R01AN7184EJ0100
Rev.1.00
Dec.8,23

RX Family

Porting Guide for RYZ014A Cellular Module Control Module Using Firmware Integration Technology

Introduction

This application note is based on the “RYZ014A Cellular Module Control Module Using Firmware Integration Technology” (R01AN6324EJ0111, Rev.1.11) and further describes how to add new AT commands, how to use AT commands for controlling communication modules other than the RYZ014A cellular module, and how to use AT commands for an RX family MCU that does not support AT commands.

Hereinafter, the software module based on the firmware integration technology (FIT) for controlling the RYZ014A cellular module is referred to as “RYZ014A Cellular FIT Module” or “Cellular FIT Module”.

For details about RYZ014A Cellular FIT Module, refer to the relevant related document.

Related Documents

[1] Firmware Integration Technology User’s Manual (R01AN1833)
[2] RX Family Board Support Package Module Using Firmware Integration Technology (R01AN1685)
[3] RX Family Adding Firmware Integration Technology Modules to Projects (R01AN1723)
[4] RX Family Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)
[5] RX Smart Configurator User’s Guide: e^2 studio (R20AN0451)
[6] RX Family RYZ014A Cellular Module Control Module Using Firmware Integration Technology (R01AN6324)
[7] RX Family SCI Module Using Firmware Integration Technology (R01AN1815)
[8] RX Family BYTEQ Module Using Firmware Integration Technology (R01AN1683)
[9] RX Family IRQ Module Using Firmware Integration Technology (R01AN1668)
[10] RYZ014 Modules User’s Manual: AT Command (R11UZ0110)
[11] RYZ014 Module System Integration Guide (R19AN0074)
[12] FreeRTOS Kernel API reference (Link to the website)
[13] FreeRTOS Cellular Interface Library (Link to the website)

Contents

1. Overview ... 5
1.1 Overview of RYZ014A Cellular FIT Module ... 5

2. About the Resources Used ... 5
2.1 Hardware ... 5
2.2 Software ... 5

3. Specifications of RYZ014A Cellular FIT Module ... 8
3.1 Operational Overview of the API Functions of RYZ014A Cellular FIT Module ... 8
3.2 About AT Command Processing ... 10
3.3 Timeout Processing of RYZ014A Cellular FIT Module ... 13
3.4 Lock Processing in RYZ014A Cellular FIT Module ... 15
3.5 Code Sizes ... 16

4. Adding a New AT Command ... 17
4.1 Steps of the Procedure for Adding a New AT Command ... 17
4.2 Files Needing Modification ... 18
4.3 AT Command Addition Example: Command That Returns Only “OK” as a Response When Processing Ends Normally ... 24
4.4 AT Command Addition Example: Command That Returns a Response Consisting of Only One Line That Contains the String “+*xxx*:” When Processing Ends Normally ... 29
4.5 AT Command Addition Example: Command That Returns a Response Consisting of Multiple Lines That Contain the String “+*xxx*:” When Processing Ends Normally ... 36
4.6 Command that Returns an Intermediate Result Code as a Response When Processing Ends Normally ... 42

5. Processing Reusable for Other Cellular Modules ... 50
5.1 R_CELLULAR_Open() ... 51
5.2 R_CELLULAR_Close() ... 52
5.3 R_CELLULAR_ APConnect() ... 53
5.4 R_CELLULAR_IsConnected() ... 53
5.5 R_CELLULAR_Disconnect() ... 54
5.6 R_CELLULAR_CreateSocket() ... 55
5.7 R_CELLULAR_ConnectSocket() ... 56
5.8 R_CELLULAR_ShutdownSocket() ... 57
5.9 R_CELLULAR_CloseSocket() ... 57
5.10 R_CELLULAR_SendSocket() ... 58
5.11 R_CELLULAR_ReceiveSocket() ... 59
5.12 R_CELLULAR_DnsQuery() ... 61
5.13 R_CELLULAR_GetTime() ... 61

5.14 R_CELLULAR_SetTime() 62
5.15 R_CELLULAR_SetEDRX() 62
5.16 R_CELLULAR_GetEDRX() 63
5.17 R_CELLULAR_SetPSM() 64
5.18 R_CELLULAR_GetPSM() 66
5.19 R_CELLULAR_GetICCID() 66
5.20 R_CELLULAR_GetIMEI() 67
5.21 R_CELLULAR_GetIMSI() 67
5.22 R_CELLULAR_GetPhonenum() 68
5.23 R_CELLULAR_GetRSSI() 68
5.24 R_CELLULAR_GetSVN() 69
5.25 R_CELLULAR_Ping() 69
5.26 R_CELLULAR_GetAPConnectState() 70
5.27 R_CELLULAR_GetCellInfo() 70
5.28 R_CELLULAR_AutoConnectConfig() 71
5.29 R_CELLULAR_SetOperator() 71
5.30 R_CELLULAR_SetBand() 72
5.31 R_CELLULAR_GetPDPAddress() 73
5.32 R_CELLULAR_FirmUpgrade() 73
5.33 R_CELLULAR_FirmUpgradeBlocking() 74
5.34 R_CELLULAR_GetUpgradeState() 75
5.35 R_CELLULAR_UnlockSIM() 75
5.36 R_CELLULAR_WriteCertificate() 76
5.37 R_CELLULAR_EraseCertificate() 76
5.38 R_CELLULAR_GetCertificate() 77
5.39 R_CELLULAR_ConfigSSLProfile() 77
5.40 R_CELLULAR_SoftwareReset() 78
5.41 R_CELLULAR_HardwareReset() 79
5.42 R_CELLULAR_FactoryReset() 80
5.43 R_CELLULAR_RTS_Ctrl() 81

6. API Functions for Software Modules 82

7. Appendix 83
7.1 Environment in Which Operation Was Verified 83
7.2 Troubleshooting 83
7.3 Recovery Operation 84
7.4 Built-in Functions of RYZ014A Cellular FIT Module Needing Modification 86
7.5 Sections Needing Modification in the r_bsp Module 106
7.6 Sections Needing Modification in the r_sci_rx Module 107
7.7 Sections Needing Modification in the r_irq_rx Module 108

7.8 Sections Needing Modification in FreeRTOS 109

8. Reference Documents 112

Revision History 113

1. Overview

1.1 Overview of RYZ014A Cellular FIT Module

RYZ014A Cellular FIT Module supports UART communication with a cellular module. For details about RYZ014A Cellular FIT Module, refer to related document [6].

2. About the Resources Used

2.1 Hardware

The MCU you use must support the following functions:

- Serial communication
- I/O ports
- Interrupt request (IRQ)
- One or more IRQ input pins that can be set as interrupt sources

2.2 Software

RYZ014A Cellular FIT Module depends on the following software and realtime operating system:

- Board Support Package Module Using Firmware Integration Technology (hereinafter referred to as "r_bsp")
- SCI Module Using Firmware Integration Technology (hereinafter referred to as "r_sci_rx")
- Byte-based Queue Buffer (BYTEQ) Module Using Firmware Integration Technology (hereinafter referred to as "r_byteq") *1
- IRQ Module Using Firmware Integration Technology (hereinafter referred to as "r_irq_rx")
- FreeRTOS

Note 1. The r_byteq FIT module is used for the r_sci_rx FIT module. RYZ014A Cellular FIT Module does not directly use the FIT module.

2.2.1 r_bsp

The API functions of the r_bsp FIT module used by RYZ014A Cellular FIT Module are listed below. The r_bsp is a module that provides support for configuring system clocks and interrupts of the MCU.

If you do not use the r_bsp FIT module, the API functions of the FIT module must be replaced by appropriate alternative processing. For details about the API, refer to related document [2]. For the purpose of each API function used in RYZ014A Cellular FIT Module, refer to section 7.5.

- R_BSP_NOP()
- R_BSP_RegisterProtectDisable()
- R_BSP_RegisterProtectEnable()

2.2.2 r_sci_rx

The API functions of the r_sci_rx FIT module used by RYZ014A Cellular FIT Module are listed below. The r_sci_rx FIT module is necessary for UART communication with the RYZ014A cellular module. It is used for exchanging AT commands and data between the RYZ014A and the MCU.

If you do not use the r_sci_rx FIT module, the API functions of the FIT module must be replaced by appropriate alternative processing. For details about the API, refer to related document [7]. For the purpose of each API function used in RYZ014A Cellular FIT Module, refer to section 7.6.

- R_SCI_Open()
- R_SCI_Close()
- R_SCI_Send()
- R_SCI_Receive()
- R_SCI_Control()

The r_sci_rx FIT module uses the r_byteq FIT module. The r_byteq FIT module is used to buffer the send data and receive data of the r_sci_rx FIT module. If you do not use the r_sci_rx FIT module, the processing that uses the API of the r_byteq FIT module must also be replaced by other processing.

2.2.3 r_irq_rx

The API functions of the r_irq_rx FIT module used by RYZ014A Cellular FIT Module are listed below. The r_irq_rx FIT module uses IRQ to handle events from MCU's external pin interrupts. The Cellular FIT Module is used to process notifications by the signal from RING pin on RYZ014A as interrupts. The notifications are used for the MCU to detect data reception requests from the cellular module in PSM mode.

If you do not use the r_irq_rx FIT module, use appropriate substitutive processing for it. For details about the r_irq_rx FIT module, refer to related document [9]. For the purpose of each API function used in RYZ014A Cellular FIT Module, refer to section 7.7.

- R_IRQ_Open()
- R_IRQ_Close()

2.2.4 FreeRTOS

RYZ014A Cellular FIT Module runs on only FreeRTOS and uses the following FreeRTOS API functions listed below.

If you do not use FreeRTOS, the FreeRTOS API functions must be replaced by appropriate alternative processing. For details about FreeRTOS, refer to related document [12]. For the purpose of each API function used in RYZ014A Cellular FIT Module, refer to section 7.8.

- xTaskCreate()
- vTaskDelay()
- xTaskGetTickCount()
- vTaskSuspend()
- vTaskDelete()
- xEventGroupCreate()
- xEventGroupWaitBits()
- xEventGroupSetBitsFromISR()
- xEventGroupSync()
- vEventGroupDelete()
- xSemaphoreCreateMutex()
- xSemaphoreTake()
- xSemaphoreGive()
- vSemaphoreDelete()
- pvPortMalloc()
- vPortFree()
- taskENTER_CRITICAL()
- taskEXIT_CRITICAL()

3. Specifications of RYZ014A Cellular FIT Module

3.1 Operational Overview of the API Functions of RYZ014A Cellular FIT Module

RYZ014A Cellular FIT Module communicates with the RYZ014A cellular module via UART communication to send AT commands and receive data. API functions are used to check arguments and parameters, acquire the semaphores for issuing AT commands, generate AT commands, execute (transmit) AT commands, and receive replies from the module.

Figure 1 shows an operational overview of the API functions of RYZ014A Cellular FIT Module.

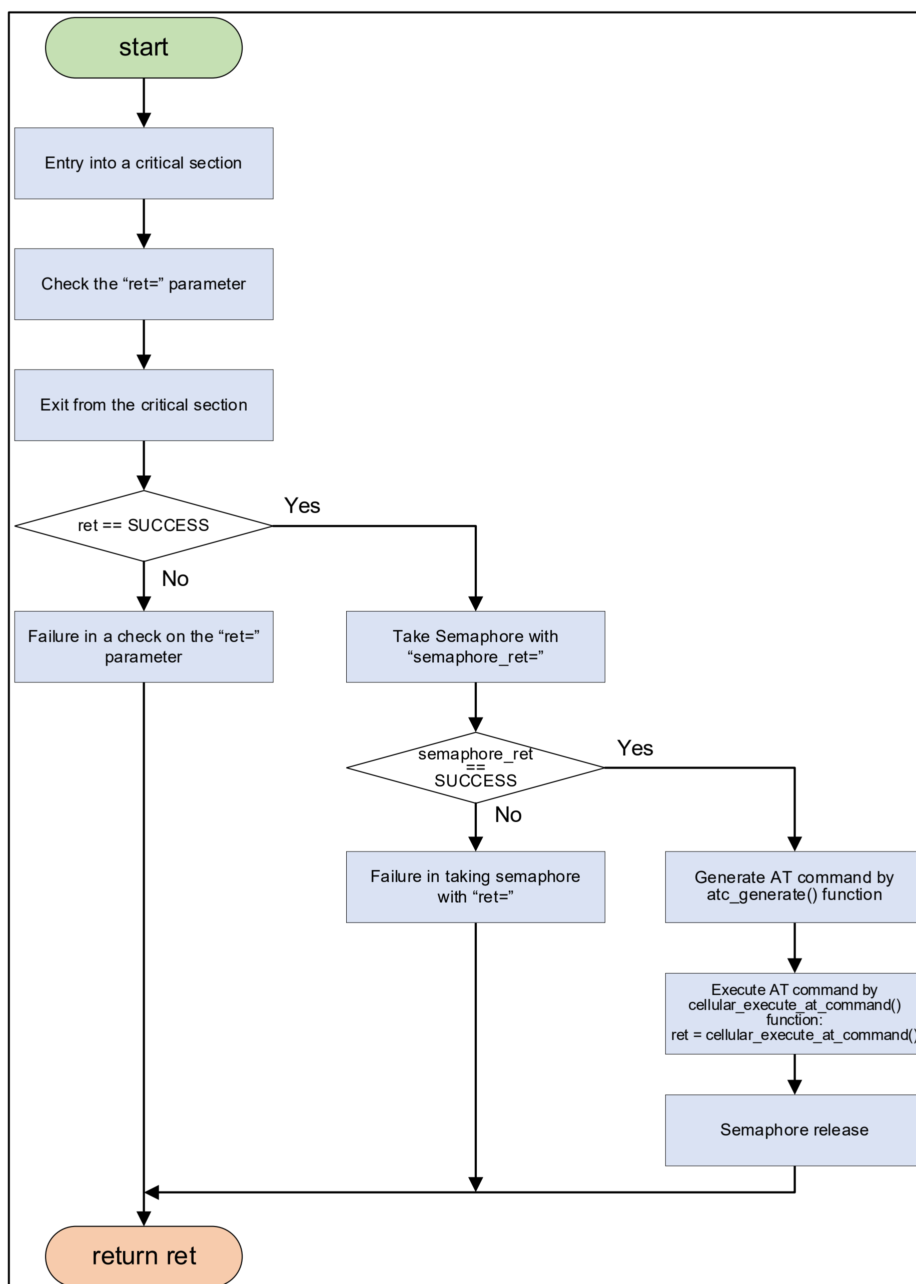

Figure 1. Operational overview of the API functions of RYZ014A Cellular FIT Module

3.1.1 Operational Overview of the cellular_execute_at_command() Function (AT Command Transmission)

Figure 2 shows an operational overview of cellular_execute_at_command(), which is a built-in function of RYZ014A Cellular FIT Module. The cellular_execute_at_command() function transmits AT commands to the cellular module via UART communication. In this function, processing that detects a timeout if there is no response to a transmitted AT command is also implemented.

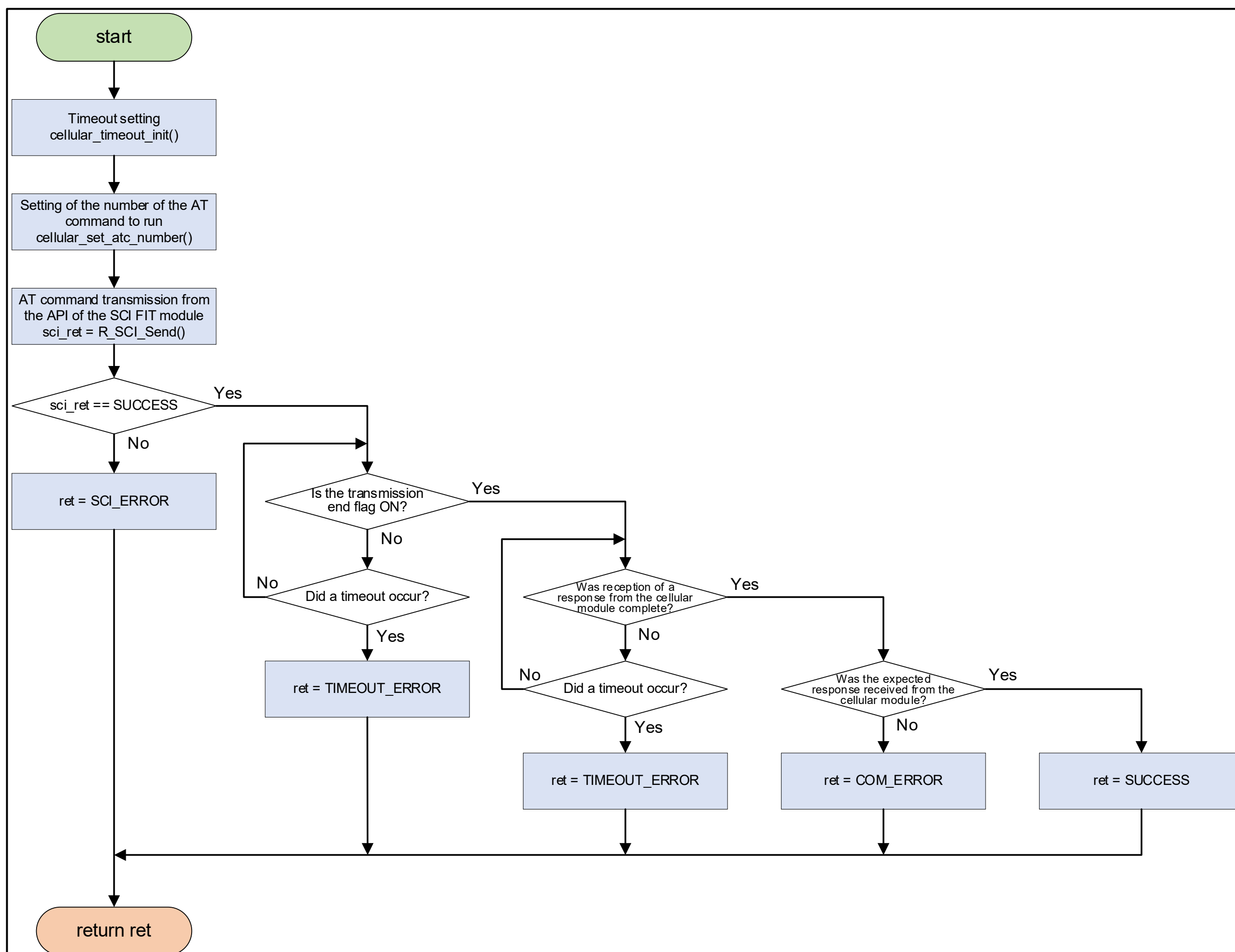

Figure 2. Operational overview of cellular_execute_at_command()

3.2 About AT Command Processing

3.2.1 atc_generate() Function

RYZ014A Cellular FIT Module uses the atc_generate() function to generate AT commands to be transmitted.

To add new AT commands, the AT command numbers of these AT commands must be added to the AT command table. For details about how to add new AT commands, refer to “4. Adding a New AT Command”.

The specifications of the atc_generate() function are as follows:

```
atc_generate(
    uint8_t * const p_command_buff,
    const uint8_t * p_command,
    const uint8_t ** pp_command_arg
)
```

- 1st argument: Sets the buffer that will store the AT command string to be transmitted.
  For RYZ014A Cellular FIT Module, “p_ctrl->sci_ctrl.atc_buff” is set as the buffer. This buffer has been declared in the st_cellular_ctrl_t management structure.
- 2nd argument: Sets the format of the AT command string to be executed as a reference to the AT command table.
  Use the AT command number that corresponds to the AT command to be executed when referencing the AT command table gp_at_command[].
  Example: gp_at_command[ATC_GET_LR_SVN]
  Figure 3 shows the AT command table gp_at_command[] and the AT command number list e_atc_list_t that are defined in ryz014_private.h. For details about ryz014_private.h, refer to section 4.2.1.
- 3rd argument: The arguments of the AT command string are set in this argument.
  Set up the arguments according to the AT command to be executed.

```
const uint8_t * const gp_at_command[ATC_LIST_MAX] =
{
    g_ryz014_echo_off,
    g_ryz014_function_level_check,
    g_ryz014_function_level,
    g_ryz014_pin_lock_check,
    g_ryz014_pin_lock_release,
             |
         (omitted)
             |

    g_ryz014_atc_get_lr_svn,
    g_ryz014_write_certificate,
    g_ryz014_erase_certificate,
    g_ryz014_get_certificate,
    g_ryz014_config_ssl_profile,
#if (CELLULAR_IMPLEMENT_TYPE == 'B')
    g_ryz014_config_ssl_socket,
#endif
    g_ryz014_no_command,
};
```

```
typedef enum
{
    ATC_ECHO_OFF          = 0,
    ATC_FUNCTION_LEVEL_CHECK,
    ATC_FUNCTION_LEVEL,
    ATC_PIN_LOCK_CHECK,
    ATC_PIN_LOCK_RELEASE,
             |
         (omitted)
             |

    ATC_GET_LR_SVN,
    ATC_WRITE_CERTIFICATE,
    ATC_ERASE_CERTIFICATE,
    ATC_GET_CERTIFICATE,
    ATC_CONFIG_SSL_PROFILE,
#if (CELLULAR_IMPLEMENT_TYPE == 'B')
    ATC_CONFIG_SSL_SOCKET,
#endif
    ATC_SQNSSENDEXT_END,
    ATC_LIST_MAX
} e_atc_list_t;
```

Figure 3. Definitions of array gp_at_command[] and enumerated type e_atc_list_t used in the 2nd argument (ryz014_private.h)

3.2.2 Processing for the Received Text

In RYZ014A Cellular FIT Module, the cellular_recv_task() function performs processing called the "receive task" for the text that is received from the cellular module as responses and unsolicited result codes (URCs).

The cellular_recv_task() function receives the text sent from the cellular module on a character-by-character basis. While receiving the text from the cellular module, the cellular_recv_task() function interprets the text sequentially. Each time the function recognizes a meaningful string, it identifies the task number corresponding to the string within the enumerated type e_atc_return_code_t, and then calls the function corresponding to the task number based on the function pointer table p_cellular_recvtask_api[]. This is the receive task that the cellular_recv_task() function performs.

The specifications of the function pointer table p_cellular_recvtask_api[] are as follows:

```
static void (* p_cellular_recvtask_api[]) (
    st_cellular_ctrl_t * p_ctrl,
    st_cellular_receive_t * cellular_receive
)
```

The task number e_atc_return_code_t is set as the index of p_cellular_recvtask_api[] according to the received character.

- 1st argument: Sets the management structure pointer to Cellular FIT Module.
- 2nd argument: Sets the management structure pointer to the cellular_recv_task() function, which is the reception task of Cellular FIT Module.

```
static void(* p_cellular_recvtask_api[])(st_cellular_ctrl_t * p_ctrl, st_cellular_receive_t * cellular_receive) =
{
    cellular_data_send_command,
    cellular_memclear,
    cellular_memclear,
    cellular_memclear,
    cellular_memclear,
    cellular_exit,
         |
      (omitted)
         |
    cellular_get_certificate,
    cellular_response_skip,
    cellular_store_data,
    cellular_response_check,
    cellular_job_check
};
```

Figure 4. Definition of p_cellular_recvtask_api[] (r_cellular_receive_task.c)

```
typedef enum
{
    CELLULAR_RES_GO_SEND = 0,           // Request for Data Transmission
    CELLULAR_RES_OK,                    // Response is OK
    CELLULAR_RES_ERROR,                 // Response is ERROR
    CELLULAR_RES_NO_CARRIER,            // Response is NO CARRIER
    CELLULAR_RES_CONNECT,               // Response is CONNECT
    CELLULAR_RES_EXIT,                  // Exit error detected (module is automatically restarted)
                 |
             (omitted)
                 |
    CELLULAR_GET_CERTIFICATE,           // Get Certificate
    CELLULAR_RES_MAX,                   // End of analysis result processing
    CELLULAR_RES_PUT_CHAR,              // Received data storage process
    CELLULAR_RES_CHECK,                 // Receipt confirmation process
    CELLULAR_RES_NONE,                  // No process
} e_atc_return_code_t;
```

Figure 5. Definition of the task number e_atc_return_code_t (cellular_receive_task.h)

RENESAS

3.3 Timeout Processing of RYZ014A Cellular FIT Module

RYZ014A Cellular FIT Module provides timeout processing to prevent a deadlock from occurring in cases such as when it fails to receive a response from the cellular module successfully. Table 3.1 lists the functions that can time out. The recommended timeout value differs depending on the cellular module. In the case of the RYZ014A cellular module, for AT commands that do not perform communication via a network, the recommended time is 60 seconds. Set the value appropriate for the module you use.

Table 3.1 Cellular FIT Module functions that can time out

Function name	Type	Timeout value
R_CELLULAR_SendSocket()	API function for users	The user can set a value by using an argument of the API function.
R_CELLULAR_ReceiveSocket()	API function for users	The user can set a value by using an argument of the API function.
cellular_take_semaphore()	Built-in function	The user can set a value from the GUI (Smart Configurator).
cellular_execute_at_command()	Built-in function	A value can be set by using a macro definition in the driver.
cellular_synchro_event_group()	Built-in function	A value can be set by using a macro definition in the driver.
cellular_pin_reset()	Built-in function	A value can be set by using a macro definition in the driver.
cellular_power_down()	Built-in function	A value can be set by using a macro definition in the driver.

3.3.1 Timeout Processing for the R_CELLULAR_SendSocket() Function

R_CELLULAR_SendSocket() is an API function that sends data by using a socket that is connected to the network.

The user sets the timeout value for the 5th argument.

Note: Set the time period before execution of the R_CELLULAR_SendSocket() function ends. The timeout setting is not applied to the AT command transmission processing (cellular_execute_at_command() function) performed within the processing of the R_CELLULAR_SendSocket() function.

If the function times out, it returns the return value CELLULAR_ERR_MODULE_TIMEOUT.

3.3.2 Timeout Processing for the R_CELLULAR_ReceiveSocket() Function

R_CELLULAR_ReceiveSocket() is an API function that allows the MCU to receive data from the cellular module via a socket connected to the network.

The user sets the timeout value for the 5th argument.

Note: Set the time period before execution of the R_CELLULAR_ReceiveSocket() function ends. The timeout setting is not applied to the AT command transmission processing (cellular_execute_at_command() function) performed within the processing of the R_CELLULAR_ReceiveSocket() function.

If the function times out, it returns the size (in bytes) of data that could be received before occurrence of the timeout.

3.3.3 Timeout Processing for the cellular_take_semaphore() Function

cellular_take_semaphore() is a built-in function that is executed when a FreeRTOS semaphore is acquired.

The value of CELLULAR_CFG_SEMAPHORE_BLOCK_TIME defined in r_cellular_config.h is used as the timeout value.

If the function times out, it returns the return value CELLULAR_SEMAPHORE_ERR_TAKE.

Note: The default timeout value is 15,000 ms.

3.3.4 Timeout Processing for the cellular_execute_at_command() Function

cellular_execute_at_command() is a built-in function that transmits AT commands to the cellular module.

The value of the 2nd argument is used as the timeout value.

If the function times out, it returns the return value CELLULAR_ERR_MODULE_TIMEOUT.

Note: When the R_CELLULAR_Open() function is executed, a value is set in p_ctrl->sci_ctrl.atc_timeout. This value is subsequently used as the timeout value. The value of CELLULAR_COMMAND_TIMEOUT defined in ryz014_private.h is used as the timeout value.

Note: The default timeout value is 60,000 ms.

3.3.5 Timeout Processing for the cellular_synchro_event_group() Function

cellular_synchro_event_group() is a built-in function that performs synchronization with the cellular_recv_task() function, which processes received data.

If the function times out, it returns the flag state at occurrence of the timeout.

Note: The value of CELLULAR_TIME_WAIT_TASK_START defined in r_cellular_private.h is used as the timeout value.

Note: The default timeout value is 10,000 ms.

3.3.6 Timeout Processing for the cellular_pin_reset() Function

cellular_pin_reset() is a built-in function that performs a hardware reset of the cellular module.

The value of CELLULAR_RESTART_LIMIT defined in cellular_module_reset.c is used as the timeout value.

If the function times out, it returns the return value CELLULAR_ERR_RECV_TASK.

Note: The default timeout value is 100 s (= 100 ms, the default CELLULAR_RESTART_LIMIT value, × 1,000).

3.3.7 Timeout Processing for the cellular_power_down() Function

cellular_power_down() is a built-in function that shuts down the cellular module.

The value of CELLULAR_SHUTDOWN_LIMIT defined in cellular_power_down.c is used as the timeout value.

If the function times out, it returns the return value CELLULAR_ERR_MODULE_COM.

Note: The default timeout value is 50 s (= 50 ms, the default CELLULAR_SHUTDOWN_LIMIT value, × 1,000).

3.4 Lock Processing in RYZ014A Cellular FIT Module

Table 3.2 lists the processes that must be controlled by locks in RYZ014A Cellular FIT Module.

Table 3.2 Processes that must be controlled by locks in Cellular FIT Module

Process	When the process is locked	When the process is unlocked
AT command issuance	Before execution of the built-in function for executing an AT command	After completion of the built-in function for executing an AT command
RTS pin control	Before the start of RTS pin control	After the end of RTS pin control
Socket data reception process	Before execution of the function that processes received data	After completion of the function that processes received data
API function execution	Immediately after the start of an API function	Immediately before the end of an API function

3.4.1 Lock Processing During Execution of AT Commands

Because the RYZ014A cellular module processes AT commands on a one-by-one basis, they must be controlled by locks so that multiple AT commands do not run concurrently.

In Cellular FIT Module, a semaphore must be acquired before an AT command is executed (when an atc_***() function is executed), and the semaphore must be released after the AT command is completed. The semaphore used for the lock processing on AT command executions can be acquired and released by performing the following operations:

- Acquiring a semaphore: cellular_take_semaphore(p_ctrl->at_semaphore)
- Releasing a semaphore: cellular_give_semaphore(p_ctrl->at_semaphore)

3.4.2 Lock Processing During the RTS Pin Control Processing

The RTS pin of the RYZ014A cellular module must be controlled so that the operating mode of the cellular module can change to PSM mode. While an API function is controlling the RTS pin, the pin must be locked to prevent other API functions from controlling the RTS pin.

In Cellular FIT Module, a semaphore must be acquired before control of the RTS pin starts and the semaphore must be released after the control is completed. The semaphore used for the lock processing on the RTS pin control processing can be acquired and released by performing the following operations:

- Acquiring a semaphore: cellular_take_semaphore(p_ctrl->ring_ctrl.rts_semaphore)
- Releasing a semaphore: cellular_give_semaphore(p_ctrl->ring_ctrl.rts_semaphore)

3.4.3 Lock Processing During the Data Reception Processing

In Cellular FIT Module, a semaphore must be acquired before data reception from a socket starts and the semaphore must be released after the data reception is completed. A semaphore can be acquired and released for each socket by performing the following operations:

- Acquiring a semaphore: cellular_take_semaphore
  (p_ctrl->p_socket_ctrl[<socket-number> – CELLULAR_START_SOCKET_NUMBER].rx_semaphore)
- Releasing a semaphore: cellular_give_semaphore
  (p_ctrl->p_socket_ctrl[<socket-number> – CELLULAR_START_SOCKET_NUMBER].rx_semaphore)

Note: CELLULAR_START_SOCKET_NUMBER is the start socket number of the cellular module. Set an appropriate value according to the specifications of the cellular module you use.

3.4.4 Lock Processing During API Function Execution (Thread-Safe Feature)

Cellular FIT Module provides the thread-safe feature for each API function. When a new API function is added, thread-safe functionality can be added to it by performing the following operations.

While an API function of Cellular FIT Module is running, if R_CELLULAR_Close() is executed, the resources being used by the module are released, thus causing the module to malfunction. The thread-safe feature can prevent this problem.

(1) Setting a critical section: preemption = cellular_interrupt_disable()

(2) Adding processing to check the execution states of other API functions:

```
if (0 != (p_ctrl->running_api_count % 2))
{
    ret = CELLULAR_ERR_OTHER_API_RUNNING;
}
else
{
    p_ctrl->running_api_count += 1 or 2;   // Set 1 if concurrent execution with
                                              other API functions is prohibited.
                                           // Set 2 if concurrent execution with
                                              other API functions is permitted.
}
```

(3) Disabling the critical section: cellular_interrupt_enable(preemption)

(4) p_ctrl->running_api_count – 1 or 2; // Value added in (2) is subtracted (only when addition was done in (2)).

3.5 Code Sizes

Table 3.3 shows the ROM and RAM space requirements for RYZ014A Cellular FIT Module, and the maximum size of stack space used by RYZ014A Cellular FIT Module.

The values shown in Table 3.3 have been confirmed under the following conditions:

Revision number of the FIT module: r_cellular rev1.11

Version of the compiler: Renesas Electronics C/C++ Compiler for RX Family V3.04.00
(The "-lang = c99" option has been added to the default settings of the integrated development environment.)

Configuration options: Default settings

Table 3.3 Code Sizes

Code sizes of the ROM, RAM, and stack			
Device	**Item**	**Memory usage**	**Remarks**
RX65N RX72N	ROM	36 KB, approx.	-
	RAM	600 bytes, approx.	-
	Maximum size of stack space used	700 bytes, approx.	-

4. Adding a New AT Command

4.1 Steps of the Procedure for Adding a New AT Command

Figure 6 shows the steps of the overall procedure for adding a new AT command.

The tasks to be performed differ depending on the type (response) of the AT command to be added.

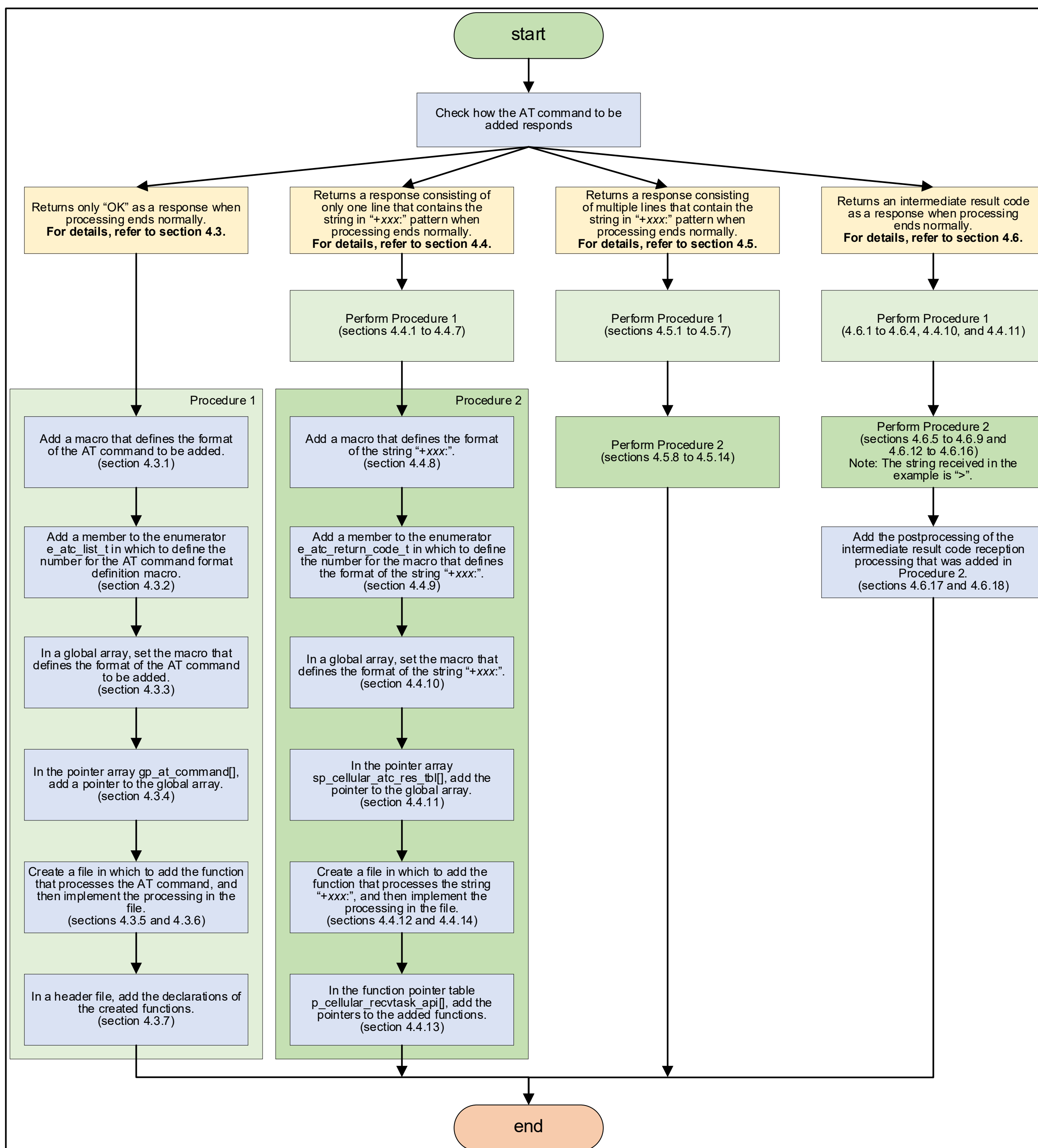

Figure 6. Flowchart of the procedure for adding a new AT command

4.2 Files Needing Modification

Figure 7 shows the folder configuration of RYZ014A Cellular FIT Module.

Figure 7 shows the locations of the files that need to be modified according to the specifications of RYZ014A Cellular FIT Module when a user adds a new AT command. For details about the specifications, refer to section 3.2.

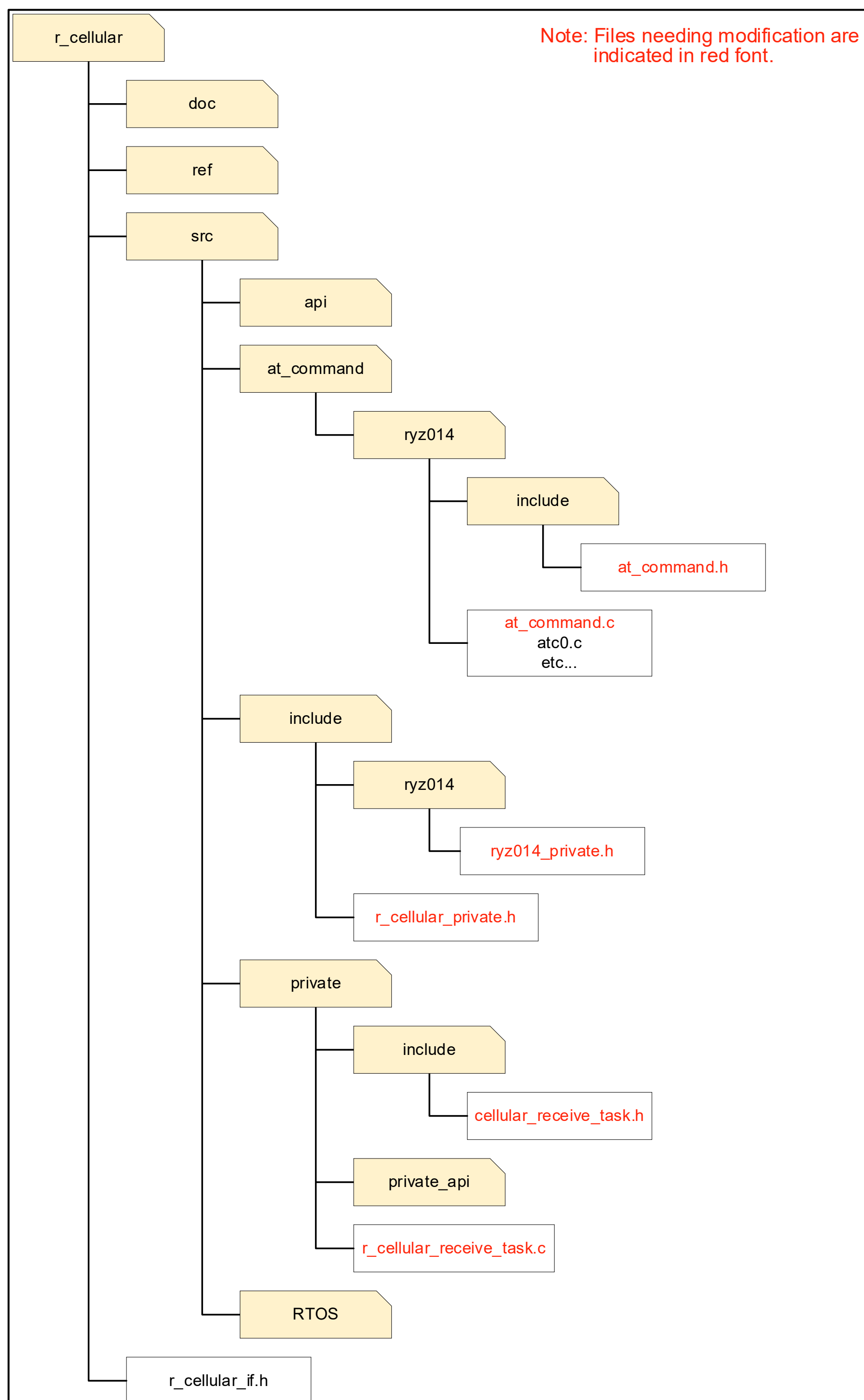

Figure 7. Locations of files to be modified

4.2.1 File Defining AT Command Formats

For Cellular FIT Module, the formats of supported AT commands are defined in the ryz014_private.h file. When you add a new AT command, you must edit the ryz014_private.h file.

To generate an AT command, set the macro definition of the AT command template (format), and then execute the sprintf function. Be sure to define an AT command format including the termination character (\r).

The ryz014_private.h file defines AT command formats and the enumerated type e_atc_list_t consisting of the numbers that correspond to the formats. Figure 8 shows AT command format definitions. Figure 9 shows the correspondence between the defined formats and the elements of the enumerated type e_atc_list_t.

If you add a new AT command, you must add its format definition to the ryz014_private.h file. Also, you must add the definition of the number associated with the new AT command to the enumerated type e_atc_list_t. Make sure that you add the new definition to a location before (above) the keyword ATC_LIST_MAX.

```
#define RYZ014_ATC_ECHO_OFF                  "ATE0\r"
#define RYZ014_ATC_FUNCTION_LEVEL_CHECK      "AT+CFUN?\r"
#define RYZ014_ATC_FUNCTION_LEVEL            "AT+CFUN=%s\r"
#define RYZ014_ATC_PIN_LOCK_CHECK            "AT+CPIN?\r"
#define RYZ014_ATC_PIN_LOCK_RELEASE          "AT+CPIN=\"%s\"\r"

                              |
                          (omitted)
                              |
#define RYZ014_ATC_WRITE_CERTIFICATE         "AT+SQNSNVW=\"%s\",%s,%s\r"
#define RYZ014_ATC_ERASE_CERTIFICATE         "AT+SQNSNVW=\"%s\",%s,0\r"
#define RYZ014_ATC_GET_CERTIFICATE           "AT+SQNSNVR=\"%s\",%s\r"
#define RYZ014_ATC_CONFIG_SSL_PROFILE        "AT+SQNSPCFG=%s,2,,%s,%s,%s,%s,\"\"\r"
#if (CELLULAR_IMPLEMENT_TYPE == 'B')
#define RYZ014_ATC_CONFIG_SSL_SOCKET         "AT+SQNSSCFG=%s,%s,%s\r"
#endif
#define RYZ014_NO_COMMAND                    "\r"


typedef enum
{
    ATC_ECHO_OFF          = 0,
    ATC_FUNCTION_LEVEL_CHECK,
    ATC_FUNCTION_LEVEL,
    ATC_PIN_LOCK_CHECK,
    ATC_PIN_LOCK_RELEASE,

               |
           (omitted)
               |
    ATC_ERASE_CERTIFICATE,
    ATC_GET_CERTIFICATE,
    ATC_CONFIG_SSL_PROFILE,
#if (CELLULAR_IMPLEMENT_TYPE == 'B')
    ATC_CONFIG_SSL_SOCKET,
#endif
    ATC_SQNSSENDEXT_END,
    ATC_LIST_MAX
} e_atc_list_t;
```

Figure 8. Definitions in the ryz014_private.h file

RENESAS

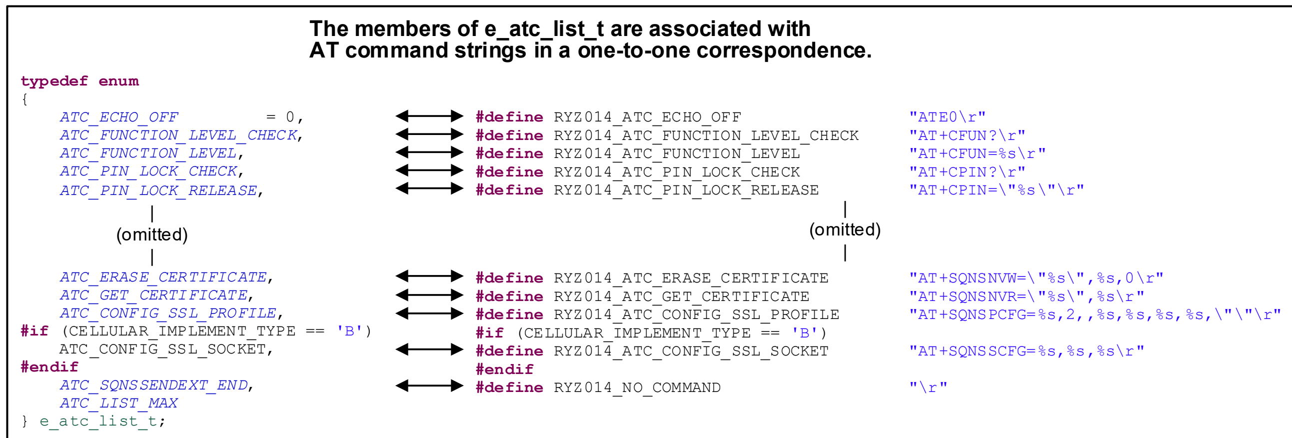

Figure 9. Correspondence between the elements of the enumerated type e_atc_list_t and AT command definitions

4.2.2 File Defining the AT Command Format Table

Cellular FIT Module manages AT command formats as table data. The AT command format table is defined in the at_command.c file. If you add a new AT command, you must edit the at_command.c file.

The at_command.c file defines character constants for AT command formats defined in the ryz014_private.h file. It also defines the gp_at_command[] pointer table that is used for referencing the character constants. Figure 10 shows the relationships among the AT command format definitions, the character constants for AT command formats, and the elements of gp_at_command[].

For the new AT command format that you added in the ryz014_private.h file in order to add a new AT command, you must add the command format's character constant definitions to the at_command.c file. Also, in gp_at_command[], you must add the pointers to the added character constants.

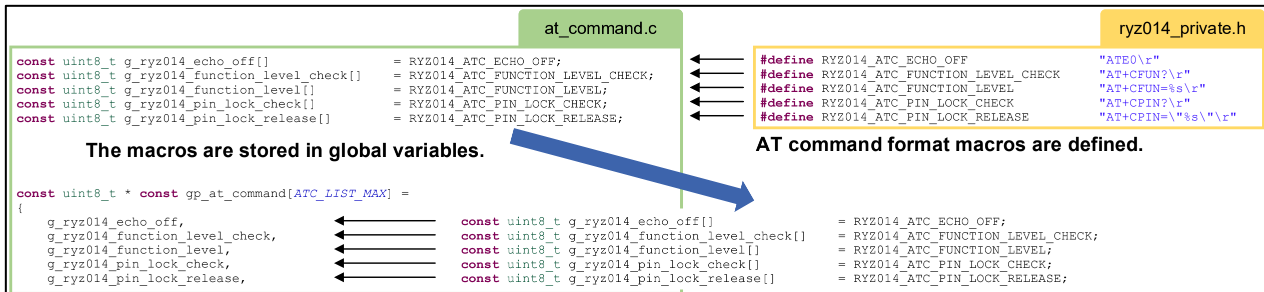

Figure 10. Relationships among the AT command format definitions, the character constants for AT command formats, and the elements of gp_at_command[]

4.2.3 File Defining Response Strings

After Cellular FIT Module executes an AT command for the cellular module, it returns a response string. The response strings that can be returned are defined in the cellular_receive_task.h file. If you add new response strings, you must edit the cellular_receive_task.h file.

In the cellular_receive_task.h file, use the enumerator e_atc_return_code_t to define response strings and the numbers corresponding to the response strings. Furthermore, in the cellular_receive_task.c file, define the function pointer table p_cellular_recvtask_api[] that contains the pointers to the functions that correspond to the members of the enumerated type e_atc_return_code_t. Figure 11 shows the correspondence between the response string definitions and the members of enumerated type e_atc_return_code_t. It also shows the correspondence between the members of enumerated type e_atc_return_code_t and the members of the function pointer table p_cellular_recvtask_api[].

If you add new response strings, add their definitions to the cellular_receive_task.h file. In addition, add the definitions of the numbers that are associated with the added response strings to the enumerated type e_atc_return_code_t. Make sure that you add the new definitions to a location before (above) the keyword CELLULAR_RES_MAX.

The members of the enumerated type e_atc_return_code_t and the members of the function pointer table p_cellular_recvtask_api[] have a one-to-one correspondence. If you add a number to e_atc_retur_code_t, you must also add the corresponding function pointer to p_cellular_recvtask_api[] defined in the cellular_receive_task.c file. The user is responsible for implementing the functions to be added. If no processing needs to be implemented, add the pointer to the cellular_memclear() function in p_cellular_recvtask_api[].

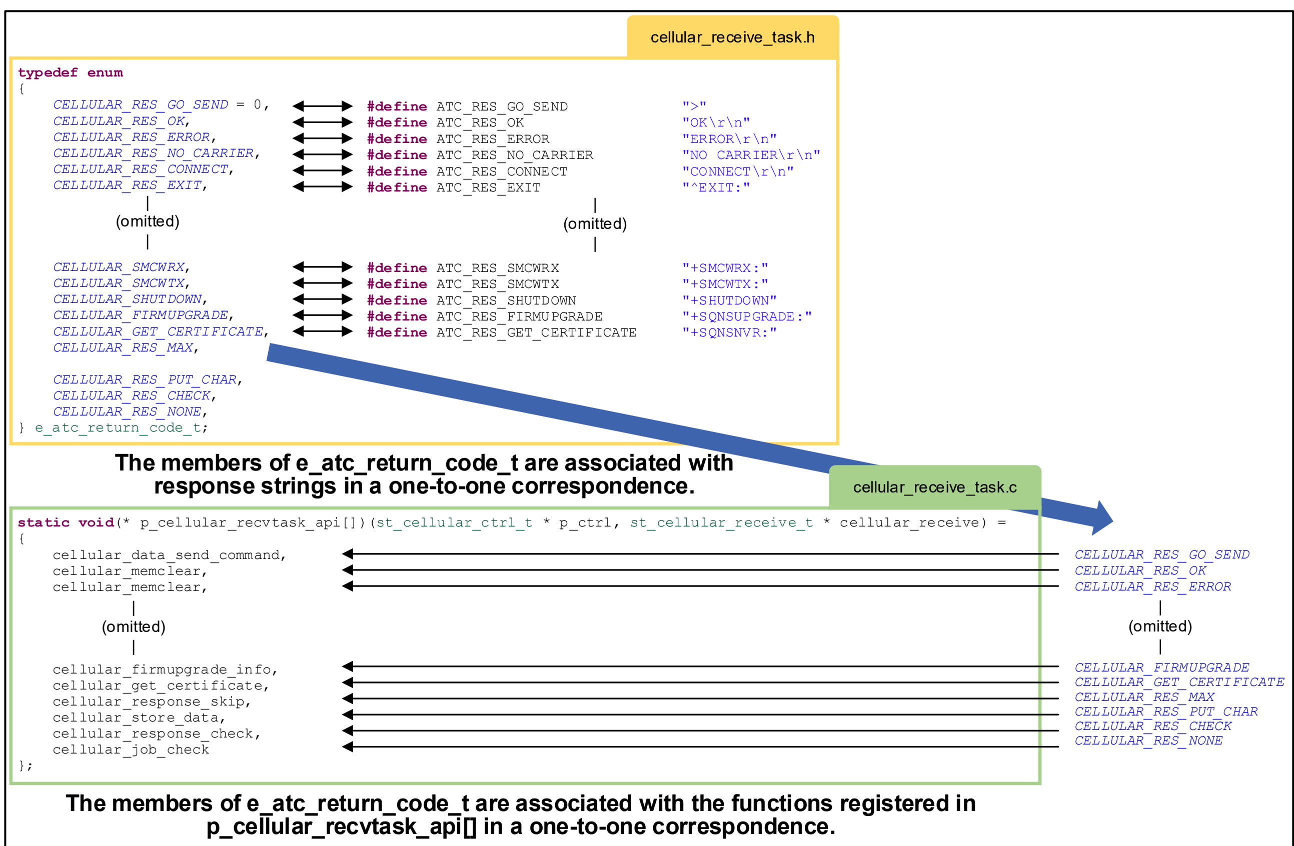

Figure 11. Relationship between the response string definitions and the members of e_atc_return_code_t

4.2.4 File Defining the Response String Table

In Cellular FIT Module, the response string table is defined in the r_cellular_receive_task.c file. If you add new response strings, you must edit the cellular_receive_task.c file.

In the cellular_receive_task.c file, define the response string constants and the sp_cellular_atc_res_tbl[] table that contains the pointers to them. Figure 12 shows the relationships among the response string definitions, response string constants, and the members of sp_cellular_atc_res_tbl[].

If you add a new response string, the pointer to the added response string constant must also be added to sp_cellular_atc_res_tbl[]. For details about how to add response string constants, refer to section 4.2.3.

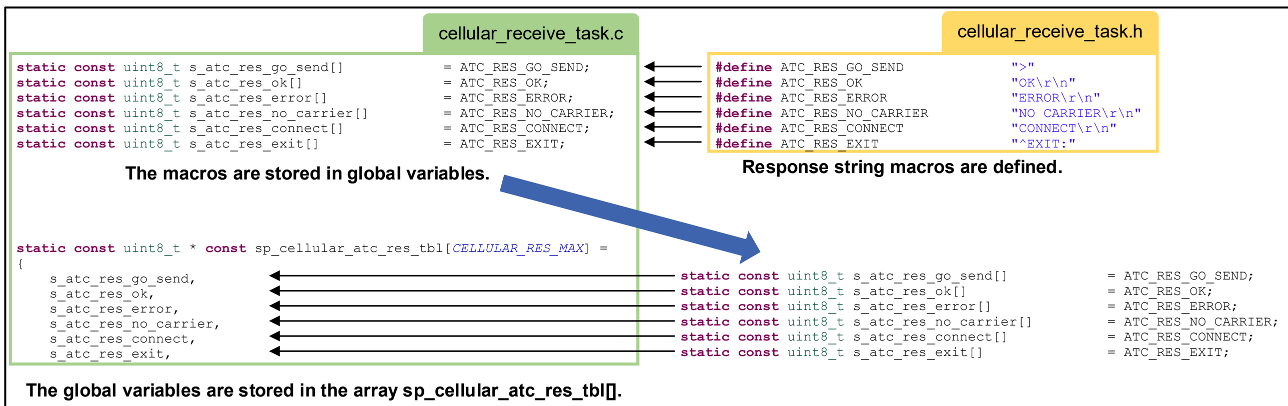

Figure 12. How response string macros are stored

4.3 AT Command Addition Example: Command That Returns Only “OK” as a Response When Processing Ends Normally

This section describes the procedure for modifying the source code in the case when adding an AT command that responds by returning only “OK” when processing ends normally.

The command you add in this example is “AT+CTZU”. For details about the AT command specifications, refer to related document [10].

```
AT+CTZU=1

OK
```

7.17 Automatic Time Zone Update: AT+CTZU

Note: This command is described in 3GPP TS 27.007. See Section References.

7.17.1 Syntax

Command	Possible Response(s)
AT+CTZU=<onoff>	+CME ERROR: <err>

Figure 13. AT+CTZU command to be added

onoff

Integer. 0 or 1. Boolean switch.

Table 146. onoff

Value	Description
0	Disable automatic time zone update via NITZ.
1	Enable automatic time zone update via NITZ.

Figure 14. Specifications of the <onoff> parameter of the AT+CTZU command

4.3.1 Adding a Macro

In the ryz014a_private.h file, add a macro that defines the format of the AT command that you want to add.

According to the AT command specifications in Figure 14, the value that can be set for the <onoff> parameter is 0 or 1. Therefore, replace the <onoff> parameter by the keyword "%s" in the AT command string.

```
158 #define RYZ014_ATC_CONFIG_SSL_PROFILE        "AT+SQNSPCFG=%s,2,,%s,%s,%s,%s,\"\"\r"
159 #define RYZ014_ATC_TIMEZONE_UPDATE           "AT+CTZU=%s\r"
160 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 15. Code section where the new command "AT+CTZU" has been added

4.3.2 Adding a New Member to the Enumeration

Add a new member to the enumerated type e_atc_list_t defined in the ryz014a_private.h file.

Figure 16 shows the member ATC_CONFIG_TIMEZONE_UPDATE that has been added in the enumerated type. Assume that this member has been added as the 71st member and therefore given a value of 70.

```
239     ATC_CONFIG_SSL_PROFILE,
240     ATC_CONFIG_TIMEZONE_UPDATE,
241 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 16. Code section (in the enumerated type e_atc_list_t) where a new member has been added

4.3.3 Adding a Constant and Storing a String

In the at_command.c file, add the definition of a string constant, and then, in the constant, store the string added in section 4.3.1.

```
110 const uint8_t g_ryz014_config_ssl_profile[]       = RYZ014_ATC_CONFIG_SSL_PROFILE;
111 const uint8_t g_ryz014_config_timezone_update[]   = RYZ014_ATC_TIMEZONE_UPDATE;
112 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 17. Code section where a string constant has been added

4.3.4 Adding the Address of a String

In the pointer array gp_at_command[] defined in the at_command.c file, add the address of the string that you added in section 4.3.3.

Make sure that you add the address at the same position as the position of the member added in Figure 16. Because ATC_CONFIG_TIMEZONE_UPDATE is the 71st member and has a value of 70, add the address to the array gp_at_command[] as the 71st member (at the position of element 70) as shown in Figure 18.

```
189     g_ryz014_config_ssl_profile,
190     g_ryz014_config_timezone_update,
191 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 18. Code section (in the array gp_at_command[]) where a new member has been added

4.3.5 Creating a New File

Create a new file in the "r_cellular/src/at_command/ryz014" folder.

We recommend that you copy and rename cfun.c or another existing file for a command that takes only one argument.

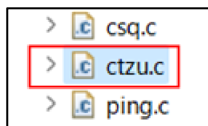

Figure 19. Folder where the new file "ctzu.c" has been created

4.3.6 Creating a Function That Executes the AT Command

In the .c file that you created in section 4.3.5, create a function that executes the "AT+CTZU" command.

Examples

```
#include "at_command.h"
#include "cellular_private_api.h"

e_cellular_err_t atc_ctzu(st_cellular_ctrl_t * const p_ctrl,
                          const uint8_t                  onoff) (1)
{
    uint8_t              str[2]                              = {0}; (2)
    const uint8_t *   p_command_arg[CELLULAR_MAX_ARG_COUNT]  = {0}; (3)
    e_cellular_err_t ret                                     = CELLULAR_SUCCESS;

    sprintf((char *)str, "%d", onoff); // (uint8_t *)->(char *)  (4)

    p_command_arg[0] = str; (5)

    atc_generate(p_ctrl->sci_ctrl.atc_buff,
                 gp_at_command[ATC_CONFIG_TIMEZONE_UPDATE],      (6)
                 p_command_arg);

    ret = cellular_execute_at_command(p_ctrl,
                                      p_ctrl->sci_ctrl.atc_timeout,  (7)
                                      ATC_RETURN_OK,
                                      ATC_CONFIG_TIMEZONE_UPDATE);

    return ret;
}
```

(1) According to the AT command specifications in Figure 14, the value that can be set for the parameter is 0 or 1. Therefore, define the argument as the uint8_t type.
"st_cellular_ctrl_t * const p_ctrl", which is a pointer to the management structure, is specified as the 1st argument.

(2) The array that will store the parameter of the AT command is declared.

(3) The pointer array to be set for the 2nd argument of the atc_generate() function is declared.

(4) By using the sprintf() function, the 2nd argument value of this function is converted to the string type and the conversion result is stored in the array declared in (2).

(5) The start address of the string converted in (4) is stored in the pointer array.

(6) The atc_generate() function is used to generate the AT command string to be transmitted to the cellular module.

- 1st argument: "p_ctrl->sci_ctrl.atc_buff" is set.
- 2nd argument: AT command table gp_at_command[] is set with the member added in Figure 16 specified in the square brackets.
- 3rd argument: The pointer array declared in (3) is set.

(7) The cellular_execute_at_command() function is used to transmit the AT command to the cellular module.

- 1st argument: "p_ctrl" is set.
- 2nd argument: The time before a wait for a response returned from the cellular module times out is set.
- 3rd argument: The expected value for the response returned from the cellular module is set. For most commands, "ATC_RETURN_OK" is set.
- 4th argument: The number of the AT command to be transmitted (the member added in Figure 16) is set.

RENESAS

4.3.7 Adding the Declaration of the New Function

In the at_command.h file, add the declaration of atc_ctzu(), which is the new function that you created.

```
645 /*********************************************************************************************
646  * Function Name  @fn            atc_ctzu
647  * Description    @details       Execute the AT command (AT+CTZU).
648  * Arguments      @param[in/out] p_ctrl -
649  *                                  Pointer to managed structure.
650  *                @param[in]     onoff -
651  *                                  Enable(1)/Disable(0) automatic time zone update via NITZ.
652  * Return Value   @retval        CELLULAR_SUCCESS -
653  *                                  Successfully executed AT command.
654  *                @retval        CELLULAR_ERR_MODULE_COM -
655  *                                  Communication with module failed.
656  *********************************************************************************************/
657 e_cellular_err_t atc_ctzu (st_cellular_ctrl_t * const p_ctrl, const uint8_t onoff);
```

Figure 20. Code section where the declaration of "atc_ctzu()" has been added

4.4 AT Command Addition Example: Command That Returns a Response Consisting of Only One Line That Contains the String “+*xxx*:” When Processing Ends Normally

This section describes the procedure for modifying the source code in the case when adding an AT command that returns a response consisting of only one line that contains the string “+*xxx*:” when processing ends normally.

Continuing from the previous section 4.3, this section also provides an example of adding a new command. The command you add in this example is “AT+CPAS”. For details about the AT command specifications, refer to related document [10].

```
AT+CPAS
+CPAS: 0
OK
```

7.11 Phone Activity Status: AT+CPAS

Note: This command is described in 3GPP TS 27.007. See Section References.

7.11.1 Syntax

Command	Possible Response(s)
AT+CPAS	+CPAS: <pas> +CME ERROR: <err>

Figure 21. AT+CPAS command to be added

7.11.3 Defined Values

pas

Integer.

Caution: Only 0, 4 and 5 values are currently implemented. All other values are reserved.

Table 136. pas

Value	Description
0	Ready (MT allows commands from TA/TE)
1	Unavailable (MT does not allow commands from TA/TE)
2	Unknown (MT is not guaranteed to respond to instructions)
3	Ringing (MT is ready for commands from TA/TE, but the ringer is active)
4	Call in progress (MT is ready for commands from TA/TE, but a call is in progress)
5	Asleep (MT is unable to process commands from TA/TE because it is in a low functionality state)
6..128	Reserved

Figure 22. Specifications of the <pas> parameter of the +CPAS: command

4.4.1 Adding a Macro

In the ryz014a_private.h file, add a macro that defines the format of the AT command that you want to add.

```
159 #define RYZ014_ATC_TIMEZONE_UPDATE          "AT+CTZU=%s\r"
160 #define RYZ014_ATC_GET_MT_STATE             "AT+CPAS\r"
161 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 23. Code section where the new command “AT+CPAS” has been added

4.4.2 Adding a New Member to the Enumeration

Add a new member to the enumerated type e_atc_list_t defined in the ryz014a_private.h file.

Figure 24 shows the member ATC_GET_MT_STATE that has been added in the enumerated type. Assume that this member has been added as the 72nd member and therefore given a value of 71.

```
240     ATC_CONFIG_TIMEZONE_UPDATE,
241     ATC_GET_MT_STATE,
242 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 24. Code section (in the enumerated type e_atc_list_t) where a new member has been added

4.4.3 Adding a Constant and Storing a String

In the at_command.c file, add a string constant, and then, in the constant, store the string added in section 4.4.1.

```
111 const uint8_t g_ryz014_config_timezone_update[]   = RYZ014_ATC_TIMEZONE_UPDATE;
112 const uint8_t g_ryz014_get_mt_state[]             = RYZ014_ATC_GET_MT_STATE;
113 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 25. Code section where a string constant has been added

4.4.4 Adding the Address of a String

In the pointer array gp_at_command[] defined in the at_command.c file, add the address of the string that you added in section 4.4.3.

Make sure that you add the address at the same position as the position of the member added in Figure 24. Because ATC_GET_MT_STATE has been added as the 72nd member, add the address to the array gp_at_command[] as the 72nd member (at the position of element 71) as shown in Figure 26.

```
190     g_ryz014_config_timezone_update,
191     g_ryz014_get_mt_state,
192 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 26. Code section (in the array gp_at_command[]) where a new member has been added

4.4.5 Creating a New File

Create a new file in the "r_cellular/src/at_command/ryz014" folder.

We recommend that you copy and rename ceer.c or another existing file for a command that takes no argument.

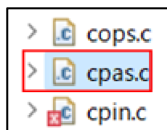

Figure 27. Folder where the new file "cpas.c" has been created

4.4.6 Creating a Function That Executes the AT Command

In the .c file that you created in section 4.4.5, create a function that executes the "AT+CPAS" command.

Examples

```
#include "at_command.h"
#include "cellular_private_api.h"

e_cellular_err_t atc_cpas(st_cellular_ctrl_t * const p_ctrl)
{
    e_cellular_err_t ret = CELLULAR_SUCCESS;

    atc_generate(p_ctrl->sci_ctrl.atc_buff,
                 gp_at_command[ATC_GET_MT_STATE],   (1)
                 NULL);

    ret = cellular_execute_at_command(p_ctrl,
                                      p_ctrl->sci_ctrl.atc_timeout,   (2)
                                      ATC_RETURN_OK,
                                      ATC_GET_MT_STATE);

    return ret;
}
```

(1) The atc_generate() function is used to generate the AT command string to be transmitted to the cellular module.

- 1st argument: "p_ctrl->sci_ctrl.atc_buff" is set.
- 2nd argument: AT command table gp_at_command[] is set with the member added in Figure 24 specified in the square brackets.
- 3rd argument: NULL is set because the AT command takes no argument.

(2) The cellular_execute_at_command() function is used to transmit the AT command to the cellular module.

- 1st argument: "p_ctrl" is set.
- 2nd argument: The time before a wait for a response returned from the cellular module times out is set.
- 3rd argument: The expected value for the response returned from the cellular module is set. For most commands, "ATC_RETURN_OK" is set.
- 4th argument: The number of the AT command to be transmitted (the member added in Figure 24) is set.

4.4.7 Adding the Declaration of the New Function

In the at_command.h file, add the declaration of atc_cpas(), which is the new function that you created.

```
659 /**********************************************************************
660  * Function Name  @fn            atc_cpas
661  * Description    @details       Execute the AT command (AT+CPAS).
662  * Arguments      @param[in/out] p_ctrl -
663  *                                  Pointer to managed structure.
664  * Return Value   @retval        CELLULAR_SUCCESS -
665  *                                  Successfully executed AT command.
666  *                @retval        CELLULAR_ERR_MODULE_COM -
667  *                                  Communication with module failed.
668  **********************************************************************/
669 e_cellular_err_t atc_cpas (st_cellular_ctrl_t * const p_ctrl);
```

Figure 28. Code section where the declaration of atc_cpas() has been added

4.4.8 Adding a Macro

In the cellular_receive_task.h file, add the macro definition of the response string “+CPAS:”.

```
79 #define ATC_RES_GET_CERTIFICATE       "+SQNSNVR:"
80 #define ATC_RES_GET_MT_STATE          "+CPAS:"
```

Figure 29. Code section where the macro definition of “+CPAS:” has been added

4.4.9 Adding a New Member to the Enumeration

Add a new member to the enumerated type e_atc_return_code_t defined in the cellular_receive_task.h file.

Figure 30 shows the member CELLULAR_GET_MT_STATE that has been added in the enumerated type. Assume that this member has been added as the 42nd member and therefore given a value of 41.

```
127     CELLULAR_GET_CERTIFICATE,           // Get Certificate
128     CELLULAR_GET_MT_STATE,              // Get Phone Activity Status
129     CELLULAR_RES_MAX,                   // End of analysis result processing
```

Figure 30. Code section (in the enumerated type e_atc_return_code_t) where a new member has been added

4.4.10 Adding a Constant and Storing a String

In the r_cellular_receive_task.c file, add the definition of a string constant, and then, in the constant, store the string added in section 4.4.8.

```
96 static const uint8_t s_atc_res_get_certificate[]     = ATC_RES_GET_CERTIFICATE;
97 static const uint8_t s_atc_res_get_mt_state[]        = ATC_RES_GET_MT_STATE;
```

Figure 31. Code section where a string constant has been added

4.4.11 Adding the Address of a String

In the pointer array sp_cellular_atc_res_tbl[] defined in the r_cellular_receive_task.c file, add the address of the string that you added in section 4.4.10.

Make sure that you add the address at the same position as the position of the member added in Figure 30. Because CELLULAR_GET_MT_STATE is the 42nd member and has a value of 41, add the address to the array sp_cellular_atc_res_tbl[] as the 42nd member (at the position of element 41) as shown in Figure 32.

```
141        s_atc_res_get_certificate,
142        s_atc_res_get_mt_state,
143  };
```

Figure 32. Code section (in the array sp_cellular_atc_res_tbl[]) where the address has been added

4.4.12 Adding the Declaration of the Function

In the r_cellular_receive_task.c file, add the declaration of the function in which to implement the processing to be performed when the response string “+CPAS:” is received.

Make sure that you specify the same arguments that are specified in the other functions.

```
186  static void cellular_get_certificate (st_cellular_ctrl_t * p_ctrl, st_cellular_receive_t * cellular_receive);
187  static void cellular_get_mt_state (st_cellular_ctrl_t * p_ctrl, st_cellular_receive_t * cellular_receive);
188  static void cellular_get_revision (st_cellular_ctrl_t * p_ctrl, st_cellular_receive_t * cellular_receive);
```

Figure 33. Code section where the function that performs processing upon reception of “+CPAS:” has been declared

4.4.13 Adding the Function to the Function Pointer Table

In the function pointer table p_cellular_recvtask_api[] defined in the r_cellular_receive_task.c file, add the function that was declared in section 4.4.12.

Make sure that you add the address at the same position as the position of the member added in

Figure 30. Because CELLULAR_GET_MT_STATE is the 42nd member and has a value of 41, add the function to the array p_cellular_recvtask_api[] as the 42nd member (at the position of element 41) as shown in Figure 34.

```
243        cellular_get_certificate,
244        cellular_get_mt_state,
245        cellular_response_skip,
```

Figure 34. Code section where the declaration of a new function has been added

4.4.14 Adding the Response Processing Function

In the r_cellular_receive_task.c file, add the function that was declared in section 4.4.12.

Examples

```
static void cellular_get_mt_state(st_cellular_ctrl_t     * p_ctrl,
                                  st_cellular_receive_t * cellular_receive)
{
    st_cellular_receive_t * p_cellular_receive = cellular_receive;
    uint8_t                 * p_state             = p_ctrl->recv_data; (1)

    if (CHAR_CHECK_4 == p_cellular_receive->data)  (2)
    {
        if (NULL != p_state)
        {
            sscanf((char *)&p_ctrl->sci_ctrl.receive_buff[p_cellular_receive- (3)
>tmp_recvcnt],
                        " %hhd", (char *)p_state);
        }
        cellular_cleardata(p_ctrl, p_cellular_receive);
    }

    return;
}
```

(1) The address of the buffer that is used to pass a value is copied to a local variable.

(2) The termination character (\n) is confirmed and processing starts.

(3) processing is performed only if an address has been stored in p_ctrl->recv_data in step (1).
The characters that follow the received string “+CPAS:” are stored in p_state.
In this example, because the string “\r\n+CPAS: <pas>\r\n” has been stored in the 1st argument of the sscanf(), the characters after a colon (a halfwidth space and the following characters) are stored in p_state.

4.4.15 Supplementary Note

For a function (atc_cpas(), in this example) to receive data via p_ctrl->recv_data as in section 4.4.14, the address of the data write destination must be stored in p_ctrl->recv_data beforehand.

After the data write is completed, be sure to set NULL for p_ctrl->recv_data.

Examples

```
/* Case where processing is performed inside the function */
void Examples_API1(st_cellular_ctrl_t * const p_ctrl)
{
    e_cellular_err_t             ret;
    e_cellular_err_semaphore_t semahore_ret;
    uint8_t                        state = 0;

    semahore_ret = cellular_take_semaphore(p_ctrl->at_semaphore);
    if (CELLULAR_SEMAPHORE_SUCCESS == semahore_ret)
    {
        p_ctrl->recv_data = (void *) &state;
        ret = atc_cpas(p_ctrl);
        p_ctrl->recv_data = NULL;
        cellular_give_semaphore(p_ctrl->at_semaphore);
    }

    return;
}

/* Case where data is returned to the user */
void Examples_API2(st_cellular_ctrl_t * const p_ctrl,
                   uint8_t               * p_state)
{
    e_cellular_err_t             ret;
    e_cellular_err_semaphore_t semahore_ret;

    semahore_ret = cellular_take_semaphore(p_ctrl->at_semaphore);
    if (CELLULAR_SEMAPHORE_SUCCESS == semahore_ret)
    {
        p_ctrl->recv_data = (void *) p_state;
        ret = atc_cpas(p_ctrl);
        p_ctrl->recv_data = NULL;
        cellular_give_semaphore(p_ctrl->at_semaphore);
    }

    return;
}
```

4.5 AT Command Addition Example: Command That Returns a Response Consisting of Multiple Lines That Contain the String “+*xxx*:” When Processing Ends Normally

This section describes the procedure for modifying the source code in the case when adding an AT command that returns a response consisting of multiple lines that contain the string “+*xxx*:” when processing ends normally.

Continuing from the previous section 4.4, this section also provides an example of adding a new command. The command you add in this example is “AT+COPN”. For details about the AT command specifications, refer to related document [10].

```
AT+COPN

+COPN: "20205","vodafone GR"
+COPN: "20404","vodafone NL"
+COPN: "20408","NL KPN"
```

(omitted)

```
+COPN: "74801","Antel"
+COPN: "90112","Telenor Maritime"
+COPN: "90171","Tampnet"

OK
```

7.9 Read Operator Names: AT+COPN

Note: This command is described in 3GPP TS 27.007. See Section References.

7.9.1 Syntax

Command	Possible Response(s)
AT+COPN	+COPN:<numeric1>,<alpha1>[<S3><S4>+COPN:<numeric2>,<alpha2>[...]] +CME ERROR: <err>

Figure 35. AT+COPN command to be added

4.5.1 Adding a Macro

In the ryz014a_private.h file, add a macro that defines the format of the AT command that you want to add.

```
160 #define RYZ014_ATC_GET_MT_STATE                "AT+CPAS\r"
161 #define RYZ014_ATC_GET_OPERATOR_LIST           "AT+COPN\r"
162 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 36. Code section where the new command "AT+COPN" has been added

4.5.2 Adding a New Member to the Enumeration

Add a new member to the enumerated type e_atc_list_t defined in the ryz014a_private.h file.

Figure 37 shows the member ATC_GET_OPERATOR_LIST that has been added in the enumerated type. Assume that this member has been added as the 71st member and therefore given a value of 72.

```
244     ATC_GET_MT_STATE,
245     ATC_GET_OPERATOR_LIST,
246 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 37. Code section (in the enumerated type e_atc_list_t) where a new member has been added

4.5.3 Adding a Constant and Storing a String

In the at_command.c file, add a string constant, and then, in the constant, store the string added in section 4.5.1.

```
112 const uint8_t g_ryz014_get_mt_state[]          = RYZ014_ATC_GET_MT_STATE;
113 const uint8_t g_ryz014_get_operator_list[]     = RYZ014_ATC_GET_OPERATOR_LIST;
114 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 38. Code section where a string constant has been added

4.5.4 Adding the Address of a String

In the pointer array gp_at_command[] defined in the at_command.c file, add the address of the string that you added in section 4.5.3.

Make sure that you add the address at the same position as the position of the member added in Figure 37. Because ATC_GET_OPERATOR_LIST is the 73rd member and has a value of 72, add the address to the array gp_at_command[] as the 73rd member (at the position of element 72) as shown in Figure 39.

```
190     g_ryz014_get_mt_state,
191     g_ryz014_get_operator_list,
192 #if (CELLULAR_IMPLEMENT_TYPE == 'B')
```

Figure 39. Code section (in the array gp_at_command) where a new member has been added

4.5.5 Creating a New File

Create a new file in the "r_cellular/src/at_command/ryz014" folder.

We recommend that you copy and rename ceer.c or another existing file for a command that takes no argument.

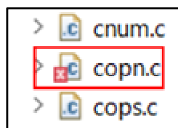

Figure 40. Folder where the new file "copn.c" has been created

4.5.6 Creating a Function That Executes the AT Command

In the .c file that you added in section 4.5.5, create a function that executes the "AT+COPN" command.

Examples

```
#include "at_command.h"
#include "cellular_private_api.h"

e_cellular_err_t atc_copn(st_cellular_ctrl_t * const p_ctrl)
{
    e_cellular_err_t ret = CELLULAR_SUCCESS;

    atc_generate(p_ctrl->sci_ctrl.atc_buff,
                 gp_at_command[ATC_GET_OPERATOR_LIST],(1)
                 NULL);

    ret = cellular_execute_at_command(p_ctrl,
                                      p_ctrl->sci_ctrl.atc_timeout,
                                      ATC_RETURN_OK,                    (2)
                                      ATC_GET_OPERATOR_LIST);

    return ret;
}
```

(1) The atc_generate() function is used to generate the AT command to be transmitted to the cellular module.

- 1st argument: "p_ctrl->sci_ctrl.atc_buff" is set.
- 2nd argument: AT command table gp_at_command[] is set with the member added in Figure 37 specified in the square brackets.
- 3rd argument: NULL is set because the AT command takes no argument.

(2) The cellular_execute_at_command() function is used to transmit the AT command to the cellular module.

- 1st argument: "p_ctrl" is set.
- 2nd argument: The time before a wait for a response returned from the cellular module times out is set.
- 3rd argument: The expected value for the response returned from the cellular module is set.
  For most commands, "ATC_RETURN_OK" is set.
- 4th argument: The number of the AT command to be transmitted (the member added in Figure 37) is set.

4.5.7 Adding the Declaration of the New Function

In the at_command.h file, add the declaration of atc_copn(), which is the new function that you created.

```
912 /**********************************************************************************************
913  * Function Name   @fn            atc_copn
914  * Description     @details       Execute the AT command (COPN). / Obtains operator names.
915  * Arguments       @param[in/out] p_ctrl -
916  *                                   Pointer to managed structure.
917  * Return Value    @retval        CELLULAR_SUCCESS -
918  *                                   Successfully executed AT command.
919  *                 @retval        CELLULAR_ERR_MODULE_COM -
920  *                                   Communication with module failed.
921  *********************************************************************************************/
922 e_cellular_err_t atc_copn (st_cellular_ctrl_t * const p_ctrl);
```

Figure 41. Code section where the declaration of “atc_copn()” has been added

4.5.8 Adding a Macro

In the cellular_receive_task.h file, add the macro definition of the response string “+COPN:”.

```
80 #define ATC_RES_GET_MT_STATE          "+CPAS:"
81 #define ATC_RES_GET_OPERATOR_LIST     "+COPN:"
```

Figure 42. Code section where the macro definition of “+COPN:” has been added

4.5.9 Adding a New Member to the Enumeration

Add a new member to the enumerated type e_atc_return_code_t defined in the cellular_receive_task.h file.

Figure 43 shows the member CELLULAR_GET_MT_STATE that has been added in the enumerated type. Assume that this member has been added as the 43rd member and therefore given a value of 42.

```
129     CELLULAR_GET_MT_STATE,              // Get Phone Activity Status
130     CELLULAR_GET_OPERATOR_LIST,         // Get Operator names
131     CELLULAR_RES_MAX,                   // End of analysis result processing
```

Figure 43. Code section (in the enumerated type e_atc_return_code_t) where a new member has been added

4.5.10 Adding a Constant and Storing a String

In the r_cellular_receive_task.c file, add the definition of a string constant, and then, in the constant, store the string added in section 4.5.8.

```
97 static const uint8_t s_atc_res_get_mt_state[]       = ATC_RES_GET_MT_STATE;
98 static const uint8_t s_atc_res_get_operator_list[]  = ATC_RES_GET_OPERATOR_LIST;
```

Figure 44. Code section where a string constant has been added

4.5.11 Adding the Address of a String

In the pointer array sp_cellular_atc_res_tbl[] defined in the r_cellular_receive_task.c file, add the address of the string that you added in section 4.5.10.

Make sure that you add the address at the same position as the position of the member added in Figure 43. Because CELLULAR_GET_MT_STATE is the 43rd member and has a value of 42, add the address to the array sp_cellular_atc_res_tbl[] as the 43rd member (at the position of element 42) as shown in Figure 45.

```
143        s_atc_res_get_mt_state,
144        s_atc_res_get_operator_list,
145    };
```

Figure 45. Code section (in the array sp_cellular_atc_res_tbl[]) where the address has been added

4.5.12 Adding the Declaration of the Function

In the r_cellular_receive_task.c file, add the declaration of the function in which to implement the processing to be performed when the response string "+COPN:" is received.

Make sure that you specify the same arguments that are specified in the other functions.

```
186 static void cellular_get_mt_state (st_cellular_ctrl_t * p_ctrl, st_cellular_receive_t * cellular_receive);
187 static void cellular_get_operator_list (st_cellular_ctrl_t * p_ctrl, st_cellular_receive_t * cellular_receive);
188 static void cellular_get_revision (st_cellular_ctrl_t * p_ctrl, st_cellular_receive_t * cellular_receive);
```

Figure 46. Code section where the function that performs processing upon reception of "+COPN:" has been declared

4.5.13 Adding the Function to the Function Pointer Table

In the function pointer table p_cellular_recvtask_api[] defined in the r_cellular_receive_task.c file, add the function that was declared in section 4.5.12.

Make sure that you add the address at the same position as the position of the member added in Figure 43. Because CELLULAR_GET_MT_STATE is the 43rd member and has a value of 42, add the function to the array p_cellular_recvtask_api[] as the 43rd member (at the position of element 42) as shown in Figure 47.

```
247        cellular_get_mt_state,
248        cellular_get_operator_list,
249        cellular_response_skip,
```

Figure 47. Code section where the declaration of a new function has been added

4.5.14 Adding the Function

In the r_cellular_receive_task.c file, add the function that was declared in section 4.5.12.

Examples

```
static void cellular_get_operator_list(st_cellular_ctrl_t     * p_ctrl,
                                       st_cellular_receive_t * cellular_receive)
{
    st_cellular_receive_t * p_cellular_receive = cellular_receive;
    sci_err_t                  sci_ret;

    if (CHAR_CHECK_4 == p_cellular_receive->data)  (1)
    {
        cellular_cleardata(p_ctrl, p_cellular_receive); (2)

        do
        {
            sci_ret = R_SCI_Receive(p_ctrl->sci_ctrl.sci_hdl,            (3)
                                     &p_cellular_receive->data, 1);
            cellular_delay_task(1);
        } while (SCI_SUCCESS != sci_ret);

        p_ctrl->sci_ctrl.receive_buff[0] = p_cellular_receive->data;
        p_cellular_receive->recv_count++;
        if (CHAR_CHECK_1 == p_cellular_receive->data)
        {                                                                     (4)
            p_cellular_receive->job_no       = CELLULAR_GET_OPERATOR_LIST;
            p_cellular_receive->tmp_recvcnt = 6;
        }
    }

    return;
}
```

(1) The termination character (\n) is confirmed and processing starts.

(2) The received string is deleted.

(3) To check whether multiple instances of the string “+COPN:” have been sent in succession, this loop is repeated until a character is received.
As shown in Figure 35, multiple instances of the string “+COPN:” have been sent in succession (+COPN: ... <S3><S4>+COPN: ...).
Note that “<S3><S4>” in the figure means “\r\n”. Therefore, <S4> is the termination character “\n” confirmed in step (1).
If multiple instances of the string “+COPN:” are sent in succession, the function will then receive the character “+”.

(4) The character received in step (3) is stored in the receive buffer for recv_task, and the number of received characters is incremented.
After that, the received character is confirmed.
If the received character is “+”, it means that multiple instances of the string “+COPN:” have been sent.
Therefore, the member added in Figure 43 is assigned to job_no so that the function in the code example is called again.

4.6 Command that Returns an Intermediate Result Code as a Response When Processing Ends Normally

This section describes the procedure for modifying the source code in the case when adding an AT command that returns an intermediate result code as a response when processing ends normally.

The explanation in this section uses the AT+SQNSSENDEXT command that has already been implemented as an example.

```
AT+SQNSSENDEXT=1,10

> 0123456789
OK
```

2.2.6 Extended Send Data In Command Mode: AT+SQNSSENDEXT

2.2.6.1 Syntax

Command	Possible Response(s)
AT+SQNSSENDEXT=<connId>,<bytesToSend>[,<RAI>]	Intermediate result code: > OK ERROR NO CARRIER +CME ERROR:<err>

Figure 48. “AT+SQNSSENDEXT”, which is an AT command that returns an intermediate result code

RENESAS

4.6.1 Adding a Macro

In the ryz014a_private.h file, add a macro that defines the format of the AT command that you want to add.

```
95  #define RYZ014_ATC_CLOSE_SOCKET              "AT+SQNSH=%s\r"
96  #define RYZ014_ATC_SEND_SCOKET               "AT+SQNSSENDEXT=%s,%s\r"
97  #define RYZ014_ATC_RECV_SCOKET               "AT+SQNSRECV=%s,%s\r"
```

Figure 49. Code section where "AT+SQNSSENDEXT" has been added

4.6.2 Adding a New Member to the Enumeration

Add a new member to the enumerated type e_atc_list_t defined in the ryz014a_private.h file.

Figure 50 shows the member ATC_SEND_SOCKET that has been added in the enumerated type. Assume that this member has been added as the 8th member and therefore given a value of 7.

```
178     ATC_CLOSE_SOCKET,
179     ATC_SEND_SOCKET,
180     ATC_RECV_SOCKET,
```

Figure 50. Code section (in the enumerated type e_atc_list_t) where a new member has been added

4.6.3 Adding a Constant and Storing a String

In the at_command.c file, add a string constant, and then, in the constant, store the string added in section 4.6.1.

```
47  const uint8_t g_ryz014_close_socket[]           = RYZ014_ATC_CLOSE_SOCKET;
48  const uint8_t g_ryz014_send_socket[]            = RYZ014_ATC_SEND_SCOKET;
49  const uint8_t g_ryz014_recv_socket[]            = RYZ014_ATC_RECV_SCOKET;
```

Figure 51. Code section where a string constant has been added

4.6.4 Adding the Address of a String

In the pointer array gp_at_command[] defined in the at_command.c file, add the address of the string that you added in section 4.6.3.

Make sure that you add the address at the same position as the position of the member added in Figure 50. Because ATC_SEND_SOCKET is the 8th member and has a value of 7, add the address to the array gp_at_command[] as the 8th member (at the position of element 7) as shown in Figure 52.

```
127     g_ryz014_close_socket,
128     g_ryz014_send_socket,
129     g_ryz014_recv_socket,
```

Figure 52. Code section (in the array gp_at_command[]) where a new member has been added

4.6.5 Adding a Macro

In the cellular_receive_task.h file, add the macro definition of the intermediate result code string.

```
39  #define ATC_RES_GO_SEND                        ">"
40  #define ATC_RES_OK                             "OK\r\n"
```

Figure 53. Code section where the macro definition of an intermediate result code has been added

4.6.6 Adding a New Member to the Enumeration

Add a new member to the enumerated type e_atc_return_code_t defined in the cellular_receive_task.h file.

```
88      CELLULAR_RES_GO_SEND = 0,          // Request for Data Transmission
89      CELLULAR_RES_OK,                   // Response is OK
```

Figure 54. Code section (in the enumerated type e_atc_return_code_t) where a new member has been added

4.6.7 Adding a Constant and Storing a String

In the r_cellular_receive_task.c file, add a string constant, and then, in the constant, store the string added in section 4.6.5.

```
56  static const uint8_t s_atc_res_go_send[]            = ATC_RES_GO_SEND;
57  static const uint8_t s_atc_res_ok[]                 = ATC_RES_OK;
```

Figure 55. Code section where a string constant has been added

4.6.8 Adding the Address of a String

In the pointer array sp_cellular_atc_res_tbl[] defined in the r_cellular_receive_task.c file, add the address of the string that you added in section 4.6.7.

Make sure that you add the address at the same position as the position of the member added in Figure 54. Because CELLULAR_RES_GO_SEND is the 1st member and has a value of 0, add the address to the array sp_cellular_atc_res_tbl[] as the 1st member (at the position of element 0) as shown in Figure 56.

```
102     s_atc_res_go_send,
103     s_atc_res_ok,
```

Figure 56. Code section (in the array sp_cellular_atc_res_tbl[]) where the address has been added

4.6.9 Adding a New Member to the Enumeration

Add a new member to the enumerated type e_cellular_atc_return_t defined in the ryz014a_private.h file.

Make sure that you add the new member to a location before (above) the keyword ATC_RETURN_ENUM_MAX.

```
252  typedef enum
253  {
254      ATC_RETURN_NONE = 0,          // No response from the module
255      ATC_RETURN_OK,                // Module response is "OK"
256      ATC_RETURN_ERROR,             // Module response is "ERROR"
257      ATC_RETURN_OK_GO_SEND,        // Module response is ">"
258      ATC_RETURN_SEND_NO_CARRIER,   // Module response is "NO CARRIER"
259      ATC_RETURN_AP_CONNECTING,     // Module response is "CONNECT"
260      ATC_RETURN_ENUM_MAX,          // Maximum enumeration value
261  } e_cellular_atc_return_t;
```

Figure 57. Code section (in the enumerated type e_cellular_atc_return_t) where a new member has been added

4.6.10 Creating a New File

Create a new file in the “r_cellular/src/at_command/ryz014” folder.

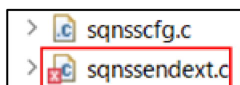

Figure 58. Folder where the New file “sqnssendext.c” has been created

4.6.11 Creating a Definition for Executing the AT Command

In the .c file that you created in section 4.6.10, create a function that executes the "AT+SQNSSENDEXT" command.

Examples

```
#include "at_command.h"
#include "cellular_private_api.h"

e_cellular_err_t atc_sqnssendext(st_cellular_ctrl_t * const p_ctrl,
                                 const uint8_t socket_no,
                                 const uint16_t length)
{
    uint8_t           str[2][10]                             = {0}; (1)
    const uint8_t * p_command_arg[CELLULAR_MAX_ARG_COUNT] = {0};
    e_cellular_err_t ret                                     =
CELLULAR_SUCCESS;

    sprintf((char *)str[0], "%d", socket_no);   // (uint8_t *)->(char *)
    sprintf((char *)str[1], "%d", length);      // (uint8_t *)->(char *)
                                                                    (2)
    p_command_arg[0] = str[0];
    p_command_arg[1] = str[1];

    atc_generate(p_ctrl->sci_ctrl.atc_buff,
                 gp_at_command[ATC_SEND_SOCKET],                 (3)
                 p_command_arg);

    ret = cellular_execute_at_command(p_ctrl,
                                      p_ctrl->sci_ctrl.atc_timeout, (4)
                                      ATC_RETURN_OK_GO_SEND,
                                      ATC_SEND_SOCKET);

    return ret;
}
```

(1) A variable is specified as an argument given to the AT command.

(2) The argument to be given to the AT command is stored in the variable "str", and then the start address of the string stored in "str" is stored in the variable "p_command_arg".

(3) The atc_generate() function is used to generate the AT command to be transmitted to the cellular module.

- 1st argument: "p_ctrl->sci_ctrl.atc_buff" is set.
- 2nd argument: AT command table gp_at_command[] is set with the member added in Figure 50 specified in the square brackets.
- 3rd argument: The variable "p_command_arg" prepared in step (2) is set.

(4) The cellular_execute_at_command() function is used to transmit the AT command to the cellular module.

- 1st argument: "p_ctrl" is set.
- 2nd argument: The time before a wait for a response returned from the cellular module times out can be specified.
- 3rd argument: The expected value for the response returned from the cellular module
  The member added in Figure 57 is set.
- 4th argument: The number of the AT command to be transmitted (the member added in Figure 50 is specified.)

4.6.12 Adding a Macro

In the r_cellular_receive_task.c file, add the macro definition of the intermediate result code. It is used to judge whether an intermediate result code was received.

```
36  #define CHAR_CHECK_1     ('+')
37  #define CHAR_CHECK_2     ('>')
```

Figure 59. Code section where the macro definition of an intermediate result code has been added

4.6.13 Modifying the Processing of the "case" Statement

In the r_cellular_receive_task.c file, add the processing shown in Figure 60 to the cellular_job_check() function.

The macro you specify in the "if" statement is the macro added in section 4.6.12. For "p_cellular_receive->job_no", set the member added in section 4.6.6.

```
347         case JOB_STATUS_FIRST_CHAR_CHECK:
348         {
349             if ((char)p_cellular_receive->data == (CHAR_CHECK_1))
350             {
351                 p_cellular_receive->job_status = JOB_STATUS_COLON_CHECK;
352             }
353             else if ((char)p_cellular_receive->data == (CHAR_CHECK_2))
354             {
355                 p_cellular_receive->job_no = CELLULAR_RES_GO_SEND;
356             }
357             else
358             {
359                 p_cellular_receive->job_no = CELLULAR_RES_CHECK;
360             }
361             break;
362         }
```

Figure 60. Code section (in the cellular_job_check() function) where processing has been added

4.6.14 Adding the Declaration of the Function

In the r_cellular_receive_task.c file, add the declaration of the function in which to implement the processing performed upon reception of an intermediate result code.

Make sure that you specify the same arguments that are specified in the other functions.

```
150 static void cellular_response_check (st_cellular_ctrl_t * p_ctrl, st_cellular_receive_t * cellular_receive);
151 static void cellular_data_send_command (st_cellular_ctrl_t * p_ctrl, st_cellular_receive_t * cellular_receive);
152 static void cellular_get_data_reception (st_cellular_ctrl_t * p_ctrl, st_cellular_receive_t * cellular_receive);
```

Figure 61. Code section where a function that performs processing upon reception of an intermediate result code is declared

4.6.15 Adding the Function to the Function Pointer Table

In the function pointer table p_cellular_recvtask_api[] defined in the r_cellular_receive_task.c file, add the function that was declared in section 4.6.14.

Make sure that you add the address at the same position as the position of the member added in Figure 54. Because CELLULAR_RES_GO_SEND is the 1st member and has a value of 0, add the function to the array p_cellular_recvtask_api[] as the 1st member (at the position of element 0) as shown in Figure 62.

```
206     cellular_data_send_command,
207     cellular_memclear,
```

Figure 62. Code section where the declaration of a new function has been added

4.6.16 Adding the Function

In the r_cellular_receive_task.c file, add the function that was declared in section 4.6.14.

Examples

```
static void cellular_data_send_command(st_cellular_ctrl_t * p_ctrl,
                                       st_cellular_receive_t * cellular_receive)
{
    st_cellular_receive_t * p_cellular_receive = cellular_receive;

    if (CHAR_CHECK_6 == p_cellular_receive->data) (1)
    {
        cellular_set_atc_response(p_ctrl, ATC_RETURN_OK_GO_SEND); (2)
        cellular_memclear(p_ctrl, p_cellular_receive);  (3)
    }

    return;
}
```

(1) The function here confirms that an incoming intermediate result code is followed by a normal string. Because the RYZ014A cellular module sends a halfwidth space after a right angle bracket (>), a macro for checking a halfwidth space is defined.

```
41  #define CHAR_CHECK_6    (' ')
```

(2) Because reception of a right angle bracket followed by a halfwidth space ("> ") was confirmed, the cellular_set_atc_response() function is used to report the normal reception result of the intermediate result code.
Make sure that the value set for the 2nd argument is the same as the value set for the 3rd argument of the cellular_execute_at_command() function called in step (4) in section 4.6.11.

(3) Because the AT command completed its processing up to the handling of the intermediate result code, the cellular_memclear() function is used to clear the information.

4.6.17 Adding a New Member to the Enumeration

Add a new member to the enumerated type e_atc_list_t defined in the ryz014a_private.h file.

An AT command string macro has not been provided for the intermediate result code unlike the macro defined in section 4.6.1. Therefore, be sure to add the new member above the keyword ATC_LIST_MAX.

```
248     ATC_SQNSSENDEXT_END,
249     ATC_LIST_MAX
250 } e_atc_list_t;
```

Figure 63. Code section (in the enumerated type e_atc_list_t) where a new member has been added

4.6.18 Creating Postprocessing

Create the processing to be performed after the receive task for the intermediate result code is completed.

When the RYZ014A cellular driver receives the intermediate result code (>), it sends the transmit data to the cellular module.

Examples

The following code is a part of the processing performed by the cellular_send_data() function in the r_cellular_sendsocket.c file.

```
ret = atc_sqnssendext(p_ctrl, socket_no, send_size);   (1)
if (CELLULAR_SUCCESS != ret)
{
    break;
}
p_ctrl->sci_ctrl.tx_end_flg = CELLULAR_TX_END_FLAG_OFF;   (2)

sci_ret = R_SCI_Send(p_ctrl->sci_ctrl.sci_hdl,
                    (uint8_t *)p_data + complete_length, send_size);   (3)

if (SCI_SUCCESS != sci_ret)
{
    ret = CELLULAR_ERR_MODULE_COM;   (4)
    break;
}

cellular_set_atc_number(p_ctrl, ATC_SQNSSENDEXT_END);   (5)

timeout = cellular_tx_flag_check(p_ctrl, socket_no);   (6)
if (CELLULAR_TIMEOUT != timeout)
{
    timeout = cellular_atc_response_check(p_ctrl, socket_no);   (7)
}

if (CELLULAR_TIMEOUT == timeout)
{
    ret = CELLULAR_ERR_MODULE_TIMEOUT;   (8)
    break;
}
```

(1) The function created in section 4.6.11 is executed.

(2) The processing in (1) ended normally, so the function then prepares for sending the transmit data to the cellular module.
The flag for checking whether the sending of the transmit data was completed normally is initialized.

(3) The R_SCI_Send() function of the SCI FIT module is executed in order to send the transmit data to the cellular module.

(4) The result of the processing in (3) is confirmed.
If the processing abnormally ended, the data transmission processing is forcibly ended.

(5) Because the R_SCI_Send() function was completed normally, the response string sent from the cellular module is confirmed.
The cellular_set_atc_number() function is executed to prepare for handling the response string.
For the 2nd argument of the cellular_set_atc_number() function, specify the member that was added in section 4.6.17.

(6) The function confirms that the data transmission to the cellular module was completed.

(7) The function confirms that the response string “OK” is returned from the cellular module.

(8) If the function could not confirm that the response string “OK” has been returned in (7), the data transmission processing is forcibly ended.

5. Processing Reusable for Other Cellular Modules

RYZ014A Cellular FIT Module uses AT commands that are compliant with the 3GPP standards and AT commands that are dedicated to the RYZ014A cellular module. The names of these dedicated commands begin with the string "SQN". The code where 3GPP-compliant AT commands are used can be reused for any cellular modules other than RYZ014A. However, the code where RYZ014A-dedicated AT commands are used and the code affected by the results of these dedicated commands must be replaced by appropriate alternative processing according to the cellular module to be used.

The following shows an example of processing if the code where 3GPP-compliant AT commands that are implemented in RYZ014A Cellular FIT Module is reused:

(1) Acquires the semaphore for AT commands.

(2) Executes the function for executing the AT command.

(3) Releases the semaphore for the AT command.

Before you can use functions for executing AT commands that have already been implemented in Cellular FIT Module, the R_CELLULAR_Open() function must have been executed.

Examples

```
e_cellular_err_t             ret         = CELLULAR_SUCCESS;
e_cellular_err_semaphore_t semahore_ret = CELLULAR_SEMAPHORE_SUCCESS;

semahore_ret = cellular_take_semaphore(p_ctrl->at_semaphore); (1)
if (CELLULAR_SEMAPHORE_SUCCESS == semahore_ret)
{
    atc_cfun(p_ctrl, CELLULAR_MODULE_OPERATING_LEVEL4); (2)
    cellular_give_semaphore(p_ctrl->at_semaphore); (3)
}
else
{
    ret = CELLULAR_ERR_OTHER_ATCOMMAND_RUNNING;
}
```

(1) Before the API function for an AT command is executed, the semaphore for the AT command is acquired.
p_ctrl is a pointer to st_cellular_ctrl_t that was set for the 1st argument when the R_CELLULAR_Open() function was executed.

(2) A function that sends the 3GPP-compliant AT command "AT+CFUN" is executed.
A value is specified for the 2nd argument.

(3) The semaphore for the AT command is released.

5.1 R_CELLULAR_Open()

The following shows the flow of the R_CELLULAR_Open() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Stores the “p_ctrl” argument in the “gp_cellular_ctrl” global variable.

(3) Applies the configuration value stored in the “p_cfg” argument. (If “p_cfg” contains NULL, the default value is applied.)

(4) Enables the serial communication function.

(5) Uses the OS functionality to obtain semaphores and to create event groups and data reception tasks.

(6) Synchronizes with the data reception task.

(6) Performs a hardware reset (by executing the cellular_module_reset() function on line 145).

(8) Issues the “ATE0” command.

(9) Issues the “AT+SQNSIMST=0” command.

(10) Issues the “AT+CEREG=x” command (x = CELLULAR_CFG_NETWORK_NOTIFY_LEVEL).

(11) Issues the “AT+CFUN=4” command.

(12) Issues the “AT+CPIN?” command.
If a SIM PIN lock is enabled, the “AT+CPIN=x” command is issued.
(x = CELLULAR_CFG_PIN_CODE or p_cfg->sim_pin_code)

(13) Disables the thread-safe feature.

5.1.1 Reusable Processes

Processes (1) to (6) and (13) can be reused for any cellular modules. Note, however, that the default value applied in process (3) may differ depending on the cellular module. Therefore, set the appropriate value in the ryz014_private.h file.

5.1.2 Processes Needing Partial Replacement

(8) and (10) to (12) are the processes for enabling issuance of major AT commands to the cellular module. For some cellular modules, additional AT commands may need to be issued. Add processing if necessary.

In the private function for performing a hardware reset in process (7), commands dedicated to the RYZ014A cellular module are executed. Therefore, this process cannot be reused for other cellular modules. In the cellular_module_reset.c file, delete line 115 or replace the processing on that line by other processing.

The atc_sqnautoconnect_chek() function executes the “AT+SQNAUTOCONNECT?” command to check whether the function level is set to 1 (CFUN=1) automatically when the cellular module is activated.

```
115            ret                = atc_sqnautoconnect_check(p_ctrl);
```

Figure 64. Line 115 in the cellular_module_reset.c file

Process (9), which uses a command dedicated to the RYZ014A cellular module, cannot be used for other cellular modules. In the r_cellular_open.c file, delete line 289 or replace the processing on that line by other processing.

```
289            ret = atc_sqnsimst(p_ctrl);
```

Figure 65. Line 289 in the r_cellular_open.c file

5.2 R_CELLULAR_Close()

The following shows the flow of the R_CELLULAR_Close() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Disables the PSM (Power Saving Mode).

(3) Uses the OS functionality to delete the event groups, semaphores, and tasks (related to PSM control).

(4) Performs a hardware reset.

(5) Shuts down the cellular module (by executing the cellular_power_down() function).
Command used:
• AT+SQNSSHDN

(6) Uses the OS functionality to delete the event groups, semaphores, and tasks (related to AT command control).

(7) Uses the OS functionality to delete the semaphores and release memory (related to sockets).

(8) Disables serial communication.

(9) Disables the thread-safe feature.

5.2.1 Reusable Processes

Processes (1) to (4) and (6) to (9) can be reused for any cellular modules.

5.2.2 Processes Needing Replacement

Process (5), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other cellular modules. In the cellular_power_down.c file, replace the processing on line 62 by appropriate alternative processing.

```
62        ret                 = atc_sqnsshdn(p_ctrl);
```

Figure 66. Line 62 in the cellular_power_down.c file

5.3 R_CELLULAR_ APConnect()

The following shows the flow of the R_CELLULAR_APConnect() function processes:

(1) Checks the arguments and enables the thread-safe feature.
(2) Acquires the semaphore for AT commands.
(3) Configures connection to an access point (by executing the cellular_apconnect_config() function).
(4) Connects to an access point (by executing the cellular_apconnect() function).
(5) Releases the semaphore for AT commands.
(6) Disables the thread-safe feature.

5.3.1 Reusable Processes

Processes (1) to (6) can be reused for any cellular modules.

5.4 R_CELLULAR_IsConnected()

The following shows the flow of the R_CELLULAR_IsConnected() function processes:

(1) Checks the arguments and enables the thread-safe feature.
(2) Acquires the access point connection state from the management structure.
(3) Disables the thread-safe feature.

5.4.1 Reusable Processes

Processes (1) to (3) can be reused for any cellular modules.

5.5 R_CELLULAR_Disconnect()

The following shows the flow of the R_CELLULAR_Disconnect() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Shuts down the socket (by executing the cellular_shutdownsocket() function).
Command used:
• AT+SQNSH

(3) Closes the socket (by executing the cellular_closesocket() function).

(4) Disconnects from the access point (by executing the cellular_disconnect() function).
Command used:
• AT+CFUN=4

(5) Disables the thread-safe feature.

5.5.1 Reusable Processes

Processes (1) and (3) to (5) can be reused for any cellular modules.

5.5.2 Processes Needing Replacement

Process (2), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the cellular_shutdownsocket.c file, replace the processing on line 76 by appropriate alternative processing.

```
76            ret = atc_sqnsh(p_ctrl, socket_no);
```

Figure 67. Line 76 in the cellular_shutdownsocket.c file

5.6 R_CELLULAR_CreateSocket()

The following shows the flow of the R_CELLULAR_CreateSocket() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Searches for the numbers of unused sockets.

(4) Performs socket configuration (by executing the cellular_socket_cfg() function).
Commands used:
• AT+SQNSCFG=%s,1,%s,%s,%s,%s
• AT+SQNSCFGEXT=%s,1,0,0

(5) Releases the semaphore for AT commands.

(6) Disables the thread-safe feature.

5.6.1 Reusable Processes

Processes (1) to (3), (5), and (6) can be reused for any cellular modules.

5.6.2 Processes Needing Replacement

Process (4), which uses AT commands dedicated to the RYZ014A cellular module, cannot be used for other modules. In the R_cellular_createsocket.c file, replace the processing on lines 154 and 157 by appropriate alternative processing.

```
154        atc_ret = atc_sqnscfg(p_ctrl, (uint8_t)(socket_num + start_num));
```

Figure 68. Line 154 in the r_cellular_createsocket.c file

```
157            atc_ret = atc_sqnscfgext(p_ctrl, (uint8_t)(socket_num + start_num));
```

Figure 69. Line 157 in the r_cellular_createsocket.c file

5.7 R_CELLULAR_ConnectSocket()

The following shows the flow of the R_CELLULAR_ConnectSocket() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Connects to a socket.
Command used:
• AT+SQNSD=%s,%s,%s,"%s",0,%s,1

(4) Updates the status of the socket management structure.

(5) Releases the semaphore for AT commands.

(6) Disables the thread-safe feature.

5.7.1 Reusable Processes

Processes (1), (2), (5), and (6) can be reused for any cellular modules.

5.7.2 Processes Needing Replacement

Process (3), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. Process (4) also cannot be reused for other cellular modules because it processes the execution results of the AT command in process (3) (processes a received string that begins with "+"). In the r_cellular_connectsocket.c file, replace the processing on lines 114 to 131 by appropriate alternative processing.

```
114            ret = atc_sqnsd(p_ctrl, socket_no, p_ip_addr, port);
115            if (CELLULAR_SUCCESS == ret)
116            {
117                p_ctrl->p_socket_ctrl[socket_no - CELLULAR_START_SOCKET_NUMBER].socket_status
118                    = CELLULAR_SOCKET_STATUS_CONNECTED;
119                if (CELLULAR_PROTOCOL_IPV4 ==
120                        p_ctrl->p_socket_ctrl[socket_no - CELLULAR_START_SOCKET_NUMBER].ipversion)
121                {
122                    strncpy((char *)p_ctrl->p_socket_ctrl[socket_no - CELLULAR_START_SOCKET_NUMBER].ip_addr.ipv4,
123                            (char *)p_ip_addr, CELLULAR_IPV4_ADDR_LENGTH); //(uint8_t *)->(char *)
124                }
125                else
126                {
127                    strncpy((char *)p_ctrl->p_socket_ctrl[socket_no - CELLULAR_START_SOCKET_NUMBER].ip_addr.ipv6,
128                            (char *)p_ip_addr, CELLULAR_IPV6_ADDR_LENGTH); //(uint8_t *)->(char *)
129                }
130                p_ctrl->p_socket_ctrl[socket_no - CELLULAR_START_SOCKET_NUMBER].port = port;
131            }
```

Figure 70. Line 114 to 131 in the r_cellular_connectsocket.c file

5.8 R_CELLULAR_ShutdownSocket()

The following shows the flow of the R_CELLULAR_ShutdownSocket() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Shuts down the socket (by executing the cellular_shutdownsocket() function).
Command used:
• AT+SQNSH=%s

(3) Disables the thread-safe feature.

5.8.1 Reusable Processes

Processes (1) and (3) can be reused for any cellular modules.

5.8.2 Processes Needing Replacement

Process (2), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the cellular_shutdownsocket.c file, replace the processing on line 76 by appropriate alternative processing.

```
76              ret = atc_sqnsh(p_ctrl, socket_no);
```

Figure 71. Line 76 in the cellular_shutdownsocket.c file

5.9 R_CELLULAR_CloseSocket()

The following shows the flow of the R_CELLULAR_CloseSocket() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Shuts down the socket (by executing the cellular_shutdownsocket() function).
Command used:
• AT+SQNSH=%s

(3) Closes the socket (by executing cellular_closesocket()).

(4) Disables the thread-safe feature.

5.9.1 Reusable Processes

Processes (1), (3), and (4) can be reused for any cellular modules.

5.9.2 Processes Needing Replacement

Process (2), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the cellular_shutdownsocket.c file, replace the processing on line 76 by appropriate alternative processing.

```
76              ret = atc_sqnsh(p_ctrl, socket_no);
```

Figure 72. Line 76 in the cellular_shutdownsocket.c file

5.10 R_CELLULAR_SendSocket()

The following shows the flow of the R_CELLULAR_SendSocket() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Sends socket data (by executing the cellular_send_data() function).
Command used:
• AT+SQNSSENDEXT=%s,%s

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.10.1 Reusable Processes

Processes (1), (2), (4), and (5) can be reused for any cellular modules.

5.10.2 Processes Needing Replacement

Process (3), which performs socket data transmission processing that uses a command dedicated to the RYZ014A cellular module, cannot be used for other cellular modules. In the cellular_send_data() function, replace the processing on line 208 by appropriate alternative processing.

Section 4.6 provides information that you can reference when implementing the alternative processing.

```
208         ret = atc_sqnssendext(p_ctrl, socket_no, send_size);
```

Figure 73. Line 208 in the r_cellular_sendsocket.c file

5.11 R_CELLULAR_ReceiveSocket()

The following shows the flow of the R_CELLULAR_ReceiveSocket() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires a semaphore for receiving socket data.

(3) Receives socket data (by executing the cellular_receive_data() function).
Command used:
• AT+SQNSRECV=%s,%s

(4) Releases the semaphore for receiving socket data.

(5) Disables the thread-safe feature.

5.11.1 Reusable Processes

Processes (1), (2), (4), and (5) can be reused for any cellular modules.

5.11.2 Processes Needing Replacement

Process (3), which performs socket data reception processing that uses a command dedicated to the RYZ014A cellular module, cannot be used for other cellular modules. In the cellular_receive_data() function, replace the processing on lines 161 to 270 by appropriate alternative processing.

5.11.3 Details of the Processing of the cellular_receive_data() Function

Line 219: The processing on this line lets the function wait for a data reception notification from the cellular module.

→ The cellular module is set to report a string in the "+SQNSRING:<connId>,<recData>" format in response to reception of data from the connection destination when the AT command "AT+SQNSCFGEXT=%s,1,0,0" is executed during execution of R_CELLULAR_CreateSocket(). This string is processed on lines 544 to 575 in the r_cellular_receive_task.c file to determine which socket data was collected in, and how much (in bytes) data was collected in the socket. This processing allows the function to check the data collection state before issuing the data reception command, thus making it possible to stop issuing the command if no receive data is collected.
Note: If the processing on this line is unnecessary, this line can be deleted without harm.

Lines 230 and 231: The processing on these lines determines the requested size of data to receive.

→ The requested size of data to receive, the size of data that can actually be received, and the maximum size of data that can be received at one time are compared, and the appropriate size is selected. You can specify the size of data that can be received at one time, depending on the specifications of the cellular module. Modify the code according to the specifications of the cellular module.

Line 235: A command that requests data reception is issued.

→ Because a command dedicated to the RYZ014A cellular module is used, this process cannot be used for cellular modules other than RYZ014A.
In the r_cellular_receivesocket.c file, replace the processing on line 235 by appropriate alternative processing.

```
235            ret = atc_sqnsrecv(p_ctrl, socket_no, receive_size);
```

Figure 74. Line 235 in the r_cellular_receivesocket.c file

The "AT+SQNSRECV" command returns a response string in the following format:
+SQNSRECV:<connId>,<maxByte><CR><LF><data>
The receive task for the string up to <LF> is performed by the processing on lines 580 to 618 in the r_cellular_receive_task.c file. The receive task for <data> is performed by the processing on lines 631 to 654 in the r_cellular_receive_task.c file. Also modify the processing on these lines according to the specifications of the cellular module.

Line 240: The processing on this line obtains the total size of data that was received during execution of cellular_receive_data().

→ The data receive task is performed on lines 631 to 654 in the r_cellular_receive_task.c file. The total size of received data is counted by using p_ctrl->p_socket_ctrl[p_cellular_receive->socket_no].total_recv_count.

5.12 R_CELLULAR_DnsQuery()

The following shows the flow of the R_CELLULAR_DnsQuery() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a DNS query.
Command used:
• AT+SQNDNSLKUP="%s",%s

(4) The string obtained in (3) is stored in the 4th argument “p_addr”.

(5) Releases the semaphore for AT commands.

(6) Disables the thread-safe feature.

5.12.1 Reusable Processes

Processes (1), (2), (5), and (6) can be reused for any cellular modules.

5.12.2 Processes Needing Replacement

Process (3), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. Also, process (4), to which the result of process (3) is applied, cannot be used for other cellular modules. In the r_cellular_dnsquery.c file, replace the processing on lines 116 and 120 by appropriate alternative processing.

```
116                 ret = atc_sqndnslkup(p_ctrl, p_domain_name, ip_version);
```

Figure 75. Line 116 in the r_cellular_dnsquery.c file

```
120                 cellular_getip(p_ctrl, ip_version, p_addr);
```

Figure 76. Line 120 in the r_cellular_dnsquery.c file

5.13 R_CELLULAR_GetTime()

The following shows the flow of the R_CELLULAR_GetTime() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects time information.
Command used:
• AT+CCLK?

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.13.1 Reusable Processes

Processes (1) to (5) can be reused for any cellular modules.

5.14 R_CELLULAR_SetTime()

The following shows the flow of the R_CELLULAR_SetTime() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that sets time information.
Command used:
• AT+CCLK="%s/%s/%s,%s:%s:%s%s"

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.14.1 Reusable Processes

Processes (1) to (5) can be reused for any cellular modules.

5.15 R_CELLULAR_SetEDRX()

The following shows the flow of the R_CELLULAR_SetEDRX() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that specifies the eDRX settings and a command that acquires the settings.
Commands used:
• AT+SQNEDRX=%s,4,"%s","%s"
• AT+SQNEDRX?

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.15.1 Reusable Processes

Processes (1), (2), (4), and (5) can be reused for any cellular modules.

5.15.2 Processes Needing Replacement

Process (3), which uses AT commands dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_setedrx.c file, replace the processing on lines 93 and 97 by appropriate alternative processing.

```
93                ret = atc_sqnedrx(p_ctrl, p_config);
```

Figure 77. Line 93 in the r_cellular_setedrx.c file

```
97                ret                   = atc_sqnedrx_check(p_ctrl);
```

Figure 78. Line 97 in the r_cellular_setedrx.c file

5.16 R_CELLULAR_GetEDRX()

The following shows the flow of the R_CELLULAR_GetEDRX() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects the eDRX settings.
Command used:
• AT+SQNEDRX?

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.16.1 Reusable Processes

Processes (1), (2), (4), and (5) can be reused for any cellular modules.

5.16.2 Processes Needing Replacement

Process (3), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_getedrx.c file, replace the processing on line 84 by appropriate alternative processing.

```
84              ret                 = atc_sqnedrx_check(p_ctrl);
```

Figure 79. Line 84 in the r_cellular_getedrx.c file

5.17 R_CELLULAR_SetPSM()

The following shows the flow of the R_CELLULAR_SetPSM() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Performs PSM configuration (by executing the cellular_psm_config() function on line 97).
Commands used:
• AT+SQNRICFG=%s,3,%s
• AT+SQNIPSCFG=%s,%s
• AT+SQNPSCFG=%s

(4) Executes a command that enables or disables PSM and a command that acquires the settings.
Commands used:
• AT+CPSMS=%s,,,"%s","%s"
• AT+CPSMS?

(5) Releases the semaphore for AT commands.

(6) Performs a hardware reset (by executing the cellular_module_reset() function on line 114).

(7) Disables the thread-safe feature.

5.17.1 Reusable Processes

Processes (1), (2), (4), (5), and (7) can be reused for any cellular modules.

5.17.2 Processes Needing Replacement

Process (3), which uses AT commands dedicated to the RYZ014A cellular module, cannot be used for other modules. In the cellular_psm_config.c file, replace the processing on lines 64, 68, 74, 116, 120, 188, and 193 by appropriate alternative processing.

```
64          ret = atc_sqnricfg(p_ctrl, CELLULAR_SQNRICFG_MODE);
116         ret = atc_sqnricfg(p_ctrl, (uint8_t)CELLULAR_PSM_MODE_INVALID);
193         atc_sqnricfg(p_ctrl, (uint8_t)CELLULAR_PSM_MODE_INVALID);
```

Figure 80. Lines 64, 116, and 193 in the cellular_psm_config.c file

```
68              ret         = atc_sqnipscfg(p_ctrl, CELLULAR_SQNIPSCFG_MODE);
120             ret = atc_sqnipscfg(p_ctrl, (uint8_t)CELLULAR_PSM_MODE_INVALID);
188         atc_sqnipscfg(p_ctrl, (uint8_t)CELLULAR_PSM_MODE_INVALID);
```

Figure 81. Lines 68, 120, and 188 in the cellular_psm_config.c file

```
74              ret         = atc_sqnpscfg(p_ctrl);
```

Figure 82. Line 74 in the cellular_psm_config.c file

In the private function for performing a hardware reset in process (6), commands dedicated to the RYZ014A cellular module are executed. Therefore, this process cannot be reused for other cellular modules. In the cellular_module_reset.c file, delete line 115 or replace the processing on that line by other processing.

The atc_sqnautoconnect_chek() function executes the “AT+SQNAUTOCONNECT?” command to check whether the function level is set to 1 (CFUN=1) automatically when the cellular module is activated.

```
115             ret               = atc_sqnautoconnect_check(p_ctrl);
```

Figure 83. Line 115 in the cellular_module_reset.c file

5.18 R_CELLULAR_GetPSM()

The following shows the flow of the R_CELLULAR_GetPSM() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects the PSM settings.
Command used:
• AT+CPSMS?

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.18.1 Reusable Processes

Processes (1) to (5) can be reused for any cellular modules.

5.19 R_CELLULAR_GetICCID()

The following shows the flow of the R_CELLULAR_GetICCID() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects the ICCID.
Command used:
• AT+SQNCCID?

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.19.1 Reusable Processes

Processes (1), (2), (4), and (5) can be reused for any cellular modules.

5.19.2 Processes Needing Replacement

Process (3), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_geticcid.c file, replace the processing on line 84 by appropriate alternative processing.

```
84            ret                    = atc_sqnccid(p_ctrl);
```

Figure 84. Line 84 in the r_cellular_geticcid.c file

5.20 R_CELLULAR_GetIMEI()

The following shows the flow of the R_CELLULAR_GetIMEI() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects the IMEI.
Command used:
• AT+CGSN

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.20.1 Reusable Processes

Processes (1) to (5) can be reused for any cellular modules.

5.21 R_CELLULAR_GetIMSI()

The following shows the flow of the R_CELLULAR_GetIMSI() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects the IMSI.
Command used:
• AT+CIMI

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.21.1 Reusable Processes

Processes (1) to (5) can be reused for any cellular modules.

5.22 R_CELLULAR_GetPhonenum()

The following shows the flow of the R_CELLULAR_GetPhonenum() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects the phone number.
Command used:
• AT+CNUM

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.22.1 Reusable Processes

Processes (1) to (5) can be reused for any cellular modules.

5.23 R_CELLULAR_GetRSSI()

The following shows the flow of the R_CELLULAR_GetRSSI() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects signal quality.
Command used:
• AT+CSQ

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.23.1 Reusable Processes

Processes (1) to (5) can be reused for any cellular modules.

5.24 R_CELLULAR_GetSVN()

The following shows the flow of the R_CELLULAR_GetRSSI() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects the software version number (SVN) and a command that collects the software revision number.
Commands used:
• AT+CGSN=3
• ATI1

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.24.1 Reusable Processes

Processes (1) to (5) can be reused for any cellular modules.

5.25 R_CELLULAR_Ping()

The following shows the flow of the R_CELLULAR_Ping() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a ping command.
Command used:
• AT+PING="%s",%s,%s,%s,%s

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.25.1 Reusable Processes

Processes (1) to (5) can be reused for any cellular modules.

5.26 R_CELLULAR_GetAPConnectState()

The following shows the flow of the R_CELLULAR_GetAPConnectState() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that sets the notification level of the access point (AP) connection state and a command that collects the access point (AP) connection state.
Commands used:
• AT+CEREG=%s
• AT+CEREG?

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.26.1 Reusable Processes

Processes (1) to (5) can be reused for any cellular modules.

5.27 R_CELLULAR_GetCellInfo()

The following shows the flow of the R_CELLULAR_GetCellInfo() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects the function level and a command that sets the function level.
Commands used:
• AT+CFUN?
• AT+CFUN=1

(4) Executes a command that collects cell information.

(5) Command used:
• AT+SQNMONI=%s

(6) Releases the semaphore for AT commands.

(7) Disables the thread-safe feature.

5.27.1 Reusable Processes

Processes (1) to (3), (5), and (6) can be reused for any cellular modules.

5.27.2 Processes Needing Replacement

Process (4), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_getcellinfo.c file, replace the processing on line 100 by appropriate alternative processing.

```
100                             ret = atc_sqnmoni(p_ctrl, type);
```

Figure 85. Line 100 in the r_cellular_getcellinfo.c file

5.28 R_CELLULAR_AutoConnectConfig()

The following shows the flow of the R_CELLULAR_AutoConnectConfig() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that sets auto-connection to an access point (AP).
Command used:
• AT+SQNAUTOCONNECT=%s

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.28.1 Reusable Processes

Processes (1), (2), (5), and (6) can be reused for any cellular modules.

5.28.2 Processes Needing Replacement

Process (3), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_autoconnectconfig.c file, replace the processing on line 83 by appropriate alternative processing.

```
83              ret = atc_sqnautoconnect(p_ctrl, type);
```

Figure 86. Line 83 in the r_cellular_autoconnectconfig.c file

5.29 R_CELLULAR_SetOperator()

The following shows the flow of the R_CELLULAR_SetOperator() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects the operator settings and a command that configures operator settings (by executing the cellular_set_operator() function).
Commands used:
• AT+SQNCTM?
• AT+SQNCTM="%s"

(4) Executes a command that collects the function level and a command that sets the function level.
Commands used:
• AT+CFUN?
• AT+CFUN=4

(5) Releases the semaphore for AT commands.

(6) Disables the thread-safe feature.

5.29.1 Reusable Processes

Processes (1), (2), and (4) to (6) can be reused for any cellular modules.

5.29.2 Processes Needing Replacement

Process (3), which uses AT commands dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_setoperator.c file, replace the processing on lines 127 and 133 by appropriate alternative processing.

```
127     ret                 = atc_sqnctm_check(p_ctrl);
133         ret                 = atc_sqnctm(p_ctrl, p_operator);
```

Figure 87. Lines 127 and 133 in the r_cellular_setoperator.c file

5.30 R_CELLULAR_SetBand()

The following shows the flow of the R_CELLULAR_SetBand() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Collects the operator settings.
Command used:
• AT+SQNCTM?

(4) Configures band settings.
Command used:
• AT+SQNBANDSEL=0,"%s","%s"

(5) Performs a soft reset.
Command used:
• AT^RESET

(6) Closes the socket (by executing the cellular_closesocket() function).

(7) Detects the string “+SYSSTART”.

(8) Checks the current function level and sets the function level to 4, after the cellular module is reset.
Commands used:
• AT+CFUN?
• AT+CFUN=4

(9) Releases the semaphore for AT commands.

(10) Disables the thread-safe feature.

5.30.1 Reusable Processes

Processes (1), (2), and (5) to (10) can be reused for any cellular modules.

5.30.2 Processes Needing Replacement

Processes (3) and (4), which use AT commands dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_setband.c file, replace the processing on lines 93 and 96 by appropriate alternative processing.

```
93          ret                 = atc_sqnctm_check(p_ctrl);
96              ret = atc_sqnbandsel(p_ctrl, ctm_name, p_band);
```

Figure 88. Lines 93 and 96 in the r_cellular_setband.c file

5.31 R_CELLULAR_GetPDPAddress()

The following shows the flow of the R_CELLULAR_GetPDPAddress() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects the PDP address.
Command used:
• AT+CGPADDR=1

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.31.1 Reusable Processes

Processes (1) to (5) can be reused for any cellular modules.

5.32 R_CELLULAR_FirmUpgrade()

The following shows the flow of the R_CELLULAR_FirmUpgrade() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that upgrades the firmware.
Command used:
• AT+SQNSUPGRADE="%s",%s,5,%s

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.32.1 Reusable Processes

Processes (1), (2), (4), and (5) can be reused for any cellular modules.

5.32.2 Processes Needing Replacement

Process (3), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_firmupgrade.c file, replace the processing on line 90 by appropriate alternative processing.

```
90            ret = atc_sqnsupgrade(p_ctrl, p_url, 0, command, spid);
```

Figure 89. Line 90 in the r_cellular_firmupgrade.c file

5.33 R_CELLULAR_FirmUpgradeBlocking()

The following shows the flow of the R_CELLULAR_FirmUpgradeBlocking() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Shuts down the socket (by executing the cellular_shutdownsocket() function).
Command used:
• AT+SQNSH=%s

(3) Closes the socket (by executing the cellular_closesocket() function).

(4) Acquires the semaphore for AT commands.

(5) Executes a command that upgrades the firmware.
Command used:
• AT+SQNSUPGRADE="%s",%s,5,%s

(6) Executes a command that collects the function level and a command that sets the function level.
Commands used:
• AT+CFUN?
• AT+CFUN=4

(7) Releases the semaphore for AT commands.

(8) Disables the thread-safe feature.

5.33.1 Reusable Processes

Processes (1), (3), (4), and (6) to (8) can be reused for any cellular modules.

5.33.2 Processes Needing Replacement

Process (2), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the cellular_shutdownsocket.c file, replace the processing on line 76 by appropriate alternative processing.

```
76              ret = atc_sqnsh(p_ctrl, socket_no);
```

Figure 90. Line 76 in the cellular_shutdownsocket.c file

Process (5), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_firmupgradeblocking.c file, replace the processing on line 141 by appropriate alternative processing.

```
141     ret             = atc_sqnsupgrade(p_ctrl, p_url, 1, CELLULAR_FIRM_UPGRADE_BLOCKING, spid);
```

Figure 91. Line 141 in the r_cellular_firmupgradeblocking.c file

5.34 R_CELLULAR_GetUpgradeState()

The following shows the flow of the R_CELLULAR_GetUpgradeState() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that upgrades the firmware.
Command used:
• AT+SQNSUPGRADE?

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.34.1 Reusable Processes

Processes (1), (2), (4), and (5) can be reused for any cellular modules.

5.34.2 Processes Needing Replacement

Process (3), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_getupgradestate.c file, replace the processing on line 87 by appropriate alternative processing.

```
87            ret                    = atc_sqnsupgrade_check(p_ctrl);
```

Figure 92. Line 87 in the r_cellular_getupgradestate.c file

5.35 R_CELLULAR_UnlockSIM()

The following shows the flow of the R_CELLULAR_UnlockSIM() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that collects the function level and a command that sets the function level.
Commands used:
• AT+CFUN?
• AT+CFUN=4

(4) Executes a command that collects the PIN lock state and a command that releases the PIN lock.
Commands used:
• AT+CPIN?
• AT+CPIN="%s"

(5) Releases the semaphore for AT commands.

(6) Disables the thread-safe feature.

5.35.1 Reusable Processes

Processes (1) to (6) can be reused for any cellular modules.

5.36 R_CELLULAR_WriteCertificate()

The following shows the flow of the R_CELLULAR_WriteCertificate() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that writes a certificate (by executing the cellular_write_certificate() function).
Command used:
• AT+SQNSNVW=\"%s\",%s,%s

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.36.1 Reusable Processes

Processes (1), (2), (4), and (5) can be reused for any cellular modules.

5.36.2 Processes Needing Replacement

Process (3), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_writecertificate.c file, replace the processing on line 181 and the processing of the cellular_write_certificate() function by appropriate alternative processing. Section 4.6 provides information that you can reference when implementing the alternative processing.

```
181        ret = atc_sqnsnvw(p_ctrl, data_type, index, size);
```

Figure 93. Line 181 in the r_cellular_writecertificate.c file

5.37 R_CELLULAR_EraseCertificate()

The following shows the flow of the R_CELLULAR_EraseCertificate() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that writes a certificate to erase the certificate (the command used deletes the certificate by writing 0).
Command used:
• AT+SQNSNVW=\"%s\",%s,0

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.37.1 Reusable Processes

Processes (1), (2), (4), and (5) can be reused for any cellular modules.

5.37.2 Processes Needing Replacement

Process (3), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_erasertificate.c file, replace the processing on line 85 by appropriate alternative processing.

```
85                ret = atc_sqnsnvw_erase(p_ctrl, data_type, index);
```

Figure 94. Line 85 in the r_cellular_erasertificate.c file

5.38 R_CELLULAR_GetCertificate()

The following shows the flow of the R_CELLULAR_GetCertificate() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that reads a certificate.
Command used:
• AT+SQNSNVR="%s",%s

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.38.1 Reusable Processes

Processes (1), (2), (4), and (5) can be reused for any cellular modules.

5.38.2 Processes Needing Replacement

Process (3), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_getcertificate.c file, replace the processing on line 87 by appropriate alternative processing.

```
87              ret                 = atc_sqnsnvr(p_ctrl, data_type, index);
```

Figure 95. Line 87 in the r_cellular_getcertificate.c file

5.39 R_CELLULAR_ConfigSSLProfile()

The following shows the flow of the R_CELLULAR_ConfigSSLProfile() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Executes a command that configures a security profile.
Command used:
• AT+SQNSPCFG=%s,2,,%s,%s,%s,%s,\"\"\r

(4) Releases the semaphore for AT commands.

(5) Disables the thread-safe feature.

5.39.1 Reusable Processes

Processes (1), (2), (4), and (5) can be reused for any cellular modules.

5.39.2 Processes Needing Replacement

Process (3), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_configsslprofile.c file, replace the processing on lines 92 and 93 by appropriate alternative processing.

```
92              ret = atc_sqnspcfg(p_ctrl, security_profile_id, cert_valid_level,
93                      ca_certificate_id, client_certificate_id, client_privatekey_id);
```

Figure 96. Line 92, 93 in the r_cellular_configsslprofile.c file

5.40 R_CELLULAR_SoftwareReset()

The following shows the flow of the R_CELLULAR_SoftwareReset() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Shuts down the socket (by executing the cellular_shutdownsocket() function).
Command used:
• AT+SQNSH=%s

(3) Closes the socket (by executing the cellular_closesocket() function).

(4) Acquires the semaphore for AT commands.

(5) Executes a command that obtains current auto-connection mode.
Command used:
• AT+SQNAUTOCONNECT?

(6) Executes a command that sets the notification level of the access point connection state.
Command used:
• AT+CEREG=2

(7) Executes a reset command.
Command used:
• AT^RESET

(8) Executes a command that sets the function level.
Command used:
• AT+CFUN=4

(9) Releases the semaphore for AT commands.

(10) Disables the thread-safe feature.

5.40.1 Reusable Processes

Processes (1), (3), (4), and (6) to (10) can be reused for any cellular modules.

5.40.2 Processes Needing Replacement

Process (2), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the cellular_shutdownsocket.c file, replace the processing on line 76 by appropriate alternative processing.

```
76            ret = atc_sqnsh(p_ctrl, socket_no);
```

Figure 97. Line 76 in the cellular_shutdownsocket.c file

Process (5), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other cellular modules. In the r_cellular_softwarereset.c file, delete line 133 or replace the processing on that line by appropriate alternative processing.

The atc_sqnautoconnect_chek() function executes the “AT+SQNAUTOCONNECT?” command to check whether the function level is set to 1 (CFUN=1) automatically when the cellular module is activated.

```
133    ret                 = atc_sqnautoconnect_check(p_ctrl);
```

Figure 98. Line 133 in the r_cellular_softwarereset.c file

5.41 R_CELLULAR_HardwareReset()

The following shows the flow of the R_CELLULAR_HardwareReset() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Performs a hardware reset (by executing the cellular_module_reset() function on line 79).

(3) Disables the thread-safe feature.

5.41.1 Reusable Processes

Processes (1) and (3) can be reused for any cellular modules.

5.41.2 Processes Needing Replacement

In the private function for performing a hardware reset in process (2), commands dedicated to the RYZ014A cellular module are executed. Therefore, this process cannot be reused for other cellular modules. In the cellular_module_reset.c file, delete line 115 or replace the processing on that line by other processing.

The atc_sqnautoconnect_chek() function executes the “AT+SQNAUTOCONNECT?” command to check whether the function level is set to 1 (CFUN=1) automatically when the cellular module is activated.

```
115            ret                 = atc_sqnautoconnect_check(p_ctrl);
```

Figure 99. Line 115 in the cellular_module_reset.c file

5.42 R_CELLULAR_FactoryReset()

The following shows the flow of the R_CELLULAR_FactoryReset() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Acquires the semaphore for AT commands.

(3) Acquires the usage state of PDP contexts.
Command used:
• AT+CGDCONT?

(4) Find an unused context in the information acquired in process (3), and then register dummy data by specifying the number of the found context.
Command used:
• AT+CGDCONT=%s,"IPV4V6","%s"

(5) Perform a factory reset.
Command used:
• AT+SQNSFACTORYRESET

(6) Performs a hardware reset (by executing the cellular_module_reset() function on line 141).

(7) Acquires the usage state of PDP contexts.
Command used:
• AT+CGDCONT?

(8) Confirm that the dummy data registered in process (4) does not remain in the PDP context information acquired in process (7).

Note: If no recovery points have been created, the “AT+SQNSFACTORYRESET” command returns the string “ERROR” as a response even when a factory reset ends normally. For this reason, this function confirms that a factory reset was successful by confirming that the involatile information that was registered immediately before a factory reset was performed has been erased.

(9) Checks the PSM setting status (enabled or disabled).
This process is necessary in the case where the PSM that was enabled before a reset is disabled after a reset.
Command used:
• AT+CPSMS?

(10) Performs settings related to PSM such that disables for the RYZ014A to indicate URCs and disables IRQ if the status retrieved in process (9) indicates that the PSM is disabled (by executing the cellular_psm_config() function on line 223).
Commands used:
• AT+SQNRICFG=%s,3,%s
• AT+SQNIPSCFG=%s,%s

(11) Releases the semaphore for AT commands.

(12) Disables the thread-safe feature.

5.42.1 Reusable Processes

Processes (1) to (5), (7), (9), (11), and (12) can be reused for any cellular modules.

5.42.2 Processes Needing Replacement

In the private function for performing a hardware reset in process (6), commands dedicated to the RYZ014A cellular module are executed. Therefore, this process cannot be reused for other cellular modules. In the cellular_module_reset.c file, delete line 115 or replace the processing on that line by other processing.

The atc_sqnautoconnect_chek() function executes the "AT+SQNAUTOCONNECT?" command to check whether the function level is set to 1 (CFUN=1) automatically when the cellular module is activated.

```
115        ret                 = atc_sqnautoconnect_check(p_ctrl);
```

Figure 100. Line 115 in the cellular_module_reset.c file

Process (5), which uses an AT command dedicated to the RYZ014A cellular module, cannot be used for other modules. In the r_cellular_factoryreset.c file, replace the processing on line 139 by appropriate alternative processing.

```
139            atc_sqnsfactoryreset(p_ctrl);
```

Figure 101. Line 139 in the r_cellular_factoryreset.c file

Process (10), which uses AT commands dedicated to the RYZ014A cellular module, cannot be used for other modules. In the cellular_psm_config.c file, replace the processing on lines 64, 68, 74, 116, and 120 by appropriate alternative processing.

```
64      ret = atc_sqnricfg(p_ctrl, CELLULAR_SQNRICFG_MODE);
116     ret = atc_sqnricfg(p_ctrl, (uint8_t)CELLULAR_PSM_MODE_INVALID);
```

Figure 102. Lines 64 and 116 in the cellular_psm_config.c file

```
68          ret         = atc_sqnipscfg(p_ctrl, CELLULAR_SQNIPSCFG_MODE);
120         ret = atc_sqnipscfg(p_ctrl, (uint8_t)CELLULAR_PSM_MODE_INVALID);
```

Figure 103. Lines 68 and 120 in the cellular_psm_config.c file

```
74          ret         = atc_sqnpscfg(p_ctrl);
```

Figure 104. Line 74 in the cellular_psm_config.c file

5.43 R_CELLULAR_RTS_Ctrl()

The following shows the flow of the R_CELLULAR_RTS_Ctrl() function processes:

(1) Checks the arguments and enables the thread-safe feature.

(2) Controls the RTS pin output.

(3) Disables the thread-safe feature.

5.43.1 Reusable Processes

Processes (1) to (3) can be reused for any cellular modules.

6. API Functions for Software Modules

Appendix provides detailed information for each API function about the handling needed when RYZ014A Cellular FIT Module is used for controlling a communication module other than the RYZ014A cellular module or when an RX family MCU that does not support RYZ014A Cellular FIT Module is used. For information about the following software, see Chapter 7.

1. Built-in functions of RYZ014A Cellular FIT Module
2. r_bsp
3. r_sci_rx
4. r_irq_rx
5. FreeRTOS

7. Appendix

7.1 Environment in Which Operation Was Verified

Table 7.1 shows the details of the environment in which operation of RYZ014A Cellular FIT Module was verified.

Table 7.1 Environment in which operation was verified

Item		Description
Integrated development environment		Renesas Electronics e^2 studio Version 2023-07
Compiler	CC-RX	Renesas Electronics C/C++ Compiler for RX Family V3.05.00 Compiler option: The following option is added to the default settings of the integrated development environment: -lang = c99
	GCC	-
Endianness		Little endian
Revision number of RYZ014A Cellular FIT Module		Rev1.11
Board used (RX)		Renesas CK-RX65N (Model: RTK5CK65N0SxxxxxBE)
		Renesas RX72N Envision Kit (Model: RTK5RX72N0C00000BJ)
Board used (RYZ014A)		PMOD Expansion Board for RYZ014A (Model: RTKYZ014A0B00000BE)
RTOS	FreeRTOS	10.4.3-rx-1.0.1
FIT	BSP FIT	Ver 7.30
	SCI FIT	Ver 4.80
	IRQ FIT	Ver 4.30

7.2 Troubleshooting

(1) Q: I added RYZ014A Cellular FIT Module to a project, but when I executed the build, I got the following error: Could not open source file "platform.h".

A: FIT modules may not have been added to the project properly. Using the following documents, check the correct procedure for adding FIT modules by referring to the relevant document according to the compiler you are using. Check the correct procedure for adding them to a project by referring to the relevant document according to the compiler you use:

- If you are using CS+:
  Application note “Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)”
- If you are using e^2 studio:
  Application note “Adding Firmware Integration Technology Modules to Projects (R01AN1723)”

If you use RYZ014A Cellular FIT Module, you must also add the board support package FIT module (BSP module) to the project. For details on how to add the BSP module, refer to the application note “Board Support Package Module Using Firmware Integration Technology (R01AN1685)”.

(2) Q: I added RYZ014A Cellular FIT Module to a project, but when I executed the build, I got an error due to incorrectly configured settings.

A: The settings in the file “r_aws_cellular_config.h” may be incorrect. Check the file “r_aws_cellular_config.h”. If there are incorrect settings, correct them. For details, refer to section 2.7 in [6].

7.3 Recovery Operation

If you encounter any events described in this section when using RYZ014A Cellular FIT Module, perform the recovery operation.

7.3.1 If the Cellular Module Reports the URC "^EXIT"

The cellular module may perform a self-reset, reporting the unsolicited result code (URC) "^EXIT". It is recommended that a callback function is used to detect the URC "^EXIT". If the URC "^EXIT" is detected, perform the following procedure.

(1) Detect the URC "+SYSSTART".
    Use a callback function to detect the URC "+SYSSTART" that follows the URC "^EXIT".
(2) Execute the R_CELLULAR_Close() function.
(3) Execute the R_CELLULAR_Open() function.

After performing the above procedure, the cellular module will become again able to connect to access points ("Completed Cellular module and FIT module initialization status" in Figure 1.3 in the related document [6]).

7.3.2 If the Cellular Module Reports the URC "+SYSSTART"

The cellular module reports the unsolicited result code (URC) "+SYSSTART" when it becomes ready to operate after activation. The cellular module reports the URC "+SYSSTART" when the module is restarted during execution of the following API functions:

- R_CELLULAR_Open()
- R_CELLULAR_SetPSM()
- R_CELLULAR_SetOperator()
- R_CELLULAR_SetBand()
- R_CELLULAR_FirmUpgradeBlocking()
- R_CELLULAR_SoftwareReset()
- R_CELLULAR_HardwareReset()
- R_CELLULAR_FactoryReset()

If the cellular module is restarted unexpectedly, it reports the URC "+SYSSTART" even during execution an API function other than the above ones. It is recommended that a callback function is used to detect the URC "+SYSSTART". If unexpected restart of the cellular module occurs, perform the following procedure:

(1) Execute the R_CELLULAR_Close() function.
(2) Execute the R_CELLULAR_Open() function.

After performing the above procedure, the cellular module will become again able to connect to access points ("Completed Cellular module and FIT module initialization status" in Figure 1.3 in the related document [6]).

7.3.3 If an API Function Times Out

If an API function times out, it notifies you by returning CELLULAR_ERR_MODULE_TIMEOUT as the return value. In this case, perform the following procedure.

(1) Execute the R_CELLULAR_HardwareReset() function.
The RYZ014A cellular module is reset with the RESETN pin.
(2) Execute the R_CELLULAR_Close() function.
(3) Execute the R_CELLULAR_Open() function.

After performing the above procedure, the cellular module will become again able to connect to access points (“Completed Cellular module and FIT module initialization status” in Figure 1.3 in the related document [6]).

7.4 Built-in Functions of RYZ014A Cellular FIT Module Needing Modification

7.4.1 cellular_apconnect_config()

The cellular_apconnect_config() function is a static function in the r_cellular_apconnect.c file. This function configures the settings for access point connection. This function can be reused for any cellular modules.

7.4.2 cellular_apconnect()

The cellular_apconnect() function is a static function in the r_cellular_apconnect.c file. This function performs the processing to connect to an access point. This function can be reused for any cellular modules.

7.4.3 cellular_sync_check()

The cellular_sync_check() function is a static function in the r_cellular_apconnect.c file. This function acquires information (PDP address and network time) after connection an access point is completed. This function can be reused for any cellular modules.

7.4.4 cellular_socket_cfg()

The cellular_socket_cfg() function is a static function in the r_cellular_createsocket.c file. This function configures the socket settings. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. In the cellular_socket_cfg() function, replace the processing on lines 154 and 157 by appropriate alternative processing.

```
154        atc_ret = atc_sqnscfg(p_ctrl, (uint8_t)(socket_num + start_num));
```

Figure 105. Line 154 in the cellular_socket_cfg() function

```
157            atc_ret = atc_sqnscfgext(p_ctrl, (uint8_t)(socket_num + start_num));
```

Figure 106. Line 157 in the cellular_socket_cfg() function

7.4.5 cellular_getip()

The cellular_getip() function is a static function in the r_cellular_dnsquery.c file. This function stores the acquired IP address in the 3rd argument. The processing of this function must be modified according to the IP address notification method of the cellular module.

7.4.6 cellular_factoryreset()

The cellular_factoryreset() function is a static function in the r_cellular_factoryreset.c file. This function performs a factory reset. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. In the cellular_factoryreset() function, replace the processing on line 139 by appropriate alternative processing.

```
139                    atc_sqnsfactoryreset(p_ctrl);
```

Figure 107. Line 139 in the cellular_factoryreset() function

7.4.7 cellular_psm_check()

The cellular_psm_check() function is a static function in the r_cellular_dnsquery.c file. This function acquires the PSM setting state or disables the PSM settings. This function can be reused for any cellular modules.

Note: This function internally uses the cellular_psm_config() function, which needs modification.

7.4.8 private_cgdcont()

The private_cgdcont() function is a static function in the r_cellular_dnsquery.c file. This function adds dummy data to a PDP context. This function can be reused for any cellular modules.

7.4.9 cellular_firmupgradeblocking()

The cellular_firmupgradeblocking() function is a static function in the r_cellular_firmupgradeblocking.c file. This function upgrades the firmware by FOTA (Firmware upgrade Over The Air) in blocking mode. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. In the cellular_firmupgradeblocking() function, replace the processing on line 141 by appropriate alternative processing.

```
141     ret                 = atc_sqnsupgrade(p_ctrl, p_url, 1, CELLULAR_FIRM_UPGRADE_BLOCKING, spid);
```

Figure 108. Line 141 in the cellular_firmupgradeblocking() function

7.4.10 cellular_init()

The cellular_init() function is a static function in the r_cellular_open.c file. This function configures the settings for communication with the cellular module. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. In the cellular_init() function, replace the processing on line 289 by appropriate alternative processing.

```
289         ret = atc_sqnsimst(p_ctrl);
```

Figure 109. Line 289 in the cellular_init() function

7.4.11 cellular_config_init()

The cellular_config_init() function is a static function in the r_cellular_open.c file. This function configures various settings. This function can be reused for any cellular modules.

7.4.12 cellular_open_fail()

The cellular_open_fail() function is a static function in the r_cellular_open.c file. This function performs the handling for when the R_CELLULAR_Open() function fails. This function can be reused for any cellular modules.

7.4.13 cellular_receive_data()

The cellular_receive_data() function is a static function in the r_cellular_receivesocket.c file. This function receives data via socket communication. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Refer to section 5.11.3 when modifying the processing.

7.4.14 cellular_receive_flag_check()

The cellular_receive_flag_check() function is a static function in the r_cellular_receivesocket.c file. This function checks whether there is data that can be received. This function performs processing related to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Refer to section 5.11.3 when modifying the processing.

7.4.15 cellular_recv_size_check()

The cellular_recv_size_check() function is a static function in the r_cellular_receivesocket.c file. This function determines the size of receive data to be requested from the cellular module. This function performs processing related to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Refer to section 5.11.3 when modifying the processing. Alternatively, you can also delete lines 332 to 336.

7.4.16 cellular_send_data()

The cellular_send_data() function is a static function in the r_cellular_sendsocket.c file. This function sends data via socket communication. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Refer to section 4.6 when modifying the processing.

7.4.17 cellular_send_size_check()

The cellular_send_size_check() function is a static function in the r_cellular_sendsocket.c file. This function determines the size of data that will actually be sent to the cellular module. This function can be reused for any cellular modules.

7.4.18 cellular_tx_flag_check()

The cellular_tx_flag_check() function is a static function in the r_cellular_sendsocket.c file. This function uses the R_SCI_Send() function to confirm that data transmission to the cellular module was completed. This function can be reused for any cellular modules.

7.4.19 cellular_atc_response_check()

The cellular_atc_response_check() function is a static function in the r_cellular_sendsocket.c file. This function sends an AT command and receives an intermediate result code from the cellular module. This function then sends additional data to the cellular module and confirms that the response string returned from the cellular module is as expected. This function can be reused for any cellular modules.

7.4.20 cellular_set_operator()

The cellular_set_operator() function is a static function in the r_cellular_setoperator.c file. This function configures the operator settings. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. In the cellular_set_operator() function, replace the processing on lines 127 and 133 by appropriate alternative processing.

```
127     ret                 = atc_sqnctm_check(p_ctrl);
133         ret                 = atc_sqnctm(p_ctrl, p_operator);
```

Figure 110. Lines 127 and 133 in the r_cellular_setoperator.c file

7.4.21 cellular_softwarereset()

The cellular_softwarereset() function is a static function in the r_cellular_softwarereset.c file. This function performs a soft reset. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. In the cellular_softwarereset() function, delete line 133 or replace the processing on that line by appropriate alternative processing.

Note: The atc_sqnautoconnect_chek() function executes the "AT+SQNAUTOCONNECT?" command to check whether the function level is set to 1 (CFUN=1) automatically when the cellular module is activated.

```
133     ret                    = atc_sqnautoconnect_check(p_ctrl);
```

Figure 111. Line 133 in the cellular_softwarereset() function

7.4.22 cellular_unlocksim()

The cellular_unlocksim() function is a static function in the r_cellular_unlocksim.c file. This function issues a command that releases the SIM PIN lock. This function can be reused for any cellular modules.

7.4.23 cellular_write_certificate()

The cellular_write_certificate() function is a static function in the r_cellular_writecertficate.c file. This function issues a command that writes certificate or private key information to involatile memory on the cellular module. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Refer to section 4.6 when modifying the processing.

7.4.24 cellular_send_size_check()

The cellular_send_size_check() function is a static function in the r_cellular_unlocksim.c file. This function uses the R_SCI_Send() function to check the size of data to be sent to the cellular module. This function can be reused for any cellular modules.

7.4.25 atc_generate()

The atc_generate() function is defined in the at_command.c file. This function generates the AT command to be sent to the cellular module. This function can be reused for any cellular modules.

7.4.26 atc_ate0()

The atc_ate0() function is defined in the ate0.c file. This function sends the following command to the cellular module: "ATE0". This function can be reused for any cellular modules.

7.4.27 atc_ati1()

The atc_ati1() function is defined in the ati0.c file. This function sends the following command to the cellular module: "ATI1". This function can be reused for any cellular modules.

7.4.28 atc_cclk()

The atc_cclk() function is defined in the cclk.c file. This function sends the following command to the cellular module: "AT+CCLK="%s/%s/%s,%s:%s:%s%s"". This function can be reused for any cellular modules.

7.4.29 atc_cclk_check()

The atc_cclk_check() function is defined in the cclk.c file. This function sends the following command to the cellular module: "AT+CCLK?". This function can be reused for any cellular modules.

7.4.30 atc_ceer()

The atc_ceer() function is defined in the ceer.c file. This function sends the following command to the cellular module: “AT+CEER”. This function can be reused for any cellular modules.

7.4.31 atc_cereg()

The atc_cereg() function is defined in the cereg.c file. This function sends the following command to the cellular module: “AT+CEREG=%s”. This function can be reused for any cellular modules.

7.4.32 atc_cereg_check()

The atc_cereg_check() function is defined in the cereg.c file. This function sends the following command to the cellular module: “AT+CEREG?”. This function can be reused for any cellular modules.

7.4.33 atc_cfun()

The atc_cfun() function is defined in the cfun.c file. This function sends the following command to the cellular module: “AT+CFUN=%s”. This function can be reused for any cellular modules.

7.4.34 atc_cfun_check()

The atc_cfun_check() function is defined in the cfun.c file. This function sends the following command to the cellular module: “AT+CFUN?”. This function can be reused for any cellular modules.

7.4.35 atc_cgact()

The atc_cgact() function is defined in the cgact.c file. This function sends the following command to the cellular module: “AT+CGACT=1,%s”. This function can be reused for any cellular modules.

7.4.36 atc_cgact_check()

The atc_cgact_check() function is defined in the cgact.c file. This function sends the following command to the cellular module: “AT+CGACT?”. This function can be reused for any cellular modules.

7.4.37 atc_cgatt()

The atc_cgatt() function is defined in the cgatt.c file. This function sends the following command to the cellular module: “AT+CGATT=%s”. This function can be reused for any cellular modules.

7.4.38 atc_cgatt_check()

The atc_cgatt_check() function is defined in the cgatt.c file. This function sends the following command to the cellular module: “AT+CGATT?”. This function can be reused for any cellular modules.

7.4.39 atc_cgauth()

The atc_cgauth() function is defined in the cgauth.c file. This function sends the following command to the cellular module: “AT+CGAUTH=1,%s,"%s","%s"”. This function can be reused for any cellular modules.

7.4.40 atc_cgauth_reset()

The atc_cgauth_reset() function is defined in the cgauth.c file. This function sends the following command to the cellular module: “AT+CGAUTH=1,0”. This function can be reused for any cellular modules.

7.4.41 atc_cgdcont()

The atc_cgdcont() function is defined in the cgdcont.c file. This function sends the following command to the cellular module: “AT+CGDCONT=%s,"IPV4V6","%s"”. This function can be reused for any cellular modules.

7.4.42 atc_cgdcont_check()

The atc_cgdcont_check() function is defined in the cgdcont.c file. This function sends the following command to the cellular module: “AT+CGDCONT?”. This function can be reused for any cellular modules.

7.4.43 atc_cgmi()

The atc_cgmi() function is defined in the cgmi.c file. This function sends the following command to the cellular module: “AT+CGMI”. This function can be reused for any cellular modules.

7.4.44 atc_cgmm()

The atc_cgmm() function is defined in the cgmm.c file. This function sends the following command to the cellular module: “AT+CGMM”. This function can be reused for any cellular modules.

7.4.45 atc_cgmr()

The atc_cgmr() function is defined in the cgmr.c file. This function sends the following command to the cellular module: “AT+CGMR”. This function can be reused for any cellular modules.

7.4.46 atc_cgpaddr()

The atc_cgpaddr() function is defined in the cgpaddr.c file. This function sends the following command to the cellular module: “AT+CGPADDR=1”. This function can be reused for any cellular modules.

7.4.47 atc_cgpiaf()

The atc_cgpiaf() function is defined in the cgpiaf.c file. This function sends the following command to the cellular module: “AT+CGPIAF=1,0,1,0”. This function can be reused for any cellular modules.

7.4.48 atc_cgsn()

The atc_cgsn() function is defined in the cgsn.c file. This function sends the following command to the cellular module: “AT+CGSN”. This function can be reused for any cellular modules.

7.4.49 atc_cgsn3()

atc_cgsn3() function is defined in the cgsn.c file. This function sends the following command to the cellular module: “AT+CGSN=3”. This function can be reused for any cellular modules.

7.4.50 atc_cimi()

The atc_cimi() function is defined in the cimi.c file. This function sends the following command to the cellular module: “AT+CIMI”. This function can be reused for any cellular modules.

7.4.51 atc_cmer()

The atc_cmer() function is defined in the cmer.c file. This function sends the following command to the cellular module: “AT+CMER=3,0,0,%s,0,0,0”. This function can be reused for any cellular modules.

7.4.52 atc_cnum()

The atc_cnum() function is defined in the cnum.c file. This function sends the following command to the cellular module: “AT+CNUM”. This function can be reused for any cellular modules.

7.4.53 atc_cops()

The atc_cops() function is defined in the cops.c file. This function sends the following command to the cellular module: “AT+COPS=%s,2,"%s%s"”. This function can be reused for any cellular modules.

7.4.54 atc_cops_check()

The atc_cops_check() function is defined in the cops.c file. This function sends the following command to the cellular module: “AT+COPS?”. This function can be reused for any cellular modules.

7.4.55 atc_cpin()

The atc_cpin() function is defined in the cpin.c file. This function sends the following command to the cellular module: “AT+CPIN="%s"”. This function can be reused for any cellular modules.

7.4.56 atc_cpin_check()

The atc_cpin_check() function is defined in the cpin.c file. This function sends the following command to the cellular module: “AT+CPIN?”. This function can be reused for any cellular modules.

7.4.57 atc_cpsms()

The atc_cpsms() function is defined in the cpsms.c file. This function sends the following command to the cellular module: “AT+CPSMS=%s,,,"%s","%s"”. This function can be reused for any cellular modules.

7.4.58 atc_cpsms_check()

The atc_cpsms_check() function is defined in the cops.c file. This function sends the following command to the cellular module: “AT+CPSMS?”. This function can be reused for any cellular modules.

7.4.59 atc_crsm()

The atc_crsm() function is defined in the crsm.c file. This function sends the following command to the cellular module: “AT+CRSM=%s,%s,%s,%s,%s,"%s","%s"”. This function can be reused for any cellular modules.

7.4.60 atc_csq()

The atc_csq() function is defined in the csq.c file. This function sends the following command to the cellular module: “AT+CSQ”. This function can be reused for any cellular modules.

7.4.61 atc_ping()

The atc_ping() function is defined in the ping.c file. This function sends the following command to the cellular module: “AT+PING="%s",%s,%s,%s,%s”. This function can be reused for any cellular modules.

7.4.62 atc_reset()

The atc_reset() function is defined in the reset.c file. This function sends the following command to the cellular module: “AT^RESET”. This function can be reused for any cellular modules.

7.4.63 atc_smcwrx()

The atc_smcwrx() function is defined in the smcwrx.c file. This function sends the following command to the cellular module: “AT+SMCWRX=%s”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.64 atc_smcwtx()

The atc_smcwtx() function is defined in the smcwtx.c file. This function sends the following command to the cellular module: “AT+SMCWTX=%s,%s,%s”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.65 atc_sqnautoconnect()

The atc_sqnautoconnect() function is defined in the sqnautoconnect.c file. This function sends the following command to the cellular module: “AT+SQNAUTOCONNECT=%s”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.66 atc_sqnautoconnect_check()

The atc_sqnautoconnect_check() function is defined in the sqnautoconnect.c file. This function sends the following command to the cellular module: “AT+SQNAUTOCONNECT?”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.67 atc_sqnbandsel()

The atc_sqnbandsel() function is defined in the sqnbandsel.c file. This function sends the following command to the cellular module: “AT+SQNBANDSEL=0,"%s","%s"”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.68 atc_sqnccid()

The atc_sqnccid() function is defined in the sqnccid.c file. This function sends the following command to the cellular module: “AT+SQNCCID?”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.69 atc_sqnctm()

The atc_sqnctm() function is defined in the sqnctm.c file. This function sends the following command to the cellular module: “AT+SQNCTM="%s"”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.70 atc_sqnctm_check()

The atc_sqnctm_check() function is defined in the sqnctm.c file. This function sends the following command to the cellular module: “AT+SQNCTM?”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.71 atc_sqndnslkup()

The atc_sqndnslkup() function is defined in the sqndnslkup.c file. This function sends the following command to the cellular module: “AT+SQNDNSLKUP="%s",%s”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.72 atc_sqnedrx()

The atc_sqnedrx() function is defined in the sqnedrx.c file. This function sends the following command to the cellular module: “AT+SQNEDRX=%s,4,"%s","%s"”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.73 atc_sqnedrx_check()

The atc_sqnedrx_check() function is defined in the sqnedrx.c file. This function sends the following command to the cellular module: “AT+SQNEDRX?”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.74 atc_sqnipscfg()

The atc_sqnipscfg() function is defined in the sqnipscfg.c file. This function sends the following command to the cellular module: “AT+SQNIPSCFG=%s,%s”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.75 atc_sqnmoni()

The atc_sqnmoni() function is defined in the sqnmoni.c file. This function sends the following command to the cellular module: “AT+SQNMONI=%s”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.76 atc_sqnpscfg()

The atc_sqnpscfg() function is defined in the sqnpscfg.c file. This function sends the following command to the cellular module: “AT+SQNPSCFG=%s”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.77 atc_sqnricfg()

The atc_sqnricfg() function is defined in the sqnricfg.c file. This function sends the following command to the cellular module: “AT+SQNRICFG=%s,3,%s”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.78 atc_sqnscfg()

The atc_sqnscfg() function is defined in the sqnscfg.c file. This function sends the following command to the cellular module: “AT+SQNSCFG=%s,1,%s,%s,%s,%s”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.79 atc_sqnscfgext()

The atc_sqnscfgext() function is defined in the sqnscfgext.c file. This function sends the following command to the cellular module: “AT+SQNSCFGEXT=%s,1,0,0”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.80 atc_sqnsd()

The atc_sqnsd() function is defined in the sqnsd.c file. This function sends the following command to the cellular module: "AT+SQNSD=%s,%s,%s,"%s",0,%s,1". This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.81 atc_sqnsfactoryreset()

The atc_sqnsfactoryreset() function is defined in the sqnsfactoryreset.c file. This function sends the following command to the cellular module: "AT+SQNSFACTORYRESET". This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.82 atc_sqnsh()

The atc_sqnsh() function is defined in the sqnsh.c file. This function sends the following command to the cellular module: "AT+SQNSH=%s". This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.83 atc_sqnsimst()

The atc_sqnsimst() function is defined in the sqnsimst.c file. This function sends the following command to the cellular module: "AT+SQNSIMST=0". This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.84 atc_sqnsl()

The atc_sqnsl() function is defined in the sqnsl.c file. This function sends the following command to the cellular module: "AT+SQNSL=%s,%s,%s,0". This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.85 atc_sqnsnvr()

The atc_sqnsnvr() function is defined in the sqnsnvr.c file. This function sends the following command to the cellular module: "AT+SQNSNVR="%s",%s". This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.86 atc_sqnsnvw()

The atc_sqnsnvw() function is defined in the sqnsnvw.c file. This function sends the following command to the cellular module: "AT+SQNSNVW="%s",%s,%s". This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.87 atc_sqnsnvw_erase()

The atc_sqnsnvw_erase() function is defined in the sqnsnvw.c file. This function sends the following command to the cellular module: "AT+SQNSNVW=\"%s\",%s,0". This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.88 atc_sqnspcfg()

The atc_sqnspcfg() function is defined in the sqnspcfg.c file. This function sends the following command to the cellular module: "AT+SQNSPCFG=%s,2,,%s,%s,%s,%s,""". This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.89 atc_sqnsrecv()

The atc_sqnsrecv() function is defined in the sqnsrecv.c file. This function sends the following command to the cellular module: "AT+SQNSRECV=%s,%s". This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.90 atc_sqnssendext()

The atc_sqnssendext() function is defined in the sqnssendext.c file. This function sends the following command to the cellular module: “AT+SQNSSENDEXT=%s,%s”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.91 atc_sqnsshdn()

The atc_sqnsshdn() function is defined in the sqnsshdn.c file. This function sends the following command to the cellular module: “AT+SQNSSHDN”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.92 atc_sqnsupgrade()

The atc_sqnsupgrade() function is defined in the sqnsupgrade.c file. This function sends the following command to the cellular module: “AT+SQNSUPGRADE="%s",%s,5,%s,%s”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.93 atc_sqnsupgrade_check()

The atc_sqnsupgrade_check() function is defined in the sqnsupgrade.c file. This function sends the following command to the cellular module: “AT+SQNSUPGRADE?”. This command is dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.94 cellular_set_atc_number()

The cellular_set_atc_number() function is defined in the cellular_at_cmd_res_ctrl.c file. This function performs preparation for receiving a response returned when an AT command is sent to the cellular module. This function can be reused for any cellular modules.

7.4.95 cellular_get_atc_response()

The cellular_get_atc_response() function is defined in the cellular_at_cmd_res_ctrl.c file. This function checks the content of the response returned from the cellular module. This function can be reused for any cellular modules.

7.4.96 cellular_closesocket()

The cellular_closesocket() function is defined in the cellular_closesocket.c file. This function initializes the socket management structure. This function can be reused for any cellular modules.

7.4.97 cellular_disconnect()

The cellular_disconnect() function is defined in the cellular_disconnect.c file. This function performs disconnection from the access point. This function can be reused for any cellular modules.

7.4.98 cellular_execute_at_command()

The cellular_execute_at_command() function is defined in the cellular_execute_at_command.c file. This function actually sends an AT command to the cellular module. This function can be reused for any cellular modules.

Note: The cellular_send_atc() and cellular_res_check() functions in this file can also be reused.

7.4.99 cellular_getpdpaddr()

The cellular_getpdpaddr() function is defined in the cellular_getpdpaddr.c file. This function stores the acquired PDP address in the 2nd argument. This function can be reused for any cellular modules.

7.4.100 cellular_irq_open()

The cellular_irq_open() function is defined in the cellular_irq_ctrl.c file. This function executes the R_IRQ_Open() function. This function can be reused for any cellular modules.

7.4.101 cellular_irq_close()

The cellular_irq_close() function is defined in the cellular_irq_ctrl.c file. This function executes the R_IRQ_Close() function. This function can be reused for any cellular modules.

7.4.102 cellular_ring_callbResponse()

The cellular_ring_callbResponse() function is a callback function defined in the cellular_irq_ctrl.c file. This function is registered when the R_IRQ_Open() function is executed. This function can be reused for any cellular modules.

7.4.103 cellular_ring_task()

The cellular_ring_task() is a task function defined in the cellular_irq_ctrl.c file. This function monitors the RING line. This function can be reused for any cellular modules.

7.4.104 cellular_module_reset()

The cellular_module_reset() function is defined in the cellular_module_reset.c file. This function performs a hardware reset. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Replace the processing of this function by appropriate alternate processing as follows:

Line 62: Modify the specification of the cellular_shutdownsocket() function.

Line 115: Replace the atc_sqnautoconnect_check() function by the AT command execution function that is appropriate for the cellular module.
Alternatively, delete the function.

7.4.105 cellular_pin_reset()

The cellular_pin_reset() function is a static function defined in the cellular_module_reset.c file. This function performs a hardware reset. This function can be reused for any cellular modules.

7.4.106 cellular_power_down()

The cellular_power_down() function is defined in the cellular_power_down.c file. This function performs a hardware reset. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Replace the processing of this function by appropriate alternate processing as follows:

Line 62: Replace the atc_sqnsshdn() function by the AT command execution function that is appropriate for the cellular module.

7.4.107 cellular_psm_config()

The cellular_psm_config() function is defined in the cellular_psm_config.c file. This function performs PSM configuration. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Replace the processing of this function by appropriate alternate processing as follows:

Lines 64 and 116: Replace the atc_sqnricfg() function by the AT command execution function that is appropriate for the cellular module.

Lines 68 and 120: Replace the atc_sqnipscfg() function by the AT command execution function that is appropriate for the cellular module.

Line 74: Replace the atc_sqnpscfg() function by the AT command execution function that is appropriate for the cellular module.

7.4.108 cellular_psm_config_fail()

The cellular_psm_config_fail() function is a static function defined in the cellular_psm_config.c file. This function performs the handling when PSM configuration fails. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Replace the processing of this function by appropriate alternate processing as follows:

Line 188: Replace the atc_sqnipscfg() function by the AT command execution function that is appropriate for the cellular module.

Line 193: Replace the atc_sqnricfg() function by the AT command execution function that is appropriate for the cellular module.

7.4.109 cellular_rts_ctrl()

The cellular_rts_ctrl() function is defined in the cellular_rts_ctrl.c file. This function controls the RTS pin. This function can be reused for any cellular modules.

7.4.110 cellular_rts_hw_flow_enable()

The cellular_rts_hw_flow_enable() function is defined in the cellular_rts_ctrl.c file. This function enables hardware flow control of the RTS pin. This function can be reused for any cellular modules.

7.4.111 cellular_rts_hw_flow_disable()

The cellular_rts_hw_flow_disable() function is defined in the cellular_rts_ctrl.c file. This function disables hardware flow control of the RTS pin. This function can be reused for any cellular modules.

7.4.112 cellular_serial_open()

The cellular_serial_open() function is defined in the cellular_sci_ctrl.c file. This function executes the R_SCI_Open() function. This function can be reused for any cellular modules.

7.4.113 cellular_serial_close()

The cellular_serial_close() function is defined in the cellular_sci_ctrl.c file. This function executes the R_SCI_Close() function. This function can be reused for any cellular modules.

7.4.114 cellular_uart_callbResponse()

The cellular_uart_callbResponse() function is a callback function defined in the cellular_sci_ctrl.c file. This function is registered when the R_SCI_Open() function is executed. This function can be reused for any cellular modules.

7.4.115 cellular_semaphore_init()

The cellular_semaphore_init() function is defined in the cellular_semaphore_ctrl.c file. This function initializes the semaphores used in RYZ014A Cellular FIT Module. This function can be reused for any cellular modules.

7.4.116 cellular_shutdownsocket()

The cellular_shutdownsocket() function is defined in the cellular_shutdownsocket.c file. This function shuts down the socket. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Replace the processing of this function by appropriate alternate processing as follows:

Line 76: Replace the atc_sqnsh() function by the AT command execution function that is appropriate for the cellular module.

7.4.117 cellular_smcwrx()

The cellular_smcwrx() function is defined in the cellular_smcwrx.c file. This function measures the received signal strength. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Replace the processing of this function by appropriate alternate processing as follows:

Line 62: Replace the atc_smcwrx() function by the AT command execution function that is appropriate for the cellular module.

7.4.118 cellular_smcwtx()

The cellular_smcwtx() function is defined in the cellular_smcwtx.c file. This function performs test transmission. This function uses a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules. Replace the processing of this function by appropriate alternate processing as follows:

Line 63: Replace the atc_smcwtx() function by the AT command execution function that is appropriate for the cellular module.

7.4.119 cellular_start_recv_task()

The cellular_start_recv_task() function is defined in the cellular_task_ctrl.c file. This function creates the data reception task used in RYZ014A Cellular FIT Module. This function can be reused for any cellular modules.

7.4.120 cellular_start_ring_task()

The cellular_start_ring_task() function is defined in the cellular_task_ctrl.c file. This function creates the RING pin control task used in RYZ014A Cellular FIT Module. This function can be reused for any cellular modules.

7.4.121 cellular_timeout_init()

The cellular_timeout_init() function is defined in the cellular_timeout_ctrl.c file. This function sets the timeout value. This function can be reused for any cellular modules.

7.4.122 cellular_check_timeout()

The cellular_check_timeout() function is defined in the cellular_timeout_ctrl.c file. This function performs timeout processing. This function can be reused for any cellular modules.

7.4.123 cellular_job_check()

The cellular_job_check() function is a static function defined in the r_cellular_receive_task.c file. This function determines the type of the response string. This function can be reused for any cellular modules.

7.4.124 cellular_response_string_check()

The cellular_response_string_check() function is a static function defined in the r_cellular_receive_task.c file. This function determines the response string. This function can be reused for any cellular modules.

7.4.125 cellular_response_check()

The cellular_response_check() function is a static function defined in the r_cellular_receive_task.c file. This function determines the result code. This function can be reused for any cellular modules.

7.4.126 cellular_data_send_command()

The cellular_data_send_command() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of the intermediate result code “>”. This function can be reused for any cellular modules. However, the intermediate result code must be changed according to the cellular module. For details, refer to section 4.6.

7.4.127 cellular_get_data_reception()

The cellular_get_data_reception() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+SQNSRING”. This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.128 cellular_request_data()

The cellular_request_data() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+SQNSRECV”. This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.129 cellular_store_data()

The cellular_store_data() function is a static function defined in the r_cellular_receive_task.c file. This function receives the data that is sent after reception of “+SQNSRECV”. Modify the processing appropriately according to the specifications of the socket data reception command of the cellular module you use.

7.4.130 cellular_get_data_reception()

The cellular_get_data_reception() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+SQNDNSLKUP”. This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.131 cellular_get_ap_connect_status()

The cellular_get_ap_connect_status() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+CGATT”. This function can be reused for any cellular modules.

7.4.132 cellular_get_ap_connect_config()

The cellular_get_ap_connect_config() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+CGDCONT”. This function can be reused for any cellular modules.

7.4.133 cellular_station_info()

The cellular_station_info() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+CREG" or "+CEREG". This function can be reused for any cellular modules.

7.4.134 cellular_control_level()

The cellular_control_level() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+CFUN". This function can be reused for any cellular modules.

7.4.135 cellular_cpin_status()

The cellular_cpin_status() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+CPIN". This function can be reused for any cellular modules.

7.4.136 cellular_get_time()

The cellular_get_time() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+CCLK". This function can be reused for any cellular modules.

7.4.137 cellular_get_imei()

The cellular_get_imei() function is a static function defined in the r_cellular_receive_task.c file. This function processes the response string returned when the "AT+CGSN" command is executed. This function can be reused for any cellular modules.

7.4.138 cellular_get_imsi()

The cellular_get_imsi() function is a static function defined in the r_cellular_receive_task.c file. This function processes the response string returned when the "AT+CIMI" command is executed. This function can be reused for any cellular modules.

7.4.139 cellular_system_start()

The cellular_system_start() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+SYSSTART". This function can be reused for any cellular modules.

7.4.140 cellular_disconnect_socket()

The cellular_disconnect_socket() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+SQNSH”. This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.141 cellular_get_timezone()

The cellular_get_timezone() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+CTZE”. This function can be reused for any cellular modules.

7.4.142 cellular_get_service_status()

The cellular_get_service_status() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+COPS”. This function can be reused for any cellular modules.

7.4.143 cellular_get_service_status()

The cellular_get_service_status() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+COPS”. This function can be reused for any cellular modules.

7.4.144 cellular_get_pdp_status()

The cellular_get_pdp_status() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+CGACT”. This function can be reused for any cellular modules.

7.4.145 cellular_get_pdp_addr()

The cellular_get_pdp_addr() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+CGPADDR”. This function can be reused for any cellular modules.

7.4.146 cellular_get_psms()

The cellular_get_psms() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+CPSMS”. This function can be reused for any cellular modules.

7.4.147 cellular_get_edrx()

The cellular_get_edrx() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+SQNEDRX”. This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.148 cellular_get_signal()

The cellular_get_signal() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+CSQ”. This function can be reused for any cellular modules.

7.4.149 cellular_res_command_send_sim()

The cellular_res_command_send_sim() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+CRSM”. This function can be reused for any cellular modules.

7.4.150 cellular_timezone_info()

The cellular_timezone_info() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+CTZV". This function can be reused for any cellular modules.

7.4.151 cellular_ind_info()

The cellular_ind_info() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+CIEV". This function can be reused for any cellular modules.

7.4.152 cellular_get_svn()

The cellular_get_svn() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+CGSN". This function can be reused for any cellular modules.

7.4.153 cellular_get_lrsvn()

The cellular_get_lrsvn() function is a static function defined in the r_cellular_receive_task.c file. This function processes the response string returned when the "ATI1" command is executed. This function can be reused for any cellular modules.

7.4.154 cellular_get_phone_number()

The cellular_get_phone_number() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+CNUM". This function can be reused for any cellular modules.

7.4.155 cellular_get_iccid()

The cellular_get_iccid() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+SQNCCID". This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.156 cellular_ping()

The cellular_ping() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+PING". This function can be reused for any cellular modules.

7.4.157 cellular_get_cellinfo()

The cellular_get_cellinfo() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+SQNMONI". This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.158 cellular_get_autoconnect()

The cellular_get_autoconnect() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+SQNAUTOCONNECT". This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.159 cellular_get_ctm()

The cellular_get_ctm() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of "+SQNCTM". This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.160 cellular_set_smcwrx()
The cellular_set_smcwrx() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+SMCWRX”. This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.161 cellular_set_smcwtx()
The cellular_set_smcwtx() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+SMCWTX”. This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.162 cellular_shutdown_info()
The cellular_shutdown_info() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+SHUTDOWN”. This function can be reused for any cellular modules.

7.4.163 cellular_firmupgrade_info()
The cellular_firmupgrade_info() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+SQNSUPGRADE”. This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.164 cellular_get_certificate()
The cellular_get_certificate() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “+SQNSNVR”. This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.165 cellular_get_revision()
The cellular_get_revision() function is a static function defined in the r_cellular_receive_task.c file. This function processes the response string returned when the “AT+CGMR” command is executed. This function can be reused for any cellular modules.

7.4.166 cellular_response_skip()
The cellular_response_skip() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of an unregistered response string. This function can be reused for any cellular modules.

7.4.167 cellular_memclear()
The cellular_memclear() function is a static function defined in the r_cellular_receive_task.c file. This function initializes the data receive buffer. This function can be reused for any cellular modules.

7.4.168 cellular_exit()
The cellular_exit() function is a static function defined in the r_cellular_receive_task.c file. This function performs the action to be taken upon reception of “^EXIT”. This function can be reused for any cellular modules.

7.4.169 cellular_system_state_change()
The cellular_system_state_change() function is a static function defined in the r_cellular_receive_task.c file. This function updates the access point connection status upon reception of “+CREG” or “+CEREG”. This function can be reused for any cellular modules.

7.4.170 cellular_get_at_command()

The cellular_get_at_command() function is a static function defined in the r_cellular_receive_task.c file. This function acquires the AT command that is currently running. This function can be reused for any cellular modules.

7.4.171 cellular_set_atc_response()

The cellular_set_atc_response() function is a static function defined in the r_cellular_receive_task.c file. This function stores the execution result of the AT command. This function can be reused for any cellular modules.

7.4.172 cellular_cleardata()

The cellular_cleardata() function is a static function defined in the r_cellular_receive_task.c file. This function notifies the user-registered callback function of the response string, outputs log data, and clear the receive buffer. This function can be reused for any cellular modules.

7.4.173 cellular_charcheck()

The cellular_charcheck() function is a static function defined in the r_cellular_receive_task.c file. This function performs the processing that is needed when the “AT+SQNSUPGRADE” command is executed in blocking mode. This function responds to a command dedicated to the RYZ014A cellular module, so it cannot be used for other cellular modules.

7.4.174 binary_conversion()

The binary_conversion() function is a static function defined in the r_cellular_receive_task.c file. This function converts a binary number to a decimal number. This function can be reused for any cellular modules.

7.4.175 Functions in the RTOS Folder

All functions that are defined in the source files placed in the RTOS folder can be reused for any cellular modules.

7.5 Sections Needing Modification in the r_bsp Module

7.5.1 R_BSP_NOP()

The R_BSP_NOP() function is used as a NOP instruction. Replace the processing of this function by appropriate alternate processing according to the MCU you use.

The R_BSP_NOP() function is used in the following two types of sections:

- Sections where the function is specified for reducing the transmission time
- Sections where the function is specified based on coding rules

7.5.1.1 Sections Where the Function Is Specified for Reducing the Transmission Time

In RYZ014A Cellular FIT Module, the control mode (control by software or hardware) can be selected for the CTS and RTS pins. If software control is selected for the CTS pin and hardware control is selected for the RTS pin, data is sent to the cellular module in bytes. In this case, if the vTaskDelay() function is used for timeout processing (in the same way as when the CTS pin is controlled by hardware and the RTS pin is controlled by software), a delay of 1 ms (smallest unit) arises each time a single byte is sent, which results in an increased data transmission time. The R_BSP_NOP() function is used to prevent this problem.

The sections where the function is used for reducing the transmission time are as follows:

- r_cellular_sendsocket.c: Line 421
- r_cellular_writecertificate.c: Line 396

7.5.1.2 Sections Where the Function is Specified Based on Coding Rules

All instances of the R_BSP_NOP() function are specified based on the coding rules except those used in section 7.5.1.1.

7.5.2 R_BSP_RegisterProtectDisable()

The R_BSP_RegisterProtectDisable() function disables write protection for registers. Replace the processing of this function by appropriate alternate processing according to the MCU you use.

The R_BSP_RegisterProtectDisable() function is used in the following sections:

- cellular_rts_ctrl.c: Lines 67 and 85

7.5.3 R_BSP_RegisterProtectEnable()

The R_BSP_RegisterProtectEnable() function enables write protection for registers. Replace the processing of this function by appropriate alternate processing according to the MCU you use.

The R_BSP_RegisterProtectEnable() function is used in the following sections:

- cellular_rts_ctrl.c: Lines 72 and 89

7.6 Sections Needing Modification in the r_sci_rx Module

7.6.1 R_SCI_Open()

The R_SCI_Open() function enables the SCI channel. This function also monitors the transmit end interrupt by registering the callback function that is called from the interrupt. Replace the processing of this function by appropriate alternate processing according to the MCU you use.

The R_SCI_Open() function is used in the following sections:

- cellular_sci_ctrl.c: Line 69

The callback function registered when the R_SCI_Open() function is executed is defined in the following file. If you do not use the r_sci module, replace the processing of this function by appropriate alternate processing.

- cellular_sci_ctrl.c: Lines 118 to 165 (cellular_uart_callbResponse() function)

7.6.2 R_SCI_Close()

The R_SCI_Open() function disables the SCI channel. Replace the processing of this function by appropriate alternate processing according to the MCU you use.

The R_SCI_Close() function is used in the following sections:

- cellular_sci_ctrl.c: Line 105

7.6.3 R_SCI_Send()

The R_SCI_Send() function sends data. Replace the processing of this function by appropriate alternate processing according to the MCU you use.

The R_SCI_Send() function is used in the following sections:

- r_cellular_sendsocket.c: Lines 217 and 248
- r_cellular_writecertificate.c: Lines 195 and 230
- cellular_execute_at_cmd.c: Lines 145 and 178

7.6.4 R_SCI_Receive()

The R_SCI_Receive() function receives data. Replace the processing of this function by appropriate alternate processing according to the MCU you use.

The R_SCI_Receive() function is used in the following sections:

- r_cellular_receive_task.c: Lines 275, 777, 1617, 1655, 1848, 1859, 1909, and 1923

7.6.5 R_SCI_Control()

The R_SCI_Control() function enables hardware control of the CTS function and sets the transmission priority level for the target channel. Replace the processing of this function by appropriate alternate processing according to the MCU you use.

The R_SCI_Control() function is used in the following sections:

- cellular_sci_ctrl.c: Line 79 (settings for hardware control of the CTS function) and line 82 (settings for the transmission priority level)

7.7 Sections Needing Modification in the r_irq_rx Module

7.7.1 R_IRQ_Open()

The R_IRQ_Open() function enables interrupt requests. This function also registers a callback function to monitor the target pin (RING pin). Replace the processing of this function by appropriate alternate processing according to the MCU you use.

The RING pin is monitored only when PSM (Power Saving Mode) is used. Therefore, you do not need to replace the processing of the R_IRQ_Open() function if PSM is not used.

The R_IRQ_Open() function is used in the following sections:

- cellular_irq_ctrl.c: Line 60

The callback function registered when the R_IRQ_Open() function is executed is defined in the following file. If you do not use the r_irq module, replace the processing of this function by appropriate alternate processing.

- cellular_sci_ctrl.c: Lines 97 to 117 (cellular_ring_callbResponse() function)

7.7.2 R_IRQ_Close()

The R_SCI_Open() function disables interrupt requests. Replace the processing of this function by appropriate alternate processing according to the MCU you use.

The R_IRQ_Close() function is used in the following sections:

- cellular_irq_ctrl.c: Line 86

7.8 Sections Needing Modification in FreeRTOS

7.8.1 xTaskCreate()

The xTaskCreate() function creates a task. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The xTaskCreate() function is used in the following sections:

- cellular_create_task.c: Line 64
  Note: Also modify the decision statement on line 71 according to the OS you use.

7.8.2 vTaskDelay()

The vTaskDelay() function delays task execution. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The vTaskDelay() function is used in the following sections:

- cellular_delay_task.c: Line 54

7.8.3 xTaskGetTickCount()

The xTaskGetTickCount() function acquires the counter value of the system clock. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The xTaskGetTickCount() function is used in the following sections:

- cellular_get_tickcount.c: Line 52

7.8.4 vTaskSuspend()

The vTaskSuspend() function suspends execution of a task. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The vTaskSuspend() function is used in the following sections:

- cellular_delete_task.c: Line 53

7.8.5 vTaskDelete()

The vTaskDelete() function deletes a task. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The vTaskDelete() function is used in the following sections:

- cellular_delete_task.c: Line 54

7.8.6 xEventGroupCreate()

The xEventGroupCreate() function is used to create an event group. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The xEventGroupCreate() function is used in the following sections:

- cellular_create_event_group.c: Line 54

7.8.7 xEventGroupWaitBits()

The xEventGroupWaitBits() function is used to wait for event bits within an event group to be set. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The xEventGroupWaitBits() function is used in the following sections:

- cellular_get_event_flg.c: Lines 57 and 65
  Note: Also modify the decision statement on line 72 according to the OS you use.

7.8.8 xEventGroupSetBitsFromISR()

The xEventGroupSetBitsFromISR() function is used to set event bits within an event group from an interrupt routine. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The xEventGroupSetBitsFromISR() function is used in the following sections:

- cellular_set_event_flg.c: Line 55
  Note: Also modify the decision statement on line 59 according to the OS you use.

7.8.9 xEventGroupSync()

The xEventGroupSync() function is used to set event bits within an event group and then wait for a combination of event bits to be set within the same event group. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The xEventGroupSync() function is used in the following sections:

- cellular_syncro_event_group.c: Lines 60 and 67

7.8.10 vEventGroupDelete()

The vEventGroupDelete() function is used to delete a generated event group. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The vEventGroupDelete() function is used in the following sections:

- cellular_delete_event_group.c: Line 53

7.8.11 xSemaphoreCreateMutex()

The xSemaphoreCreateMutex() function is used to create a mutex. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The xSemaphoreCreateMutex() function is used in the following sections:

- cellular_create_semaphore.c: Line 54

7.8.12 xSemaphoreTake()

The xSemaphoreTake() function is used to create a semaphore. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The xSemaphoreTake() function is used in the following sections:

- cellular_take_semaphore.c: Line 52

7.8.13 xSemaphoreGive()

The xSemaphoreGive() function is used to release a semaphore. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The xSemaphoreGive() function is used in the following sections:

- cellular_give_semaphore.c: Line 53

7.8.14 vSemaphoreDelete()

The vSemaphoreDelete() function is used to delete a semaphore. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The vSemaphoreDelete() function is used in the following sections:

- cellular_delete_semaphore.c: Line 56

7.8.15 pvPortMalloc()

The pvPortMalloc() function is used to secure heap memory. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The pvPortMalloc() function is used in the following sections:

- cellular_malloc.c: Line 52

7.8.16 vPortFree()

The vPortFree() function is used to free the secured heap memory. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The vPortFree() function is used in the following sections:

- cellular_free.c: Line 53

7.8.17 taskENTER_CRITICAL()

The taskENTER_CRITICAL() function is used to enter into a critical section. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The taskENTER_CRITICAL() function is used in the following sections:

- cellular_interrupt_ctrl.c: Line 53

7.8.18 taskEXIT_CRITICAL()

The taskEXIT_CRITICAL() function is used to exit from a critical section. If you use an OS other than FreeRTOS, replace the processing of this function by appropriate alternate processing.

The taskEXIT_CRITICAL() function is used in the following sections:

- cellular_interrupt_ctrl.c: Line 77

8. Reference Documents

User’s Manual: Hardware
The latest versions can be downloaded from the Renesas Electronics website.

Technical Update/Technical News
The latest information can be obtained from the Renesas Electronics website.

User’s Manual: Development Environment
RX Family CC-RX Compiler User’s Manual (R20UT3248)
The latest versions can be downloaded from the Renesas Electronics website.

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Dec. 8, 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 be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. 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.
6. 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.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. 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.5.0-1 October 2020)

Corporate Headquarters

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

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/.

Trademarks

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

© 2023 Renesas Electronics Corporation. All rights reserved.