Modbus Server Stack

Integration Guide

Version 1.0

Table of Contents

1. Overview

This Quick Start Guide walks you through everything needed to integrate the RAPIDSEA Modbus Server stack into your application in the shortest possible time. By the end of this guide, you will have a working Modbus server that listens for client requests, invokes your register callbacks, and returns correct responses.

The RAPIDSEA Modbus Server stack supports:

- All eight standard Modbus function codes (FC 01–06, 0F, 10)
- RTU, ASCII, and TCP/IP physical layers via a single unified API
- Up to 255 simultaneous TCP client connections
- Custom function code extension via a callback hook
- Run-time reconfiguration without closing the channel

2. Prerequisites

2.1 Required Files

File	Purpose
rs_modbus_server.h	Public API — types, macros, function declarations
rs_modbus_server.c	Server stack implementation
modbus.c / modbus.h	Common Modbus frame encode/decode engine
rs_lib.h	RAPIDSEA platform types and handle manager
modbus_server_demo_app.c	Reference application with callback examples

2.2 Include in Your Build

```
#include <rs_lib.h>
#include "rs_modbus_server.h"
```

NOTE:
rs_lib.h must be on the include path. Link rs_modbus_server.c and modbus.c into your project.

3. Environment Setup

3.1 Windows - Visual Studio

Open the Visual Studio workspace at rs_windows_ws/. The project RSModbusServer contains pre-configured sources, include paths, and library references.

Setting	Value
Preprocessor macro	RS_PLAT_WIN32
Additional Include Dirs	rs_common\\inc; rs_platform\\inc; rs_modbus\\inc; rs_hal_windows\\inc
Library	rs_modbus\\libs\\release\\ or debug\\ (.lib)
Serial port	Pass COM port name e.g. "COM3" in config.addr[]
TCP	WinSock2 included automatically by win32_arch.h

NOTE:
Define RS_PLAT_WIN32 as a project-level preprocessor macro so rs_lib.h selects the correct HAL.

3.2 Linux — Eclipse / GCC

Import the Eclipse workspace at rs_linux_ws/. The project RSModbusServer is ready to build.

Setting	Value
Preprocessor macro	RS_PLAT_LINUX_SDL
Additional Include Dirs	rs_common/inc rs_platform/inc rs_modbus/inc rs_hal_linux/inc
Library	rs_modbus/libs/release/ or debug/ (.a)
Serial port	Pass /dev/ttyUSBx or /dev/ttySx in config.addr[]

TCP	POSIX sockets; linux_socket.h included automatically

NOTE:
For embedded Linux (e.g. iMX6) define RS_PLAT_LINUX_IMX6 instead.

3.3 Embedded MCUs/MPUs

For other MCUs/MPUs open the given reference project in corresponding development tools. Following macros can be set to have the corresponding HAL layers enabled.

Platform	Macro
Renesas RA6M3 / RA8D1	RS_PLAT_RA6M3 / RS_PLAT_RA8D1
NXP IMXRT	RS_PLAT_IMXRT
MX1170	RS_PLAT_MX1170
STM32	RS_PLAT_STM32
Renesas RH850	RS_PLAT_RH850
NXP LPC55S16	RS_PLAT_LPC55S16
NXP LPC54628	RS_PLAT_LPC54628

4. HAL Layer Reference

The RAPIDSEA HAL abstracts serial ports and TCP sockets behind a platform-independent API. These macros are defined in rs_serial.h and rs_socket.h.

4.1 Serial HAL Macros

Category	Macro	Value
Baud Rate	RS_SERIAL_PORT_BAUD_9600	9600
	RS_SERIAL_PORT_BAUD_19200	19200
	RS_SERIAL_PORT_BAUD_57600	57600
	RS_SERIAL_PORT_BAUD_115200	115200
	RS_SERIAL_PORT_BAUD_1MBPS	1000000
Parity	RS_SERIAL_PORT_PARITY_DISABLED	0 — No parity
	RS_SERIAL_PORT_PARITY_ODD	1 — Odd
	RS_SERIAL_PORT_PARITY_EVEN	2 — Even
Flow Control	RS_SERIAL_PORT_FLOW_CTRL_DISABLED	0 — None
	RS_SERIAL_PORT_FLOW_CTRL_RTS_CTS	1 — RTS/CTS
Data Bits	RS_SERIAL_PORT_DATA_LENGTH_8BITS	3 — 8 bits (standard for Modbus)
Stop Bits	RS_SERIAL_PORT_STOP_BITS_ONE	0 — 1 stop bit
	RS_SERIAL_PORT_STOP_BITS_TWO	1 — 2 stop bits
I/O Mode	RS_SERIAL_BLOCKING	0 — Blocking
	RS_SERIAL_RX_MODE_FIFO	0 — FIFO receive (recommended)
	RS_SERIAL_TX_MODE_FIFO	0 — FIFO transmit (recommended)

4.2 Socket HAL Macros

Macro	Value	Meaning
RS_SOCKET_BLOCKING	0	Blocking socket
RS_SOCKET_NON_BLOCKING	1	Non-blocking — required for TCP server with multiple clients

5. Quick Start Steps

Step 1: Allocate Instance and Configuration Structures

Select the instance structure that matches your chosen transport mode. Declare it at file scope (global or static) so it persists for the lifetime of the server.

RTU Mode

```
/* RTU — declare at file scope */
static rs_modbus_rtu_server_instance_t  g_rtu_instance;
static rs_modbus_server_config_t        g_server_cfg;
```

ASCII Mode

```
/* ASCII — declare at file scope */
static rs_modbus_ascii_server_instance_t  g_ascii_instance;
static rs_modbus_server_config_t          g_server_cfg;
```

TCP/IP Mode

```
/* TCP/IP — declare at file scope */
static rs_modbus_tcp_server_instance_t  g_tcp_instance;
static rs_modbus_server_config_t        g_server_cfg;
```

Structure	Transport
rs_modbus_rtu_server_instance_t	RTU
rs_modbus_ascii_server_instance_t	ASCII
rs_modbus_tcp_server_instance_t	TCP

Step 2: Fill the Configuration Structure

Populate rs_modbus_server_config_t before calling rs_modbus_server_open. Fields that are not used by a particular transport may be left at zero.

RTU Configuration Example (from modbus_server_demo_app.c)

```
g_server_cfg.server_id       = 1;      /* Modbus slave address 1–247 */
g_server_cfg.comm_type       = RS_MODBUS_COMM_TYPE_RTU; /* RTU framing */
g_server_cfg.addr[0]         = '0';                     /* port index */
g_server_cfg.baud_rate       = 115200;                  /* bps */
g_server_cfg.parity          = RS_SERIAL_PORT_PARITY_DISABLED;
g_server_cfg.flow_ctrl       = RS_SERIAL_PORT_FLOW_CTRL_DISABLED;
g_server_cfg.data_length     = RS_SERIAL_PORT_DATA_LENGTH_8BITS;
g_server_cfg.stop_bits       = RS_SERIAL_PORT_STOP_BITS_ONE;

/* RTU-specific instance settings */
g_rtu_instance.rtu_serial_config.non_blocking = RS_SERIAL_BLOCKING;
g_rtu_instance.rtu_serial_config.rx_mode      = RS_SERIAL_RX_MODE_FIFO;
g_rtu_instance.rtu_serial_config.tx_mode      = RS_SERIAL_TX_MODE_FIFO;
```

ASCII Configuration Example

```
g_server_cfg.comm_type       = RS_MODBUS_COMM_TYPE_ASCII;
/* other serial fields same as RTU */
g_ascii_instance.ascii_serial_config.non_blocking = RS_SERIAL_BLOCKING;
g_ascii_instance.ascii_serial_config.rx_mode      =
RS_SERIAL_RX_MODE_FIFO;
g_ascii_instance.ascii_serial_config.tx_mode      =
RS_SERIAL_TX_MODE_FIFO;
```

TCP/IP Configuration Example

```
g_server_cfg.server_id       = 1;
g_server_cfg.comm_type       = RS_MODBUS_COMM_TYPE_TCP_IP;
memcpy(g_server_cfg.addr, "192.168.1.10", 13); /* server IP */
g_server_cfg.port            = 502;            /* standard port */
```

```
g_server_cfg.num_conn_accept = 5;              /* max TCP clients */
g_tcp_instance.tcp_socket_config.non_blocking = RS_SOCKET_NON_BLOCKING;
```

Field	RTU	ASCII	TCP/IP
server_id	1–247	1–247	1–247
comm_type	COMM_TYPE_RTU	COMM_TYPE_ASCII	COMM_TYPE_TCP_IP
addr[]	Port index / name	Port index	IP address string
baud_rate	Required	Required	—
parity	Required	Required	—
stop_bits	Required	Required	—
port	—	—	502 (standard)
num_conn_accept	—	—	1–5

Step 3: Open and Start the Server

Call rs_modbus_server_open() once at startup. Pass the instance pointer cast to the base type. Store the returned handle — it is required for all subsequent calls.

```
rs_handle_t g_mb_handle;

void app_modbus_init(void)
{
    rs_lib_init();   /* initialise RAPIDSEA platform layer */

    /* Open – cast instance to base type */
    g_mb_handle = rs_modbus_server_open(
                      (rs_modbus_server_instance_t *)&g_rtu_instance,
                      &g_server_cfg);

    if (g_mb_handle <= 0) {
        /* handle error */
        return;
    }
```

```
    /* Start listening for client requests */
    rs_modbus_server_run(g_mb_handle, 1 /*run*/);
}
```

NOTE:
For TCP/IP mode, also call rs_modbus_server_tcp_comm_handler(g_mb_handle) periodically alongside rs_modbus_server_process().

Step 4: Call the Process Function Periodically

rs_modbus_server_process() drives the entire server state machine. It must be called from your main task loop or a periodic timer. A 1 ms resolution is sufficient.

```
/* Application main task loop  (or 1 ms timer callback) */
void app_task_1ms(void)
{
    rs_modbus_server_process(g_mb_handle);

    /* TCP/IP only – manage client connections */
    /* rs_modbus_server_tcp_comm_handler(g_mb_handle); */
}
```

What process() does internally
Receives bytes from the serial/TCP transport layer
Validates the incoming Modbus frame (address, CRC/LRC, length)
Decodes the function code and dispatches to the appropriate rcb_ callback
Encodes the response frame (positive or exception)
Transmits the response back to the requesting client

Step 5: Implement the Register Callbacks

The stack calls these six functions from within rs_modbus_server_process(). You must implement each one and map the register address to your application data. Return 0 on

success; return a negative RS_MODBUS_ERROR_* value to send an exception response to the client.

Callback Summary

Callback	Triggered by FC	Direction
rcb_mb_get_coil_status_value	01 — Read Coils	Read from app → send to client
rcb_mb_set_coil_status_value	05 / 15 — Write Coil(s)	Receive from client → write to app
rcb_mb_get_discrete_input_value	02 — Read Discrete Inputs	Read from app → send to client
rcb_mb_get_holding_reg_val	03 — Read Holding Registers	Read from app → send to client
rcb_mb_set_holding_register	06 / 16 — Write Register(s)	Receive from client → write to app
rcb_mb_get_input_register_value	04 — Read Input Registers	Read from app → send to client

Minimal Implementation Example (from modbus_server_demo_app.c)

```
/* ---- Application data ---- */
static uint8_t  g_coil[16]    = {0};
static uint16_t g_holding[64] = {0};
static uint16_t g_input[32]   = {0};

/* NOTE: the stack passes (address - 1); add 1 to recover the user-facing
address */

rs_ret_val_t rcb_mb_get_coil_status_value(
    rs_handle_t handle, uint16_t addr, uint8_t *val)
{
    addr += 1;
    if (addr < 1 || addr > 16)
```

```
        return -RS_MODBUS_ERROR_INVALID_DATA_ADDR;
    *val = g_coil[addr - 1];
    return 0;
}

rs_ret_val_t rcb_mb_set_coil_status_value(
    rs_handle_t handle, uint16_t addr, uint8_t *val)
{
    addr += 1;
    if (addr < 1 || addr > 16)
        return -RS_MODBUS_ERROR_INVALID_DATA_ADDR;
    g_coil[addr - 1] = *val;
    return 0;
}

rs_ret_val_t rcb_mb_get_holding_reg_val(
    rs_handle_t handle, uint16_t addr, uint16_t *val)
{
    addr += 1;
    if (addr < 1 || addr > 64)
        return -RS_MODBUS_ERROR_INVALID_DATA_ADDR;
    *val = g_holding[addr - 1];
    return 0;
}

rs_ret_val_t rcb_mb_set_holding_register(
    rs_handle_t handle, uint16_t addr, uint16_t *val)
{
    addr += 1;
    if (addr < 1 || addr > 64)
        return -RS_MODBUS_ERROR_INVALID_DATA_ADDR;
    g_holding[addr - 1] = *val;
    return 0;
}
```

```
rs_ret_val_t rcb_mb_get_input_register_value(
    rs_handle_t handle, uint16_t addr, uint16_t *val)
{
    addr += 1;
    if (addr < 1 || addr > 32)
         return -RS_MODBUS_ERROR_INVALID_DATA_ADDR;
    *val = g_input[addr - 1];
    return 0;
}

rs_ret_val_t rcb_mb_get_discrete_input_value(
    rs_handle_t handle, uint16_t addr, uint8_t *val)
{
    /* implement similarly */
    (void)handle; (void)addr; *val = 0; return 0;
}

rs_ret_val_t rcb_mb_handle_custom_request(
    rs_handle_t handle, uint8_t *buf)
{
    /* handle non-standard function codes if needed */
    return 0;
}
```

WARNING:
The stack internally passes (register_address - 1) to the callbacks. Always add 1 inside the callback to recover the user-facing register address, consistent with the pattern used in modbus_server_demo_app.c.

Step 6: Complete Minimal Integration Example

The snippet below shows the full server startup sequence in one place (RTU mode). Combine Steps 1–5 into your application file.

```
/* ========================================================
 * Minimal RAPIDSEA Modbus Server — RTU mode
 * Sources: rs_modbus_server.h, rs_modbus_server.c
 * ======================================================== */
#include <rs_lib.h>
#include "rs_modbus_server.h"

/* Step 1 — instance and config */
static rs_modbus_rtu_server_instance_t  g_rtu_inst;
static rs_modbus_server_config_t        g_cfg;
static rs_handle_t                      g_hdl;

/* Step 2 — init */
void app_modbus_server_init(void)
{
    g_cfg.server_id   = 1;
    g_cfg.comm_type   = RS_MODBUS_COMM_TYPE_RTU;
    g_cfg.addr[0]     = '0';   /* serial port index */
    g_cfg.baud_rate   = 115200;
    g_cfg.parity      = RS_SERIAL_PORT_PARITY_DISABLED;
    g_cfg.flow_ctrl   = RS_SERIAL_PORT_FLOW_CTRL_DISABLED;
    g_cfg.data_length = RS_SERIAL_PORT_DATA_LENGTH_8BITS;
    g_cfg.stop_bits   = RS_SERIAL_PORT_STOP_BITS_ONE;

    g_rtu_inst.rtu_serial_config.non_blocking = RS_SERIAL_BLOCKING;
    g_rtu_inst.rtu_serial_config.rx_mode      = RS_SERIAL_RX_MODE_FIFO;
    g_rtu_inst.rtu_serial_config.tx_mode      = RS_SERIAL_TX_MODE_FIFO;

    rs_lib_init();
```

```
    /* Step 3 – open */
    g_hdl = rs_modbus_server_open(
                (rs_modbus_server_instance_t *)&g_rtu_inst, &g_cfg);

    /* Step 4 – run */
    rs_modbus_server_run(g_hdl, 1);
}

/* Step 4 – periodic process (1 ms task) */
void app_modbus_server_task(void)
{
    rs_modbus_server_process(g_hdl);
}

/* Step 5 – callbacks (see previous step for full implementations) */
```

6. Run-Time Reconfiguration

To change the baud rate, parity, or slave address without restarting the application, update the relevant field(s) in the config structure and call rs_modbus_server_reconfig(). The server briefly stops, reconfigures the transport, and restarts automatically. This pattern is used in the demo to allow a Modbus client to update baud rate/parity via holding register writes.

```
/* Example: client has written a new baud rate into the holding register
*/
g_cfg.baud_rate = new_baud_rate;
rs_modbus_server_reconfig(g_hdl);
```

TIP:
The handle remains valid after rs_modbus_server_reconfig(). No need to re-open.

7. Error Codes

The server automatically sends Modbus exception responses when a callback returns a negative error code. The following exception codes are defined in rs_modbus_server.h:

Macro	Value	Modbus Name	When to Return
RS_MODBUS_ERROR_INVALID_FUNCTION	1	Illegal Function	Function code not supported for this register
RS_MODBUS_ERROR_INVALID_DATA_ADDR	2	Illegal Data Address	Address out of range for this register type
RS_MODBUS_ERROR_INVALID_VALUE	3	Illegal Data Value	Written value out of valid range
RS_MODBUS_ERROR_SLAVE_DEVICE_FAILURE	4	Server Device Failure	Unrecoverable internal error
RS_MODBUS_ERROR_TIMEOUT	5	Acknowledge	Long processing time / timeout
RS_MODBUS_ERROR_SLAVE_DEVICE_BC	6	Server Busy	Still processing a previous command
RS_MODBUS_ERROR_PARITY	8	Memory Parity Error	Parity error reading extended memory

```
/* Return from callback to trigger exception EC 02 */
return -RS_MODBUS_ERROR_INVALID_DATA_ADDR;
```

8. Thread Safety and RTOS Considerations

The RAPIDSEA Modbus stack is **single-threaded by design**. All calls — rs_modbus_*_open(), rs_modbus_*_run(), rs_modbus_*_process(), and the rcb_ callbacks — must execute from the **same task or execution context**. The register callbacks are called synchronously from inside process(), on the same call stack as the caller.

RULE:
One task owns the stack. All other tasks share data only through protected application variables.

FreeRTOS — recommended task structure

```
/* Modbus owns this task entirely. No other task calls process(). */
void modbus_server_task(void *pvParams)
{
    app_modbus_server_init();
    for (;;)
    {
        rs_modbus_server_process(g_hdl);
        vTaskDelay(pdMS_TO_TICKS(1));
    }
}
```

9. Protecting shared data inside callbacks

If application data (sensor readings, control outputs) is written by a second task or an ISR, protect it with a mutex **inside the callback** — not outside:

```
/* FreeRTOS / CMSIS-RTOS2 example */

rs_ret_val_t rcb_mb_get_holding_reg_val(rs_handle_t handle,

                                        uint16_t addr, uint16_t *val)

{

    addr += 1;

    if (addr < 1 || addr > 64)

        return -RS_MODBUS_ERROR_INVALID_DATA_ADDR;

    osMutexAcquire(g_sensor_mutex, osWaitForever);

    *val = g_holding[addr - 1];  /* g_holding updated by sensor task */

    osMutexRelease(g_sensor_mutex);

    return 0;

}
```

Keep callbacks short. process() blocks until every callback returns. A callback that waits on a semaphore or performs I2C reads will stall the Modbus response and may cause the client to time out.

10. Rules summary

Rule	Detail
One task calls process()	Never call process() from two tasks or an ISR
Callbacks are synchronous	They run on the Modbus task's stack; keep them fast
Shared data needs a mutex	Protect every variable read/written by a second task
run() / reconfig() from another task	Requires external synchronisation before calling

Bare-metal superloop	No locking needed; single execution context throughout

Note for TCP servers: rs_modbus_server_tcp_comm_handler() must also be called from the same task as rs_modbus_server_process(). Both functions access the same server instance without internal locking.

11. Troubleshooting

Symptom	Likely Cause	Action
rs_modbus_server_open returns ≤ 0	Wrong instance type or config	Verify comm_type matches the instance struct; check rs_lib_init() was called
No response to client requests	rs_modbus_server_process not called	Ensure the process function is called every ~1 ms in your task loop
Client receives exception EC 02	Callback returning - RS_MODBUS_ERROR_INVALID_DATA_ADDR	Add 1 to addr inside callback; check address range validation
Client receives exception EC 03	Callback returning - RS_MODBUS_ERROR_INVALID_VALUE	Validate the incoming value range before storing
TCP client cannot connect	tcp_comm_handler not called, or wrong IP/port	Call rs_modbus_server_tcp_comm_handler periodically; verify num_conn_accept > 0
CRC errors on RTU	Baud rate or parity mismatch	Ensure baud_rate, parity, data_length, stop_bits match the client settings
Reconfiguration has no effect	Config struct not updated before reconfig	Update g_cfg fields, then call rs_modbus_server_reconfig(g_hdl)

12. Quick Reference Card

Task	Function / Macro
Initialise platform	rs_lib_init()
Open server channel	rs_modbus_server_open(instance, config)
Start server	rs_modbus_server_run(handle, 1)
Stop server	rs_modbus_server_run(handle, 0)
Periodic maintenance	rs_modbus_server_process(handle)
TCP connection management	rs_modbus_server_tcp_comm_handler(handle)
Reconfigure at run time	rs_modbus_server_reconfig(handle)
Read coil callback	rcb_mb_get_coil_status_value(...)
Write coil callback	rcb_mb_set_coil_status_value(...)
Read discrete input callback	rcb_mb_get_discrete_input_value(...)
Read holding register callback	rcb_mb_get_holding_reg_val(...)
Write holding register callback	rcb_mb_set_holding_register(...)
Read input register callback	rcb_mb_get_input_register_value(...)
Custom FC callback	rcb_mb_handle_custom_request(...)
Invalid address error	-RS_MODBUS_ERROR_INVALID_DATA_ADDR
Invalid value error	-RS_MODBUS_ERROR_INVALID_VALUE