RX72M Group

CPU Card Modbus Startup Manual

R01AN6757EJ0100
Rev.1.00
Jan 31, 2023

Introduction

This application note is a quick start guide for Modbus communication with the RX72M communication board for industrial network evaluation.

This stack runs on e-Force's real-time OS "µC3 (micro-C-cube)" and TCP/IP protocol stack "µNet 3 (micro-net-cube)".

Target Device

RX72M Group

Contents

1. Overview

This is a document of Modbus protocol stack that runs on RX72M, and gives an outline of functions, application programming interface (API), and application sample when developing and implementing an application that uses the protocol stack.

Modbus protocol stack for RX72M supports Ethernet-based Modbus TCP, RS-485 serial communication-based Modbus RTU and Modbus ASCII protocols.

This package supports Ethernet-based Modbus TCP server for operation with the RX72M CPU card.

1.1 Feature

The Modbus protocol is a communication protocol developed by Modicon Inc. (Schneider Electric SA.) for programmable logic controllers (PLCs), the specifications of which are publicly available.

Refer to the protocol specification (PI-MBUS-300 Rev. J).

The Modbus protocol stack for RX72M allows easy development of the following applications: The stack mode is specified by the initialization API at application execution time.

- Modbus TCP server (Build configuration support)
- Modbus TCP gateway
- Modbus RTU master
- Modbus RTU slave
- Modbus ASCII master
- Modbus ASCII slave

The Modbus protocol stack for RX72M supports the following nine function codes:

- 1 (0x01)-Read coils
- 2 (0x02)-Read discrete input
- 3 (0x03) – Read holding registers
- 4 (0x04) – Read input registers
- 5 (0x05)-Write single coil
- 6 (0x06)-Write single register
- 15 (0x0F)-Write multiple coils
- 16 (0x10) – Write multiple registers
- 23 (0x17) – Read / Write multiple registers

For more information about Modbus, please refer to the following site.

http://www.modbus.org

「Modicon Modbus Protocol Reference Guide Rev.J」 (PI_MBUS_300.pdf)

「Modbus Application Protocol Specification V1.1b3」(Modbus_Application_Protocol_V1_1b3.pdf)

Note) The version number may be different due to the update. Please refer to the latest manual.

1.2 Operating Environment

The sample program covered in this manual run in the environment below.

Table 1.1 Operating Environment

Item	Description
Board	Renesas Electronics RX72M CPU Card Model name: RTK0EMXDE0C00000BJ
CPU	RX CPU (RXv3)
Operating frequency	CPU clock (CPUCLK): 240 MHz
Operating voltage	3.3 V
Operating modes	• Single chip mode • Boot mode (SCI interface) • Boot mode (USB interface) • Boot mode (FINE interface)
Device requirements	R5F572MNDDBD • Code flash memory Capacity: 2/4 Mbytes ROM cache: 8 Kbytes • Data flash memory Capacity: 32 Kbytes • RAM/extended RAM Capacity: 512 Kbytes/512 Kbytes
Communications protocol	Modbus
Integrated development environment	e2Studio V7.5.0 or later
Tool chain	C/C ++ compiler package V3.01.00 or later for RX family
Emulator (ICE)	Renesas Electronics On-board emulator E2 On-Board (E2OB)
Evaluation tool	ModbusDemoApplication.exe: Modbus evaluation test program

1.3 Reference document

Technical information on Modbus is available from the Modbus Organization site, and information on the RX72M communication board is available from the Renesas Electronics site.

- Modbus Organization's site : http://www.modbus.org
- Renesas Electronics website : http://www.renesas.com

Table 1.2 Modbus related documents

Item	Description
1	Modbus_Application_Protocol_V1_1b3.pdf
2	PI_MBUS_300.pdf
3	Modbus_over_serial_line_V1_02.pdf
4	Modbus_Messaging_Implementation_Guide_V1_0b.pdf

Table 1.3 µC3/µNet3 related documents

Item	Description
1	ProcessorDependentManual_RXv3.pdf: µC3/Standard User's Guide Processor dependent part RXv3
2	TutorialGuide_RXv3_RX72M.pdf: µC3 / Standard Tutorial Guide RXv3-RX72M
3	uC3Std_UsersGuide.pdf: µC3 / Standard User's Guide
4	uNet3_BSD_UsersGuide.pdf: µNet3 / BSD User's Guide
5	uNet3_DriverGuide.pdf: uNet3 Ethernet Driver Interface
6	uNet3_UsersGuide.pdf: µNet3 User's Guide

Table 1.4 RX72M Communication board related documents

Item	Description
1	RX72M Group User's Manual: Hardware (R01UH0804EJ)
2	RX72M CPU Card with RDC-IC User's Manual (R12UZ0098EJ0100)
3	RX72M CPU Card with RDC-IC Schematic （R12TU0146EJ0100）

Table 1.5 Emulator related documents

Item	Description
1	E1 / E20 Emulator, E2 Emulator Lite User's Manual Supplement (RX User System Design) (R20UT0399EJ)
2	RX Family E1 / E20 Emulator User's Manual (R20UT0398EJ)

2. Hardware

For detailed information on the board, refer to the *RX72M CPU Card with RDC-IC User's Manual (R12UZ0098EJ0100)*.

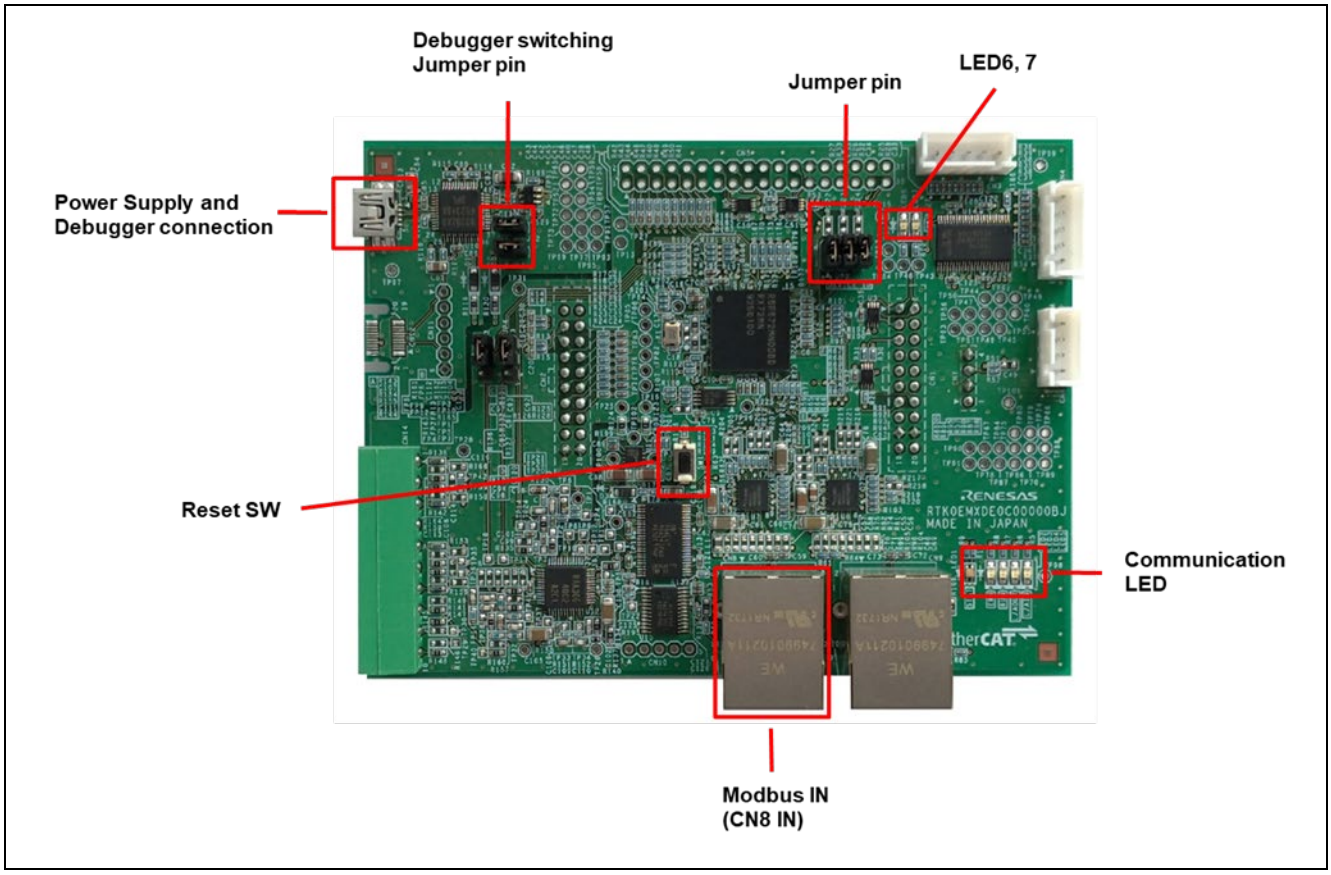

Figure 2.1 Configuration of the RX72M CPU Card

2.1 Setting up the CPU Card

Set the jumper pins before powering on the CPU card.

When using E2OB: Short the Debugger switching Jumper pin.

2.2 Power connection

CPU card does not have DC Jack, please input DC5V from USB connector.

3. Installing the e² studio

Download RX72M compatible e2studio (V7.5.0 or later) from the following website.

https://www.renesas.com/e2studio_download

3.1 Installing the CC-RX Compiler V3.01.00

The compiler selection screen appears while installing e2studio. By selecting [Renesas CCRX v3.0 1.00] and selecting [Next], CC-RX V3.0 1.00 compiler compatible with RX72M will be installed together.

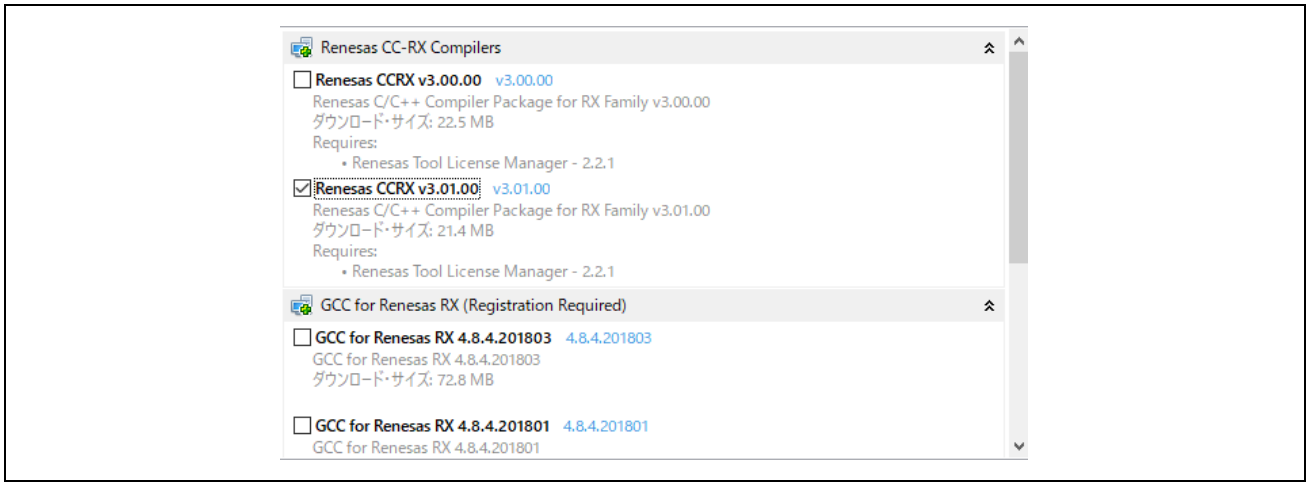

Figure 3.1 e2studio-Compiler selection

To start e2studio, please run "e2studio.exe" located in the installed folder below.

e2_studio_rx72m\eclipse

3.2 Registering the Tool Chain

Register the CC-RX compiler v3.01.00 so that it can be used with the e^2 studio for RX72M.

(1) Start the e^2 studio for RX72M.

(2) Select [File] → [New] → [C/C++Project] → [Next].

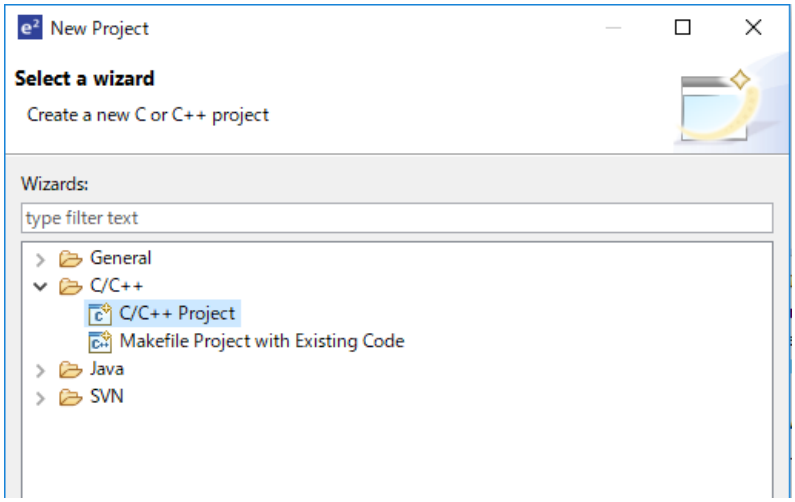

Figure 3.2 e2studio-Project selection

(3) In the [Templates for New C/C++ Project] dialog box, select [Renesas RX] → [Renesas CC-RX C/C++ Executable Project] → [Next].

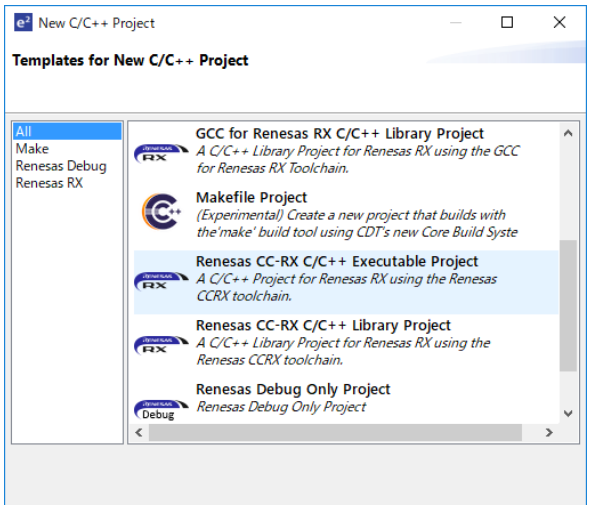

Figure 3.3 e2studio-Project selection

(4) In the [New Renesas CC-RX C/C++ Executable Project] dialog box, enter a desired project name and select [Next].

(5) In the [Select toolchain, device & debug settings] dialog box, select [Toolchain Management] under [Toolchain Settings].

(6) In the [Renesas Toolchain Management] dialog box, select [Add] → [Browse...] to refer to the installation folder "C:¥Renesas¥RX¥3_0_1".
The registration was successful if "v3.01.00 has been added under "Renesas CCRX".

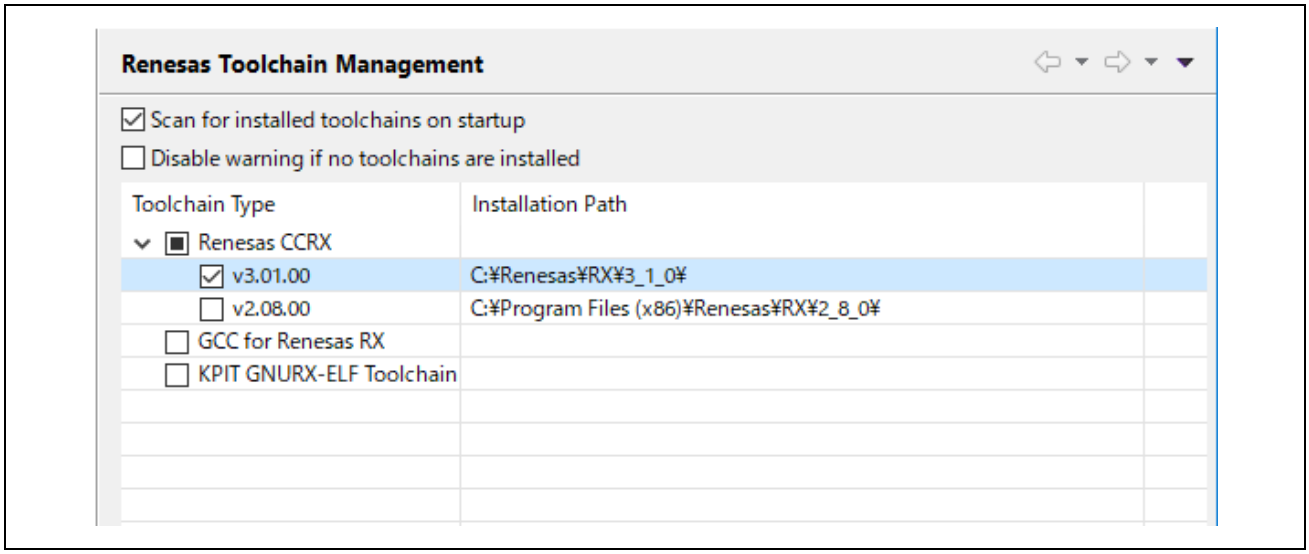

Figure 3.4 e2studio-Project selection

4. Sample application

4.1 Overview

This sample application program can be divided three blocks broadly.

1. Real time OS µC3 (micro C cube) and TCP/IP stack µNet3 (micro Net cube) provided by eForce.
2. Modbus protocol stack sample program, which uses µC3 and µNet3
3. Modbus application sample program, which uses Modbus protocol stack sample program.

4.1.1 µC3/µNet3 (Evaluation version)

This sample application includes evaluation softwares of real time OS µC3 (micro C cube) and TCP/IP stack µNet3 (micro Net cube) provided by eForce.

Please refer to chapter 6. Basic concept of TCP/IP stack for RX72M in detail.

4.1.2 Modbus protocol stack (Sample program)

This sample application includes sample program of Modbus protocol stack, which provides TCP or serial communication based on Modbus protocol. This sample program uses µC3 as RTOS and µNet3 as TCP/IP stack (for Modbus TCP protocol).

Please refer to chapter 7 Basic concept of Modbus stack for RX72M ~ chapter 11. Implementation in detail.

4.1.3 Modbus application (Sample program)

This sample application includes Modbus application sample program, which use µC3, µNet3 and Modbus protocols stack sample program.

Please refer to this chapter and chapter 5. Test communication by sample application in detail.

4.2 Block diagram

A block diagram of this sample application is shown in Figure 4.1.

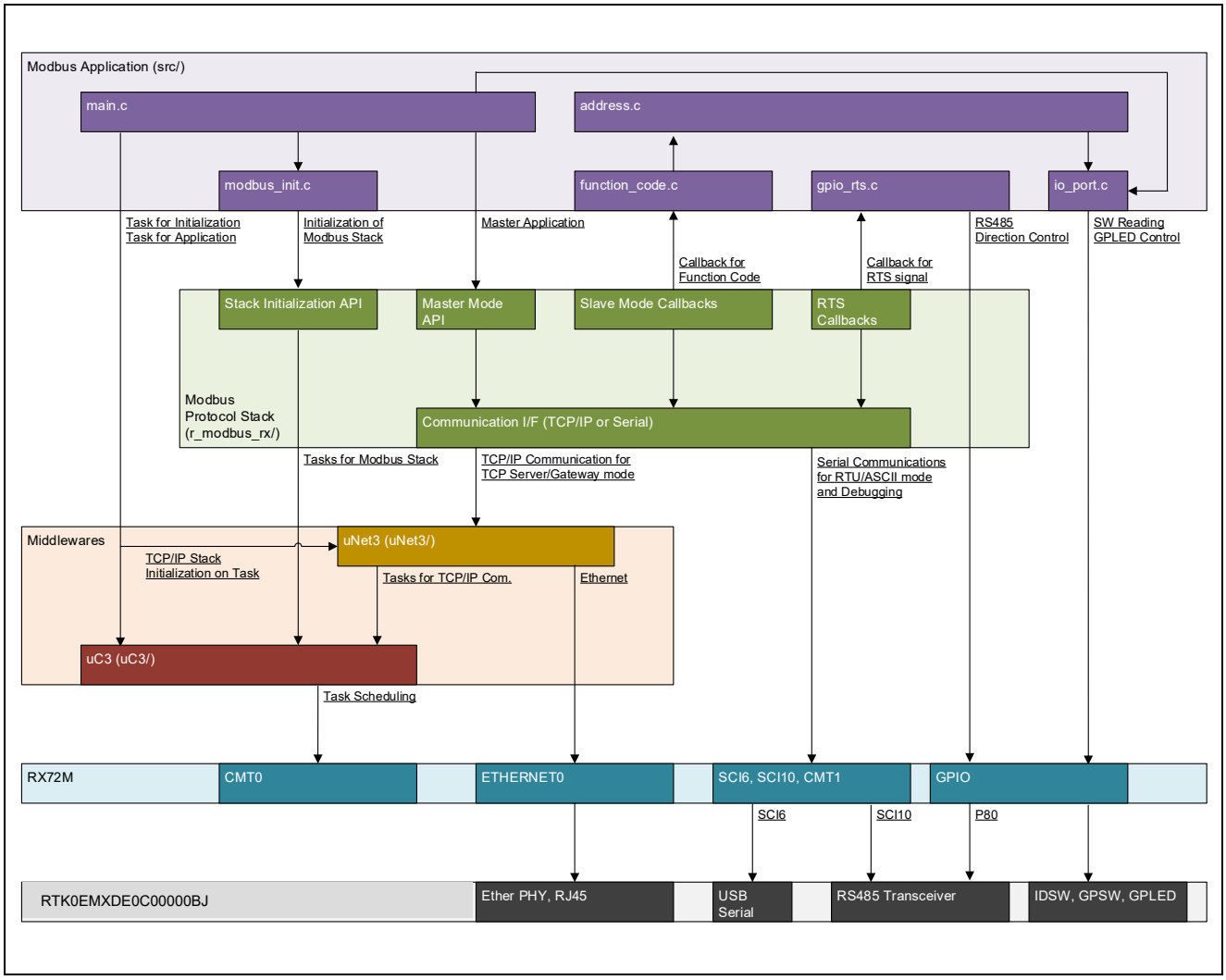

Figure 4.1 Feature block diagram of sample application

4.3 File structure

The file structure of the sample application is shown in Table 4.1.

Table 4.1 File structure of the sample application

Folder or File name	Description
src/	Modbus application sample programs
main.c	Initialization and main application task of Modbus application
io_port.c (.h)	GPIO access processing for Modbus application.
modbus_init.c (.h)	Initialization of Modbus protocol stack sample program.
mbapp_serial/	Implementation for Modbus RTU/ASCII serial communication.
gpio_rts.c (.h)	Call-back function for RS485 directional control.
mbapp_slave/	Implementation for Modbus salve mode.
function_code.c (.h)	Function code call-back functions for Modbus slave mode.
address.c (.h)	Data address implementation for Modbus salve mode.
r_modbus_rx/	Modbus protocol stack sample program.
inc/ (src/)	Header and source files of Modbus protocol stack.
master/	Master mode implementation based on Modbus protocol.
port/	Communication ports for Modbus protocol communication.
serial/	Implementation for Modbus RTU/ASCII protocol.
slave/	Slave mode implementation based on Modbus protocol.
tcp/	Implementation for Modbus TCP protocol.
modbusCommon.h	Modbus protocol genera definition.
modbusTyperdef.h	Typedef variables for Modbus protocol stack.
modbus.h	Include header for API of Modbus protocol stack.
uNet3/	µNet3 TCP/IP stack files
uC3/	µC3 RTOS files
generate/	Heap configuration files generated by e2studio

4.4 Build configuration

The build configurations of the e2studio project of this sample application shown in Table 4.2. This sample application prepares build configurations for each mode of Modbus protocol. This sample application supports only the Modbus TCP server stack as an operation on the RX72M CPU card.

Table 4.2 Build configuration

Stack mode	Build structure name
Modbus TCP server stack	TCP_SERVER_UC3
Server stack with Modbus TCP gateway	Not Support
Modbus RTU master stack	Not Support
Modbus RTU slave stack	Not Support
Modbus ASCII master Stack	Not Support
Modbus ASCII slave stack	Not Support

4.5 Resource

4.5.1 HW Module used on sample program

The HW modules on RX72M, which are used in each software blocks, are shown in Table 4.3. If you add the new features on this sample program, please note resource contention.

Table 4.3 HW Module used on sample program

SW block	HW module	I/O Port	Use
µC3 (Real time OS)	CMT0	-	RTOS Task scheduling
µNet3 (TCP/IP stack)	ETHERNET0	P74~5 PK0~5 PL2~7 PM4~7	Modbus TCP Server / TCP Gateway mode communication.
Modbus application	GPIO port	P15 PH3 PK6 PK7	LED control (CPU Card LD2~5)
		PB4 PB6 PB7	Jumper Switch input (CPU Card JP5)

* CMT : Compare match timer

4.5.2 Driver SW Module

The device driver software of each HW module are generated by µC3/Configurator, and it is modified for this sample application.

- **DDR_COM.c** : Micro C Cube Standard, DEVICE DRIVER Standard Communication Interface
  - Common modules of serial communication interface module.

- **DDR_RX_CMT0.c** : Micro C Cube Compact, DEVICE DRIVER Interval Timer code for RX CMT
  - Driver SW for CMT0 used on RTOS µC3.

- **DDR_RX_CMT1.c** : Micro C Cube Compact, DEVICE DRIVER Interval Timer code for RX CMT
  - Driver SW for CMT1 used on Modbus protocol stack.

- **DDR_RX_SCI3.c** : Micro C Cube Compact, DEVICE DRIVER Serial Interface for RX
  - Driver SW for SCI6 used for debug console of Modbus protocol.
  - Driver SW for SCI10 used on Modbus RTU/ASCII communication.

- **DDR_RX_ETH0.c** : Micro C Cube Compact, DEVICE DRIVER Ethernet driver for RX
  - Driver SW for ETHERNET0 used on Modbus TCP communication.

4.5.3 Real time OS configuration

The real time OS configuration files are generated by µC3/Configurator, and it is modified for this sample application.

- **" uC3/ kernel_cfg.h"**
  - Configures each parameter for real-time OS.

- **" uC3/ hw_init.c"**
  - init_peripheral ()
    - Implement the configurations of ETHERNET module port and Ether PHY clock.
  - _ddr_init ()
    - Call initialization functions of CMT driver SWs.

- **"uC3/kernel_cfg.c"**
  - Generate an instance of real time OS and generates tasks regarding each build configuration.
    - Start-up CMTs with _ddr_init() function before generating task.
  - Configure stack size as followings.
    - `#pragma stacksize su = 0x400 /* Time Event Handler */`
    - `#pragma stacksize si = 0x400 /* Isr Service Routine */`

4.5.4 Configure Heap size

The heap size of sample application is configured on following file and define macro.

- **" generate/sbrk.h"**
    - `#define HEAPSIZE 0x10000`

4.6 Modbus application overview

This section explains the overview of Modbus application sample program using Modbus protocol stack sample program.

4.6.1 Initialization and application implementation

4.6.1.1 main.c

This file starts up Modbus application with driver SWs, real time OS and TCP/IP stack by following procedure.

1. Initialize the peripheral HW modules and their driver SWs.

2. Start-up µC3 RTOS and generate all tasks.
   - After starting up µC3, initialization task automatically wake-up.
   - Subsequent initializations are processed on tasks.

3. Open serial port for debug console.
   - After this, serial console output can be used with debug_printf() function.
   - This feature is implemented for debug in Modbus protocol stack.

4. Start-up µNet3 TCP/IP stack for waking up TCP/IP communication tasks.
   - If it starts-up normally, the IP and port configured on device are shown via serial console output.
   - This process is required when using Modbus TCP communication.

5. Start-up Modbus protocol stack.
   - Please refer to explanation of modbus_init.c (.h) in detail.
   - When Modbus protocol stack use RTU/ASCII communication, serial communication configuration is shown via serial console output.

6. Wake-up main task and transits to Modbus application for each Modbus stack mode.
   - The application for Modbus RTU/ASCII mode only is implemented. Please refer to chapter 5 in detail.
   - The applications for other modes (slave or gateway) are not implemented because these modes passively work on each task when the tasks receive requests from opposed master device.

4.6.2 Initialization of Modbus protocol stack

4.6.2.1 modbus_init.c (.h)

These files describe an implementation example of initialization of Modbus protocol stack. Please refer to chapter 10 for API in detail.

This file starts up Modbus protocol stack by following procedure.

- Map callback functions for Modbus function code.
  - Used API：Modbus_slave_map_init
  - Implement callback functions for processing Modbus function code required Modbus master, and map the callbacks into dedicated function pointers.
    - An implementation example of function code is explained in the section of mbapp_slave/function_code.c (.h) .
  - This initialization process is required by Modbus RTU/ASCII slave mode and Modbus TCP Server mode only.
    - In exceptional cases, Modbus TCP Gateway mode requires initialization process, but does not require to set call back functions into dedicated function pointers.

- Enable host IP list and register IPs allowed to access.
  - Used API：Modbus_tcp_init_ip_table、Modbus_tcp_add_ip_addr
  - Enable host IP list and register master device IPs allowed to access.
  - Modbus TCP server mod and Modbus TCP gateway mode can use this feature.
  - In this sample code, allows all IPs to access by disabling this feature for make the tutorial easy.

- Configure serial communication configuration and start-up Modbus protocol stack with each mode.
  - Used API：Modbus_serial_stack_init、Modbus_tcp_init_stack
  - Set serial communication configuration, for Modbus RTU/ASCII communication, into dedicated struct variable.
  - Set callback functions, to control RTS signal for RS485 transceiver, into dedicated struct variable.
    - An implementation example of the callback functions to control RTS signal is shown in the section of mbapp_serial/gpio_rts.c (.h) .
  - Input the struct variables as arguments into each initialization API to start-up Modbus protocol stack with each stack mode.
  - Modbus TCP server mode does NOT use serial communication then it inputs NULL into struct variable argument.

- Initialize implementation sample of call back functions for Modbus function code
  - In detail, Initialize Modbus data model accessed form call back function for Modbus function code.
  - An implementation example of Modbus data model is explained in mbapp_slave/address.c (.h) section.

4.6.3 Implementation for slave (server) mode

The following files becomes targets of build when RTU_SLAVE_UC3, ASCII_SLAVE_UC3, and TCP_SERVER_UC3 build configurations, which runs Modbus protocol stack as slave or server mode, are specified.

4.6.3.1 mbapp_slave/function_code.c (.h)

This file describes an implementation example of callback functions for processing each behavior based on Modbus function code in requests from Modbus master.

- The Modbus protocol stack can register callback functions corresponding to function codes shown in Table 4.4. This source file describes examples for all callback functions.
- Modbus protocol has unique data model which composed of 4 data types and address space of each data type. The four data types supported by this protocol stack are shown in Table 4.5.
- The callback functions are required to process function code which access to each data type corresponding to each function code.
- In Modbus protocol, slaves can have up to 65536 (0x10000) data for each data type and can map the data into any reference data addresses in range 1~65536 (0x00001~0x10000).
- The reference data address can assign any physical address on device.
- This Modbus protocol stack implements processes of function codes as callback functions to access any Modbus data model which can be arbitrarily designed by user.

Table 4.4 Modbus function code supported by Modbus protocol stack sample program

Code	Code (Hex)	Function Name	Description
1	1h	Read Coils	Read multiple data of specified Coils addresses.
2	2h	Read Discrete Inputs	Read multiple data of specified Discrete Inputs addresses.
3	3h	Read Holding Registers	Read multiple data of specified Holding Registers addresses.
4	4h	Read Input Registers	Read multiple data of specified Input Registers addresses.
5	5h	Write Single Coil	Write single data of specified Coils addresses.
6	6h	Write Single Register	Write single data of specified Holding Registers addresses.
15	Fh	Write Multiple Coils	Write multiple data of specified Coils addresses.
16	10h	Write Multiple Registers	Write multiple data of specified Holding Registers addresses.
23	17h	Read/Write Multiple Registers	First, write multiple data of specified Coils addresses. Second, write multiple data of specified Holding registers address.

Table 4.5 Modbus data types supported by Modbus protocol stack sample program

Name	Bits	Type of access	Description
Discrete Inputs	1	Read	Data type can be provided by I/O system
Coils	1	Read/Write	Data type can be modified by application program.
Input Registers	16	Read	Data type can be provided by I/O system
Holding Registers	16	Read/Write	Data type can be modified by application program.

(Continued on next page)

The callback functions for Modbus function code should be implemented the following processes.

1. Set a result of function code process based on a request struct variable passed as an argument into a response struct variable passed as an argument.
2. Set 0x02: illegal data address exception into the response struct variable when a reference address which a designed Modbus data model does NOT have are referred.

And, this callback function can set 0x04: slave device failure exception into the response struct variable when unrecoverable error occurs during function code process.

Table 4.6 Modbus exception codes can be return in callback functions.

Code	Code (Hex)	Function Name	Description
2	2h	Illegal Data Address	Return this code when specified addresses in request includes incorrect address for user-designed Modbus data model.
4	4h	Slave Device Failure	Return this code when unrecoverable error occurs during function code process.

4.6.3.2 mbapp_slave/address.c (.h)

This source file describes an implementation example of Modbus data model. The map of reference address and physical address of each data type show in Figure 4.2.

- Each Modbus data type has 32768 data and uses reference address 0x0001~0x8000 (*1).
- The reference addresses are assigned to RAM memory as array buffer variables having static global attributes.
- Functions to access(read/write) each array buffer of each Modbus data type, which are used in callback functions for Modbus function code, are implemented.
- The read/write access functions assign the buffer array to RX72M peripheral registers when the functions access to specific reference address. (*2)
  - Coils addresses 0x0001~0x0004: GPIO port assigned to LED on RX72M COM board
  - Discrete Inputs address 0x0001~0x0008: GPIO port assigned to SW5 on RX72M COM board
- When reference address 0x5002 of each data type is referred, the read/write access functions return 0x04 : slave device failure exception code with assuming that unrecoverable error occurs. (*3)
- Functions to check whether the specified addresses include incorrect address for each Modbus data type are implemented.
- The callback functions for Modbus function code can return 0x02: illegal data address exception before accessing to buffer array to detect incorrect address by the checking functions. (*4)

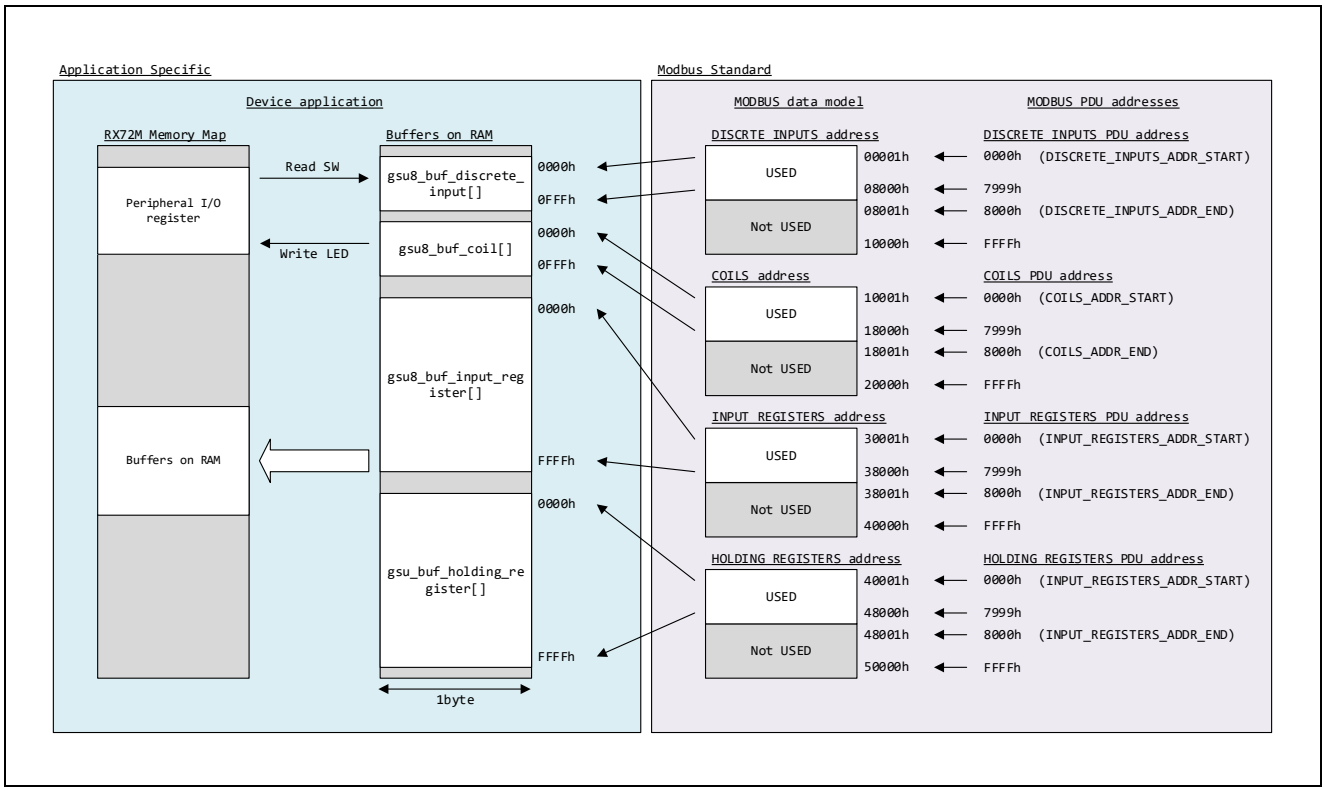

Figure 4.2 Modbus data model and memory mapping

(Continued on next page)

*1

The range of used reference address can be specified by constant macro for each Modbus data type.

- For example, COILS_ADDR_START and COILS_ADDR_END constant macros can specify the start and end of reference address for Coils data type.
  - In default, the start address is set to 0x0000 and the end address is set to 0x8000 for each Modbus data type, respectively.
  - Here, please note the specified address value starting form 0x0000 but not 0x0001 because this is based on address value representation of address in PDU (protocol data unit) of which Modbus request composes. Hereinafter, this address representation is called PDU address.
  - This sample program processes all reference addresses as PDU address.
  - Please note that the end address values such as COILS_ADDR_END is “exclusive” so that the address specified by the macro becomes “incorrect address” for designed Modbus data model.

*2

In this sample program, the reference addresses assigned to peripheral registers of RX72M are defined by relative address from COILS_ADDR_START and DISCRETE_INPUT_ADDR_START.

- For example, when COILS_ADDR_START is set to 0x1000, Coils reference addresses 0x1001~0x1004 assign GPIO ports corresponding to LED on CPU Card.

*3

The reference address which returns 0x04: Slave device failure can be specified constant macro PUSEDO_SLAVE_FAILURE_ADDR. Please note that the specified value is PUD address.

*4

Any address can be designed as incorrect address by modifying the checking functions to detect incorrect reference address.

4.6.4 IO port implementation for application

4.6.4.1 ioport.c/.h

This source file describes the processing of the GPIO port corresponding to the CPU card LED and jumper switch (JP5).

If Modbus protocol stack runs as slave mode, these GPIO ports are operated from accessing function for Modbus data implemented in address.c(.h).

If Modbus protocol stack runs as master mode, these GPIO ports are operated from application function for master program implemented in main.c.

4.7 Others

4.7.1 Introduction of µC3/µNet3 official version

The evaluation version of µC3/µNet3 in sample program includes some limitation for the behavior. Please refer to below website to purchase of official version of µC3/µNet3.

- https://www.eforce.co.jp/

The official version of µC3/µNet3 can be applied to this sample application by replacing following libraries for evaluation version of µC3/µNet3 to corresponding libraries for official version of µC3/µNet3.

	Targets libs (evaluation ver.) for replacing	New libs in install folder of official ver. of µC3/µNet3
uC3	{project_root}\uC3\ uC3RXv3b.lib uC3Rxv3l.lib uC3RXv3rbb.lib uC3Rxc3rbl.lib	{install_root}\Kernel\Standard\lib\RXv3\e2studio uC3RXv3b.lib uC3Rxv3l.lib uC3RXv3rbb.lib uC3Rxc3rbl.lib
uNet3	{project_root}\uNet3\ uNet3BSDRXv3b.lib uNet3BSDRXv3l.lib uNet3RXv3b.lib uNet3RXv3l.lib	{install_root}\Network\TCPIP\lib\RXv3\e2studio uNet3BSDRXv3b_Std.lib uNet3BSDRXv3l_Std.lib uNet3RXv3b_Std.lib uNet3RXv3l_Std.lib

{project_root} : Root folder of this sample application.

{install_root} : Install folder of official version µC3/µNet3 for RX72M

5. Test communication by sample application

5.1 Hardware connection

The Modbus protocol stack has different hardware connection methods depending on the stack mode.

5.1.1 Modbus TCP Server stack mode

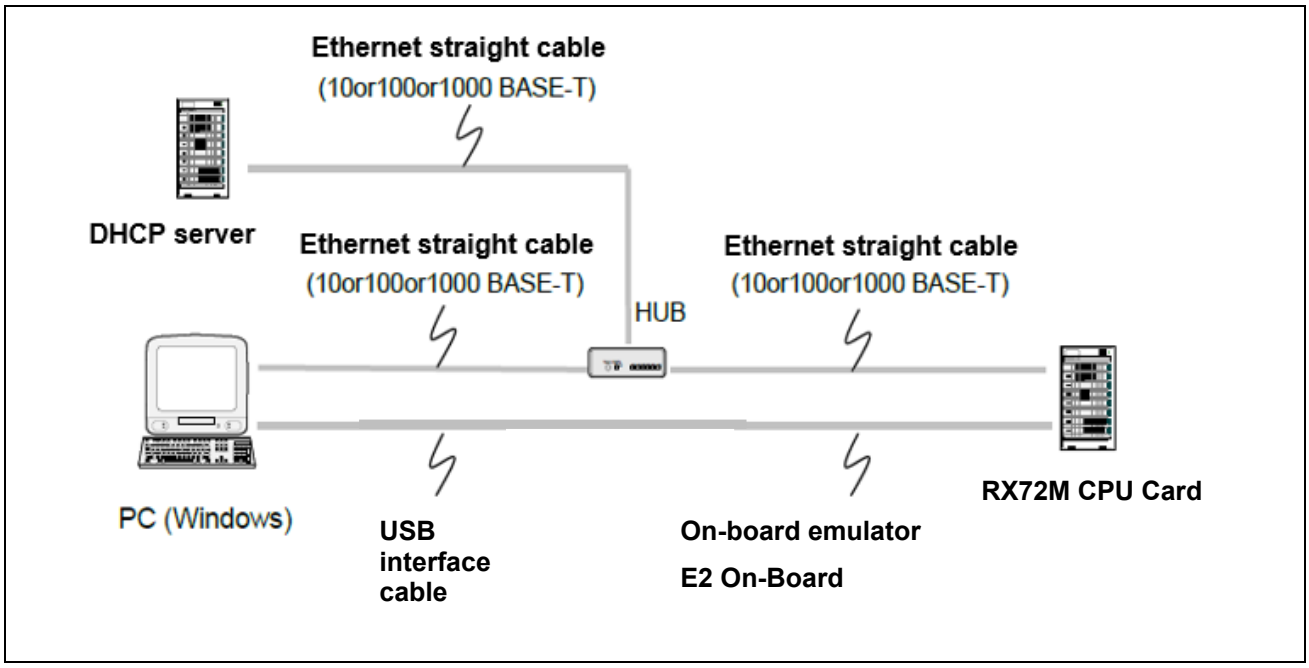

Figure 5.1 Hardware connection example in Modbus TCP server stack mode

In this sample program, the IP address is obtained automatically from the DHCP server.
After a certain period of time, it will switch to a fixed IP (192.168.1.103).

5.2 Execution procedure

Describes the procedure to execute communication in the sample application.

Complete the hardware connection according to the protocol stack mode to be operated by the sample application referring to 5.1 Hardware Connection in advance.

(1) After starting e2studio, click "File"-> "Import".

(2) In the "Select" dialog, select "General" → "To an existing project to workspace" and click "Next".

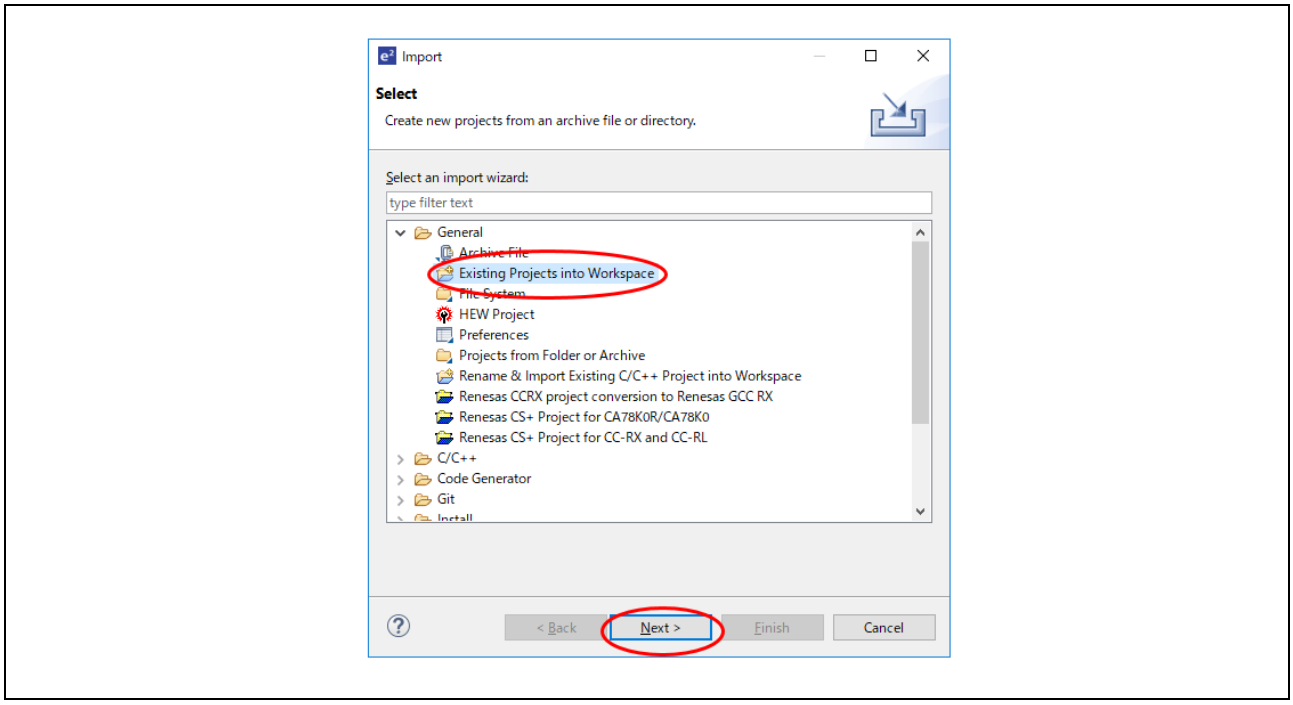

Figure 5.5 Project import

(3) Select the "Select archive file" check box in the "Import project" dialog and click "Browse". Select “rx72m_cpucard_modbus_eva.zip” and click “Open”. Click Finish to complete the project import.

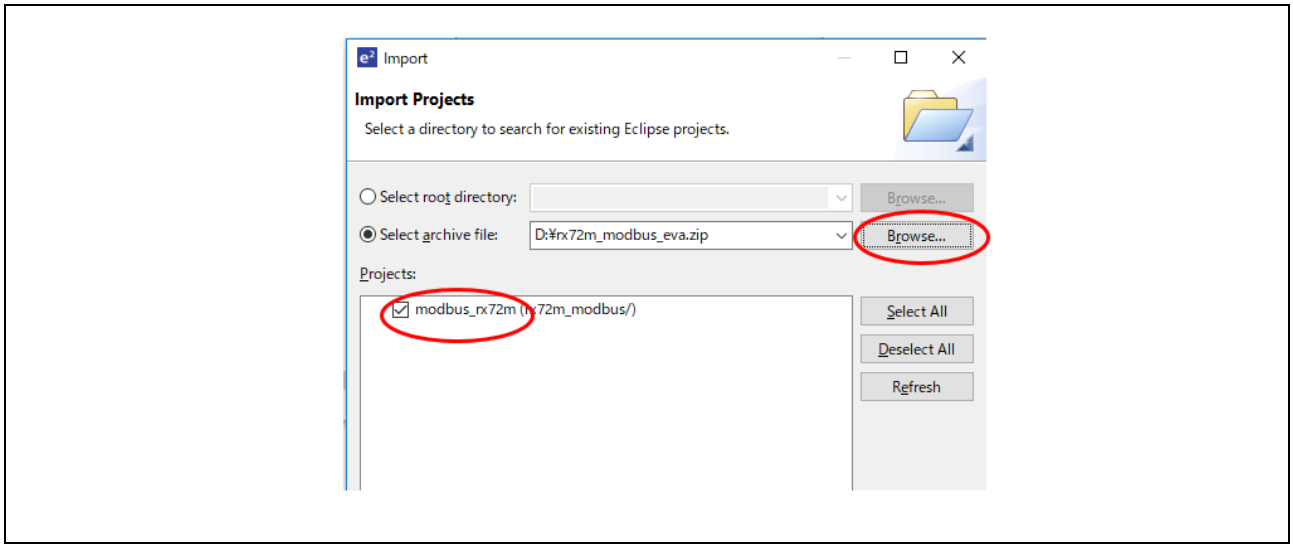

Figure 5.6 Archive file selection

(4) Right-click the sample project, check that "TCP_SERVER_UC3" is active from "Activate" in "Build Configuration", and execute the build.

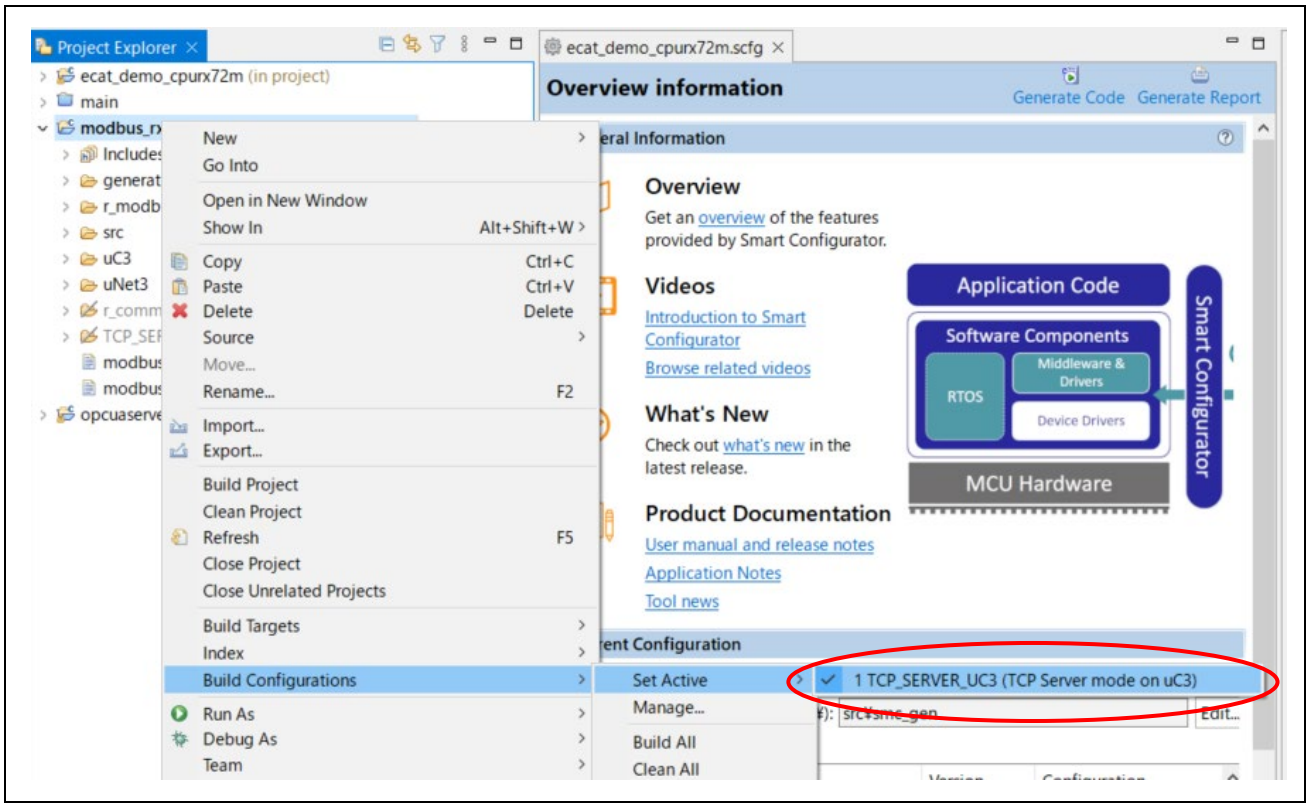

Figure 5.7 Build configuration settings

(5) Debug is started from "Configure Debug" under "Run".

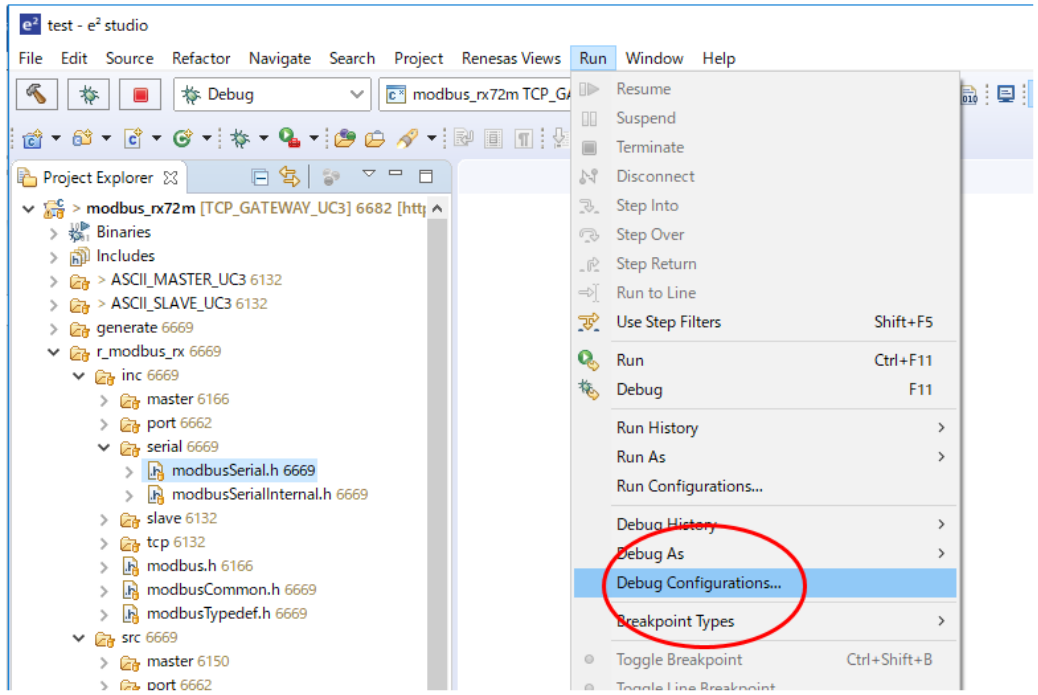

Figure 5.8 Debug Configuration Selection

(6) Select "Debug Configuration" according to the build configuration from Renesas GDB Hardware Debugging below, and click the "Debug" button to start the debugger.

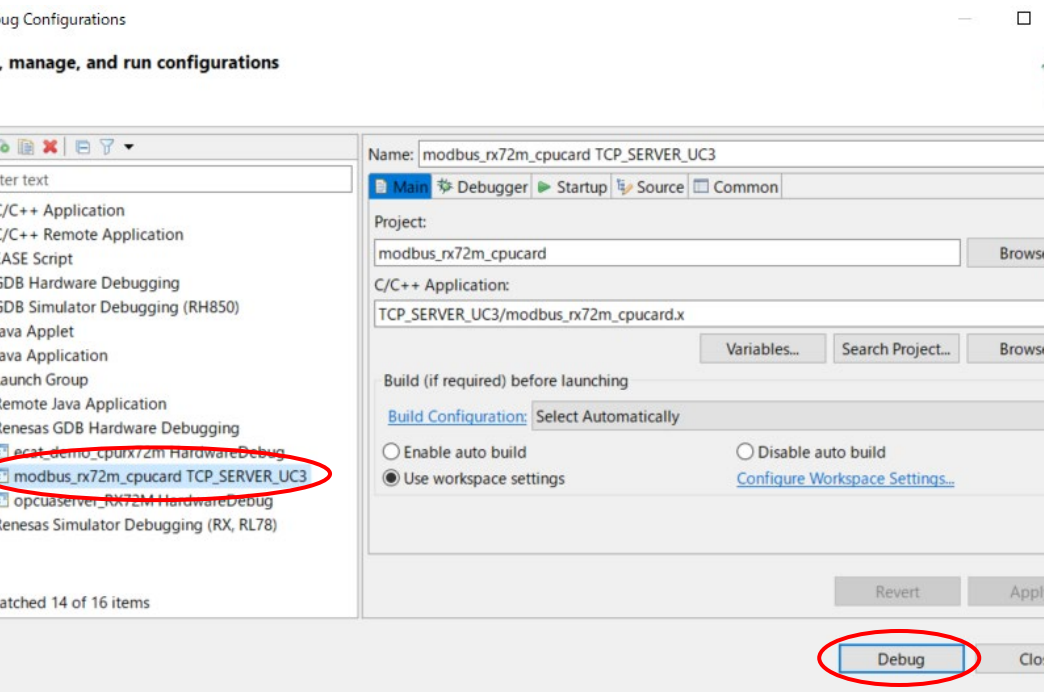

Figure 5.9 Debugger start

(7) After pressing the "Restart" button, press the "Resume" button to operate the sample application.

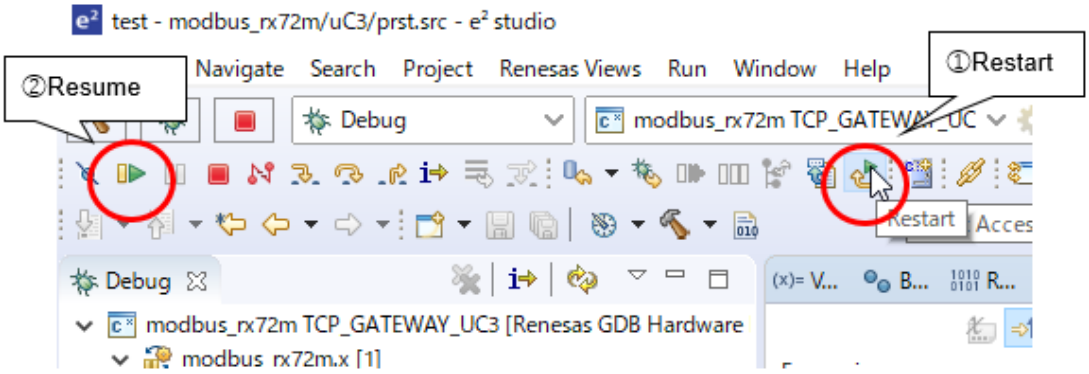

Figure 5.10 Start sample application

5.3 Modbus Communication tests with evaluation tool

This chapter makes sure the work of sample programs (Modbus protocol stack, and its applications) by using an evaluation tool (ModbusDemoApplication.exe).

5.3.1 Overview of evaluation tool

The evaluation tool runs on PC, and runs as an opposite software of Modbus application sample program on RX72M CPU Card.

RX72M CPU Card controls the light of LED or send the state of JP5 in response to Modbus communication with evaluation tool.

Coils address and assigned LEDs are shown in Table5.1 and Discrete inputs address and assigned SW5s are shown in Table 5.2

Table5.1 Coil address and assigned LED

Coils address	Assigned LED
0001h	LED5
0002h	LED4
0003h	LED3
0004h	LED2

Table 5.2 Discrete Input address and assigned SW

Discrete Inputs address	Assigned SW
0001h	JP5 1-4
0002h	JP5 2-5
0003h	JP5 3-6

(Continued to next page)

When the sample application runs as Modbus slave, the evaluation tool runs as Modbus master.

The behavior can be made sure by following steps.

- The evaluation tool sends a request of Read Discrete Inputs.
- The sample application receives the request of Read Discrete Inputs and send a state of JP5 on RX72M CPU Card as a response.
- The evaluation tool decides a way of update LED on RX72M CPU Card depending on the state of JP5 received as a response of a request of Read Discrete Inputs and sends a request of Write Multiple Coils.
  - JP5 1-4 Open : Change the lighting LED regularly.
  - JP5 1-4 Short : Apply values specified by Coils text box on Evaluation tool to LED.
- Sample application receives a Write Multiple Coils request and updates the lighting state of the specified RX72M CPU card LED.

When the sample application runs as Modbus master, the evaluation tool runs as Modbus slave.

The behavior can be made sure by following steps.

- The sample application sends request of Read Coils every 1 second.
- The evaluation tool receives the request of Read Coils and sends values set in Coils text box as a response.
- The sample application receives the values as a response of a request of Read Coils, and apply the values to LEDs

5.3.2 How to use evaluation tool.

Select the operation mode with "Connection" and set various parameters.

- Connection
  - ➢ Select TCP server.

- Serial setting
  - ➢ No setup required.

- Remote Modbus Server
  - ➢ Set the IP address and port number according to the device.
  - ➢ When connecting with a fixed IP address, please set as follows.
    IP Address: 192.168.1.103 Port: 502

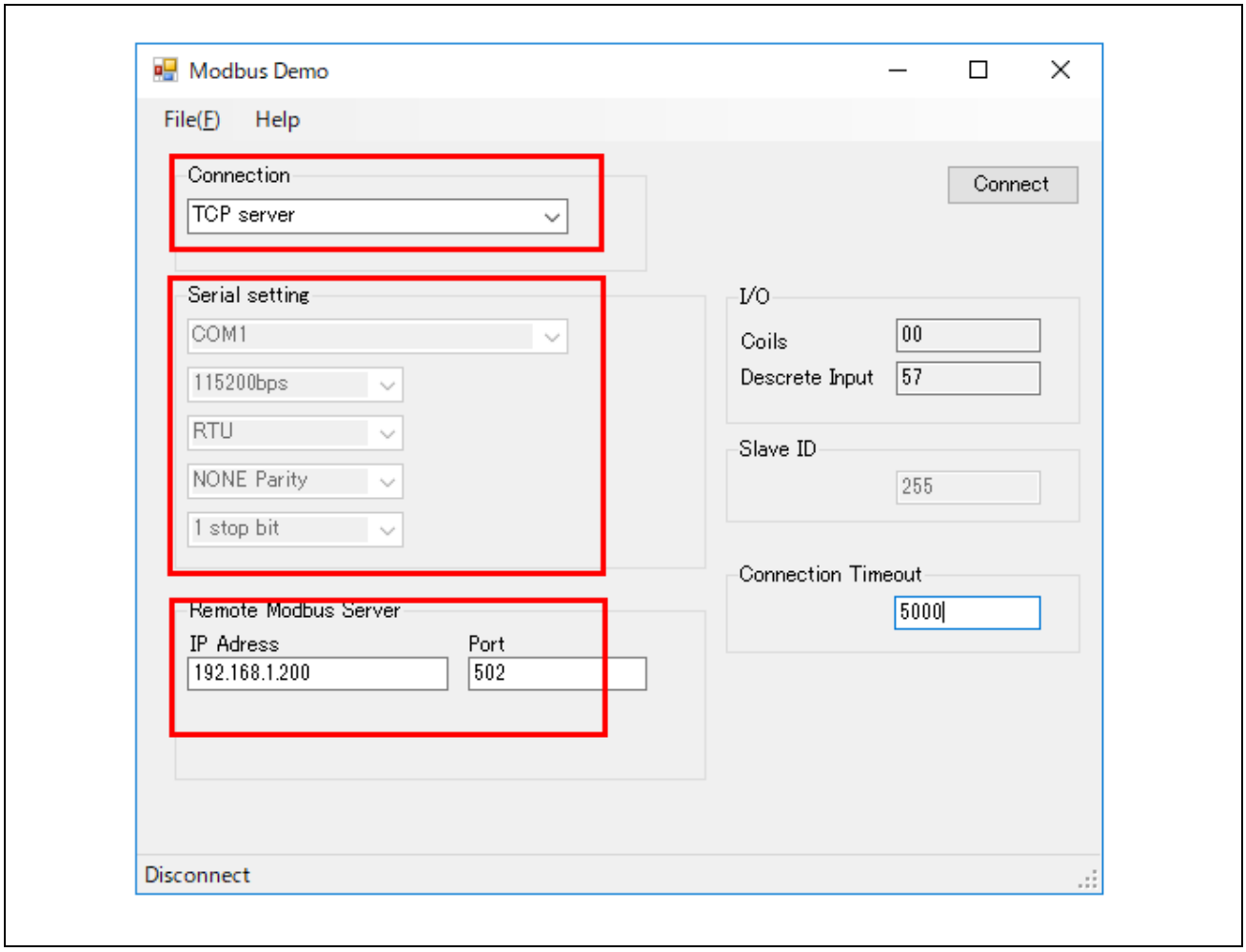

Figure 5.15 Evaluation tool screen

(Continued to next page)

When the evaluation tool runs as Modbus master;

- In case JP5 1-4 of RX72M CPU card is open, Coils text box will be changed automatically.
- In case JP5 1-4 of RX72M CPU card is shorted, Coils text box can be changed manually.
- The value in the Coils textbox is reflected in the LEDs on the RX72M CPU card at regular intervals.

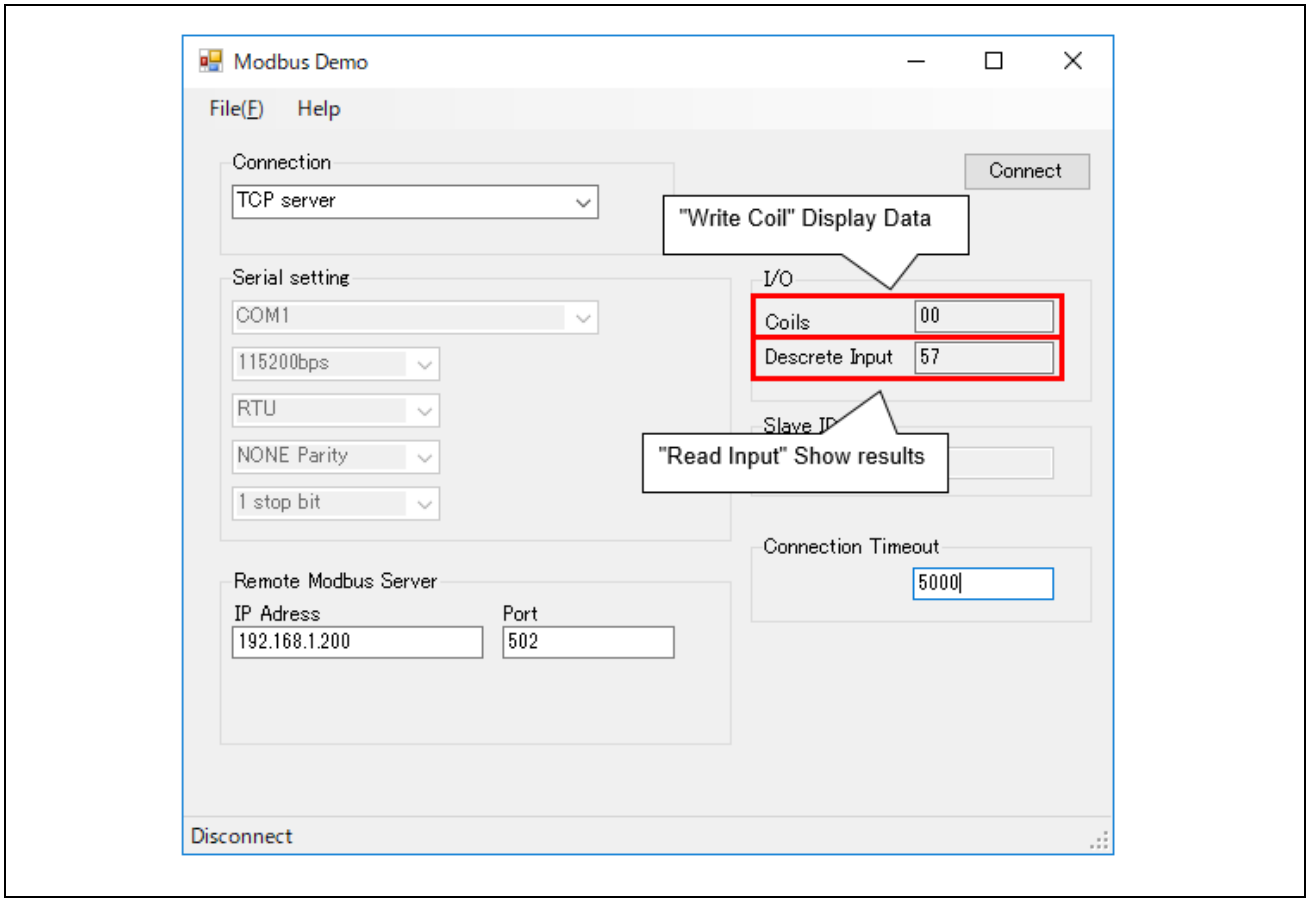

Figure 5.16 The evaluation tool running as Modbus master

(Continued to next page)

When the evaluation tool runs as Modbus slave;

- the Coils text box can be manually input by user.
- The values on Coils text box is applied to LEDs on RX72M CPU Card at regular intervals.

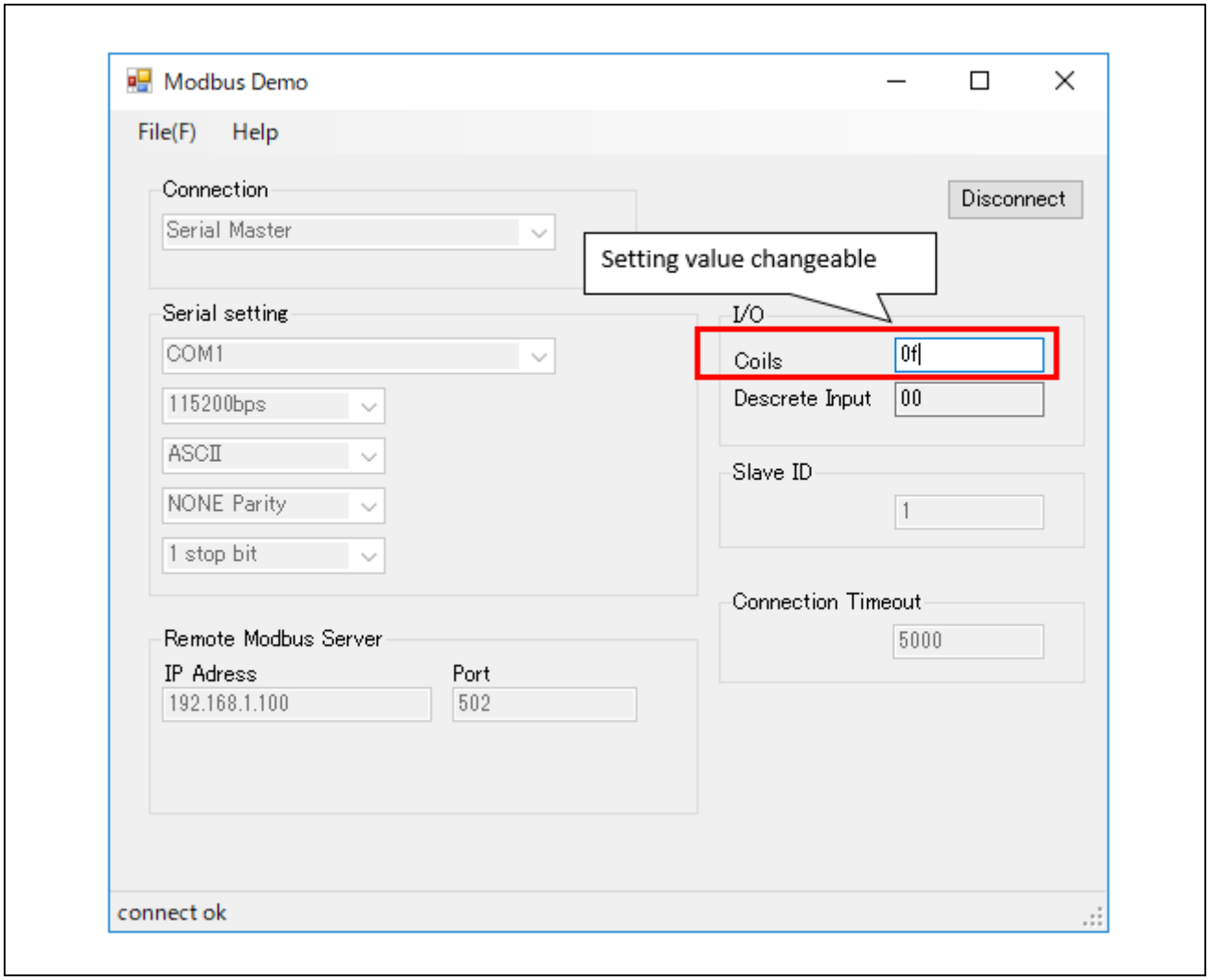

Figure 5.17 The evaluation tool running as Modbus slave

note The sample application side will be the server, so enable communication before the slave device starts operating.

6. Basic concept of TCP/IP stack for RX72M

Modbus TCP stack for RX72M uses eforce TCP/IP protocol stack "µNet3" for Ethernet network interface. This is a TCP/IP protocol stack that is also implemented for RTOS "µC3" from eforce.

6.1 µNet3 Module Configuration

µNet3 consists of three modules: Application Interface, TCP/IP Protocol Stack, and Network Device Control API.

- Application interface

Provides an interface (API) to use various network services such as establishing connection with remote host, sending / receiving data, etc.

- TCP/IP protocol stack

It handles network protocols such as TCP, UDP, ICMP, IGMP, IP, and ARP.

- Network device control API

There may be various network devices in the network system, and each device needs a device driver. The network device control API absorbs the differences between these devices and provides an interface for uniform access. Access each device from the application program using the device number.

Please refer to the following document for details.

- UNet3_UsersGuide.pdf: µNet3 User's Guide
- UNet3_DriverGuide.pdf: µNet3 Ethernet Driver Interface
- UNet3_BSD_UsersGuide.pdf: µNet3 / BSD User's Guide

6.2 Procedure of Development

Describe the configuration such as network configuration (IP address, socket definition), device driver configuration (MAC address, device I / F definition, driver specific feature), µNet3 initialization routine call etc. in net_cfg.c source file.

Describe each configuration definition in the net_cfg.h header file.

6.2.1 Configuration definition list

Table 6.1 lists the configurable parameters and settings in the Modbus protocol stack.

Table 6.1 **Configuration definition list**

Configuration define	Value	Item
CFG_NET_DEV_MAX	1	Number of data link devices
CFG_NET_SOC_MAX	10	Maximum number of used sockets
CFG_NET_TCP_MAX	8	Maximum number of used TCP sockets
CFG_NET_ARP_MAX	8	ARP entry number
CFG_NET_MGR_MAX	8	Number of multicast entries
CFG_NET_IPR_MAX	2	IP rebuilding queue number
CFG_NET_BUF_SZ	1576	Network buffer size
CFG_NET_BUF_CNT	8	Network buffer number
CFG_NET_BUF_OFFSET	2	Network buffer data write position
CFG_PATH_MTU	1560	MTU size
CFG_ARP_RET_CNT	3	ARP retry count
CFG_ARP_RET_TMO	1*1000	ARP retry timeout
CFG_ARP_CLR_TMO	20*60*1000	ARP cache clear timeout
CFG_IP4_TTL	64	IP header TTL value
CFG_IP4_TOS	0	IP header TOS value
CFG_IP4_IPR_TMO	10*1000	IP reassembly packet wait time
CFG_IP4_MCAST_TTL	1	IP header TTL (multicast packet)
CFG_IGMP_V1_TMO	400*1000	IGMPV1 timeout
CFG_IGMP_REP_TMO	10*1000	IGMP report timeout
CFG_TCP_MSS	1460	MSS
CFG_TCP_RTO_INI	3	TCP retry timeout initial value
CFG_TCP_RTO_MIN	500	TCP retry timeout minimum value
CFG_TCP_RTO_MAX	60	TCP retry timeout maximum value
CFG_TCP_SND_WND	1024	Send buffer size
CFG_TCP_RCV_WND	1024	Receive buffer size (Window size)
CFG_TCP_DUP_CNT	4	Number of retry start duplicate ACKs
CFG_TCP_CON_TMO	75*1000	SYN timeout
CFG_TCP_SND_TMO	64*1000	Transmission timeout
CFG_TCP_CLS_TMO	75*1000	FIN timeout
CFG_TCP_CLW_TMO	20*1000	Close Wait timeout
CFG_TCP_ACK_TMO	200	ACK timeout
CFG_TCP_KPA_CNT	0	Number of KeepAlive notifications before disconnecting
CFG_TCP_KPA_INT	1*1000	Notification interval after starting KeepAlive
CFG_TCP_KPA_TMO	7200*1000	No communication time until starting KEepAlive
CFG_PKT_RCV_QUE	1	Received packet queuing number
CFG_TCP_RCV_OSQ_MAX	6	Receive sequence guarantee queuing number
CFG_PKT_CTL_FLG	0x0000	Received packet checksum verification invalid flag
CFG_ARP_PRB_WAI	1*1000	ARP Probe transmission wait time when net_acd () is executed
CFG_ARP_PRB_NUM	3	Number of ARP Probe transmissions when net_acd () is executed
CFG_ARP_PRB_MIN	1*1000	Minimum wait time until next ARP Probe transmission
CFG_ARP_PRB_MAX	2*1000	Maximum wait time until next ARP Probe transmission
CFG_ARP_ANC_WAI	2*1000	Detection wait time after sending ARP Probe
CFG_ARP_ANC_NUM	2	Number of ARP Announce transmissions
CFG_ARP_ANC_INT	2*1000	Wait time until next ARP Announce transmission

6.2.2 Initialize of Protocol stack

To use eforce's TCP/IP protocol stack "µNet 3", it is necessary to initialize the protocol stack and initialize the network device.

The following shows a sample of initialization.

- Initialize of Protocol stack

```
ercd = net_ini();
if (ercd != E_OK) {
  return ercd;
}
```

- Initialize of Network device (Device# is N)

```
ercd = net_dev_ini(N);
if (ercd != E_OK) {
  return ercd;
}
```

The initialization code is implemented in the net_setup function of net_cfg.c.

6.3 Ethernet Driver interface

Describes porting methods for implementing the µNet3 protocol stack on network devices and the interfaces necessary for developing network device drivers.

6.3.1 File structure

In order to implement the network device driver, source code needs to be written in the following file.

- Net net_cfg.c

  Describe the configuration of µNet3 and the interface with the network device driver.

- DDR DDR_RX_ETH0.c / DDR_RX_ETH0.h

  Network device driver itself. Describe the device name etc. This file needs to be newly prepared when developing a new network device driver.

- DDR DDR_RX_ETH0_cfg.h

  Describe configuration macros for network device drivers such as PHY ID and MII / RMII mode. This file needs to be newly prepared when developing a new network device driver.

6.3.2 Interface

µNet3 uses the device number to access the network device. The network device driver is implemented by setting the device initialization function, frame transmission function, device release function, device control function, and device state acquisition function to gNET_DEV [] (* 1) pointed to by its device number.

In order for µNet3 to process frames received from network devices, it is necessary to implement appropriate reception processing functions in the network device driver.

(* 1) µNet3 manages control of all network device drivers by gNET_DEV [] (device object). gNET_DEV [] is an array of T_NET_DEV type in the order of device numbers, and is defined in the net_cfg.c file.

- Device initialization function: ER eth_ini (UH dev_num)

  When the application calls the device initialization function net_devini (device number) API, µNet3 calls the network device driver initialization function corresponding to the device number.

  Set it to the member "ini" of gNET_DEV [] that corresponds to the device number.

- Frame transmission function: ER eth_snd (UH dev_num, T_NET_BUF * pkt)

  When the application calls send socket snd_soc () API, or when µNet3 sends a packet, µNet3 calls the send function of the network device driver corresponding to the device number.

  Set it to the member "out" of gNET_DEV [] that corresponds to the device number.

- Device release function: ER eth_cls (UH dev_num)

  When the application calls the device release function net_dev_cls (device number) API, µNet3 calls the nice driver release function on the network corresponding to the device number.

  Set it to the member "cls" of gNET_DEV [] that corresponds to the device number.

- Device control function: ER eth_ctl (UH dev_num, UH opt, VP val)

  When the application calls the device control function net_dev_ctl (device number, control code, setting value) API, µNet3 calls the control function of the network device driver corresponding to the device number.

  Set to the member "ctl" of gNET_DEV [] that corresponds to the device number.

- Device state acquisition function: ER eth_ref (UH dev_num, UH opt, VP val)

  When the application calls the device status acquisition function net_dev_sts (device number, control code, acquired value) API, µNet3 calls the network device driver status acquisition function corresponding to the device number.

  Please set it to the member "ref" of gNET_DEV [] that corresponds to the device number.

- Frame reception processing: net_pkt_rcv ()

  Call the net_pkt_rcv () API with the following procedure to transfer frames received from a network device to µNet3. The network buffer used is released by µNet3.

  (1) Get network buffer (T_NET_BUF *) using net_buf_get () API.
  (2) Set its own gNET_DEV [] in the network buffer.
  (3) Write the received frame to the network buffer.
  (4) Set the start address (header address) of the written frame in the network buffer.
  (5) Set the data address of the written frame in the network buffer.
  (6) Set the header size (Ether header size) of the written frame in the network buffer.
  (7) Set the data size (frame size-header size) of the written frame in the network buffer.
  (8) Call net_pkt_rcv () with the network buffer as an argument.
  (9) Turn on the power of the RZ / N1 board.
  (10) Download the program using the [Project] → [Download and Debug] menu in EWARM or the [Download and Debug] button in the following figure

6.4 µNet3/BSD

µNet3 / BSD provides a BSD interface to run BSD applications on µNet3. By using µNet3 / BSD, socket applications running on Linux and BSD will be able to work seamlessly on µNet3.

µNet3 / BSD provides a socket API equivalent to 4.4BSD-Lite.

By using µNet3 / BSD, the application can use BSD socket API and µNet3 proprietary API

- Socket API multiple calls
- select () function
- Loopback address
- Per socket multicast group
- TCP socket Listen queue
- Socket error

6.4.1 Symbol Name Compatibility

The API, structures and macros provided by µNet3 / BSD are prefixed with the unique prefix "unet3_" to avoid symbol collisions caused by the compiler environment.

The application includes /bsd/unet3_posix/unet3_socket.h to replace the POSIX standard symbol names used in the application with these prefixed symbols. Therefore, applications using BSD sockets will run under µNet3 / BSD with the same source files.

6.4.2 Socket API

Table 6.2 shows the API provided by the TCP / IP protocol stack "µNet3 / BSD" manufactured by eforce.

Table 6.2 Socket API List

µNet3 API	µNet3/BSD API	Function
unet3_bsd_init		Initialize µNet3 / BSD
unet3_socket	socket	Create a new socket
unet3_bind	bind	Name the socket
unet3_listen	listen	Wait for a connection on the socket
unet3_accept	accept	Receive connection to socket
unet3_connect	connect	Connect socket
unet3_send	send	Send a message to the socket
unet3_sendto	sendto	Send a message to the socket
unet3_recv	recv	Receive a message from a socket
unet3_recvfrom	recvfrom	Receive a message from a socket
unet3_shutdown	shutdown	Close part of a full duplex connection
unet3_close	close	Close the socket
unet3_getsockname	getsockname	Get the socket name
unet3_getpeername	getpeername	Get name of connected socket
unet3_getsockopt	getsockopt	Get socket options
unet3_setsockopt	setsockopt	Configure socket options
unet3_inet_aton	inet_aton	Internet address manipulation routine
unet3_inet_addr	inet_addr	Internet address manipulation routine
unet3_inet_ntoa	inet_ntoa	Internet address manipulation routine
unet3_rresvport	rresvport	Get a socket bound to a port
unet3_getifaddrs	getifaddrs	Get address of interface
unet3_freeifaddrs	freeifaddrs	Release interface information

6.4.3 BSD application settings

(1) Source code

Import the four source code directly under uNet3 / into the project.

- unet3_lodev.c Virtual loopback device
- unet3_option.c socket option functions
- unet3_socket.c BSD socket API
- unet3_wrap.c µNet3 / Wrapper task

  Also, the µNet3 main library links the BSD library.
- uNet3BSDRXv3l.lib µNet3 library for BSD

(2) Include path

Add the include path settings.

Header files are included under uNet3 / bsd / unet3_posix / folder including POSIX compliant files.

(3) configuration

The macro is defined in unet3_cfg.h with the maximum number of sockets and the number of application tasks to be used by the application.

- Maximum number of sockets

  #define BSD_SOCKET_MAX CFG_NET_SOC_MAX
- Number of tasks for application

  #define NUM_OF_TASK_ERRNO (CFG_TASK_MAX + 1)

(4) Resource definition

Prepare the resources needed to run µNet3 / BSD. Resources are tables for µNet3 / BSD to manage information.

- BSD socket management table

  T_UNET3_BSD_SOC gNET_BSD_SOC [BSD_SOCKET_MAX];
- Error number management table

  UW tsk_errno [NUM_OF_TASK_ERRNO];

(5) Kernel object

Kernel objects used by µNet3 / BSD are shown in Table 6.3.

Table 6.3 Kernel object list

Resource name	Usage	ID
Task	BSD Wrapper task	ID TSK_BSD_API
	Loopback device task	ID_LO_IF_TSK
MailBox	BSD Wrapper intertask communication	ID MBX_BSD_REQ
	Loopback device intertask communication	ID_LO_IF_MBX
MemoryPool	Message buffer	ID MPF_BSD_MSG

(6) Initialize

The application must call the unet3_bsd_init () function to initialize the µNet3 / BSD module before using the socket API. Note that to initialize µNet3 / BSD, µNet3 initialization and device driver initialization must have been completed successfully.

7. Basic concept of Modbus stack for RX72M

7.1 Design method

(1) A set of software stacked in a hierarchy is called a protocol stack, selecting the functions required to realize the functions that are on the network. This stack is structured as shown in Figure 7.1.

(2) This stack creates tasks using RTOS "µC3" manufactured by e-force. The stack works in multitasking using RTOS.

(3) This stack can not use multiple timer channels for Modbus frame timing.

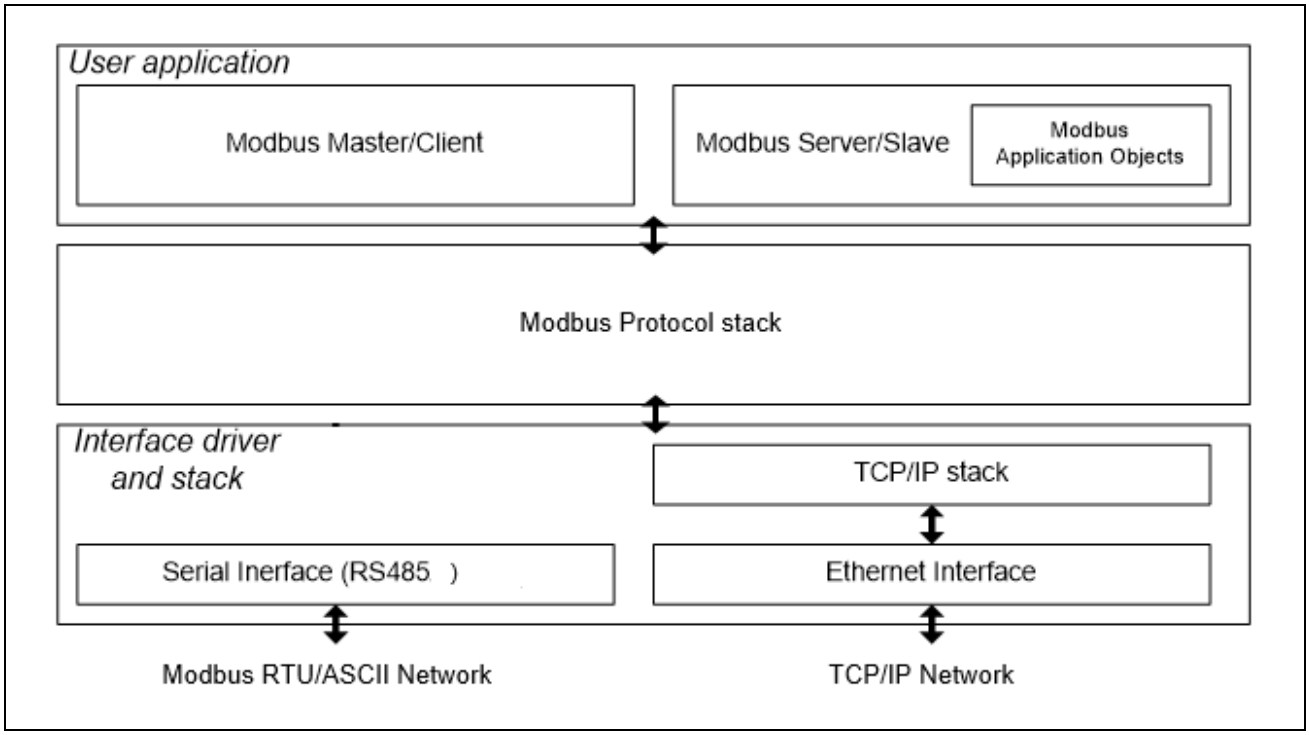

Figure 7.1 Outline of Modbus stack for RX72M

7.2 Communication format

The Modbus TCP communication format includes the Modbus RTU excluding the CRC. Modbus RTU needs to add a CRC check code at the end for error check, but Modbus TCP is not necessary because it uses the check mechanism of the TCP / IP protocol.

In Modbus TCP communication format, it is necessary to add transaction identifier, protocol identifier, message length and unit identifier, which did not exist in Modbus RTU.

The difference between Modbus RTU format and Modbus TCP / IP format is shown in Figure 7.2.

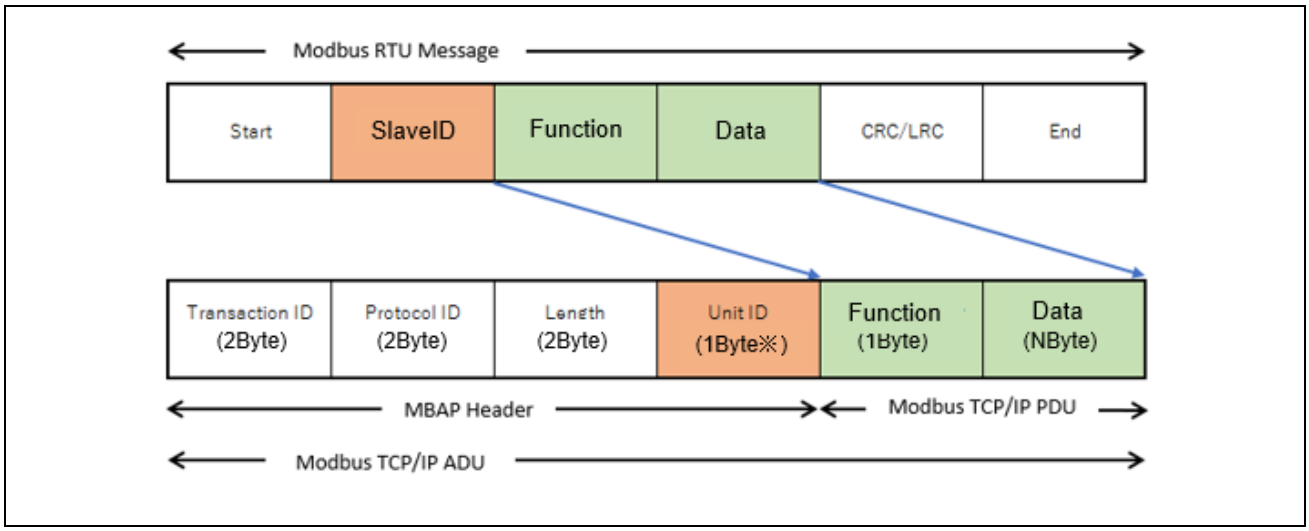

Figure 7.2 Difference between Modbus RTU format and Modbus TCP / IP format

ADU：Application Data Unit

MBAP Header：Modbus Application Header

PDU：Protocol Data Unit

8. System configuration-Modbus TCP protocol stack

This section describes the details of the Modbus TCP and Modbus TCP serial gateway stack.

Modbus TCP Serial Gateway mode uses the Modbus RTU / ASCII master stack as a gateway to the serial network. The initialization of the Modbus RTU / ASCII master stack takes place within the initialization of the gateway stack. Users can select either Modbus RTU or Modbus ASCII Gateway Stack.

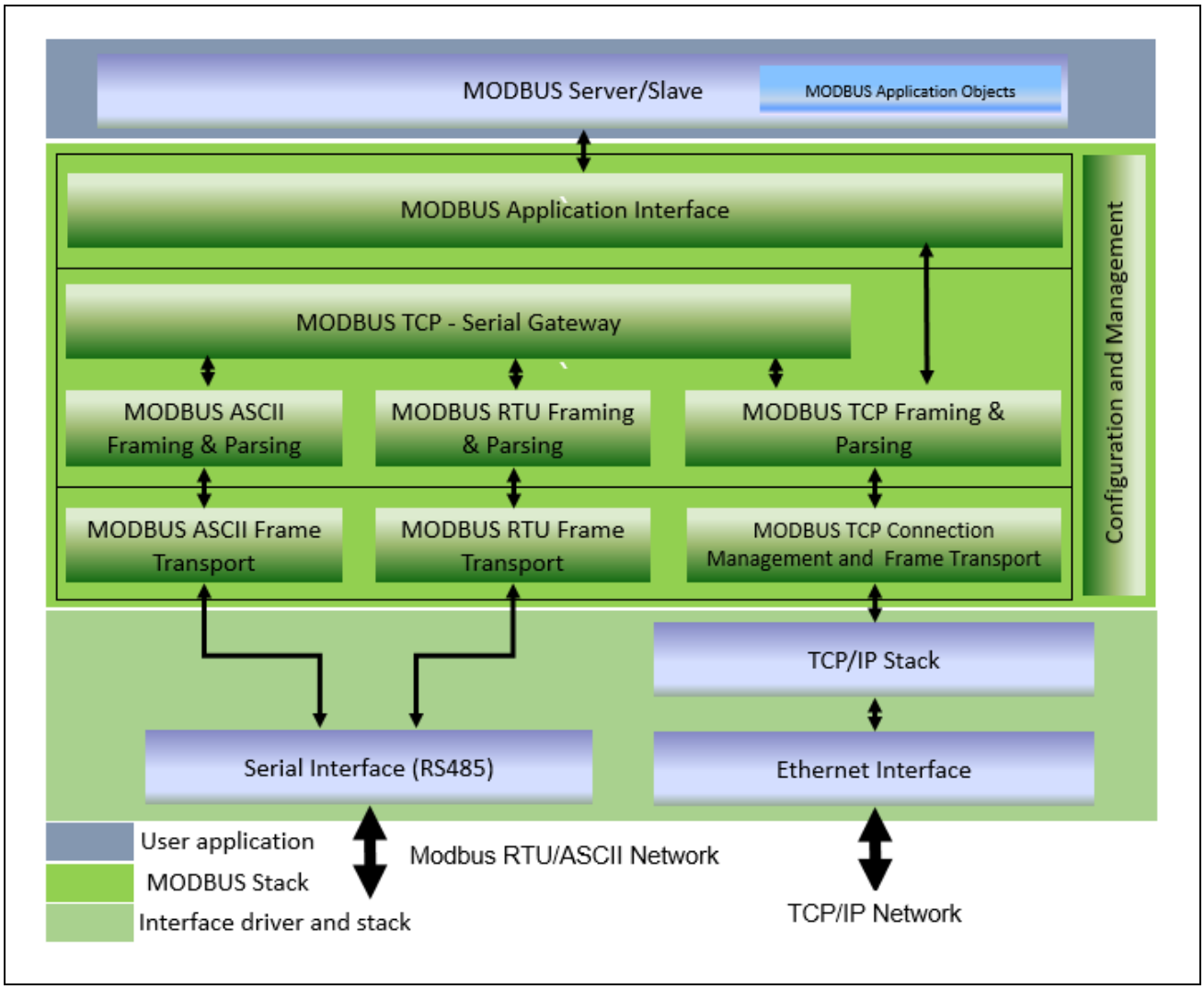

Figure 8.1 Modbus TCP stack software configuration

8.1 Module configuration

The TCP server and gateway stacks are divided into several layers based on functionality.

At the top layer, the application interface layer consists of API functions that map two tasks and callback functions to each function code.

The middle layer packet construction and analysis layer consists of functions and queues for packet construction / analysis and mailbox operation. All of these functions are performed by upper layer tasks.

The lowest layer connection management and packet transmission / reception layer consists of functions and tasks that connect and transmit TCP communication. All functions except for the function to send response TCP messages are executed by the task.

The configuration layer is a layer that is called up from the top layer to the bottom layer, and includes the necessary functions along with the configuration API.

8.1.1 Application interface layer

This layer consists of two tasks and multiple functions, and tasks and functions operate based on the specified mode. The gateway task does not work if the stack is in TCP server mode. It is activated only when the gateway function works. However, even with the gateway function only, the TCP server task works.

Processing in TCP server mode

(1) The Modbus initialization function (modbus_init ()) is called from the main task (main_task) of the sample program.

(2) The Modbus initialization function starts the following tasks required for Modbus TCP communication.

- Connection waiting task (Modbus_tcp_soc_wait_task)
- Packet Packet reception task (Modbus_tcp_recv_data_task)
- Modbus function processing task (Modbus_tcp_req_process_task)

Processing in TCP serial gateway mode

(1) The Modbus initialization function (modbus_init ()) is called from the main task (main_task) of the sample program.

(2) The Modbus initialization function starts the following tasks required for Modbus TCP communication.

- Connection waiting task (Modbus_tcp_soc_wait_task)
- Packet reception task (Modbus_tcp_recv_data_task)
- Modbus function processing task (Modbus_tcp_req_process_task)
- Serial gateway task (Modbus_gateway_task)
- Serial task (Modbus_serial_task)
- Serial data reception task (Serial_recv_task)
- Serial packet reception task (Modbus_serial_recv_task)

8.1.1.1 Modbus TCP server task

This task runs when the stack is configured as a Modbus TCP server. When the received Modbus request is sent to the TCP Receive Data task, it waits to get data from the mailbox. When a packet arrives in the mailbox, it copies and processes the packet.

Operation differs depending on the presence or absence of the gateway function**.**

Without gateway function, packets with slave ID other than 0xFF will be discarded and only packets with slave ID 0xFF will be processed. On the other hand, with the gateway function, when the slave ID receives a request other than 0xFF, the request packet is forwarded to the TCP-serial gateway task.

The task state transition diagrams with and without the gateway function are shown respectively.

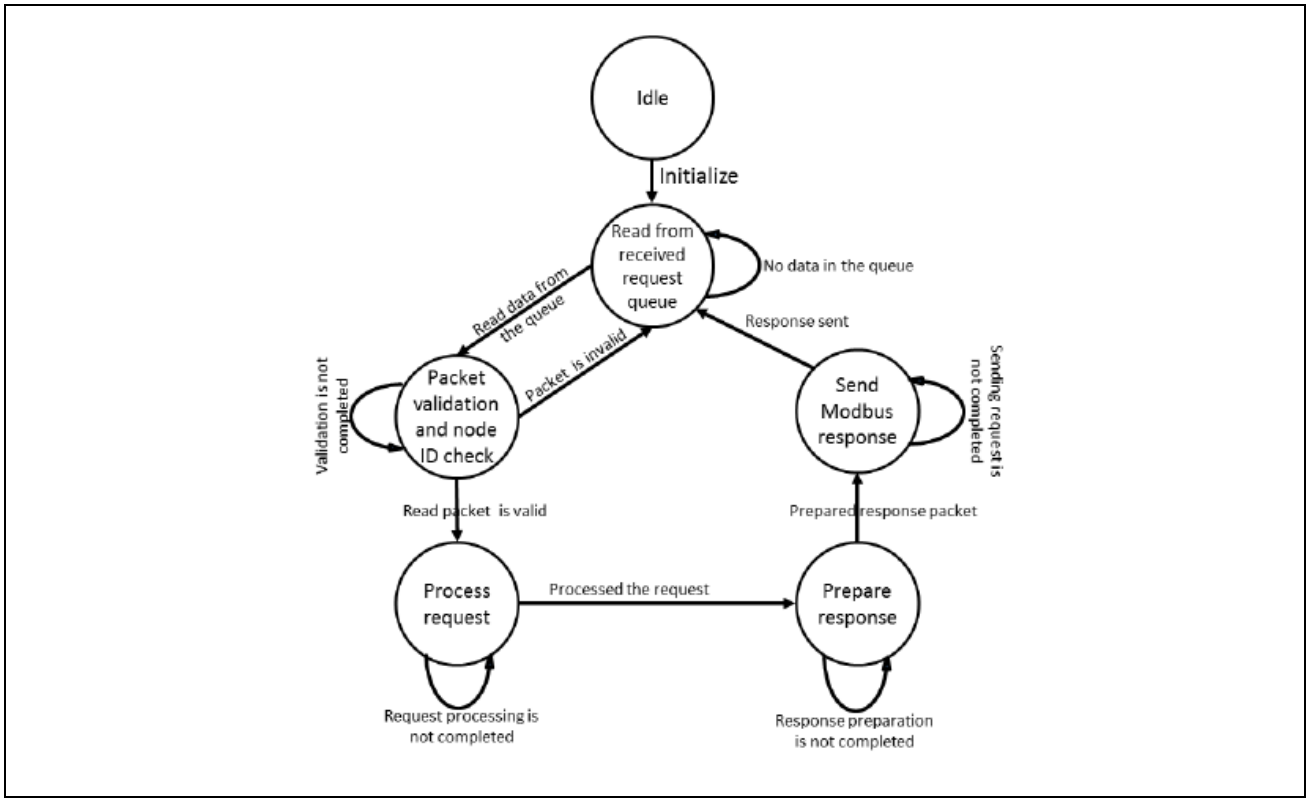

Figure 8.2 Modbus TCP server task (without gateway function)

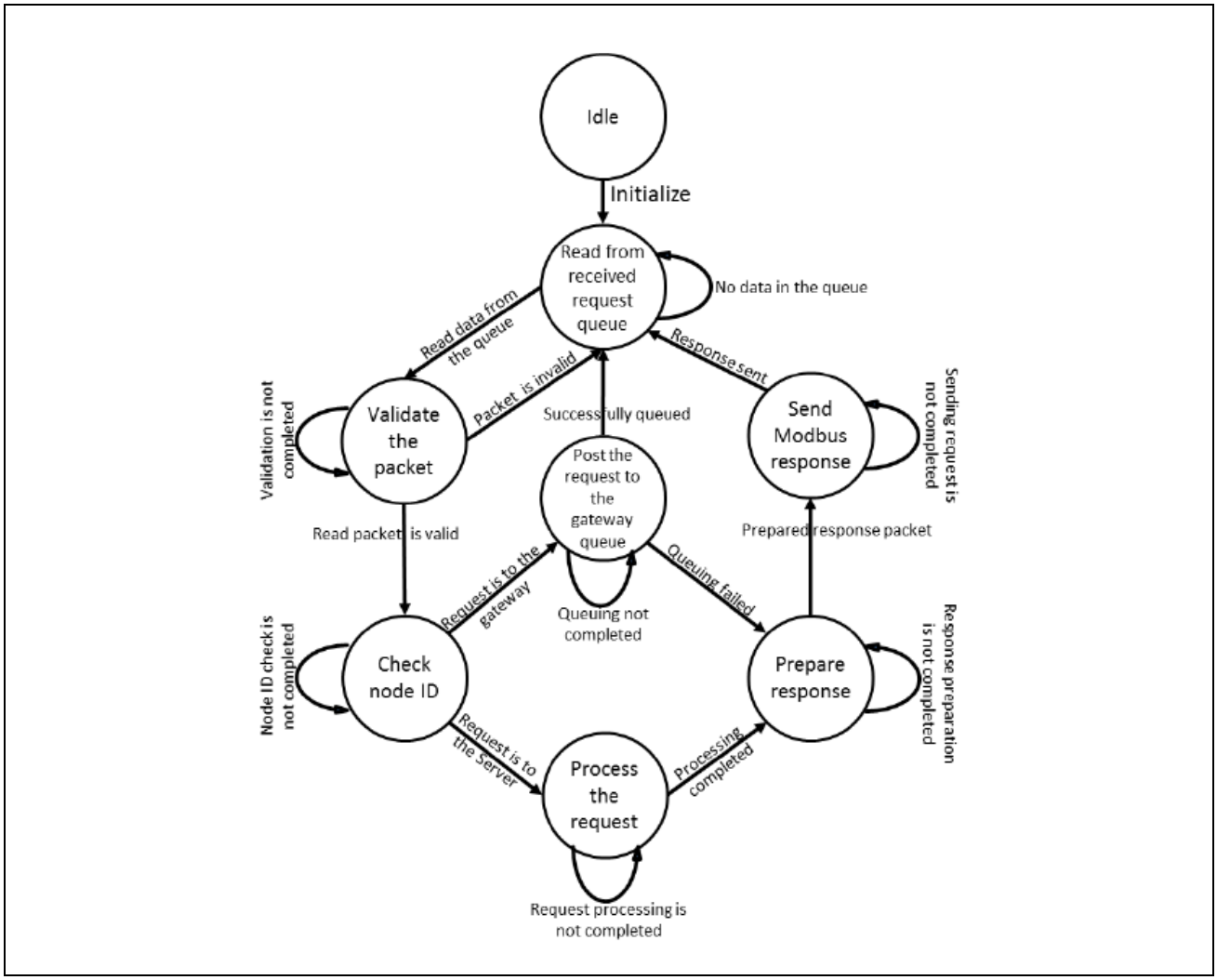

Figure 8.3 Modbus TCP server task (with gateway function)

8.1.1.2 Modbus TCP-Serial Gateway Task

This task uses the Modbus RTU / ASCII master stack function to respond to communication via the serial interface.

- Receive requests from Modbus TCP clients.
- When the slave ID receives a request other than 0xFF, it sends the request to the RTU / ASCII master stack.
- Receives a response from the RTU / ASCII master stack and returns the response to the Modbus TCP client.

Figure 8.4 shows the task state transition diagram.

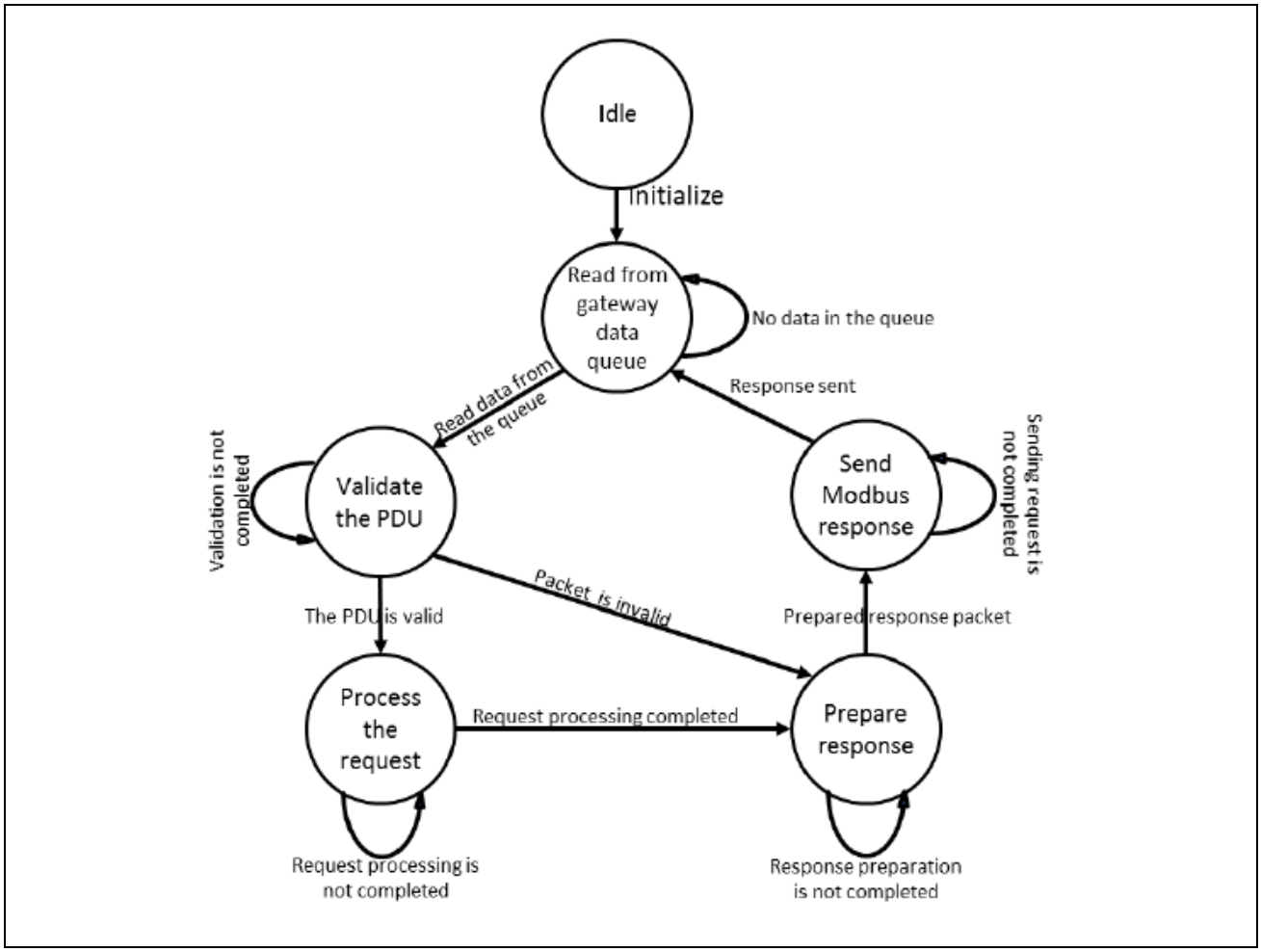

Figure 8.4 Modbus TCP-Serial Gateway Task (Modbus_gateway_task)

8.1.1.3 Error judgment and report

- Allocate dynamic memory for packet construction. An error is reported if memory can not be allocated.
- The gateway task queues messages up to MAX_GW_MBX_SIZE. If the gateway task can not queue, the TCP server returns an exception code (06) as a response packet to the request packet.

8.1.2 Packet construction and analysis layer

This layer is a layer that implements Modbus packet construction and analysis. It consists of functions and data structures that analyze and construct Modbus packets.

- Received packet analysis (Modbus_tcp_parse_pkt)
- Received packet validation (Modbus_tcp_validate_pkt)
- Transmission packet construction (Modbus_tcp_frame_pkt)

8.1.2.1 Analysis of received packet

If there is an abnormality in the packet length and integrity of the specified value, etc., the received packet is discarded and an error is reported. If the received packet is normal, call the callback function registered by the user to process the request.

8.1.2.2 Construct transmit packet

The response packet is constructed and sent back based on the result of execution in each callback function. Also, if an unsupported function code is specified, such as when an exception code needs to be returned, a response packet is constructed and sent.

8.1.2.3 Error judgment and reporting

- Based on the function code, it verifies that the packet length in the received packet and the specified data length conform to the protocol, and reports an error if there is an error.
- Receive packet analysis dynamically allocates memory. An error is reported if memory can not be allocated.

8.1.3 Communication connection management and packet transmission / reception layer

This layer consists of tasks and functions that accept connections from clients and send and receive packets.

8.1.3.1 Modbus TCP Connection Acceptance Task

This task is initialized when the stack is initialized by the user, and starts waiting for a connection request from the client for port 502 or the port set by the user (only when specified by the user at stack initialization).

When a task receives a connection request, it checks its IP list against it to see if it is connected or acceptable. After accepting the connection, register the IP in the connected list.

The maximum number of connections is limited by the setting value of the "MAXIMUM_NUMBER_OF_CLIENTS" macro defined in "modbusTcpConfig.h" in the r_modbus_rx folder.

Figure 8.5 shows this task state transition diagram.

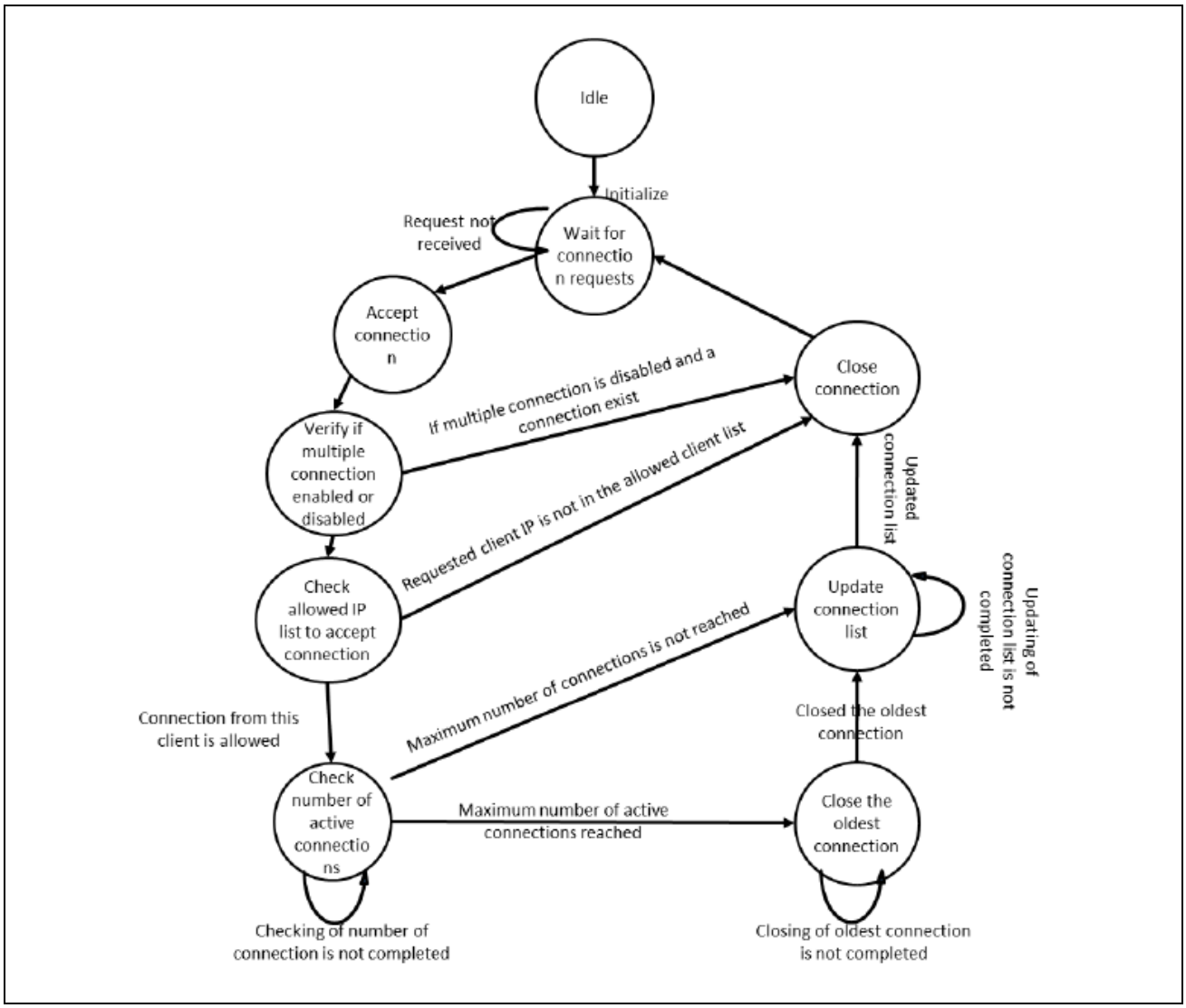

Figure 8.5 Modbus TCP connection acceptance task (Modbus_tcp_soc_wait_task)

8.1.3.2 Modbus TCP Receive Data Task

This task is initialized when the stack is initialized. The task waits for data from the connected client, and when it receives a valid packet, it forwards it to the mailbox.

When a request is received from the client, Modbus_post_to_mailbox () is called to send the request to the mailbox. Mail messages are read by the TCP server task using Modbus_fetch_from_mailbox ().

Figure 8.6 shows this task state transition diagram.

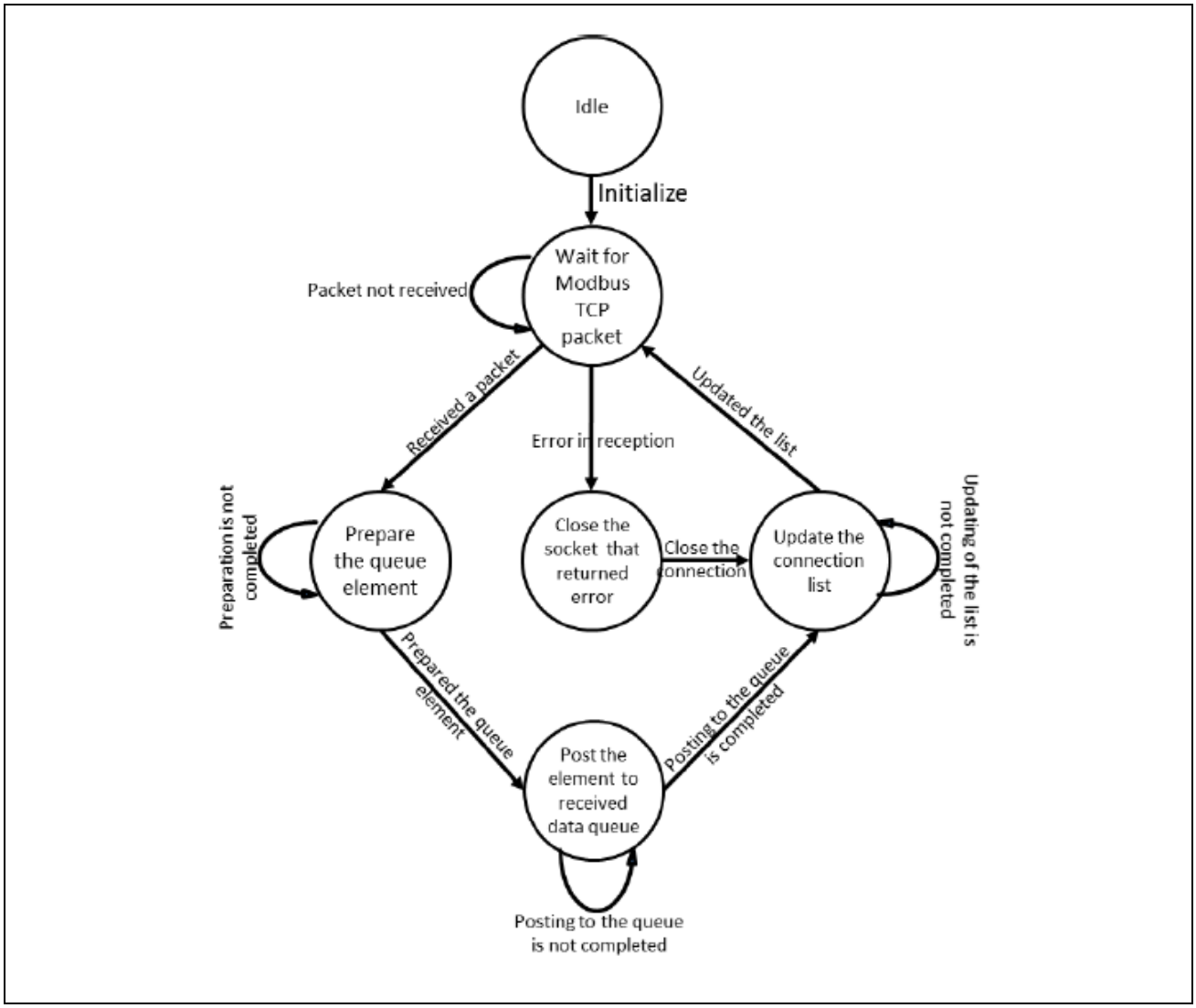

Figure 8.6 Modbus TCP receive data task (Modbus_tcp_recv_data_task)

8.1.3.3 Error judgment and report

- Allocate dynamic memory for packet analysis. An error is reported if memory can not be allocated.
- The TCP server task queues messages up to MAX_RCV_MBX_SIZE. If the TCP server task can not queue, the TCP receive data task returns an exception code (06) as a response packet to the request packet.

9. System Configuration-Modbus RTU / ASCII Protocol Stack

This section describes the details of the Modbus RTU / ASCII stack.

This stack can be used as a master or server / slave application by setting the necessary configuration. The stack can be configured to support either Modbus RTU or Modbus ASCII. Mode selection is performed when the initialization API is called on the user application.

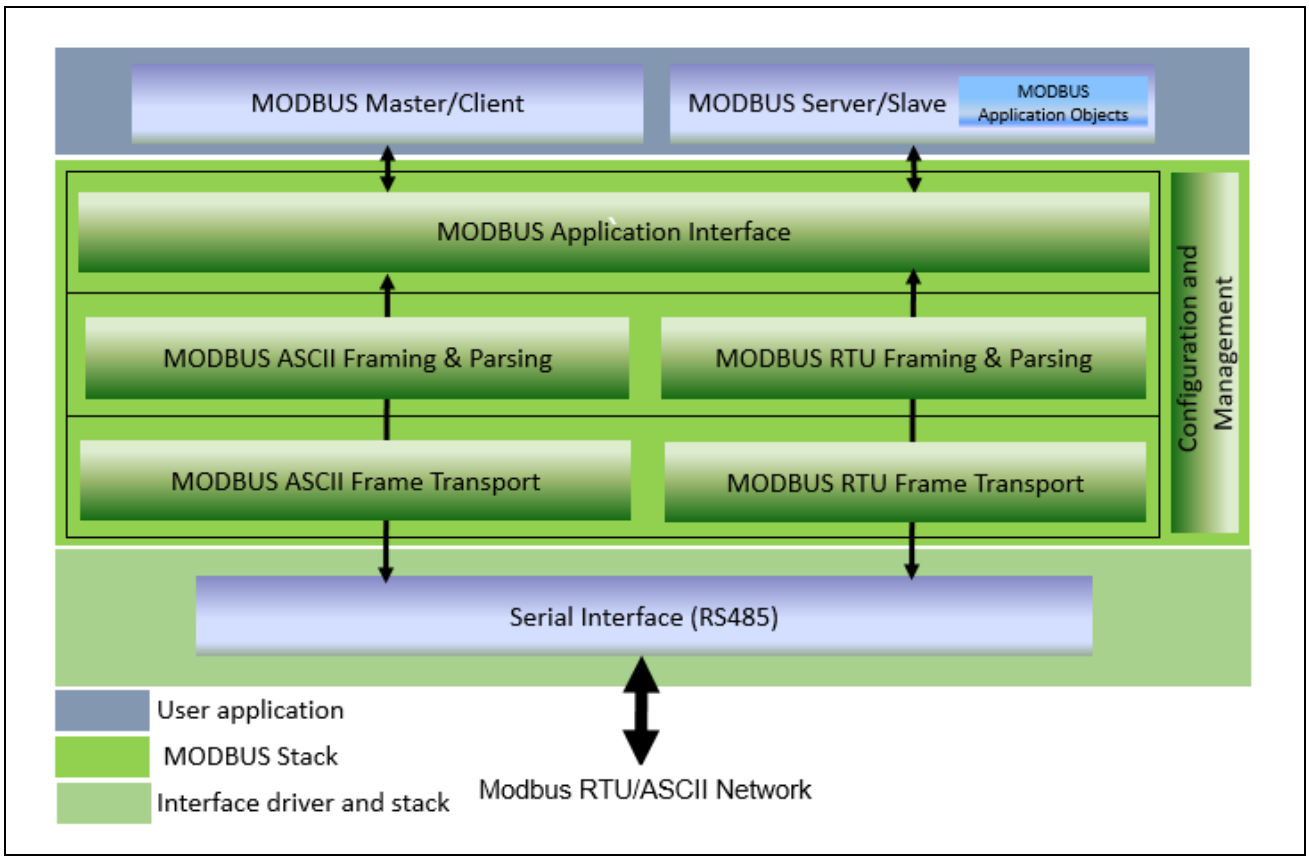

Figure 9.1 Modbus stack software configuration

The following sections describe each functional layer.

9.1 Module configuration

The stack is divided into the following layers based on functionality.

The application interface layer, the top layer, interacts directly with the user application in both master and slave modes.

The packet construction and analysis layer, which is the middle layer, is responsible for Modbus frame construction, analysis and verification.

The lowest level communication connection management and frame transmission / reception layer manages transmission / reception of Modbus frame with communication logical connection.

There is a configuration layer including configuration API etc. in the form of straddling three layers from the top layer to the bottom layer. The details of each layer are shown below.

9.1.1 Application interface layer

The application interface layer contains the functionality needed to interact with the user application. It also contains threads that maintain the state of the stack. Based on the configured master or slave stack mode, the thread provides the user with the ability to operate in stack mode. This layer has the same function even if the communication mode is RTU or ASCII.

The main components of the application interface layer in Modbus server / slave mode are the serial task function and Modbus_slave_map_init (). Using Modbus_slave_map_init (), the user application registers the callback function corresponding to the specific function code to be called when a valid Modbus request is received.

Analysis of request message and construction of response message are done by serial task function. The serial task function calls the appropriate callback handler when it receives a valid Modbus request. It also sends the response message from the callback handler back to the requesting master.

> note **If use a user-specified callback handler, be aware of the execution time. If an error occurs in the handler and processing is delayed, the stack can not process the next command during that time.**

The main component of the application interface layer in Modbus master mode is the various APIs that interface the serial task function with the user application. When the stack is initialized by the user, the serial task function is activated, and Modbus communication is performed by calling various APIs from the user application.

The user application calls the user application interface API corresponding to each function code to send Modbus requests from the stack to the Modbus slave device. The serial task function sends and receives requests.

The User Application Interface API can be called in blocking mode or nonblocking mode. If a callback function is provided by the user as an argument to the API, it will be called in non-blocking mode, and the serial task function will call the callback function provided upon receipt of the response or cause a timeout. If no callback function is provided, these APIs will be blocked until a response from the slave is received or a timeout occurs.

(1) **Modbus serial task function**

The Modbus serial task function is used in both Modbus master and slave stacks, regardless of the mode (RTU/ASCII).

For example, if the stack mode is defined as a Modbus RTU master, the Modbus serial task function will function as a Modbus master function. Switching between master and slave is determined when the mode is specified in the initialization API.

Figure 9.2 shows the state transition diagram when the Modbus serial task function is in slave mode. The main_task (void) function calls the Modbus serial task function and checks that a message is sent. It acts as a master or slave, depending on the type of message received. When acting as a slave, it retains its state until it receives a Modbus request from the Modbus master.

When receiving a request from the client, the function does the following:

- Analyzes and verifies the received request packet.
- If the received packet is normal, it constructs a response packet and sends it to the master device.

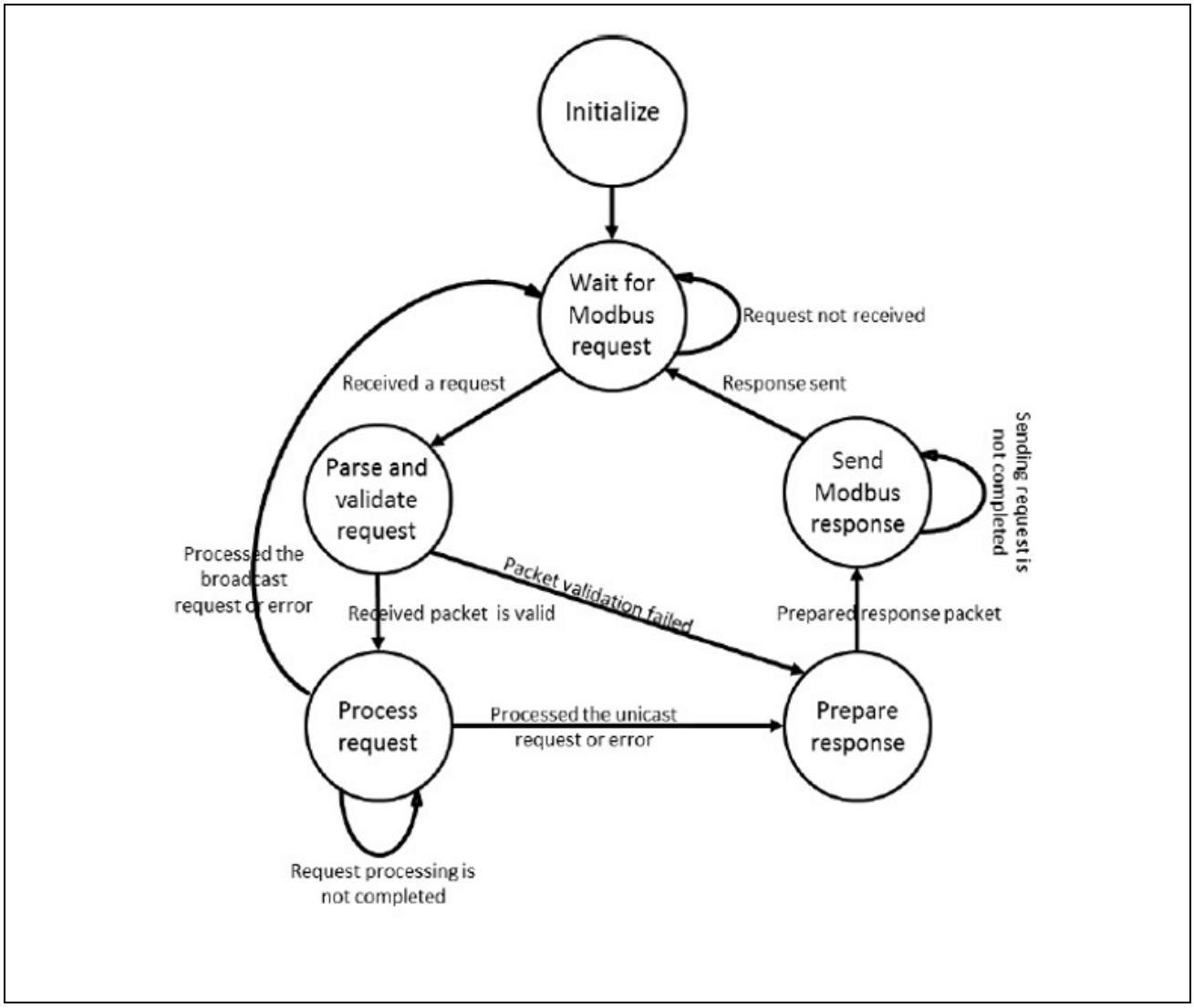

Figure 9.2 State transition diagram of Modbus serial task function in slave mode

Figure 9.3 shows the state transition diagram when the serial task function is in master mode.

In slave mode, the stack is initialized in Modbus_serial_stack_init () in master mode. The user application calls the API to request the serial task function to perform Modbus communication with the slave device. If you receive a request, do the following.

- Prepare Modbus request packet and send it to slave device.
- If the transmission request is a broadcast request, it waits for the response from the slave device to return until the "turn around delay time".
- If the send request is a unicast request, wait for a response from the slave device until "response timeout time".
- If a valid response is received from the slave device before the response timeout, the response table is updated from the received data and handed over to the user application.
- If it can not receive a response within the response timeout time, it sends the same request up to the maximum number of retries.
- If a response to the retry can not be received, a timeout error is passed to the callback handler.
-

The user application can provide a callback handler with the API function call if notification is required when the processing of the command request is complete. Also, API function calls operate in nonblocking mode, and other functions can operate until the request is completed. If no callback function is provided, the API function call operates in blocking mode.

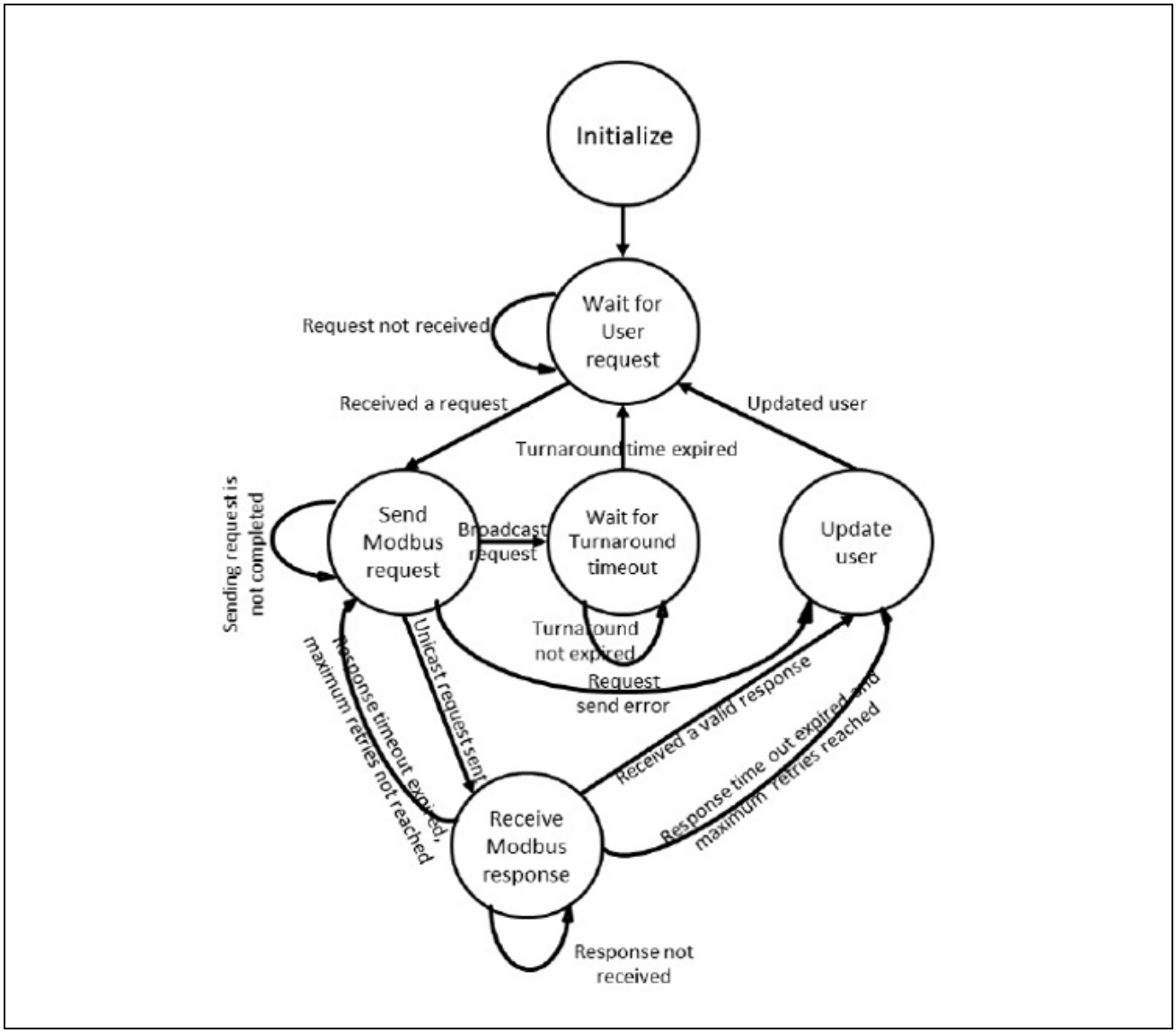

Figure 9.3 State transition diagram of Modbus serial task function in master mode

(2) **Error judgment and reporting**

(a) Modbus RTU / ASCII slave

- In the case of a unicast request, the serial task function constructs an exception response and sends it back to the master device when it receives a request for a function code for which a callback function has not been registered by the user.
- The callback function written by the user needs to generate an exception response to requests for unimplemented registers or coils.
- The initialization API performs basic checks on specified parameters and returns the status.
- In the case of a broadcast request, no response is returned to the client.

(b) Modbus RTU / ASCII master

- The initialization API performs basic checks on specified parameters.
- If the response to the request can not be received even after the specified number of retries, a timeout error is returned.

9.1.2 Packet construction and analysis layer

This layer is a layer that implements Modbus packet construction and analysis. It consists of functions and data structures that analyze and construct Modbus packets. These functions are processed according to the set stack mode, but since Modbus packets are internally processed in RTU format, they are also converted to RTU format and processed in ASCII mode.

- Received packet analysis (Modbus_master_parse_pkt, Modbus_slave_parse_pkt)
- Received packet validation (Modbus_master_validate_pkt, Modbus_slave_validate_pkt)
- Transmission packet construction (Modbus_master_frame_request, Modbus_slave_frame_response)

9.1.2.1 Analysis of received packet

- In ASCII mode, packet conversion from ASCII to RTU is performed.
- Receive packet validation is discarded if length check, packet integrity, slave ID mismatch, etc. occur.
- If the received packet is normal, call the callback function registered by the user to process the request.
- When a unicast request is received, if there is an error in the processing of the function code, an exception response message is created and sent to the master device.
- When a broadcast request is received, if the received packet is a light function code, it is accepted as a normal packet, but a response message is not sent to the master device. If the received packet is a read function code, the request packet is discarded as a slave ID error.

9.1.2.2 Construct transmit packet

In master mode, request packets are constructed, and in slave mode, response packets are constructed based on the contents generated by the API. After that, add CRC / LRC to the constructed packet. In ASCII mode, RTU → ASCII conversion is performed because internal processing is performed in RTU format.

9.1.2.3 Error judgment and reporting

- Based on the function code, it verifies that the packet length in the receive and transmit packets, specified data and slave ID conform to the protocol.
- Verify the integrity of the packet using the CRC / LRC embedded in the received packet.
- Allocate dynamic memory for packet analysis. An error is reported if memory can not be allocated.

9.1.3 Communication connection management and frame transmission / reception layer

This layer contains functions and data structures for sending and receiving data through the communication interface. The frame timing of serial communication is managed in this layer.

9.1.3.1 Serial receive task function

Serial I / F and timer interrupts are registered by Modbus RTU / ASCII stack initialization processing. The serial reception task function is called when a serial I / F and timer interrupt occurs.

The serial reception task function reads reception data with the serial I / F driver function when a serial I / F interrupt occurs. After successfully receiving data, Modbus_ascii_recv_char () or Modbus_rtu_recv_char () is called according to the stack mode. The data is stored in a buffer with these functions.

When a timer interrupt occurs, Modbus_timer_handler () is called. The frame timing is determined by this function.

9.1.3.2 Modbus serial interface configuration

- The serial interface is used to send and receive packets according to the configuration parameters established during stack initialization.
- If an error occurs during receive operation, a status interrupt event is generated. For details on reception errors, refer to the "RX72M Communication Board Hardware Manual".
- The timer is used to measure the idle time.
- The RS485 mode switching is done using the GPIO pin.

9.1.3.3 Error identification and reporting

If an error occurs during reception, discard the received packet and continue processing.

10. Description of application programming interface

This chapter describes the application programming interface (API) specifications in detail.

10.1 User interface API

This chapter describes the APIs used by user applications.

10.1.1 Modbus TCP/IP

10.1.1.1 Initialization of protocol stack

The following API is used in initialization of protocol stack.

Modbus_tcp_init_stack	Modbus TCP stack initialization API

【Format】

```
uint32_t Modbus_tcp_init_stack(uint8_t u8_stack_mode,
                               uint8_t u8_tcp_gw_slave,
                               uint8_t u8_tcp_multiple_client,
                               uint32_t u32_additional_port,
                               p_serial_stack_init_info_t pt_serial_stack_init_info,
                               p_serial_gpio_cfg_t pt_serial_gpio_cfg_t);
```

【Parameter】

uint8_t	u8_stack_mode	Variable to store the stack mode
uint8_t	u8_tcp_gw_slave	Status whether gateway enabled as TCP server
uint8_t	u8_tcp_multiple_client	Status whether multiple client is enabled
uint32_t	u32_additional_port	Additional port configured by user
p_serial_stack_init_info_t	pt_serial_stack_init_info	Structure pointer to serial stack initialization parameters
p_serial_gpio_cfg_t	pt_serial_gpio_cfg_t	Pointer to the structure with hardware configuration parameters.

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful initialization
ERR_STACK_INIT	On failure

【Explanation】

This API initialize Modbus stack based on the user provided information. If the serial stack information structure is NULL, Modbus_tcp_server_init_stack() is invoked. If the serial stack information structure is provided by user, Modbus_tcp_init_gateway_stack() is invoked with the required serial parameter configuration.

For this few initializing parameters are provided in the APIs.

a. u8_stack_mode of type uint8_t is an argument in order to select the Modbus TCP stack type. The user specifies the following macro in this parameter. If user wants to use the gateway mode, please specify the mode to be used to communicate with the serial device.

Stack mode parameter code	Meaning
MODBUS_RTU_MASTER_MODE	Used to select Modbus Stack RTU master mode
MODBUS_RTU_SLAVE_MODE	Used to select Modbus Stack RTU slave mode
MODBUS_ASCII_MASTER_MODE	Used to select Modbus Stack ASCII master mode
MODBUS_ASCII_SLAVE_MODE	Used to select Modbus Stack ASCII slave mode
MODBUS_TCP_SERVER_MODE	Used to select Modbus Stack TCP server mode

b. u8_tcp_gw_slave of type uint8_t is an argument in order to select the Modbus gateway mode type. The user specifies the following macro in this parameter.

Stack gateway parameter code	Meaning
MODBUS_TCP_GW_SLAVE_DISABLE	Modbus stack gateway slave disable
MODBUS_TCP_GW_SLAVE_ENABLE	Modbus stack gateway slave enable

c. u8_tcp_multiple_client of type uint8_t is an argument in order to select whether accept communication from multiple clients. The user specifies the following macro in this parameter.

Multiple client connection parameter code	Meaning
DISABLE_MULTIPLE_CLIENT_CONNECTION	By setting this value, multiple client connection is disabled
ENABLE_MULTIPLE_CLIENT_CONNECTION	By setting this value, multiple client connection is enabled

d. Additional port (other Modbus default port 502) provided by user for MODBUS communication can also be used. If user does not want to add the port, please specify 0.

e. Structure of type p_serial_stack_init_info_t is an argument in order to provide information specific to serial communication. If want to use in TCP server mode, please specify NULL to this argument.

・Structure of serial stack initialization parameters (serial_stack_init_info_t)

```
typedef struct _stack_init_info{
    uint32_t    u32_baud_rate;          /* Baud rate for serial port configuration */
    uint8_t     u8_parity;              /* Parity for serial port configuration */
    uint8_t     u8_stop_bit;            /* Stop bit for serial port configuration */
    uint8_t     u8_uart_channel;        /* The hardware UART channel to be used by the Modbus serial
                                           stack */
    uint8_t     u8_timer_channel;       /* The hardware timer channel to be used by the Modbus serial
                                           stack */
    uint32_t    u32_response_timeout_ms; /* Response shall be received within this time out */
    uint32_t    u32_turnaround_delay_ms; /* Delay in between consecutive requests in broadcast mode */
    uint32_t    u32_interframe_timeout_us; /* Inter frame delay for the RTU packet */
    uint32_t    u32_interchar_timeout_us; /* Inter char delay for the ASCII packet */
    uint8_t     u8_retry_count;         /* Number of retries to be done in case of an error */
}serial_stack_init_info_t, *p_serial_stack_init_info_t;
```

Use the following macro to the parameters of the structure.

Boud rate parameter code	Meaning
MODBUS_BAUDRATE	Specify any baud rate from 9600 bps, 19200 bps, 115200 bps

Parity parameter code	Meaning
PARITY_NONE	No parity
PARITY_ODD	Odd parity
PARITY_EVEN	Even parity

Stop bit parameter code	Meaning
UART_STOPBIT_1	One stop bit
UART_STOPBIT_2	Two stop bit

f. Structure of type p_serial_gpio_cfg_t is an argument in order to provide function pointers to control the GPIO port for RS485 communication. If want to use in TCP server mode, please specify NULL to this argument.

・Structure of I/O port configuration information (serial_gpio_cfg_t)

```
typedef struct _serial_gpio_cfg_t{
    fp_gpio_callback_t    fp_gpio_init_ptr;    /* Callback function pointer to invoke the initialize the GPIO used
                                                  for RS485 direction control */
    fp_gpio_callback_t    fp_gpio_set_ptr;     /* Callback function pointer to set the GPIO used for RS485
                                                  direction control */
    fp_gpio_callback_t    fp_gpio_reset_ptr;   /* Callback function pointer to reset the GPIO used for RS485
                                                  direction control */
}serial_gpio_cfg_t, *p_serial_gpio_cfg_t;
```

Modbus_slave_map_init	Modbus function code mapping API

【Format】

uint32_t Modbus_slave_map_init(p_slave_map_init_t p_serial_slave_map_init_t);

【Parameter】

p_slave_map_init_t	p_serial_slave_map_init_t	structure pointer to function code mapping table

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On success
ERR_INVALID_STACK_INIT_PARAMS	If parameter is null
ERR_MEM_ALLOC	If memory allocation failed

【Explanation】

This API does the mapping of user defined functions for processing requests from clients depending on function code. When the Modbus Slave stack receives a request, it invokes the corresponding handler function registered.

This API is only valid when the Modbus stack is configured as Slave mode.

・Structure of function code mapping table (slave_map_init_t)

```
typedef struct _slave_map_init{
    fp_function_code1_t     fp_function_code1;     /* Call back function pointer for Modbus function code 1
                                                      (Read Coils) operation */
    fp_function_code2_t     fp_function_code2;     /* Call back function pointer for Modbus function code 2
                                                      (Read Discrete Inputs) operation */
    fp_function_code3_t     fp_function_code3;     /* Call back function pointer for Modbus function code 3
                                                      (Read Holding Registers) operation */
    fp_function_code4_t     fp_function_code4;     /* Call back function pointer for Modbus function code 4
                                                      (Read Input Registers) operation */
    fp_function_code5_t     fp_function_code5;     /* Call back function pointer for Modbus function code 5
                                                      (Write Single Coil) operation */
    fp_function_code6_t     fp_function_code6;     /* Call back function pointer for Modbus function code 6
                                                      (Write Single Register) operation */
    fp_function_code15_t    fp_function_code15;    /* Call back function pointer for Modbus function code 15
                                                      (Write Multiple Coils) operation */
    fp_function_code16_t    fp_function_code16;    /* Call back function pointer for Modbus function code 16
                                                      (Write Multiple Registers) operation */
    fp_function_code23_t    fp_function_code23;    /* Call back function pointer for Modbus function code 23
                                                      (Read/Write Multiple Registers) operation */
}slave_map_init_t, *p_slave_map_init_t;
```

Callback function corresponding to each function code, to the definition in the following format. For more information on the structure to be used in the callback function, please refer to each API of Chapter10.1.2.2.

fp_function_code1_t	Call back function pointer for Modbus function code 1(Read Coils) processing

【Format】

```
uint32_t (*fp_function_code1_t)(p_req_read_coils_t  pt_req_read_coils,
                                p_resp_read_coils_t pt_resp_read_coils );
```

【Parameter】

p_req_read_coils_t	pt_req_read_coils	structure pointer from stack to user with read coils request information
p_resp_read_coils_t	pt_resp_read_coils	structure pointer to stack from user with read coils response data

【Return value】

uint32_t	0 : success ,1 : failure

fp_function_code2_t	Call back function pointer for Modbus function code 2(Read Discrete Inputs) processing

【Format】

```
uint32_t (*fp_function_code2_t)(p_req_read_inputs_t pt_req_read_inputs,
                                p_resp_read_inputs_t pt_resp_read_inputs );
```

【Parameter】

p_req_read_inputs_t	pt_req_read_inputs	structure pointer from stack to user with read discrete inputs request information
p_resp_read_inputs_t	pt_resp_read_inputs	structure pointer to stack from user with read discrete inputs response data

【Return value】

uint32_t	0 : success ,1 : failure

fp_function_code3_t	Call back function pointer for Modbus function code 3(Read Holding Registers) processing

【Format】

```
uint32_t (*fp_function_code3_t)(p_req_read_holding_reg_t pt_req_read_holding_reg,
                                p_resp_read_holding_reg_t pt_resp_read_holding_reg);
```

【Parameter】

p_req_read_holding_reg_t	pt_req_read_holding_reg	structure pointer from stack to user with read holding registers request information
p_resp_read_holding_reg_t	pt_resp_read_holding_reg	structure pointer to stack from user with read holding registers response data

【Return value】

uint32_t	0 : success ,1 : failure

fp_function_code4_t	Call back function pointer for Modbus function code 4(Read Input Registers) processing

【Format】

```
uint32_t (*fp_function_code4_t)(p_req_read_input_reg_t pt_req_read_input_reg,
                                p_resp_read_input_reg_t pt_resp_read_input_reg);
```

【Parameter】

p_req_read_input_reg_t	pt_req_read_input_reg	structure pointer from stack to user with read input registers request information
p_resp_read_input_reg_t	pt_resp_read_input_reg	structure pointer to stack from user with read input registers response data

【Return value】

uint32_t	0 : success ,1 : failure

fp_function_code5_t	Call back function pointer for Modbus function code 5(Write Single Coil) processing

【Format】

```
uint32_t (*fp_function_code5_t)(p_req_write_single_coil_t  pt_req_write_single_coil,
                                p_resp_write_single_coil_t pt_resp_write_single_coil );
```

【Parameter】

p_req_write_single_coil_t	pt_req_write_single_coil	structure pointer from stack to user with write single coil request information
p_resp_write_single_coil_t	pt_resp_write_single_coil	structure pointer to stack from user with write single coil response

【Return value】

uint32_t	0 : success ,1 : failure

fp_function_code6_t	Call back function pointer for Modbus function code 6(Write Single Register) processing

【Format】

```
uint32_t (*fp_function_code6_t)(p_req_write_single_reg_t  pt_req_write_single_reg,
                                p_resp_write_single_reg_t pt_resp_write_single_reg);
```

【Parameter】

p_req_write_single_reg_t	pt_req_write_single_reg	structure pointer from stack to user with write single register request information
p_resp_write_single_reg_t	pt_resp_write_single_reg	structure pointer to stack from user with write single register response

【Return value】

uint32_t	0 : success ,1 : failure

fp_function_code15_t	Call back function pointer for Modbus function code 15(Write Multiple Coils) processing

【Format】

```
uint32_t (*fp_function_code15_t) (p_req_write_multiple_coils_t pt_req_write_multiple_coils,
                                  p_resp_write_multiple_coils_t pt_resp_write_multiple_coils);
```

【Parameter】

p_req_write_multiple_coils_t	pt_req_write_multiple_coils	structure pointer from stack to user with write multiple coils request information
p_resp_write_multiple_coils_t	pt_resp_write_multiple_coils	structure pointer to stack from user with write multiple coils response

【Return value】

uint32_t	0 : success ,1 : failure

fp_function_code16_t	Call back function pointer for Modbus function code 16(Write Multiple Registers) processing

【Format】

```
uint32_t (*fp_function_code16_t) (p_req_write_multiple_reg_t pt_req_write_multiple_reg,
                                  p_resp_write_multiple_reg_t pt_resp_write_multiple_reg);
```

【Parameter】

p_req_write_multiple_reg_t	pt_req_write_multiple_reg	structure pointer from stack to user with write multiple registers request information
p_resp_write_multiple_reg_t	pt_resp_write_multiple_reg	structure pointer to stack from user with write multiple registers response

【Return value】

uint32_t	0 : success ,1 : failure

fp_function_code23_t	Call back function pointer for Modbus function code 23(Read/Write Multiple Registers) processing

【Format】

```
uint32_t (*fp_function_code23_t) (p_req_read_write_multiple_reg_t pt_req_read_write_multiple_reg,
                                  p_resp_read_write_multiple_reg_t pt_resp_read_write_multiple_reg);
```

【Parameter】

p_req_read_write_multiple_reg_t	pt_req_read_write_multiple_reg	structure pointer from stack to user with read/write multiple registers request information
p_resp_read_write_multiple_reg_t	pt_resp_read_write_multiple_reg	structure pointer to stack from user with read/write multiple registers response

【Return value】

uint32_t	0 : success ,1 : failure

10.1.1.2 IP management

The following API is used in IP management.

Modbus_tcp_init_ip_table	Modbus set host IP list properties

【Format】

```
void_t Modbus_tcp_init_ip_table(ENABLE_FLAG e_flag,
                                TABLE_MODE e_mode);
```

【Parameter】

ENABLE_FLAG	e_flag	Status is whether the connection table enabled or disabled enabled: ENABLE, disabled: DISABLE
TABLE_MODE	e_mode	Status indicating the list contain IP to be accepted or rejected accepted: ACCEPT, rejected: REJECT

【Return value】

void_t

【Error code】

—

【Explanation】

This function is used for specifying mode (accept/reject) and to Enable or Disable the list of IP address by the user. By default the host IP list is disabled.

Modbus_tcp_add_ip_addr	Modbus add an IP to host IP list

【Format】

```
uint32_t Modbus_tcp_add_ip_addr(pchar_t pu8_add_ip);
```

【Parameter】

pchar_t	pu8_add_ip	Host IP address in numbers and dots notation ex. 192.168.1.100

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful addition.
ERR_IP_ALREADY_PRESENT	If address already present in list.
ERR_MAX_CLIENT	If maximum connections reached.
ERR_TABLE_DISABLED	If list is disabled.

【Explanation】

This function is used for adding a IP to the host IP list.

Modbus_tcp_delete_ip_addr	remove an IP from host IP list

【Format】

uint32_t Modbus_tcp_delete_ip_addr(pchar_t pu8_del_ip);

【Parameter】

pchar_t	pu8_del_ip	Host IP address in numbers and dots notation

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful search.
ERR_IP_NOT_FOUND	If IP is not in the list
ERR_TABLE_EMPTY	If the list is empty
ERR_TABLE_DISABLED	If the table is disabled, i.e., server accepts connection request from any host

【Explanation】

This function is used for removing a host IP from the list.

10.1.1.3 Task

The following function is the main processing task operates in the protocol stack.

Modbus_tcp_recv_data_task	TCP Receive data Task

【Format】

void_t Modbus_tcp_recv_data_task(void_t);

【Parameter】

void_t

【Return value】

void_t

【Error code】

—

【Explanation】

This task waits for a request received in the selected socket ID. It verifies the packet is for Modbus protocol. If so, write the request to the receive mailbox. If the mailbox is found full, send an error server busy to the client.

Modbus_tcp_req_process_task	TCP server task

【Format】

void_t Modbus_tcp_req_process_task(void_t);

【Parameter】

void_t

【Return value】

void_t

【Error code】

—

【Explanation】

This task wait for a request in the queue. Verify the slave ID in the request packet to determine the packet is for the TCP server or the device connected to it serially. If the packet is for the TCP server process the request read from the queue, prepare the response packet and send it to the TCP client. If the packet is for the serial device connected, write the request to the gateway mailbox.

Modbus_gateway_task	TCP – Serial Gateway task

【Format】

```
void_t Modbus_gateway_task(void_t);
```

【Parameter】

void_t

【Return value】

void_t

【Error code】

—

【Explanation】

This task wait for a request in the gateway queue for processing the data in the serial device connected to the TCP server. It process the request read from the queue, prepare the response packet and send it to the TCP client.

Modbus_tcp_soc_wait_task	TCP accept connection task

【Format】

```
void_t Modbus_tcp_soc_wait_task(void_t);
```

【Parameter】

void_t

【Return value】

void_t

【Error code】

—

【Explanation】

This task waits for a connection from a client in the default Modbus port (502) and an additional port if configured by user. Verify whether the IP table is enabled or not. If enabled, verify the list contains the IP list to accepted or rejected. Accordingly save the socket descriptor to the connection list.

Modbus_tcp_terminate_stack	Modbus terminate TCP stack API

【Format】

uint32_t Modbus_tcp_terminate_stack(void_t);

【Parameter】

void_t

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful termination
ERR_STACK_TERM	If termination failed

【Explanation】

This API terminate Modbus stack. Depending upon the stack mode, corresponding APIs are invoked. If the stack mode is MODBUS_TCP_SERVER_MODE, Modbus_tcp_terminate_stack() is invoked. If the stack mode is gateway mode, call Modbus_tcp_terminate_gateway_stack () in addition to Modbus_tcp_server_terminate_stack ().

10.1.2 Modbus Serial

10.1.2.1 Initialization of protocol stack

The following API is used in initialization of protocol stack.

Modbus_serial_stack_init	Modbus Serial Stack initialization API

【Format】

```
uint32_t Modbus_serial_stack_init(p_serial_stack_init_info_t pt_serial_stack_init_info,
                                  p_serial_gpio_cfg_t  pt_serial_gpio_cfg_t,
                                  uint8_t u8_stack_mode,
                                  uint8_t u8_slave_id);
```

【Paramter】

p_serial_stack_init_info_t	pt_serial_stack_init_info	Pointer to the structure with serial configuration parameters.
p_serial_gpio_cfg_t	pt_serial_gpio_cfg_t	Pointer to the structure with GPIO configuration parameters.
uint8_t	u8_stack_mode	Mode in which the stack should be initialized(RTU/ASCII master/slave)
uint8_t	u8_slave_id	Slave ID of device; valid only for slave mode

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful initialization of serial stack
ERR_INVALID_STACK_MODE	If stack mode specified is invalid
ERR_INVALID_SLAVE_ID	If slave id specified is invalid
ERR_INVALID_STACK_INIT_PARAMS	If invalid stack init information from user
ERR_STACK_INIT	If stack activation or clear flag fails

【Explanation】

This API is to initialize the serial stack as per the user provided configuration parameters. By providing different configurations, stack could function in the way user requires.

For this few initializing parameters are provided in the APIs.

a. Structure of type p_serial_stack_init_info_t is an argument in order to provide information specific to serial communication.
b. Structure of type p_serial_gpio_cfg_t is an argument in order to provide function pointers to control the GPIO port for RS485 communication.
c. u8_stack_mode of type uint8_t is an argument in order to select the Modbus serial stack type. According to the value assigned for this parameter, stack works in either of the following mode:

Stack mode parameter code	Meaning
MODBUS_RTU_MASTER_MODE	Used to select Modbus Stack RTU master mode
MODBUS_RTU_SLAVE_MODE	Used to select Modbus Stack RTU slave mode
MODBUS_ASCII_MASTER_MODE	Used to select Modbus Stack ASCII master mode
MODBUS_ASCII_SLAVE_MODE	Used to select Modbus Stack ASCII slave mode

d. u8_slave_id of type uint8_t is an argument in order to set the device ID in slave mode. This parameter is used when the stack is in either ASCII/RTU Slave mode. This parameter can hold any value within the range 1 to 247.

For details on the parameters of the p_serial_stack_init_info_t structure and the p_serial_gpio_cfg_t structure, refer to Chapter10.1.1.1.

Modbus_slave_map_init	Modbus function code mapping API

【Format】

uint32_t Modbus_slave_map_init (p_slave_map_init_t p_tcp_slave_map_init_t);

【Parameter】

p_slave_map_init_t	p_tcp_slave_map_init_t	Structure pointer to function code mapping table

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On success
ERR_INVALID_STACK_INIT_PARAMS	If parameter is null
ERR_MEM_ALLOC	If memory allocation failed

【Explanation】

This API is the same function as when the Modbus TCP. Please refer to Chapter 10.1.1.1 for detail of the function.

10.1.2.2 Master Mode API

The following API is used in master mode.

Modbus_read_coils	Modbus read coils

【Format】

```
uint32_t Modbus_read_coils(p_req_read_coils_t pt_req_read_coils,
                           p_resp_read_coils_t pt_resp_read_coils,
                           fp_callback_notify_t fp_callback_notify);
```

【Parameter】

p_req_read_coils_t	pt_req_read_coils	Structure pointer to read coil request
p_resp_read_coils_t	pt_resp_read_coils	Structure pointer to read coil response
fp_callback_notify_t	fp_callback_notify	Function pointer argument for the call back notification in non blocking API mode. If this argument is set to NULL API become blocking.

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If coil read successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_COILS	If the number of coils provided is not within the specified limit
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If the memory allocation fail
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_CRC_CHECK	If CRC validation fails for RTU stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII stack mode
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid

【Explanation】

This API is used to read data from coils when requested.If this API returns an error, the data field in the response structure will be invalid.

・Structure of read coils request (req_read_coils_t)

```
typedef struct _req_read_coils{
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected */
    uint16_t    u16_start_addr;         /* Specifies address of the first coil */
    uint16_t    u16_num_of_coils;       /* Specifies the number of coils to be read */
}req_read_coils_t, *p_req_read_coils_t;
```

・Structure of read coils response (resp_read_coils_t)

```
struct _resp_read_coils{
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected(Own ID) */
    uint8_t     u8_exception_code;      /* Error detected during processing the request. On
                                           success the exception code should  be zero, if the
                                           exception code is non zero the aru8_data will be null */
    uint8_t     u8_num_of_bytes;        /* Specifies the number of bytes of data */
    uint8_t     aru8_data[MAX_DISCRETE_DATA]; /* Data to be read */
}resp_read_coils_t, *p_resp_read_coils_t;
```

Modbus_read_discrete_inputs	Modbus read discrete inputs

【Format】

```
uint32_t Modbus_read_discrete_inputs(p_req_read_inputs_t pt_req_read_inputs,
                                     p_resp_read_inputs_t pt_resp_read_inputs,
                                     fp_callback_notify_t fp_callback_notify);
```

【Parameter】

p_req_read_inputs_t	pt_req_read_inputs	Structure pointer to read input request
p_resp_read_inputs_t	pt_resp_read_inputs	Structure pointer to read input response
fp_callback_notify_t	fp_callback_notify	Function pointer argument for the call back notification in non blocking API mode. If this argument is set to NULL API become blocking.

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If input read successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_INPUTS	If the number of inputs provided is not within the specified limit
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If the memory allocation fail
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_CRC_CHECK	If CRC validation fails for RTU stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII stack mode
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid

【Explanation】

This API is used to read data from discrete input when requested. If this API returns an error, the data field in the response structure will be invalid.

・Structure of read inputs request (req_read_inputs_t)

```
typedef struct _req_read_inputs{
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected */
    uint16_t    u16_start_addr;         /* Specifies address of the first discrete input */
    uint16_t    u16_num_of_inputs;      /* Specifies the number of discrete inputs to be read */
}req_read_inputs_t, *p_req_read_inputs_t;
```

・Structure of read inputs response (resp_read_inputs_t)

```
typedef struct _resp_read_inputs{
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected */
    uint8_t     u8_exception_code;      /* Error detected during processing the request. On
                                           success the exception code should  be zero, if the
                                           exception code is non zero the aru8_data will be null */
    uint8_t     u8_num_of_bytes;        /* Specifies the number of bytes of data */
    uint8_t     aru8_data[MAX_DISCRETE_DATA]; /* Buffer to store the read data */
}resp_read_inputs_t, *p_resp_read_inputs_t;
```

Modbus_read_holding_registers	Modbus read holding registers.

【Format】

```
uint32_t Modbus_read_holding_registers(p_req_read_holding_reg_t pt_req_read_holding_reg,
                                       p_resp_read_holding_reg_t pt_resp_read_holding_reg,
                                       fp_callback_notify_t fp_callback_notify);
```

【Parameter】

p_req_read_holding_reg_t	pt_req_read_holding_reg	Structure pointer to read holding reg. request
p_resp_read_holding_reg_t	pt_resp_read_holding_reg	Structure pointer to read holding reg. response
fp_callback_notify_t	fp_callback_notify	Function pointer argument for the call back notification in non blocking API mode. If this argument is set to NULL API become blocking.

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If holding register read successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_REG	If the number of registers provided is not within the specified limit
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If the memory allocation fail
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_CRC_CHECK	If CRC validation fails for RTU stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII stack mode
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid

【Explanation】

This API is used to read data from holding registers when requested. If this API returns an error, the data field in the response structure will be invalid.

・Structure of read holding registers request (req_read_holding_reg_t)

```
typedef struct _req_read_holding_reg{
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected */
    uint16_t    u16_start_addr;         /* Specifies address of the first holding register */
    uint16_t    u16_num_of_reg;         /* Specifies the number of registers to be read */
}req_read_holding_reg_t, *p_req_read_holding_reg_t;
```

・Structure of read holding registers response (resp_read_holding_reg_t)

```
typedef struct _resp_read_holding_reg{
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected */
    uint8_t     u8_exception_code;      /* error detected during processing the request. On success
                                           the exception code should  be zero, if the exception code is
                                           non zero the aru16_data will be null */
    uint8_t     u8_num_of_bytes;        /* specifies the number of bytes of data */
    uint16_t    aru16_data[MAX_REG_DATA]; /* buffer to store the read data */
}resp_read_holding_reg_t, p_resp_read_holding_reg_t;
```

Modbus_read_input_registers	Modbus read input registers.

【Format】

```
uint32_t Modbus_read_input_registers(p_req_read_input_reg_t pt_req_read_input_reg,
                                     p_resp_read_input_reg_t pt_resp_read_input_reg,
                                     fp_callback_notify_t fp_callback_notify);
```

【Parameter】

p_req_read_input_reg_t	pt_req_read_input_reg	Structure pointer to read input reg. request
p_resp_read_input_reg_t	pt_resp_read_input_reg	Structure pointer to read input reg. response
fp_callback_notify_t	fp_callback_notify	Function pointer argument for the call back notification in non blocking API mode. If this argument is set to NULL API become blocking.

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If input register read successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_REG	If the number of registers provided is not within the specified limit
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If the memory allocation fail
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_CRC_CHECK	If CRC validation fails for RTU stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII stack mode
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid

【Explanation】

This API is used to read data from input registers when requested. If this API returns an error, the data field in the response structure will be invalid.

・Structure of read input registers request (req_read_input_reg_t)

```
typedef struct _req_read_input_reg{
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected */
    uint16_t    u16_start_addr;         /* Specifies address of the first input register */
    uint16_t    u16_num_of_reg;         /* Specifies the number of registers to be read */
}req_read_input_reg_t, *p_req_read_input_reg_t;
```

・Structure of read input registers response (resp_read_input_reg_t)

```
typedef struct _resp_read_input_reg{
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected */
    uint8_t     u8_exception_code;      /* Error detected during processing the request. On success
                                           the exception code should  be zero, if the exception code is
                                           non zero the aru16_data will be null */
    uint8_t     u8_num_of_bytes;        /* Specifies the number of bytes of data */
    uint16_t    aru16_data[MAX_REG_DATA]; /* Buffer to store the read data */
}resp_read_input_reg_t, p_resp_read_input_reg_t;
```

Modbus_write_single_coil	Modbus write single coil

【Format】

```
uint32_t Modbus_write_single_coil(p_req_write_single_coil_t pt_req_write_single_coil,
                                  p_resp_write_single_coil_t pt_resp_write_single_coil,
                                  fp_callback_notify_t fp_callback_notify);
```

【Parameter】

p_req_write_single_coil_t	pt_req_write_single_coil	Structure pointer to write single coil request
p_resp_write_single_coil_t	pt_resp_write_single_coil	Structure pointer to write single coil response
fp_callback_notify_t	fp_callback_notify	Function pointer argument for the call back notification in non blocking API mode. If this argument is set to NULL API become blocking.

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If single coil write is successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_OUTPUT_VALUE	If the value of the registers is invalid
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If the memory allocation fail
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_CRC_CHECK	If CRC validation fails for RTU stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII stack mode
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid

【Explanation】

This API is used to write data to single coil when requested.

・Structure of write single coil request (req_write_single_coil_t)

```
typedef struct _req_write_single_coil
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected */
    uint16_t    u16_output_addr;        /* Specifies address of the coil */
    uint16_t    u16_output_value;       /* Data to be written */
}req_write_single_coil_t, *p_req_write_single_coil_t;
```

・Structure of write single coil response (resp_write_single_coil_t)

```
typedef struct _resp_write_single_coil{
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected */
    uint8_t     u8_exception_code;      /* Error detected during processing the request. On success the
                                           exception code should be zero */
    uint16_t    u16_output_addr;        /* Specifies address of the coil */
    uint16_t    u16_output_value;       /* Data to be written */
}resp_write_single_coil_t, *p_resp_write_single_coil_t;
```

Modbus_write_single_reg	Modbus write single register

【Format】

```
uint32_t Modbus_write_single_reg(p_req_write_single_reg_t pt_req_write_single_reg,
                                 p_resp_write_single_reg_t pt_resp_write_single_reg,
                                 fp_callback_notify_t      fp_callback_notify);
```

【Parameter】

p_req_write_single_reg_t	pt_req_write_single_reg	Structure pointer to write single reg. request
p_resp_write_single_reg_t	pt_resp_write_single_reg	Structure pointer to write single reg. response
fp_callback_notify_t	fp_callback_notify	Function pointer argument for the call back notification in non blocking API mode. If this argument is set to NULL API become blocking.

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If single register write is successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If the memory allocation fail
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_CRC_CHECK	If CRC validation fails for RTU stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII stack mode
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid

【Explanation】

This API is used to write data to single register when requested.

・Structure of write single register request (req_write_single_reg_t)

```
typedef struct _req_write_single_reg{
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected */
    uint16_t    u16_register_addr;      /* Specifies address of the register */
    uint16_t    u16_register_value;     /* Data to be written */
}req_write_single_reg_t, *p_req_write_single_reg_t;
```

・Structure of write single register response (resp_write_single_reg_t)

```
typedef struct _resp_write_single_reg{
    uint16_t    u16_transaction_id;     /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;        /* Specifies the protocol ID */
    uint8_t     u8_slave_id;            /* Identification of a remote slave connected */
    uint8_t     u8_exception_code;      /* Error detected during processing the request. On success the
                                           exception code should be zero */
    uint16_t    u16_register_addr;      /* Specifies address of the register */
    uint16_t    u16_register_value;     /* Data to be written */
}resp_write_single_reg_t, *p_resp_write_single_reg_t;
```

Modbus_write_multiple_coils	Modbus write multiple coils

【Format】

```
uint32_t Modbus_write_multiple_coils(p_req_write_multiple_coils_t pt_req_write_multiple_coils,
                                     p_resp_write_multiple_coils_t pt_resp_write_multiple_coils,
                                     fp_callback_notify_t fp_callback_notify);
```

【Parameter】

p_req_write_multiple_coils_t	pt_req_write_multiple_coils	Structure pointer to write multiple coils request
p_resp_write_multiple_coils_t	pt_resp_write_multiple_coils	Structure pointer to write multiple coils response
fp_callback_notify_t	fp_callback_notify	Function pointer argument for the call back notification in non blocking API mode. If this argument is set to NULL API become blocking.

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If write multiple coil write successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_OUTPUTS	If the number of outputs is invalid
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If the memory allocation fail
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_CRC_CHECK	If CRC validation fails for RTU stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII stack mode
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid

【Explanation】

This API is used to write data to multiple coils when requested.

- Structure of write multiple coils request (req_write_multiple_coils_t)

```
typedef struct _req_write_single_reg{
    uint16_t    u16_transaction_id;             /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;                /* Specifies the protocol ID */
    uint8_t     u8_slave_id;                    /* Identification of a remote slave connected */
    uint16_t    u16_start_addr;                 /* Specifies address of the first coil */
    uint16_t    u16_num_of_outputs;             /* Specifies the number of coils to be written */
    uint8_t     u8_num_of_bytes;                /* Specifies the number of bytes of data */
    uint8_t     aru8_data[MAX_DISCRETE_DATA];   /* Data to be written */
}req_write_single_reg_t, *p_req_write_single_reg_t;
```

- Structure of write multiple coils response (resp_write_multiple_coils_t)

```
typedef struct _resp_write_multiple_coils{
    uint16_t    u16_transaction_id;             /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;                /* Specifies the protocol ID */
    uint8_t     u8_slave_id;                    /* Identification of a remote slave connected */
    uint8_t     u8_exception_code;              /* Error detected during processing the request. On
                                                   success the exception code should be zero */
    uint16_t    u16_start_addr;                 /* Specifies address of the coils */
    uint16_t    u16_num_of_outputs;             /* Specifies the number of coils to be written */
}resp_write_multiple_coils_t, *p_resp_write_multiple_coils_t;
```

Modbus_write_multiple_reg	Modbus write multiple registers

【Format】

```
uint32_t Modbus_write_multiple_reg(p_req_write_multiple_reg_t pt_req_write_multiple_reg,
                                   p_resp_write_multiple_reg_t pt_resp_write_multiple_reg,
                                   fp_callback_notify_t fp_callback_notify);
```

【Parameter】

p_req_write_multiple_reg_t	pt_req_write_multiple_reg	Structure pointer to write multiple reg. request
p_resp_write_multiple_reg_t	pt_resp_write_multiple_reg	Structure pointer to write multiple reg. response
fp_callback_notify_t	fp_callback_notify	Function pointer argument for the call back notification in non blocking API mode. If this argument is set to NULL API become blocking.

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If write multiple register write successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_REG	If the number of registers is invalid
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If the memory allocation fail
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_CRC_CHECK	If CRC validation fails for RTU stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII stack mode
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid

【Explanation】

This API is used to write data to multiple registers when requested.

・Structure of write multiple registers request (req_write_multiple_reg_t)

```
typedef struct _req_write_multiple_reg{
    uint16_t    u16_transaction_id;         /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;            /* Specifies the protocol ID */
    uint8_t     u8_slave_id;                /* Identification of a remote slave connected */
    uint16_t    u16_start_addr;             /* Specifies address of the first register */
    uint16_t    u16_num_of_reg;             /* Specifies the number of registers to be written */
    uint8_t     u8_num_of_bytes;            /* Specifies the number of bytes of data */
    uint16_t    aru16_data[MAX_REG_DATA];   /* Data to be written */
}req_write_multiple_reg_t, *p_req_write_multiple_reg_t;
```

・ Structure of write multiple registers response (resp_write_multiple_reg_t)

```
typedef struct _resp_write_multiple_reg{
    uint16_t    u16_transaction_id;         /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;            /* Specifies the protocol ID */
    uint8_t     u8_slave_id;                /* Identification of a remote slave connected */
    uint8_t     u8_exception_code;          /* Error detected during processing the request. On success the
                                               exception code should be zero */
    uint16_t    u16_start_addr;             /* Specifies address of the first register */
    uint16_t    u16_num_of_reg;             /* Specifies the number of registers to be written */
}resp_write_multiple_reg_t, *p_resp_write_multiple_reg_t;
```

Modbus_read_write_multiple_reg	Modbus read and write multiple registers

【Format】

uint32_t Modbus_read_write_multiple_reg(p_req_read_write_multiple_reg_t pt_req_read_write_multiple_reg,
p_resp_read_write_multiple_reg_t pt_resp_read_write_multiple_reg,
fp_callback_notify_t fp_callback_notify);

【Parameter】

p_req_read_write_multiple_reg_t	pt_req_read_write_multiple_reg	Structure pointer to read and write multiple reg request
p_resp_read_write_multiple_reg_t	pt_resp_read_write_multiple_reg	Structure pointer to read and write multiple reg response
fp_callback_notify_t	fp_callback_notify	Function pointer argument for the call back notification in non blocking API mode. If this argument is set to NULL API become blocking.

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If read/write multiple register is successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_OUTPUT_VALUE	If the value of the registers is invalid
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If the memory allocation fail
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_CRC_CHECK	If CRC validation fails for RTU stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII stack mode
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid

【Explanation】

This API is used to read and write data to multiple registers when requested. If this API returns an error, the data field in the response structure will be invalid.

・Structure of read and write multiple registers request (req_read_write_multiple_reg_t)

```
typedef struct _req_read_write_multiple_reg{
    uint16_t    u16_transaction_id;         /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;            /* Specifies the protocol ID */
    uint8_t     u8_slave_id;                /* Identification of a remote slave connected */
    uint16_t    u16_read_start_addr;        /* Specifies address of the first register to be read from */
    uint16_t    u16_num_to_read;            /* Specifies the number of registers to be read */
    uint16_t    u16_write_start_addr;       /* Specifies address of the first register to be written */
    uint16_t    u16_num_to_write;           /* Specifies the number of registers to be written */
    uint8_t     u8_write_num_of_bytes;      /* Specifies the number of bytes of data */
    uint16_t    aru16_data[MAX_REG_DATA];   /* Data to be written */
}req_read_write_multiple_reg_t, *p_req_read_write_multiple_reg_t;
```

・Structure of read and write multiple registers response (resp_read_write_multiple_reg_t)

```
typedef struct _resp_read_write_multiple_reg{
    uint16_t    u16_transaction_id;         /* Specifies the transaction ID */
    uint16_t    u16_protocol_id;            /* Specifies the protocol ID */
    uint8_t     u8_slave_id;                /* Identification of a remote slave connected */
    uint8_t     u8_exception_code;          /* Error detected during processing the request. On
                                               success the exception code should  be zero, if the
                                               exception code is non zero the aru16_read_data will be
                                               null */
    uint16_t    u8_num_of_bytes;            /* Specifies the number of complete bytes of data */
    uint16_t    aru16_read_data[MAX_REG_DATA];  /* Data to be read */
}resp_read_write_multiple_reg_t, *p_resp_read_write_multiple_reg_t;
```

Modbus_callback_notify	Call back function for notification

【Format】

void Modbus_callback_notify(uint32_t u32_resp_code);

【Parameter】

uint32_t	u32_resp_code	Response code

【Return value】

void_t

【Error code】

—

【Explanation】

This is the default call back function invoked by the master stack if the caller has not registered their own call back handler. It is only applicable in master mode configuration of Modbus stack.

Stack invokes the registered call back function when read/write request get response from slave side.

10.1.2.3 Task

The following function is the main processing task operates in the protocol stack.

Modbus_serial_task	Modbus serial task

【Format】

```
void_t Modbus_serial_task(void_t);
```

【Parameter】

void_t

【Return value】

void_t

【Error code】

—

【Explanation】

This task runs either as slave task or as master task depending on the stack mode when the stack is in master mode, this task waits for a request from the user. Validate the information provided by the user. If validation is successful, frame the packet and send the packet to the slave device. It waits for the response from the slave. If the callback is provided by the user, task invokes the callback when the response data is received.

When the stack is in slave mode, this task waits for a request. If so process the packet and send the response.

Modbus_serial_recv_task	Modbus serial packet receive task

【Format】

```
void_t Modbus_serial_recv_task(void_t);
```

【Parameter】

void_t

【Return value】

void_t

【Error code】

—

【Explanation】

This task is used to receive data from serial I / F. Calls processing according to the event that occurred in the interrupt.

When a serial I / F receive interrupt event occurs, the receive data is read from the serial I / F, and buffering processing corresponding to each RTU / ASCII mode is called.

When the serial I / F status interrupt is detected, invoke driver function for UART status interrupt. Please refer to "RX72M Communication Board Hardware Manual" for serial I / F status interrupt details.

When the Timer interrupt is detected, invoke the buffering stop process.

Serial_recv_task	Modbus serial data receive task

【Format】

```
void Serial_recv_task(void);
```

【Parameter】

void

【Return value】

void

【Error code】

—

【Explanation】

This task detects a data receive interrupt from the serial I / F and generates a serial I / F receive interrupt event.

Modbus_serial_stack_terminate	Modbus terminate serial stack API

【Format】

```
uint32_t Modbus_serial_stack_terminate(void_t);
```

【Parameter】

void_t

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful
ERR_STACK_TERM	if termination failed

【Explanation】

This API terminate MODBUS serial stack.

10.1.3 User-Defined Functions

User-defined functions are defined in the following file.
/ r_modbus_rx / src / src / modbus_user.c

In slave mode, use the user-defined Read / Write function to process each function.
The corresponding Read / write function and its table of each address of Coil / Discrete Input / holding resister / input register are prepared.

If a user-defined function is set as shown below, when a read request for Coil address 00001 (offset 0) is received, the processing function cd_fun_code01 () of function code 1 is set from Coil's Read processing function table MB_Coils_Read [] to Coil Call Coil Read processing function MB_Coil_Read_00001 at address 00001.

```
/*Read processing function of Coil address 00001 (offset 0) */
uint8_t MB_Coil_Read_00001(uint8_t *coil)
{
   *coil = 0;
   if (LED0 == 0)
   {
      *coil |= 1;
   }
   g_Coils_Area[0]=*coil;
   return ERR_OK;
}

/* Coil Read Processing Function Table */
uint8_t (*MB_Coils_Read[])(uint8_t *data)={
   MB_Coil_Read_00001,    /* 00001 */
   MB_Coil_Read_00002,    /* 00002 */
…
```

- Coil function (Read) Function table (MB_Coils_Read)

```
uint8_t (*MB_Coils_Read[])(uint8_t *data)={
        MB_Coil_Read_00001,       /* 00001 */
        MB_Coil_Read_00002,       /* 00002 */
        MB_Coil_Read_00003,       /* 00003 */
        MB_Coil_Read_00004,       /* 00004 */
        MB_Reg_Exp_Addr_p8        /* 00005 */
        MB_Reg_Exp_Addr_p8        /* 00006 */
        MB_Reg_Exp_Addr_p8        /* 00007 */
        MB_Reg_Exp_Addr_p8        /* 00008 */
};
```

・ Coil function (Write) Function table (MB_Coils_Write)

```
uint8_t (*MB_Coils_Write[])(uint8_t data)={
    MB_Coil_Write_00001,      /* 00001 */
    MB_Coil_Write_00002,      /* 00002 */
    MB_Coil_Write_00003,      /* 00003 */
    MB_Coil_Write_00004,      /* 00004 */
    MB_Reg_Exp_Addr_p8        /* 00005 */
    MB_Reg_Exp_Addr_p8        /* 00006 */
    MB_Reg_Exp_Addr_p8        /* 00007 */
    MB_Reg_Exp_Addr_p8        /* 00008 */
};
```

・Function table for Discrete Input (MB_Discretes_Input)

```
uint8_t (*MB_Discretes_Input[])(uint8_t *data)={
    MB_D_Read_10001,          /* 10001 */
    MB_D_Read_10002,          /* 10002 */
    MB_D_Read_10003,          /* 10003 */
    MB_D_Read_10004,          /* 10004 */
    MB_D_Read_10005,          /* 10005 */
    MB_D_Read_10006,          /* 10006 */
    MB_D_Read_10007,          /* 10007 */
    MB_D_Read_10008,          /* 10008 */
    MB_D_Read_10009,          /* 10009 */
    MB_Reg_Exp_Addr_p8,       /* 10010 */
    MB_D_Read_10011,          /* 10011 */
    MB_D_Read_10012,          /* 10012 */
};
```

・Function (Read) table for holding resister (MB_HoldingRegs_Read)

```
uint8_t (*MB_HoldingRegs_Read[])(uint16_t *data)={
    MB_Reg_Read_40001,        /* 40001 */
    MB_Reg_Read_40002,        /* 40002 */
    MB_Reg_Read_40003,        /* 40003 */
    MB_Reg_Exp_Addr_p16,      /* 40004 */
    MB_Reg_Exp_Addr_p16,      /* 40005 */
    MB_Reg_Exp_Addr_p16,      /* 40006 */
    MB_Reg_Read_40007,        /* 40007 */
};
```

・Function table for Input register (MB_Input_Regs)

```
uint8_t (*MB_HoldingRegs_Read[])(uint16_t *data)={
    MB_IReg_Read_30001,     /* 30001 */
    MB_IReg_Read_30002,     /* 30002 */
    MB_IReg_Read_30003,     /* 30003 */
    MB_Reg_Exp_Addr_p16,    /* 30004 */
    MB_Reg_Exp_Addr_p16,    /* 30005 */
    MB_Reg_Exp_Addr_p16,    /* 30006 */
    MB_IReg_Read_30008,     /* 30007 */
};
```

・Function (Write) table for Holding resistance (MB_HoldingRegs_Write)

```
uint8_t (*MB_HoldingRegs_Write[])(uint16_t data)={
    MB_Reg_Write_40001,     /* 40001 */
    MB_Reg_Write_40002,     /* 40002 */
    MB_Reg_Write_40003,     /* 40003 */
    MB_Reg_Exp_Addr_p16,    /* 40004 */
    MB_Reg_Exp_Addr_p16,    /* 40005 */
    MB_Reg_Exp_Addr_p16,    /* 40006 */
    MB_Reg_Write_40007,     /* 40007 */
};
```

10.2 Internal API

This chapter explains the interface of the API that is used internally.

10.2.1 Packet Framing and Parsing API

10.2.1.1 Serial Connection Management

The following API has been used in the packet processing of serial communication.

Modbus_serial_frame_pkt	Modbus serial frame packet

【Format】

```
void_t Modbus_serial_frame_pkt(puint8_t pu8_mb_snd_pkt,
                               puint32_t pu32_snd_pkt_len,
                               p_mbserial_queue_elmnt_t pt_mbserial_queue_elmnt);
```

【Parameter】

puint8_t	pu8_mb_snd_pkt	Pointer to the array storing packet to be send
puint32_t	pu32_snd_pkt_len	Length of the packet framed
pt_mbserial_queue_elmnt	pt_mbserial_queue_elmnt	Structure pointer containing user information

【Return value】

void_t

【Error code】

—

【Explanation】

This function frames a packet with the information provided by the user application. Depending on the mode of the stack, structure is passed to corresponding functions.

For master mode, Modbus_master_frame_request() is invoked. Similarly for slave mode, Modbus_slave_frame_response() is invoked, and then collect the necessary information.

For RTU mode, Modbus_rtu_frame_pkt() is invoked. Similarly for ASCII mode, ,Modbus_ascii_frame_pkt() is invoked, and generate a packet.

・Structure of serial packet queue (mbserial_queue_elmnt_t)

```
typedef struct _mbserial_queue_elmnt{
    fp_callback_notify_t    fpt_callback_notify;          /* Function pointer for the call back
                                                          notification in non blocking API mode.If this
                                                          argument is set to NULL then it is in blocking
                                                          mode. */
    void*                   pu8_output_response;          /* Pointer to the response structure from the
                                                          API */
    void*                   pu8_input_request;            /* Pointer to the request structure from the
                                                          API */
    uint32_t                u32_num_of_bytes;             /* Specifies the number of bytes in the data
                                                          packet field */
    uint8_t                 aru8_data_packet[MAX_DATA_SIZE]; /* Contains the data provided by the user
                                                          application */
    uint8_t                 u8_cmd_mode;                  /* Contains whether the stack is in unicast or
                                                          broadcast mode */
    uint8_t                 u8_slave_id;                  /* Contains slave id in the request */
    uint8_t                 u8_func_code;                 /* Contains function code in the request */
}mbserial_queue_elmnt_t, *p_mbserial_queue_elmnt_t;
```

This structure is the following macro is used.

Packet processing mode	Meaning
UNICAST_MODE	This packet is processed as a Unicast packet
BROADCAST_MODE	This packet is processed as a Broadcast packet

Modbus_rtu_frame_pkt	Modbus RTU frame packet

【Format】

```
void_t Modbus_rtu_frame_pkt(puint8_t     pu8_mb_snd_pkt,
                            puint32_t pu32_snd_pkt_len,
                            p_mbserial_queue_elmnt_t pt_mbserial_queue_elmnt);
```

【Paramter】

puint8_t	pu8_mb_snd_pkt	Pointer to the array storing packet to be send
puint32_t	pu32_snd_pkt_len	Length of the packet framed
pt_mbserial_queue_elmnt	pt_mbserial_queue_elmnt	Structure pointer containing user information

【Return value】

void_t

【Error code】

—

【Explanation】

This function frames a packet for RTU device with the information provided by the user application. For calculating CRC, calculate_crc() is used.

Modbus_ascii_frame_pkt	Modbus ASCII frame packet

【Format】

```
void_t Modbus_ascii_frame_pkt(puint8_t  pu8_mb_snd_pkt,
                              puint32_t pu32_snd_pkt_len,
                              p_mbserial_queue_elmnt_t pt_mbserial_queue_elmnt);
```

【Parameter】

puint8_t	pu8_mb_snd_pkt	Pointer to the array storing packet to be send
puint32_t	pu32_snd_pkt_len	Length of the packet framed
pt_mbserial_queue_elmnt	pt_mbserial_queue_elmnt	Structure pointer containing user information

【Return value】

void_t

【Error code】

—

【Explanation】

This function frames a packet for ASCII device with the information provided by the user application. For calculating LRC, calculate_lrc() is used.

Modbus_serial_send_pkt	Modbus serial send packet

【Format】

```
void_t Modbus_serial_send_pkt(puint8_t pu8_mb_snd_pkt,
                              uint32_t u32_snd_pkt_len);
```

【Parameter】

puint8_t	pu8_mb_snd_pkt	Pointer to the MODBUS send packet array
uint32_t	u32_snd_pkt_len	Length of the MODBUS send packet

【Return value】

void_t

【Error code】

—

【Explanation】

This function is the wrapper function of Modbus_serial_send ().

Modbus_serial_send	Modbus serial send packet

【Format】

```
void_t Modbus_serial_send(puint8_t u8_mb_snd_pkt,
                          uint32_t u32_snd_pkt_len);
```

【Parameter】

puint8_t	pu8_mb_snd_pkt	Pointer to the MODBUS send packet array
uint32_t	u32_snd_pkt_len	Length of the MODBUS send packet

【Return value】

void_t

【Error code】

—

【Explanation】

This function sends the prepared packet through serial I / F. During transmission, by RS485 control function that has been registered in the stack initialization function, communication direction is switched to the sender.

Modbus_serial_parse_pkt	MODBUS serial parse packet

【Format】

```
uint32_t Modbus_serial_parse_pkt(puint8_t pu8_mb_recv_pkt,
                                 puint32_t pu32_recv_pkt_len,
                                 p_mbserial_queue_elmnt_t pt_mbserial_queue_elmnt);
```

【Parameter】

puint8_t	pu8_mb_recv_pkt	Pointer to the Modbus receive packet array.
uint32_t	u32_recv_pkt_len	Length of the Modbus receive packet
p_mbserial_queue_elmnt_t	pt_mbserial_queue_elmnt	Structure pointer that contains information to be provided to the user

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On parsing of packet received is successful
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_CRC_CHECK	If CRC validation fails for RTU stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII stack mode
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_INVALID_SLAVE_ID	If the slave ID is invalid
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_OK_WITH_NO_RESPONSE	Return status for broadcast requests
ERR_MSG_SIZE_OVER	Received request or response data size exceeds maximum length

【Explanation】

This function parses the received packet and updates the structure that contains information to be provided to the user.

Depending on the mode of the stack, the corresponding functions are invoked. For RTU, Modbus_rtu_parse_pkt() is invoked. Similarly for ASCII, Modbus_ascii_parse_pkt() is invoked.

Modbus_rtu_parse_pkt	Modbus RTU parse packet

【Format】

```
uint32_t Modbus_rtu_parse_pkt(puint8_t  pu8_mb_recv_pkt,
                              puint32_t pu32_recv_pkt_len,
                              p_mbserial_queue_elmnt_t pt_mbserial_queue_elmnt);
```

【Parameter】

puint8_t	pu8_mb_recv_pkt	Pointer to the Modbus receive packet array
uint32_t	pu32_recv_pkt_len	Length of the Modbus receive packet
p_mbserial_queue_elmnt_t	pt_mbserial_queue_elmnt	Structure pointer that contains information to be provided to the user

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On parsing of packet received is successful
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_CRC_CHECK	Validation fails for RTU stack mode
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_INVALID_SLAVE_ID	If the slave ID is invalid
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_OK_WITH_NO_RESPONSE	Return status for broadcast requests
ERR_MSG_SIZE_OVER	Received request or response data size exceeds maximum length

【Explanation】

This API parses the packet received from serial I / F. Depending on the mode of the stack, Modbus_master_parse_pkt() is invoked for master mode, Modbus_slave_parse_pkt() is invoked for slave mode.

Modbus_ascii_parse_pkt	Modbus ASCII parse packet

【Format】

```
uint32_t Modbus_ascii_parse_pkt (puint8_t pu8_mb_recv_pkt,
                                 puint32_t pu32_recv_pkt_len,
                                 p_mbserial_queue_elmnt_t pt_mbserial_queue_elmnt);
```

【Parameter】

puint8_t	pu8_mb_recv_pkt	Pointer to the Modbus receive packet array
uint32_t	pu32_recv_pkt_len	Length of the Modbus receive packet
p_mbserial_queue_elmnt_t	pt_mbserial_queue_elmnt	Structure pointer that contains information to be provided to the user

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On parsing of packet received is successful
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_CRC_CHECK	Validation fails for RTU stack mode
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_INVALID_SLAVE_ID	If the slave ID is invalid
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_OK_WITH_NO_RESPONSE	Return status for broadcast requests
ERR_MSG_SIZE_OVER	Received request or response data size exceeds maximum length

【Explanation】

This API parses the packet received from serial I / F. In this function, after converting the specified ASCII packet to RTU packet, call each packet analysis APIs corresponding to the stack mode.

Modbus_master_parse_pkt	Modbus Master parse packet

【Format】

```
uint32_t Modbus_master_parse_pkt(puint8_t pu8_mb_recv_pkt,
                                 puint32_t pu32_recv_pkt_len,
                                 p_mbserial_queue_elmnt_t    pt_mbserial_queue_elmnt);
```

【Parameter】

puint8_t	pu8_mb_recv_pkt	Pointer to the Modbus receive packet array
uint32_t	pu32_recv_pkt_len	Length of the Modbus receive packet
p_mbserial_queue_elmnt_t	pt_mbserial_queue_elmnt	Structure pointer that contains information to be provided to the user

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On parsing of packet received is successful
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_LRC_CHECK	If LRC validation fails for ASCII master stack mode
ERR_CRC_CHECK	If CRC validation fails for RTU master stack mode
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_INVALID_SLAVE_ID	If the slave ID is invalid
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid

【Explanation】

This API parses the packet received from serial I / F. The structure that contains information to be provided to the user is updated.

Modbus_slave_parse_pkt	Modbus Slave parse packet

【Format】

```
uint32_t Modbus_slave_parse_pkt(puint8_t pu8_mb_recv_pkt,
                                puint32_t pu32_recv_pkt_len,
                                p_mbserial_queue_elmnt_t pt_mbserial_queue_elmnt);
```

【Parameter】

puint8_t	pu8_mb_recv_pkt	Pointer to the Modbus receive packet array
uint32_t	pu32_recv_pkt_len	Length of the Modbus receive packet
p_mbserial_queue_elmnt_t	pt_mbserial_queue_elmnt	Structure pointer that contains information to be provided to the user

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On parsing of packet received is successful
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_INVALID_SLAVE_ID	If the slave ID is invalid
ERR_MEM_ALLOC	If memory allocation fails
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_OK_WITH_NO_RESPONSE	Return status for broadcast requests

【Explanation】

This API parse the specified packet, perform the callback function corresponding to each function code that the user has registered. After callback perform, API updates the structure of serial queue(pt_mbserial_queue_elmnt) based on the execution results. In this function, dynamically allocate memory request and response table for each callback perform. Request table will be released within this function, but response table will be released at the stage of generating a response packet.

Modbus_master_validate_pkt	Modbus master validate packet

【Format】

uint32_t Modbus_master_validate_pkt(p_mbserial_queue_elmnt_t pt_mbserial_queue_elmnt);

【Paramter】

p_mbserial_queue_elmnt_t	pt_mbserial_queue_elmnt	Structure pointer that contains information to be provided to the user

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On parsing of packet received is successful
ERR_SLAVE_ID_MISMATCH	If master receives a response from another slave (not from the requested slave)
ERR_FUN_CODE_MISMATCH	If master receives a response for another function code(not for the requested function code)
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_INVALID_SLAVE_ID	If the slave ID is invalid
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid

【Explanation】

This function validates the packet received from serial I / F and returns error if validation fails. Slave ID and function code in the packet is verified in this function.

Modbus_slave_validate_pkt	Modbus slave validate packet

【Format】

uint32_t Modbus_slave_validate_pkt(p_mbserial_queue_elmnt_t pt_mbserial_queue_elmnt);

【Parameter】

p_mbserial_queue_elmnt_t	pt_mbserial_queue_elmnt	Structure pointer that contains information to be provided to the user

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On validation of packet received is successful
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_ILLEGAL_FUNCTION	If the function code is invalid or if function code is disabled in the stack
ERR_INVALID_SLAVE_ID	If the slave ID is invalid
ERR_SLAVE_ID_MISMATCH	If the slave id in the request is not its own slave id or broadcast id
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_OK_WITH_NO_RESPONSE	Return status for broadcast requests

【Explanation】

This function validates the packet received from serial I / F and returns error if validation fails. Slave ID in the packet is verified in this function.

Modbus_master_frame_request	Modbus Master frame response

【Format】

void_t Modbus_master_frame_request(p_mbserial_queue_elmnt_t pt_mbserial_queue_elmnt);

【Parameter】

p_mbserial_queue_elmnt_t	pt_mbserial_queue_elmnt	Structure pointer that contains information to be provided to the user

【Return value】

void_t

【Error code】

—

【Explanation】

This function is invoked when the stack is in master mode. The mb_serial structure is updated using the information from response structure provided by the user Application.

Modbus_slave_frame_response	Modbus Slave frame response

【Format】

void_t Modbus_slave_frame_response(p_mbserial_queue_elmnt_t pt_mbserial_queue_elmnt);

【Parameter】

p_mbserial_queue_elmnt_t	pt_mbserial_queue_elmnt	Structure pointer containing user information

【Return value】

void_t

【Error code】

—

【Explanation】

This function is invoked when the stack is in slave mode. The mb_serial structure is updated using the information from response structure provided by the user Application.

Modbus_serial_write	Modbus serial I / F write

【Format】

```
void_t Modbus_serial_write(puint8_t pu8_mb_snd_data,
                           uint32_t u32_data_size);
```

【Parameter】

puint8_t	pu8_mb_snd_data	Starting address of the data to be send
uint32_t	u32_data_size	Length of data to send in bytes

【Return value】

void_t

【Error code】

—

【Explanation】

This API writes the specified number of data to serial I / F. The serial I / F driver function is used to write to the channel number defined by g_modbus_sci_handle.

Modbus_serial_read	Modbus serial I / F read

【Format】

```
uint32_t Modbus_serial_read(puint8_t pu8_mb_read_char);
```

【Parameter】

puint8_t	pu8_mb_read_char	Pointer to the location to read the 8bit character

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful.
ERR_UART_RECV_OPERATION	Read operation failed

【Explanation】

This API reads 1-byte data from serial I / F channel. Reads from the channel number defined by g_modbus_sci_handle using serial I / F driver function.

Modbus_rtu_crc_calculate	Modbus serial cyclic Redundancy check calculation

【Format】

```
uint32_t Modbus_rtu_crc_calculate(puint8_t pu8_mb_pkt,
                                  uint32_t u32_pkt_len);
```

【Parameter】

puint8_t	pu8_mb_pkt	Pointer to the Modbus packet array
uint32_t	u32_pkt_len	Length of the Modbus packet

【Return value】

uint32_t	Calculated CRC value

【Error code】

—

【Explanation】

This function calculates the CRC of the packet.

Modbus_rtu_crc_validate	Modbus serial cyclic Redundancy check validation

【Format】

```
uint32_t Modbus_rtu_crc_validate(puint8_t pu8_mb_pkt,
                                 uint32_t u32_pkt_len);
```

【Parameter】

puint8_t	pu8_mb_pkt	Pointer to the Modbus packet array
uint32_t	u32_pkt_len	Length of the Modbus packet

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If validation of CRC is successful
ERR_CRC_CHECK	If validation fails

【Explanation】

This function validates the CRC of the received packet. The CRC of the received packet is calculated and compared with the value present in the packet. If both values are same, CRC validation is successful.

Modbus_ascii_lrc_calculate	Modbus serial longitudinal Redundancy check calculation

【Format】

```
uint8_t Modbus_ascii_lrc_calculate(puint8_t pu8_mb_pkt,
                                   uint32_t u32_pkt_len);
```

【Parameter】

puint8_t	pu8_mb_pkt	Pointer to the Modbus packet array
uint32_t	u32_pkt_len	Length of the Modbus packet

【Return value】

uint32_t	Calculated LRC value

【Error code】

—

【Explanation】

This function calculates the LRC of the packet.

Modbus_ascii_lrc_validate	Modbus serial longitudinal Redundancy check validation

【Format】

```
uint32_t Modbus_ascii_lrc_validate(puint8_t pu8_mb_pkt,
                                   uint32_t u32_pkt_len);
```

【Parameter】

puint8_t	pu8_mb_pkt	Pointer to the Modbus packet array
uint32_t	u32_pkt_len	Length of the Modbus packet

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On validation of LRC is successful
ERR_LRC_CHECK	If LRC validation fails for ASCII stack mode

【Explanation】

This function validates the LRC of the received packet. The LRC of the received packet is calculated and compared with the value present in the packet. If both values are same, LRC validation is successful.

Modbus_rtu_to_ascii	Modbus RTU to ASCII Conversion

【Format】

```
void_t Modbus_rtu_to_ascii(puint8_t pu8_rtu_pkt,
                           uint32_t u32_rtu_pkt_size,
                           puint8_t pu8_ascii_pkt,
                           puint32_t pu32_ascii_pkt_size);
```

【Parameter】

puint8_t	pu8_rtu_pkt	Pointer to the input RTU array
uint32_t	u32_rtu_pkt_size	Number of bytes in the in the input RTU array
puint8_t	pu8_ascii_pkt	Pointer to the output ASCII array
puint32_t	pu32_ascii_pkt_size	Pointer to return number of bytes in the in the output ASCII array

【Return value】

void_t

【Error code】

—

【Explanation】

This function converts the modbus PDU in hex form to its ASCII values.

Modbus_ascii_to_rtu	Modbus ASCII to RTU Conversion

【Format】

```
void_t Modbus_ascii_to_rtu(puint8_t pu8_ascii_pkt,
                           uint32_t u32_ascii_pkt_size,
                           puint8_t pu8_rtu_pkt,
                           puint32_t pu32_rtu_pkt_size);
```

【Parameter】

puint8_t	pu8_ascii_pkt	Pointer to the input ASCII array
uint32_t	u32_ascii_pkt_size	Number of bytes in the input ASCII array
puint8_t	pu8_rtu_pkt	Pointer to return the output RTU array
puint32_t	pu32_rtu_pkt_size	Pointer to return the number of bytes in the in the output RTU array

【Return value】

void_t

【Error code】

—

【Explanation】

This function converts the array of ASCII values to its equivalent hex values.

Modbus_RS485_TX_enable	RS485 Transmit enable

【Format】

```
void_t Modbus_RS485_TX_enable( void_t );
```

【Parameter】

void_t

【Return value】

void_t

【Error code】

—

【Explanation】

This function switches RS485 transceiver to transmission mode.

Modbus_RS485_TX_disable	RS485 Transmit disable

【Format】

```
void_t Modbus_RS485_TX_enable( void_t );
```

【Parameter】

void_t

【Return value】

void_t

【Error code】

—

【Explanation】

This function switches RS485 transceiver to reception mode.

Modbus_ascii_recv_char	Receive character for Modbus ASCII

【Format】

void_t Modbus_ascii_recv_char(uint8_t u8_read_char);

【Parameter】

uint8_t	u8_read_char	received character

【Return value】

void_t

【Error code】

—

【Explanation】

This function is responsible for buffering of the received data to Modbus ASCII mode. Buffering is done until process detects termination character or process gets the maximum number of characters (MAX_ASCII_PACKET_LEN). Upon detecting the termination character, the packet to each task depending on the stack mode will report to the effect that could be received.

When this function is invoked, the timer is started by specified Inter frame delay at stack initialization in order to measure non-communicate time.

Modbus_rtu_recv_char	Receive character for Modbus RTU

【Format】

void_t Modbus_rtu_recv_char(uint8_t u8_recv_char)

【Parameter】

uint8_t	u8_recv_char	received character

【Return value】

void_t

【Error code】

—

【Explanation】

This function is responsible for buffering of the received data to Modbus RTU mode. Buffering is done until process gets the maximum number of characters (MAX_RTU_PACKET_LEN). Termination decision of packets is done in the timer handler to detect the non-communication time.

When this function is invoked, the timer is started by specified Inter frame delay at stack initialization in order to measure non-communicate time.

Modbus_timer_handler	Timer handler

【Format】

void Modbus_timer_handler(void);

【Parameter】

void_t

【Return value】

void_t

【Error code】

—

【Explanation】

This function is invoked when the timer interrupt event has occurred in the serial data receive task.

For ASCII mode, reset the buffering process of the receiving data. If this function is invoked before the end character is detected, packet will be discarded.

For RTU mode, Stop buffering of the received data, according to the stack mode, it reports that the packet reception has been completed to each task.

10.2.1.2 TCP/IP Connection Management

The following API has been used in the TCP/IP packet processing.

Modbus_tcp_send	Modbus TCP send packet

【Format】

```
uint32_t Modbus_tcp_send(puint8_t pu8_mb_snd_pkt,
                         uint32_t u32_snd_pkt_len,
                         uint8_t u8_soc_id);
```

【Parameter】		
puint8_t	pu8_mb_snd_pkt	Pointer to the Modbus send packet array
uint32_t	u32_snd_pkt_len	Length of the Modbus send packet
uint8_t	u8_soc_id	Socket ID to which the data is to be transmitted

【Return value】	
uint32_t	Error code

【Error code】	
ERR_OK	On successfully send the packet
ERR_SEND_FAIL	If packet sending failed

【Explanation】

Modbus function packets are transmitted using this API. The actual sending process is performed by the Modbus_tcp_send_pkt function.

Modbus_tcp_send_pkt	Modbus TCP send packet

【Format】

```
uint32_t Modbus_tcp_send_pkt(puint8_t pu8_mb_snd_pkt,
                             uint32_t u32_snd_pkt_len,
                             uint8_t u8_soc_id);
```

【Parameter】		
puint8_t	pu8_mb_snd_pkt	Pointer to the Modbus send packet array
uint32_t	u32_snd_pkt_len	Length of the Modbus send packet
uint8_t	u8_soc_id	Socket ID to which the data is to be transmitted

【Return value】	
uint32_t	Error code

【Error code】	
ERR_OK	On successfully send the packet
ERR_SEND_FAIL	If packet sending failed

【Explanation】

This API writes the specified packet to a connected socket. TCP / IP stack API is used for writing.

Modbus_tcp_recv	Modbus TCP receive packet

【Format】

```
uint32_t Modbus_tcp_recv(puint8_t pu8_mb_recv_pkt,
                         uint32_t u32_recv_pkt_len,
                         uint8_t u8_soc_id);
```

【Parameter】

puint8_t	pu8_mb_recv_pkt	Receive buffer pointer
uint32_t	u32_recv_pkt_len	Receive buffer size
uint8_t	u8_soc_id	Socket ID

【Return value】

uint32_t	Receive packet size

【Error code】

—

【Explanation】

Modbus function packets are received using this API. The actual reception processing is performed by the Modbus_tcp_recv_pkt function.

Modbus_tcp_recv_pkt	Modbus TCP receive packet

【Format】

```
uint32_t Modbus_tcp_recv_pkt(puint8_t pu8_mb_recv_pkt,
                             uint32_t u32_recv_pkt_len,
                             uint8_t u8_soc_id);
```

【Parameter】

puint8_t	pu8_mb_recv_pkt	Receive buffer pointer
uint32_t	u32_recv_pkt_len	Receive buffer size
uint8_t	u8_soc_id	Socket ID

【Return value】

uint32_t	Receive packet size

【Error code】

—

【Explanation】

This API uses the unet3_recv () function to receive Modbus function packets from the communication partner.

Modbus_tcp_frame_pkt	Modbus TCP frame packet

【Format】

```
void_t Modbus_tcp_frame_pkt(puint8_t pu8_mb_snd_pkt,
                            puint32_t pu32_snd_pkt_len,
                            p_mb_tcp_pkt_info_t pt_mb_tcp_pkt_info);
```

【Parameter】

puint8_t	pu8_mb_snd_pkt	Pointer to the array storing packet to be send
puint32_t	pu32_snd_pkt_len	Length of the packet framed
p_mb_tcp_pkt_info_t	pt_mb_tcp_pkt_info	Structure pointer containing response information

【Return value】

void_t

【Error code】

—

【Explanation】

This function is used to update TCP packet information structure from response structure provided by the user application.

Modbus_tcp_parse_pkt	Modbus TCP parse packet

【Format】

```
uint32_t Modbus_tcp_parse_pkt(puint8_t pu8_mb_rcv_pkt,
                              uint32_t u32_recv_pkt_len,
                              p_mb_tcp_pkt_info_t pt_mb_tcp_pkt_info);
```

【Parameter】

puint8_t	pu8_mb_rcv_pkt	Pointer to the array storing the received packet
puint32_t	u32_recv_pkt_len	Length of the Modbus receive packet
p_mb_tcp_pkt_info_t	pt_mb_tcp_pkt_info	Structure pointer containing response information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful parsing the packet
EXP_ILLEGAL_DATA_VALUE	If data value is not in the valid range
ERR_ILLEGAL_FUNCTION	If function code is not supported or enabled
ERR_MEM_ALLOC	If memory allocation fails

【Explanation】

This API parse the specified packet, perform the callback function corresponding to each function code that the user has registered. After callback perform, API updates the structure of TCP packet information (pt_mb_tcp_pck_info) based on the execution results. In this function, dynamically allocate memory request and response table for each callback perform. Request table will be released within this function, but response table will be released at the stage of generating a response packet.

Modbus_tcp_validate_pkt	Modbus TCP validate packet

【Format】

```
uint32_t Modbus_tcp_validate_pkt(p_mb_tcp_pkt_info_t pt_mb_tcp_pkt_info,
                                 uint32_t u32_pdu_len);
```

【Parameter】

p_mb_tcp_pkt_info_t	pt_mb_tcp_pkt_info	Structure pointer containing request information
uint32_t	u32_pdu_len	Length of the PDU received

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful validation
EXP_ILLEGAL_DATA_VALUE	If data value is not in the valid range
ERR_ILLEGAL_FUNCTION	If function code is not supported or enabled

【Explanation】

This function validates a packet recevived by the TCP device.

Modbus_tcp_init_socket	function for creating server socket

【Format】

```
int8_t Modbus_tcp_init_socket(uint16_t u16_port,
                              pint32_t ps32_listen_fd);
```

【Parameter】

uint16_t	u16_port	Port number to which socket is to be bound
pint32_t	ps32_listen_fd	Socket descriptor bound

【Return value】

int8_t	Error code

【Error code】

ERR_OK	On successful completion
ERR_SOCK_ERROR	If socket creation fails
ERR_BIND_ERROR	If binding fails
ERR_LISTEN_ERROR	If listening fails

【Explanation】

This function is used for creating the server socket and turns the server to accept mode for monitoring client connections.

Modbus_tcp_frame_response	Modbus TCP frame response

【Format】

uint32_t Modbus_tcp_frame_response(uint8_t u8_fn_code,
p_mb_tcp_pkt_info_t pt_mb_tcp_pkt_info);

【Parameter】

uint8_t	u8_fn_code	Variable storing the function code
p_mb_tcp_pkt_info_t	pt_mb_tcp_pkt_info	Structure pointer containing user information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On framing packet is successful

【Explanation】

This function is used to update TCP packet information structure from response structure provided by the user application.

10.2.2 Stack Configuration and Management API

10.2.2.1 Initialization of protocol stack

The following API has been used in the initialization process of the stack.

Modbus_tcp_server_init_stack	Modbus TCP server (without gateway)stack initialization

【Format】

```
uint32_t Modbus_tcp_server_init_stack(uint32_t u32_additional_port,
                                      uint8_t u8_tcp_multiple_client);
```

【Parameter】		
uint32_t	u32_additional_port	Additional port value configured by user
uint8_t	u8_tcp_multiple_client	Status whether multiple client is enabled

【Return value】	
uint32_t	Error code

【Error code】	
ERR_OK	On successful initialization of the task or mailbox
ERR_STACK_INIT	If initialization of the task or mailbox failed

【Explanation】

This API is used to initialize the TCP stack. Specifically, this function to start the three tasks of the following required for the operation of the stack.

- The task of monitoring the connection from the client using the port number(default 502) that is specified by the user.
- The task of receiving the data sent from the client side.
- The task of analyzes the received data and performs an operation corresponding to each function code provided by the user.

Modbus_tcp_init_gateway_stack	Modbus TCP gateway initialization

【Format】

```
uint32_t Modbus_tcp_init_gateway_stack(uint8_t u8_stack_mode,
                                       uint8_t u8_tcp_gw_slave,
                                       p_serial_stack_init_info_t pt_serial_stack_init_info,
                                       p_serial_gpio_cfg_t        pt_serial_gpio_cfg_t);
```

【Parameter】

uint8_t	u8_stack_mode	Variable to store the stack mode RTU/ASCII
uint8_t	u8_tcp_gw_slave	Status whether gateway enabled as TCP server
p_serial_stack_init_info_t	pt_serial_stack_init_info	Structure pointer to serial stack initialization parameters
p_serial_gpio_cfg_t	pt_serial_gpio_cfg_t	Pointer to the structure with hardware configuration parameters

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful initialization of the task or mailbox
ERR_STACK_INIT	If initialization of the task or mailbox failed

【Explanation】

This API is used to initialize the stack with gateway functionality. Initialize Modbus TCP stack along with the serial stack. Activate a gateway task to process the request for serial devices connected to the TCP device. Create a mailbox to communicate with this task.

10.2.2.2 IP managementt

The following API has been used in the IP management.

Modbus_tcp_search_ip_addr	Modbus search a host IP

【Format】

```
uint32_t Modbus_tcp_search_ip_addr(pchar_t pu8_search_IP,
                                   puint8_t pu8_ip_idx);
```

【Parameter】		
pchar_t	pu8_search_IP	IP address to be searched
puint8_t	pu8_ip_idx	Index at which IP address is placed
【Return value】		
uint32_t	Error code	
【Error code】		
ERR_OK	On successful search	
ERR_IP_NOT_FOUND	If IP is not in the list	
ERR_TABLE_EMPTY	If the list is empty	
ERR_TABLE_DISABLED	If the table is disabled, i.e., server accepts connection request from any host	

【Explanation】

This function is used for search an IP from the host IP list given.

Modbus_tcp_shift_conn_list	Modbus shift TCP connection list

【Format】

```
void_t Modbus_tcp_shift_conn_list(puint8_t pu8_conn_list,
                                  puint8_t pu8_conn_idx);
```

【Parameter】		
puint8_t	pu8_conn_list	Pointer to array containing the connection list
puint8_t	pu8_conn_idx	Index from which the socket ID is to be shifted
【Return value】		
void_t		
【Error code】		
—		

【Explanation】

This API is used to shift the connection list according to the latest active connection.

Modbus_tcp_remove_from_conn_list	Modbus TCP remove from connection list

【Format】

```
void_t Modbus_tcp_remove_from_conn_list(puint8_t pu8_soc_id,
                                        puint8_t pu8_conn_list);
```

【Parameter】

puint8_t	pu8_soc_id	Variable storing the socket ID.
puint8_t	pu8_conn_list	Pointer to array containing the connection list.

【Return value】

void_t

【Error code】

—

【Explanation】

This API is used to remove a connection established, from the connection list kept by the server. Verify the location at which the socket ID is specified and shift the all connection by one.

Modbus_tcp_add_to_conn_list	Modbus add a connection to Modbus TCP connection list

【Format】

```
void_t Modbus_tcp_add_to_conn_list(puint8_t pu8_soc_id,
                                   puint8_t pu8_conn_list);
```

【Parameter】

puint8_t	pu8_soc_id	Variable storing the socket ID
puint8_t	pu8_conn_list	Pointer to array containing the connection list

【Return value】

void_t

【Error code】

—

【Explanation】

This API is used to add a new connection received by the TCP server to the connection list kept by the server. Append the new connection to the array one by one.

Modbus_tcp_get_loc_from_list	Modbus get location of a connection

【Format】

```
uint32_t Modbus_tcp_get_loc_from_list(puint8_t pu8_soc_id,
                                      puint8_t pu8_conn_list,
                                      puint8_t pu8_soc_loc);
```

【Parameter】

puint8_t	pu8_soc_id	Variable storing the socket ID
puint8_t	pu8_conn_list	Pointer to array containing the connection list
puint8_t	pu8_soc_loc	Pointer variable storing the socket location referenced in the connection list

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If the location of the socket ID obtained
ERR_SOC_NOT_FOUND	If referenced socket not present in the connection array list

【Explanation】

This API is used to obtain the location of a socket ID referenced from the MODBUS TCP connection list.

Modbus_tcp_update_conn_list	Modbus update TCP connection list

【Format】

```
void_t Modbus_tcp_update_conn_list(uint8_t u8_soc_id,
                                   puint8_t pu8_conn_list,
                                   uint8_t u8_add_remove);
```

【Parameter】

uint8_t	u8_soc_id	Variable storing the socket ID
puint8_t	pu8_conn_list	Pointer to array containing the connection list
uint8_t	u8_add_remove	Update type

【Return value】

void_t

【Error code】

—

【Explanation】

This API is used to update the connection list with the latest connection. The latest connection should be placed at the last of the array and the oldest connection is placed initially. In this function, the following macro is used as an argument.

Update type	Meaning
ADD_TO_CONN_LIST	Add socket ID to the connection list
REMOVE_FROM_CONN_LIST	Remove socket ID from the connection list

10.2.2.3 Task terminate

The following API has been used in the task termination processing.

Modbus_tcp_server_terminate_stack	Modbus terminate TCP Server stack API

【Format】

uint32_t Modbus_tcp_server_terminate_stack(void_t)

【Parameter】

void_t

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful termination
ERR_STACK_TERM	If termination failed

【Explanation】

This API terminate Modbus TCP stack related task and the mailbox used for the TCP task.

Modbus_tcp_gateway_terminate_stack	Modbus terminate TCP gateway stack API

【Format】

uint32_t Modbus_tcp_gateway_terminate_stack(void_t)

【Parameter】

void_t

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful termination
ERR_STACK_TERM	If termination failed

【Explanation】

This API terminate Modbus TCP gateway stack related task and the mailbox used for the TCP gateway task.

10.2.2.4 Mailbox

The following API has been used in the mailbox management.

Modbus_post_to_mailbox	Modbus post a request to the mailbox

【Format】

```
uint32_t Modbus_post_to_mailbox(uint16_t u16_mbx_id,
                                p_mb_req_mbx_t pt_req_recvd);
```

【Parameter】

uint16_t	u16_mbx_id	Variable containing the mailbox ID to which request is to be posted
p_mb_req_mbx_t	pt_req_recvd	Pointer to the structure containing request information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful write to mailbox
ERR_MAILBOX	If write of mailbox failed
ERR_TCP_SND_MBX_FULL	If mailbox is full

【Explanation】

This API is used to send the request received from the client to the receive mailbox or gateway mailbox. Increment the number of elements in mailbox if the request sent successfully.

・Structure of mailbox queue (mb_req_mbx_t)

```
typedef struct _req_mbx{
    uint32_t    u32_soc_id;     /* Socket ID at which the request arrived */
    puint8_t    pu8_req_pkt;    /* Pointer to the requested packet */
    uint32_t    u32_pkt_len;    /* Packet length */
}mb_req_mbx_t, *p_mb_req_mbx_t;
```

Modbus_fetch_from_mailbox	Modbus read a request from the mailbox.

【Format】

```
uint32_t Modbus_fetch_from_mailbox(uint16_t u16_mbx_id,
                                   p_mb_req_mbx_t* pt_req_recvd);
```

【Parameter】		
uint16_t	u16_mbx_id	Variable containing the mailbox ID to which request is read
p_mb_req_mbx_t	pt_req_recvd	Pointer to the structure containing request information

【Return value】	
uint32_t	Error code

【Error code】	
ERR_OK	On successful read from mailbox
ERR_MAILBOX	If read from mailbox failed

【Explanation】

This API is used to read the request posted to the receive mailbox or gateway mailbox depending upon the mailbox ID specified. Decrement the number of elements in mailbox if the request read successfully.

Modbus_check_mailbox	Modbus verify the number of elements in mailbox.

【Format】

```
uint32_t Modbus_check_mailbox(uint16_t u16_mbx_id);
```

【Parameter】		
uint16_t	u16_mbx_id	Variable containing the mailbox ID to which request is read.

【Return value】	
uint32_t	Number of mailbox used or error code

【Error code】	
ERR_TCP_SND_MBX_FULL	If mailbox is full

【Explanation】

This API is used to verify the number of elements in mailbox. The maximum number that can be processed by each mailbox is defined by the following macros. If the number of messages being processed has reached the maximum value, it is determined that the message full.

Macro name	Meaning
MAX_RCV_MBX_SIZE	Maximum number of receive mailbox
MAX_GW_MBX_SIZE	Maximum number of gateway mailbox

Modbus_delete_mailbox	Modbus Delete mailbox

【Format】

uint32_t Modbus_delete_mailbox(uint16_t u16_mbx_id);

【Parameter】

uint16_t	u16_mbx_id	Variable storing the mailbox ID.

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On successful deletion of mailbox
ERR_MAILBOX	If deletion of mailbox failed

【Explanation】

This API is used to delete a mailbox specified by the mailbox ID.

10.2.3 Gateway mode API

This chapter describes the function that will be called from the gateway task.

Modbus_gw_read_coils	Modbus gateway function to read the coil

【Format】

```
uint32_t Modbus_gw_read_coils(puint8_t pu8_recvd_pkt,
                              uint32_t u32_recv_pkt_len,
                              p_mb_tcp_pkt_info_t pt_gw_tcp_pkt_info);
```

【Parameter】

puint8_t	pu8_recvd_pkt	Pointer to the array storing the received packet.
uint32_t	u32_recv_pkt_len	Length of received packet
p_mb_tcp_pkt_info_t	pt_gw_tcp_pkt_info	Structure to fill with the response information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On read coil successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_COILS	If the number of coils provided is not within the specified limit
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If the slave id in the request is not its own slave id or broadcast id
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_FUN_CODE_MISMATCH	If the function code is not supported by the stack
ERR_ILLEGAL_FUNCTION	If function code is not supported or enabled
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_INSUFFICIENT_DATA	If receive packet length is invalid

【Explanation】

This API invokes the function provided in the master to read the data of the coil. In this function, memory is allocated for both request and response structure dynamically and fill the request structure with the information in the received packet. Invoke the serial master API, frame a packet with the response information obtained and send the response packet to the client. After that, the memory allocated for both response and request structures to be freed.

・Structure of response information (mb_tcp_pkt_info_t)

```
typedef struct _mb_tcp_pkt_info{
    puint8_t    pu8_output_response;          /* pointer to the response structure from the API */
    uint16_t    u16_transaction_id;           /* specifies the transaction identifier */
    uint16_t    u16_protocol_id;              /* specifies the protocol ID */
    uint8_t     u8_slave_id;                  /* specifies the slave ID of the device */
    uint16_t    u16_num_of_bytes;             /* specifies the number of bytes in the data packet
                                                 PDU field */
    uint8_t     aru8_data_packet[MAX_DATA_SIZE]; /* contains the function and data */
}mb_tcp_pkt_info_t, *p_mb_tcp_pkt_info_t;
```

Modbus_gw_read_discrete_inputs	Modbus gateway function to read the discrete input

【Format】

```
uint32_t Modbus_gw_read_discrete_inputs(puint8_t pu8_recvd_pkt,
                                        uint32_t u32_recv_pkt_len,
                                        p_mb_tcp_pkt_info_t pt_gw_tcp_pkt_info);
```

【Parameter】

puint8_t	pu8_recvd_pkt	Pointer to the array storing the received packet
uint32_t	u32_recv_pkt_len	Length of received packet
p_mb_tcp_pkt_info_t	pt_gw_tcp_pkt_info	Structure to fill with the response information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	On discrete input read successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_INPUTS	If the number of inputs provided is not within the specified limit
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If the slave id in the request is not its own slave id or broadcast id
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_FUN_CODE_MISMATCH	If the function code is not supported by the stack
ERR_ILLEGAL_FUNCTION	If function code is not supported or enabled
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_INSUFFICIENT_DATA	If receive packet length is invalid

【Explanation】

This API invokes the function provided in the master to read the data of the discrete input. In this function, memory is allocated for both request and response structure dynamically and fill the request structure with the information in the received packet. After that, the memory allocated for both response and request structures to be freed.

Modbus_gw_read_holding_regs	Modbus gateway function to read the holding register

【Format】

```
uint32_t Modbus_gw_read_holding_regs(puint8_t pu8_recvd_pkt,
                                     uint32_t u32_recv_pkt_len,
                                     p_mb_tcp_pkt_info_t pt_gw_tcp_pkt_info);
```

【Parameter】

puint8_t	pu8_recvd_pkt	Pointer to the array storing the received packet.
uint32_t	u32_recv_pkt_len	Length of received packet
p_mb_tcp_pkt_info_t	pt_gw_tcp_pkt_info	Structure to fill with the response information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If holding register read successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_REG	If the number of registers provided is not within the specified limit
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If the slave id in the request is not its own slave id or broadcast id
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_FUN_CODE_MISMATCH	If the function code is not supported by the stack
ERR_ILLEGAL_FUNCTION	If function code is not supported or enabled
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_INSUFFICIENT_DATA	If receive packet length is invalid

【Explanation】

This API invokes the function provided in the master to read the data of the holding register. In this function, memory is allocated for both request and response structure dynamically and fill the request structure with the information in the received packet. After that, the memory allocated for both response and request structures to be freed.

Modbus_gw_read_input_regs	Modbus gateway function to read the input register

【Format】

```
uint32_t Modbus_gw_read_input_regs(puint8_t pu8_recvd_pkt,
                                   uint32_t u32_recv_pkt_len,
                                   p_mb_tcp_pkt_info_t pt_gw_tcp_pkt_info);
```

【Parameter】

puint8_t	pu8_recvd_pkt	Pointer to the array storing the received packet
uint32_t	u32_recv_pkt_len	Length of received packet
p_mb_tcp_pkt_info_t	pt_gw_tcp_pkt_info	Structure to fill with the response information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If input register read successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_REG	If the number of registers provided is not within the specified limit
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If the slave id in the request is not its own slave id or broadcast id
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_FUN_CODE_MISMATCH	If the function code is not supported by the stack
ERR_ILLEGAL_FUNCTION	If function code is not supported or enabled
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_INSUFFICIENT_DATA	If receive packet length is invalid

【Explanation】

This API invokes the function provided in the master to read the data of the input register. In this function, memory is allocated for both request and response structure dynamically and fill the request structure with the information in the received packet. After that, the memory allocated for both response and request structures to be freed.

Modbus_gw_write_single_coil	Modbus gateway function to write a single coil

【Format】

```
uint32_t Modbus_gw_write_single_coil(puint8_t pu8_recvd_pkt,
                                     uint32_t u32_recv_pkt_len,
                                     p_mb_tcp_pkt_info_t pt_gw_tcp_pkt_info);
```

【Parameter】

puint8_t	pu8_recvd_pkt	Pointer to the array storing the received packet
uint32_t	u32_recv_pkt_len	Length of received packet
p_mb_tcp_pkt_info_t	pt_gw_tcp_pkt_info	Structure to fill with the response information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If single coil write is successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_OUTPUT_VALUE	If the value of the registers is invalid
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If the slave id in the request is not its own slave id or broadcast id
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_FUN_CODE_MISMATCH	If the function code is not supported by the stack
ERR_ILLEGAL_FUNCTION	If function code is not supported or enabled
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_INSUFFICIENT_DATA	If receive packet length is invalid

【Explanation】

This API invokes the function provided in the master to write a single coil. In this function, memory is allocated for both request and response structure dynamically and fill the request structure with the information in the received packet. After that, the memory allocated for both response and request structures to be freed.

Modbus_gw_write_single_reg	Modbus gateway function to write a single register

【Format】

```
uint32_t Modbus_gw_write_single_reg(puint8_t pu8_recvd_pkt,
                                    uint32_t u32_recv_pkt_len,
                                    p_mb_tcp_pkt_info_t pt_gw_tcp_pkt_info);
```

【Parameter】

puint8_t	pu8_recvd_pkt	Pointer to the array storing the received packet
uint32_t	u32_recv_pkt_len	Length of received packet
p_mb_tcp_pkt_info_t	pt_gw_tcp_pkt_info	Structure to fill with the response information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If single register write is successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If the slave id in the request is not its own slave id or broadcast id
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_FUN_CODE_MISMATCH	If the function code is not supported by the stack
ERR_ILLEGAL_FUNCTION	If function code is not supported or enabled
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_INSUFFICIENT_DATA	If receive packet length is invalid

【Explanation】

This API invokes the function provided in the master to write a single register. In this function, memory is allocated for both request and response structure dynamically and fill the request structure with the information in the received packet. After that, the memory allocated for both response and request structures to be freed.

Modbus_gw_write_multiple_coils	Modbus gateway function to write multiple coils

【Format】

```
uint32_t Modbus_gw_write_multiple_coils(puint8_t pu8_recvd_pkt,
                                        uint32_t u32_recv_pkt_len,
                                        p_mb_tcp_pkt_info_t pt_gw_tcp_pkt_info);
```

【Parameter】

puint8_t	pu8_recvd_pkt	Pointer to the array storing the received packet
uint32_t	u32_recv_pkt_len	Length of received packet
p_mb_tcp_pkt_info_t	pt_gw_tcp_pkt_info	Structure to fill with the response information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If multiple coils write is successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_OUTPUTS	If the number of outputs is invalid
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If the slave id in the request is not its own slave id or broadcast id
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_FUN_CODE_MISMATCH	If the function code is not supported by the stack
ERR_ILLEGAL_FUNCTION	If function code is not supported or enabled
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_INSUFFICIENT_DATA	If receive packet length is invalid

【Explanation】

This API invokes the function provided in the master to write a data to multiple coils. In this function, memory is allocated for both request and response structure dynamically and fill the request structure with the information in the received packet. After that, the memory allocated for both response and request structures to be freed.

Modbus_gw_write_multiple_reg	Modbus gateway function to write multiple registers

【Format】

```
uint32_t Modbus_gw_write_multiple_reg(puint8_t pu8_recvd_pkt,
                                      uint32_t u32_recv_pkt_len,
                                      p_mb_tcp_pkt_info_t pt_gw_tcp_pkt_info);
```

【Parameter】

puint8_t	pu8_recvd_pkt	Pointer to the array storing the received packet
uint32_t	u32_recv_pkt_len	Length of received packet
p_mb_tcp_pkt_info_t	pt_gw_tcp_pkt_info	Structure to fill with the response information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If single register write is successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_REG	If the number of registers is invalid
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If the slave id in the request is not its own slave id or broadcast id
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_FUN_CODE_MISMATCH	If the function code is not supported by the stack
ERR_ILLEGAL_FUNCTION	If function code is not supported or enabled
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_INSUFFICIENT_DATA	If receive packet length is invalid

【Explanation】

This API invokes the function provided in the master to write a data to multiple registers. In this function, memory is allocated for both request and response structure dynamically and fill the request structure with the information in the received packet. After that, the memory allocated for both response and request structures to be freed.

Modbus_gw_read_write_multiple_reg	Modbus gateway function to read/write multiple registers

【Format】

```
uint32_t Modbus_gw_read_write_multiple_reg(puint8_t pu8_recvd_pkt,
                                           uint32_t u32_recv_pkt_len,
                                           p_mb_tcp_pkt_info_t pt_gw_tcp_pkt_info);
```

【Parameter】

puint8_t	pu8_recvd_pkt	Pointer to the array storing the received packet
uint32_t	u32_recv_pkt_len	Length of received packet
p_mb_tcp_pkt_info_t	pt_gw_tcp_pkt_info	Structure to fill with the response information

【Return value】

uint32_t	Error code

【Error code】

ERR_OK	If read/write multiple register is successful
ERR_SYSTEM_INTERNAL	For mailbox send or receive failure
ERR_ILLEGAL_OUTPUT_VALUE	If the value of the registers is invalid
ERR_INVALID_SLAVE_ID	If the slave ID is not valid
ERR_MEM_ALLOC	If memory allocation fails
ERR_SLAVE_ID_MISMATCH	If the slave id in the request is not its own slave id or broadcast id
ERR_CRC_CHECK	If CRC validation fails for RTU slave stack mode
ERR_LRC_CHECK	If LRC validation fails for ASCII slave stack mode
ERR_FUN_CODE_MISMATCH	If the function code is not supported by the stack
ERR_ILLEGAL_FUNCTION	If function code is not supported or enabled
ERR_ILLEGAL_DATA_VALUE	If data value given is invalid
ERR_INSUFFICIENT_DATA	If receive packet length is invalid

【Explanation】

This API invokes the function provided in the master to read/write a data from/to multiple registers. In this function, memory is allocated for both request and response structure dynamically and fill the request structure with the information in the received packet. After that, the memory allocated for both response and request structures to be freed.

10.3 Error Codes

Along with the response, the error code is also mentioned to inform the user application about the command processing status. For this different error codes are generated while processing the request/response. Following are the different error codes used:

Table 10.1 Description for each error

Error Code	Value	Description
ERR_OK	0x00	Specifies success code
ERR_ILLEGAL_FUNCTION	0x01	Specifies the function code received in the request is not an allowable action for the server (or slave) or the function code is not implemented. This value must be a constant, cannot change the value from 0x01.
ERR_ILLEGAL_DATA_ADDRESS	0x02	Specifies the data address received in the request is not an allowable address for the server (or slave) or the addressed register is not implemented. This value must be a constant, cannot change the value from 0x02
ERR_ILLEGAL_DATA_VALUE	0x03	Specifies a value contained in the request data field is not an allowable value for the server (or slave). This value must be a constant, cannot change the value from 0x03.
ERR_SLAVE_DEVICE_FAILURE	0x04	Specifies an unrecoverable error occurred while the server (or slave) was attempting to perform the requested action. This value must be a constant, cannot change the value from 0x04.
ERR_STACK_INIT	0x05	In stack initialization failure
ERR_ILLEGAL_SERV_BSY	0x06	Specifies the maximum transaction reached. This value must be a constant, cannot change the value from 0x06
ERR_CRC_CHECK	0x07	Specifies the CRC check has failed
ERR_LRC_CHECK	0x08	Specifies the LRC check has failed
ERR_INVALID_SLAVE_ID	0x09	Specifies the slave ID is invalid
ERR_TCP_SND_MBX_FULL	0x0A	Specifies that the mailbox is full
ERR_STACK_TERM	0x0B	In stack termination failure
ERR_TIME_OUT	0x0C	Timeout error added
ERR_MEM_ALLOC	0x0D	Memory allocation failure
ERR_SYSTEM_INTERNAL	0x0E	Mailbox send or receive failure
ERR_ILLEGAL_NUM_OF_COILS	0x0F	Specifies the number of coils provided is not within the specified limit
ERR_ILLEGAL_NUM_OF_INPUTS	0x10	Specifies the number of inputs provided is not within the specified limit
ERR_ILLEGAL_NUM_OF_REG	0x11	Specifies the number of registers provided is not within the specified limit
ERR_ILLEGAL_OUTPUT_VALUE	0x12	Specifies the value of the registers is invalid
ERR_ILLEGAL_NUM_OF_OUTPUTS	0x13	Specifies the number of outputs is invalid
ERR_INVALID_STACK_INIT_PARAMS	0x14	Specifies invalid stack init information from user
ERR_INVALID_STACK_MODE	0x15	Stack mode specified is invalid
ERR_FUN_CODE_MISMATCH	0x16	Master receives a response for another function code(not for the requested function code)
ERR_SLAVE_ID_MISMATCH	0x17	Master receives a response from another slave (not from the requested slave)
ERR_OK_WITH_NO_RESPONSE	0x18	Return status for broadcast requests
ERR_MSG_SIZE_OVER	0x19	Received request or response data size exceeds maximum length

11. Implementation

This chapter explains the software implementation procedure.

11.1 Modbus TCP

It's explained to Modbus TCP stack in this chapter. In carrying out the implementation of Modbus TCP, TCP / IP protocol stack also must be implement.

Please refer to “programming manual (TCP/IP edition)” for implementation of TCP/IP protocol stack.

11.1.1 Server mode

The following are the items required when using the slave mode.

(1) Task ID definition

To use the following API as a task, and a Task ID defined in any value.

Task API	Task ID	Function
Modbus_tcp_soc_wait_task	ID_CONN_TASK	Wait for TCP connection task
Modbus_tcp_recv_data_task	ID_RECV_SOC	TCP receive data Task
Modbus_tcp_req_process_task	ID_SERV_TSK	TCP request processing task

(2) Mailbox ID definition

The following Mailbox ID is required.

Mailbox ID	Meaning
ID_MB_TCP_RECV_MBX	TCP receive mailbox

(3) Initialization of Modbus stack

It performs various initialization, and then start the Modbus stack. This initialization needs to execute after the initialization of the TCP / IP protocol stack.

TCP server mode must perform the following by each APIs.

- Register callback functions corresponding to each the function code
- Initialize Modbus routine and start up related task

Please refer to Chapter 10.1.1.1 for specification of each APIs.
Basically initialization is as follows:

```
/* register callback functions */
st_slave_map.fp_function_code1  = cb_func_code01;  /* Read Coils operation */
st_slave_map.fp_function_code2  = cb_func_code02;  /* Read Discrete Inputs operation */
st_slave_map.fp_function_code3  = cb_func_code03;  /* Read Holding Registers operation */
st_slave_map.fp_function_code4  = cb_func_code04;  /* Read Input Registers operation */
st_slave_map.fp_function_code5  = cb_func_code05;  /* Write Single Coil operation */
st_slave_map.fp_function_code6  = cb_func_code06;  /* Write Single Register operation */
st_slave_map.fp_function_code15 = cb_func_code15;  /* Write Multiple Coils operation */
st_slave_map.fp_function_code16 = cb_func_code16;  /* Write Multiple Registers operation */
st_slave_map.fp_function_code23 = cb_func_code23;  /* Read/Write Multiple Registers operation */
Modbus_slave_map_init(&st_slave_map);

/* Initialize MODBUS stack by TCP server mode */
ercd = Modbus_tcp_init_stack(MODBUS_TCP_SERVER_MODE,
MODBUS_TCP_GW_SLAVE_DISABLE,
ENABLE_MULTIPLE_CLIENT_CONNECTION,
0,
NULL,
NULL);
```

(4) Implement call back functions

If the function code is instructed to implements the callback function for performing.

Please refer to the item of Section 10.1.1.1 of the Modbus_slave_map_init API. Interface specification of the callback function has been described.

11.1.2 Gateway mode

Gateway mode is the structure that connects Modbus Serial and Modbus TCP by gateway task. The following are the items required when using the gateway mode.

(1) Task ID definition

To use the following API as a task, and a Task ID defined in any value.

Task API	Task ID	function
Modbus_gateway_task	ID_CONN_TASK	TCP server⇔Serial device Gateway task
Modbus_tcp_soc_wait_task	ID_RECV_SOC	Wait for connection task
Modbus_tcp_recv_data_task	ID_SERV_TSK	TCP receive data Task
Modbus_tcp_req_process_task	ID_GATEWAY_TSK	TCP request processing task
Modbus_serial_recv_task	ID_MB_SERIAL_TSK	Serial receive data task
Modbus_serial_task	ID_MB_SERIAL_RECV_TSK	Serial request processing task
Serial_recv_task	ID_SERIAL_TSK	Serial receive data task

(2) Event flag ID definition

The following Event flag ID is required.

Event flag	Meaning
ID_FLG_SERIAL	Timer and UART interrupt event
ID_FLG_RESP_RDY	Response event in Blocking mode
ID_SERIAL_RESP	Recveive response event

(3) Mailbox ID definition

The following Mailbox ID is required.

Mailbox ID	Meaning
ID_MB_GATEWAY_MBX	Receive gateway process mailbox
ID_MB_TCP_RECV_MBX	TCP receive mailbox
ID_MB_SERIAL_MBX	Serial event mailbox

(4) Initialization of Modbus stack

This is the initialization as well as the TCP server mode, but with the following exceptions.

- No registration by Modbus_slave_map_init().(But, in order to ensure the internal memory, that needs to be executed.)
- Initialize in gateway mode the Modbus_tcp_init_stack, to add a set of serial communication.

Please refer to Chapter 10.1.1.1 for specification of each APIs.
Basically initialization is as follows:

```
/* serial connection setting */
st_init_info.u32_baud_rate            = MODBUS_BAUDRATE;
st_init_info.u8_parity                = MODBUS_PARITY;
st_init_info.u8_stop_bit              = MODBUS_STOPBITS;
st_init_info.u8_uart_channel          = SCI_CH;
st_init_info.u32_response_timeout_ms   = 2000;
st_init_info.u32_turnaround_delay_ms   = 200;
st_init_info.u32_interframe_timeout_us = INTER_FRAME_DELAY_TIMEOUT_US(MODBUS_BAUDRATE);
st_init_info.u32_interchar_timeout_us  = INTER_CHAR_DELAY_TIMEOUT_US(MODBUS_BAUDRATE);
st_init_info.u8_retry_count           = 3;

/* register functions that performs RS485 direction control */
st_gpio_cfg.fp_gpio_init_ptr  = gpio_init;
st_gpio_cfg.fp_gpio_set_ptr   = gpio_set;
st_gpio_cfg.fp_gpio_reset_ptr = gpio_reset;

/* register callback functions(only memory allocation) */
Modbus_slave_map_init(&st_slave_map);

/* Initialize MODBUS stack by TCP server mode */
ercd = Modbus_tcp_init_stack(MODBUS_RTU_MASTER_MODE,
                             MODBUS_TCP_GW_SLAVE_ENABLE,
                             ENABLE_MULTIPLE_CLIENT_CONNECTION,
                             0,
                             &st_init_info,
                             &st_gpio_cfg);
```

11.2 Modbus RTU/ASCII

It's explained to Modbus RTU/ASCII stack in this chapter.

11.2.1 Slave mode

The following are the items required when using the slave mode.

(1) Task ID definition

To use the following API as a task, and a Task ID defined in any value.

Task API	Task ID	function
Modbus_serial_recv_task	ID_MB_SERIAL_TSK	Serial receive data task
Modbus_serial_task	ID_MB_SERIAL_RECV_TSK	Serial request processing task
Serial_recv_task	ID_SERIAL_TSK	Serial receive data task

(2) Event flag ID definition

The following Event flag ID is required.

Event flag	Meaning
ID_FLG_SERIAL	Timer and UART interrupt event
ID_FLG_RESP_RDY	Response event in Blocking mode
ID_SERIAL_RESP	Recveive response event

(3) Mailbox ID definition

The following Mailbox ID is required.

Mailbox ID	Meaning
ID_MB_SERIAL_MBX	Serial event mailbox

(4) Initialization of Modbus stack

It performs various initialization, and then start the Modbus stack. Serial slave mode must perform the following by each APIs.

- Register callback functions corresponding to each the function code.
- Initialize MODBUS routine and start up related task. Since it also performs this initialization in the serial communication related settings, it set to match the master side.

Please refer to Chapter 10.1.2.1 for specification of each APIs.
Basically initialization is as follows:

```
/* register callback functions */
st_slave_map.fp_function_code1  = cb_func_code01;
st_slave_map.fp_function_code2  = cb_func_code02;
st_slave_map.fp_function_code3  = cb_func_code03;
st_slave_map.fp_function_code4  = cb_func_code04;
st_slave_map.fp_function_code5  = cb_func_code05;
st_slave_map.fp_function_code6  = cb_func_code06;
st_slave_map.fp_function_code15 = cb_func_code15;
st_slave_map.fp_function_code16 = cb_func_code16;
st_slave_map.fp_function_code23 = cb_func_code23;
Modbus_slave_map_init(&st_slave_map);

/* serial connection setting */
st_init_info.u32_baud_rate           = MODBUS_BAUDRATE;
st_init_info.u8_parity               = MODBUS_PARITY;
st_init_info.u8_stop_bit             = MODBUS_STOPBITS;
st_init_info.u8_uart_channel         = SCI_CH;
st_init_info.u32_response_timeout_ms   = 2000;
st_init_info.u32_turnaround_delay_ms   = 200;
st_init_info.u32_interframe_timeout_us = INTER_FRAME_DELAY_TIMEOUT_US(MODBUS_BAUDRATE);
st_init_info.u32_interchar_timeout_us  = INTER_CHAR_DELAY_TIMEOUT_US(MODBUS_BAUDRATE);
st_init_info.u8_retry_count          = 3;

/* register functions that performs RS485 direction control */
st_gpio_cfg.fp_gpio_init_ptr  = gpio_init;
st_gpio_cfg.fp_gpio_set_ptr   = gpio_set;
st_gpio_cfg.fp_gpio_reset_ptr = gpio_reset;

/* Initialize MODBUS stack by Serial mode */
ercd = Modbus_serial_stack_init(&st_init_info,
                        &st_gpio_cfg,
                        MODBUS_RTU_SLAVE_MODE,
                        1);
```

If ASCII mode is used, API argument will have to change from MODBUS_RTU_SLAVE_MODE to MODBUS_ASCII_SLAVE_MODE.

(5) Implement call back functions

If the function code is instructed to implements the callback function for performing.

Please refer to the item of Section 10.1.1.1 of the Modbus_slave_map_init API. Interface specification of the callback function has been described.

11.2.2 Master mode

Master mode uses the same OS resources as slave mode, so please refer to Chapter11.2.1. The following items are required to use the master mode.

(1) Initialization of Modbus stack

Initialized in master mode will be the only Modbus_serial_stack_init.
Please refer to Chapter 10.1.2.1for a description method of the API.
Basically initialization is as follows:

```
/* serial connection setting */
st_init_info.u32_baud_rate            = MODBUS_BAUDRATE;
st_init_info.u8_parity                = MODBUS_PARITY;
st_init_info.u8_stop_bit              = MODBUS_STOPBITS;
st_init_info.u8_uart_channel          = SCI_CH;
st_init_info.u32_response_timeout_ms   = 2000;
st_init_info.u32_turnaround_delay_ms   = 200;
st_init_info.u32_interframe_timeout_us = INTER_FRAME_DELAY_TIMEOUT_US(MODBUS_BAUDRATE);
st_init_info.u32_interchar_timeout_us  = INTER_CHAR_DELAY_TIMEOUT_US(MODBUS_BAUDRATE);
st_init_info.u8_retry_count           = 3;

/* register functions that performs RS485 direction control */
st_gpio_cfg.fp_gpio_init_ptr  = gpio_init;
st_gpio_cfg.fp_gpio_set_ptr   = gpio_set;
st_gpio_cfg.fp_gpio_reset_ptr = gpio_reset;

/* Initialize MODBUS stack by Serial mode */
ercd = Modbus_serial_stack_init(&st_init_info,
                        &st_gpio_cfg,
                        MODBUS_RTU_SLAVE_MODE,
                        1);
```

If ASCII mode is used, API argument will have to change from MODBUS_RTU_MASTER_MODE to MODBUS_ASCII_MASTER_MODE.

12. Limitations

Limitations on µC3 / Standard + µNet 3-Professional RX700 (RX72M) Series e2studio Edition Release 2.0.0

- Ethernet communication of RX72M CPU card
  This sample does not support Ethernet1 driver.

Website and Support

Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/contact/

Revision History

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Jan. 31, 2023	—	First edition issued

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)
   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on
   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state
   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins
   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals
   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin
   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses
   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products
   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.