transmit operations. Must

be called periodically alongside rs_modbus_server_process when the server is configured for TCP/IP transport. Calling this function on an RTU or ASCII handle has no effect.

Parameters

Parameter	Direction	Description
handle	[in]	Handle returned by rs_modbus_server_open (TCP mode)

Return value: 0 on success. Negative error code on failure.

4.5 rs_modbus_server_reconfig

```
rs_ret_val_t  rs_modbus_server_reconfig(
    rs_handle_t  handle );
```

Reconfigures the Modbus server with updated configuration values.

Applies any changes made to the rs_modbus_server_config_t structure since the server was opened. The application may update fields such as baud_rate, parity, or stop_bits in the configuration structure and then call this function to apply them without closing and reopening the channel.

Parameters

Parameter	Direction	Description
handle	[in]	Handle returned by rs_modbus_server_open

Return value: 0 on success. Negative error code on failure.

5. Callback Reference

The Modbus server calls the following application-defined callbacks to access the device's register data. Every callback MUST be implemented by the application — the linker will report errors for any missing implementation.

Address convention: the stack passes (user_address − 1) to each callback. For example, if the client requests holding register address 100, the callback receives u16_reg_addr = 99. Add 1 inside the callback to obtain the 1-based register number used in your register map.

5.1 rcb_mb_get_holding_reg_val

```
rs_ret_val_t  rcb_mb_get_holding_reg_val(
    rs_handle_t   handle,
    uint16_t      u16_reg_addr,
    uint16_t     *ptr_reg_value );
```

Read a holding register value (FC 03).

Called when a Modbus client issues a Read Holding Registers (FC 03) request. The application reads the register at (u16_reg_addr + 1) from its data model and writes the 16-bit value to *ptr_reg_value.

Parameters

Parameter	Direction	Description
handle	[in]	Handle to the Modbus server instance
u16_reg_addr	[in]	Register address (0-based; add 1 to get 1-based register number)
ptr_reg_value	[out]	Pointer to the 16-bit value to return to the client

Return value: 0 on success. Negative RS_MODBUS_ERROR_* code to return a Modbus exception to the client.

5.2 rcb_mb_set_holding_register

```
rs_ret_val_t  rcb_mb_set_holding_register(
    rs_handle_t   handle,
    uint16_t      u16_reg_addr,
    uint16_t     *ptr_reg_value );
```

Write a holding register value (FC 06 / FC 16).

Called when a Modbus client issues a Write Single Register (FC 06) or Write Multiple Registers (FC 16) request. The application writes *ptr_reg_value to the register at (u16_reg_addr + 1) in its data model.

Parameters

Parameter	Direction	Description
handle	[in]	Handle to the Modbus server instance
u16_reg_addr	[in]	Register address (0-based; add 1 to get 1-based register number)
ptr_reg_value	[in]	Pointer to the 16-bit value written by the client

Return value: 0 on success. Negative RS_MODBUS_ERROR_* code to return a Modbus exception.

5.3 rcb_mb_get_coil_status_value

```
rs_ret_val_t  rcb_mb_get_coil_status_value(
    rs_handle_t   handle,
    uint16_t      u16_reg_addr,
    uint8_t      *ptr_reg_value );
```

Read a coil value (FC 01).

Called when a client issues a Read Coils (FC 01) request. Write 1 or 0 to *ptr_reg_value reflecting the coil state at (u16_reg_addr + 1).

Parameters

Parameter	Direction	Description
handle	[in]	Handle to the Modbus server instance
u16_reg_addr	[in]	Coil address (0-based)
ptr_reg_value	[out]	Pointer to the 8-bit coil value (0 = OFF, 1 = ON)

Return value: 0 on success. Negative RS_MODBUS_ERROR_* code to return a Modbus exception.

5.4 rcb_mb_set_coil_status_value

```
rs_ret_val_t  rcb_mb_set_coil_status_value(
    rs_handle_t   handle,
    uint16_t      u16_reg_addr,
    uint8_t      *ptr_reg_value );
```

Write a coil value (FC 05 / FC 15).

Called when a client issues a Write Single Coil (FC 05) or Write Multiple Coils (FC 15) request. The application sets the output at (u16_reg_addr + 1) to *ptr_reg_value (0 or 1).

Parameters

Parameter	Direction	Description
handle	[in]	Handle to the Modbus server instance
u16_reg_addr	[in]	Coil address (0-based)
ptr_reg_value	[in]	Pointer to the 8-bit value (0 = OFF, 0xFF00 = ON)

Return value: 0 on success. Negative RS_MODBUS_ERROR_* code to return a Modbus exception.

5.5 rcb_mb_get_discrete_input_value

```
rs_ret_val_t  rcb_mb_get_discrete_input_value(
    rs_handle_t   handle,
    uint16_t      u16_reg_addr,
    uint8_t      *ptr_reg_value );
```

Read a discrete input value (FC 02).

Called when a client issues a Read Discrete Inputs (FC 02) request. Write the current digital input state at (u16_reg_addr + 1) to *ptr_reg_value.

Parameters

Parameter	Direction	Description
handle	[in]	Handle to the Modbus server instance
u16_reg_addr	[in]	Discrete input address (0-based)
ptr_reg_value	[out]	Pointer to the 8-bit input value (0 = OFF, 1 = ON)

Return value: 0 on success. Negative RS_MODBUS_ERROR_* code to return a Modbus exception.

5.6 rcb_mb_get_input_register_value

```
rs_ret_val_t  rcb_mb_get_input_register_value(
    rs_handle_t   handle,
    uint16_t      u16_reg_addr,
    uint16_t     *ptr_reg_value );
```

Read an input register value (FC 04).

Called when a client issues a Read Input Registers (FC 04) request. Write the 16-bit read-only value at (u16_reg_addr + 1) to *ptr_reg_value.

Parameters

Parameter	Direction	Description
handle	[in]	Handle to the Modbus server instance
u16_reg_addr	[in]	Input register address (0-based)
ptr_reg_value	[out]	Pointer to the 16-bit register value

Return value: 0 on success. Negative RS_MODBUS_ERROR_* code to return a Modbus exception.

5.7 rcb_mb_handle_custom_request

```
rs_ret_val_t  rcb_mb_handle_custom_request(
    rs_handle_t   handle,
    uint8_t      *ptr_buff );
```

Handle a custom (non-standard) function code request.

Called when the server receives a Modbus request whose function code is not natively handled by the stack (i.e. not FC 01, 02, 03, 04, 05, 06, 15, or 16). This callback is the stack's primary extension point for proprietary function codes. ptr_buff points to the start of the inbound Modbus PDU (function code byte first). The application decodes the request, performs the custom action, and writes the complete response PDU back into ptr_buff starting at offset 0 before returning 0. To send a standard Modbus exception instead, return a negated RS_MODBUS_ERROR_* constant — the stack builds and transmits the exception frame automatically.

Parameters

Parameter	Direction	Description
handle	[in/out]	Handle to the Modbus server instance
ptr_buff	[in/out]	On entry: ptr_buff[0] = custom function code, ptr_buff[1..N] = request data bytes. On exit (return 0): ptr_buff[0] = function code echo, ptr_buff[1..M] = response data bytes. The stack re-applies all transport framing (CRC / LRC / MBAP) around the response PDU.

Return value: 0 — stack transmits the response PDU written to ptr_buff. Negative RS_MODBUS_ERROR_* — stack sends a standard Modbus exception response (e.g. return -(int32_t)RS_MODBUS_ERROR_INVALID_FUNCTION to reject an unknown FC).

5.7.1 PDU Buffer Layout

ptr_buff always points to the Modbus PDU — the stack strips all transport-level framing before the callback is invoked and re-applies it around the response. This means the callback works identically for RTU, ASCII, and TCP transports.

Inbound buffer on entry:

```
ptr_buff[0]   = Function code  (the custom FC that triggered this
callback)
ptr_buff[1]   = Request data byte 0
ptr_buff[2]   = Request data byte 1
  ...
ptr_buff[N]   = Request data byte N-1
```

Response buffer the application writes before returning 0:

```
ptr_buff[0]   = Function code echo  (same FC as received)
ptr_buff[1]   = Response data byte 0
ptr_buff[2]   = Response data byte 1
  ...
ptr_buff[M]   = Response data byte M-1

Note: do NOT prepend the slave address or append CRC/LRC —
      the stack adds all transport framing automatically.
```

5.7.2 Maximum Response PDU Length

Transport	Buffer Macro	Max PDU Bytes	Notes
RTU	RS_MODBUS_RTU_BUF_MAX_LEN = 256	252	256 − 1 addr − 2 CRC − 1 FC = 252 data bytes
ASCII	RS_MODBUS_ASCII_BUF_MAX_LEN = 513	252	Same PDU limit; ASCII encoding is handled by the stack
TCP	RS_MODBUS_TCP_BUF_MAX_LEN = 260	253	260 − 6 MBAP − 1 FC = 253 data bytes

NOTE:
Never write more bytes than the maximum for the active transport. Exceeding the buffer boundary will corrupt internal stack state.

5.7.3 Returning a Standard Exception

If the custom request is invalid or cannot be fulfilled, return a negated RS_MODBUS_ERROR_* constant. The stack constructs and transmits the standard 1-byte Modbus exception response automatically:

```
/* Reject with Illegal Data Value exception — no need to build the frame
*/
return -(int32_t)RS_MODBUS_ERROR_INVALID_VALUE; /* sends exception 03 */
/* Reject unknown custom FC */
return -(int32_t)RS_MODBUS_ERROR_INVALID_FUNCTION; /* sends exception 01*/
```

5.7.4 Example — Custom FC 0x41: Read Device Serial Number

```
#define MY_FC_READ_SERIAL   0x41U
static const uint8_t g_serial[8] = {
    0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF };

rs_ret_val_t rcb_mb_handle_custom_request(
        rs_handle_t handle, uint8_t *ptr_buff)
{
    /* Reject any FC other than our custom one */
    if (ptr_buff[0] != MY_FC_READ_SERIAL)
        return -(int32_t)RS_MODBUS_ERROR_INVALID_FUNCTION;

    /* Build the response PDU in-place */
    ptr_buff[0] = MY_FC_READ_SERIAL;     /* echo function code   */
    ptr_buff[1] = 8;                      /* byte count field     */
    memcpy(&ptr_buff[2], g_serial, 8);   /* 8 serial number bytes */

    /* Wire format the stack sends (RTU example):                 */
```

```
    /* [addr][0x41][0x08][b0..b7][CRC_lo][CRC_hi]                    */
    return 0;   /* 0 = transmit the response */
}
```

6. Error Codes Reference

API functions return 0 (RS_ERR_OK / RS_STATUS_SUCCESS) on success and a negative value on failure. Callbacks return 0 on success and a negated RS_MODBUS_ERROR_* value to signal a Modbus exception to the requesting client.

Context	Value	Macro / Meaning
API return	0	RS_ERR_OK — success
API return	−1	RS_ERR_BUSY — resource busy or not yet initialised
Callback return	−1	−RS_MODBUS_ERROR_INVALID_FUNCTION (01)
Callback return	−2	−RS_MODBUS_ERROR_INVALID_DATA_ADDR (02) — address out of range
Callback return	−3	−RS_MODBUS_ERROR_INVALID_VALUE (03) — value not accepted
Callback return	−4	−RS_MODBUS_ERROR_SLAVE_DEVICE_FAILURE (04) — internal error
Callback return	−5	−RS_MODBUS_ERROR_TIMEOUT (05)
Callback return	−6	−RS_MODBUS_ERROR_SLAVE_DEVICE_BC (06) — device busy
Callback return	−8	−RS_MODBUS_ERROR_PARITY (08) — memory parity error

NOTE:
The server transmits the exception response automatically after a callback returns a negative value. The application only needs to return the correct code.