# RZ/A2M Group

## Example of Initialization

### Introduction

This document describes required settings for initialization using the RZ/A2M boot mode 3 (Serial Flash booting , 3.3V product).

### Target Device

RZ/A2M

When using this application note with other Renesas MCUs, careful evaluation is recommended after making modifications to comply with the alternative MCU.

## Contents

# 1. Specifications

After the reset is cancelled, the following devices are initialized by the program located in the serial flash memory assigned in SPI Multi I/O bus space.

SPI Multi I/O Bus Controller
Clock Pulse Generator
Interrupt Controller
OS Timer
General I/O Ports
Memory Management Unit
Primary Cache
Secondary Cache

Hereinafter, in this application note, SPI Multi I/O Bus Controller, Clock Pulse Generator, Interrupt Controller, OS Timer, Serial Communication Interface with FIFO, General I/O Ports, Power-down Mode and Memory Management Unit are referred to as SPIBSC, CPG, INTC, OSTM, SCIFA, STB and MMU, respectively.

Table 1.1 lists the peripheral functions supported in this application note. Also, Figure 1.1 shows the operation overview.

**Table 1.1 Peripheral Function to be Used**

| Peripheral Function | Usage |
|---|---|
| SPI Multi I/O Bus Controller (SPIBSC) | Configure SPIBSC as external address space read mode and generate the signal for CPU to read data from serial flash memory assigned in SPI Multi I/O bus space directly |
| Clock Pulse Generator (CPG) | Generate the operating frequency of RZ/A2M |
| Interrupt Controller (INTC) | Control the interrupt fired by OSTM channel 0, OSTM channel 2 and SCIFA channel 4 |
| OS Timer (OSTM) | Use OSTM channel 0 and channel2<br>• Channel 0<br>Control the cycle for blinking LED<br>• Channel 2<br>Use for time management by OS Abstraction Layer |
| Serial Communication Interface with FIFO (SCIFA) | Control the communication between RZ/A2M and host PC using SCIFA channel 4 |
| General I/O Ports (GPIO) | Switch the multiplexed pin functions for SCIFA channel 4<br>Control pins for blinking LED |
| Power-down Mode (STB) | Cancel the RZ/A2M peripheral I/O module standby<br>Enable writing to the on-chip data retention RAM |
| Memory Management Unit (MMU), Primary (L1) Cache and Secondary (L2) Cache | Generate MMU translation table for configuring the area where L1 cache is enabled, memory attribute of each region of memory (i.e. Normal, Device, Strongly-Ordered), etc.<br>Enable L1 and L2 Cache |

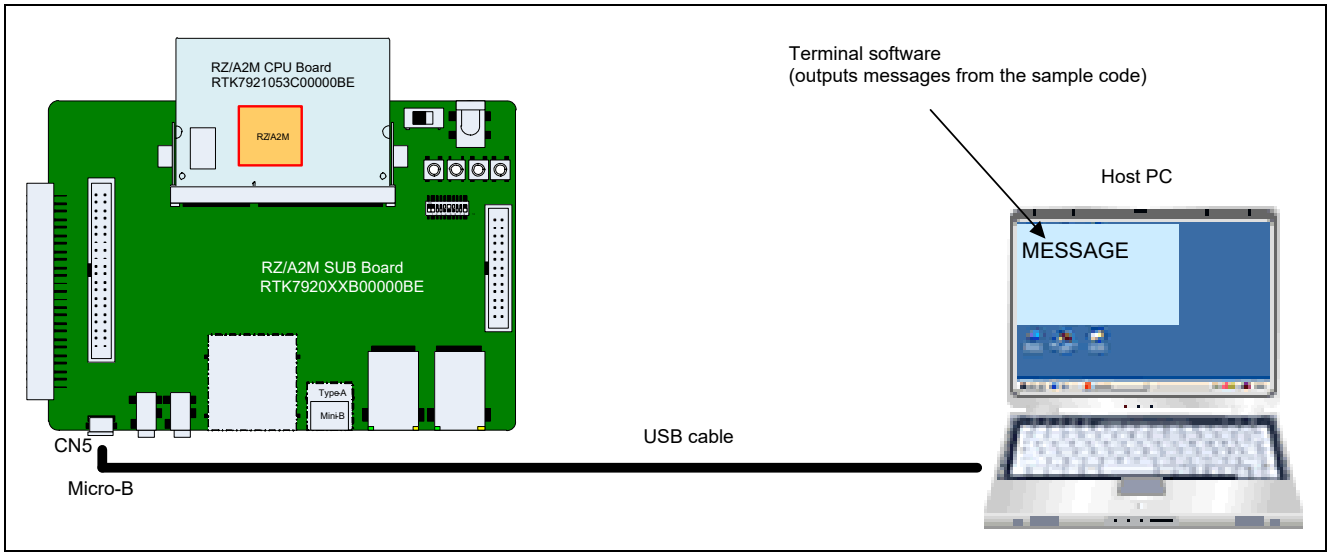



**Figure 1.1 Operation Overview**

## 2. Operation Confirmation Conditions

It was confirmed that the sample program accompanying this application note works as expected under the condition shown in Table 2.1 and Table 2.2.

**Table 2.1 Operation Conformation Condition (1/2)**

| Item | Description |
|---|---|
| MCU used | RZ/A2M |
| Operating frequency<br>(Note) | CPU Clock (Iϕ) : 528MHz<br>Image processing clock (Gϕ) : 264MHz<br>Internal Bus Clock (Bϕ) : 132MHz<br>Peripheral Clock 1 (P1ϕ) : 66MHz<br>Peripheral Clock 0 (P0ϕ) : 33MHz<br>QSPI0_SPCLK : 66MHz<br>CKIO : 132MHz |
| Operating voltage | Power supply voltage (I/O): 3.3 V<br>Power supply voltage (either 1.8V or 3.3V I/O (PVcc SPI)) : 3.3V<br>Power supply voltage (internal): 1.2 V |
| Integrated development environment | e2 studio V7.6.0 |
| C compiler | "GNU Arm Embedded Tool chain 6-2017-q2-update"<br>compiler options(except directory path)<br><br>Release Configuration:<br>-mcpu=cortex-a9 -march=armv7-a -marm<br>-mlittle-endian -mfloat-abi=hard -mfpu=neon -mno-unaligned-access<br>-Os -ffunction-sections -fdata-sections -Wunused -Wuninitialized<br>-Wall -Wextra -Wmissing-declarations -Wconversion -Wpointer-arith<br>-Wpadded -Wshadow -Wlogical-op -Waggregate-return -Wfloat-equal<br>-Wnull-dereference -Wmaybe-uninitialized -Wstack-usage=100<br>-fabi-version=0<br><br>Hardware Debug Configuration:<br>-mcpu=cortex-a9 -march=armv7-a -marm<br>-mlittle-endian -mfloat-abi=hard -mfpu=neon -mno-unaligned-access<br>-Og -ffunction-sections -fdata-sections -Wunused -Wuninitialized<br>-Wall -Wextra -Wmissing-declarations -Wconversion -Wpointer-arith<br>-Wpadded -Wshadow -Wlogical-op -Waggregate-return -Wfloat-equal<br>-Wnull-dereference -Wmaybe-uninitialized -g3 -Wstack-usage=100<br>-fabi-version=0 |

Note: The operating frequency used in clock mode 1 (Clock input of 24MHz from EXTAL pin)

**Table 2.2 Operation Conformation Condition (2/2)**

| Item | Description |
|---|---|
| Operation mode | Boot mode 3<br>(Serial Flash booting , 3.3V product) |
| Terminal software communication settings | • Communication speed: 115200bps<br>• Data length: 8 bits<br>• Parity: None<br>• Stop bits: 1 bit<br>• Flow control: None |
| Board to be used | RZ/A2M CPU board RTK7921053C00000BE<br>RZ/A2M SUB board RTK79210XXB00000BE |
| Device<br>(functionality to be used on the board) | • Serial flash memory allocated to SPI multi-I/O bus space<br>- Manufacturer : Macronix Inc.<br>- Model Name : MX25L51245GXD<br>• RL78/G1C (USB-to-serial converter, which is used for communicating with host PC)<br>• LED1 |

## 3. Reference Application Notes

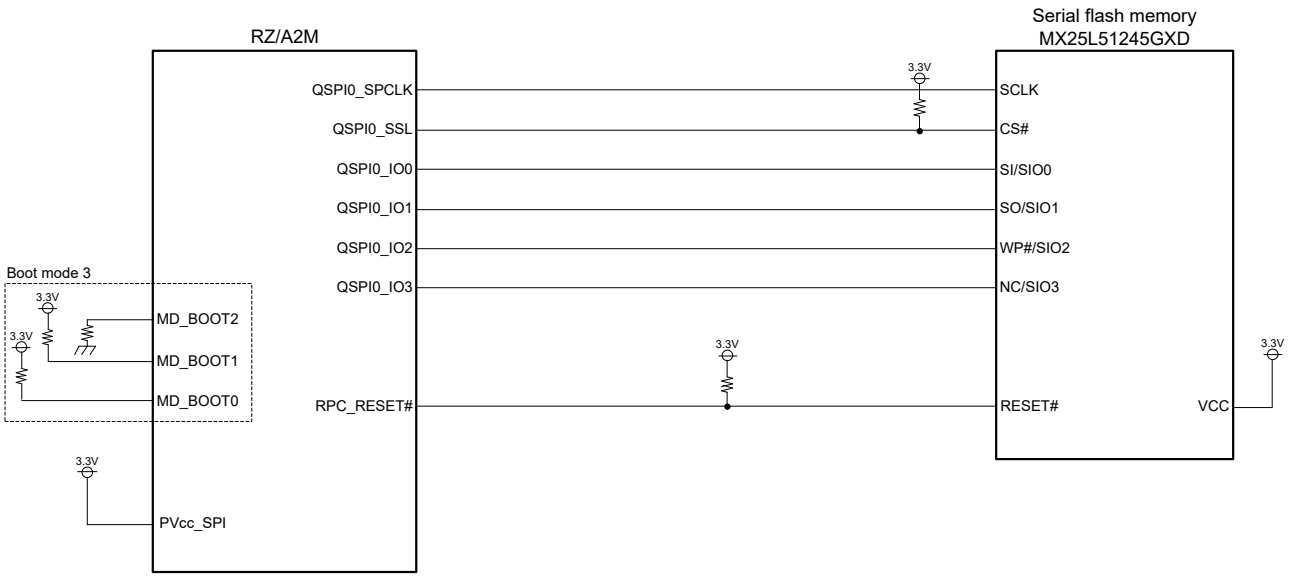
In this chapter, application note referenced in this note are listed:

- RZ/A2M Group Example of booting from serial flash memory (R01AN4333EJ)

# 4. Hardware

## 4.1 Hardware Configuration

In the example of initial setup described in this application note, boot mode is configured as boot mode 3 and so, all the processing runs by the program located in the serial flash memory connected to SPI Multi I/O bus space. Figure 4.1 shows the example of hardware connection diagram when RZ/A2M is booted up from serial flash memory assigned in SPI Multi I/O bus area under boot mode 3.




Note: The symbol of "#" indicates negative logic (or active low).

**Figure 4.1 Example of Connection Diagram under Boot Mode 3**

## 4.2 Pins Used

Table 4.1 lists the pins used in this application note.

**Table 4.1 Pins to be Used and Their Function**

| Pin Name | I/O | Description |
|---|---|---|
| MD_BOOT2 | Input | Boot mode selection (Select boot mode 3) |
| MD_BOOT1 | Input | MD_BOOT2: "L", MD_BOOT1: "H", MD_BOOT0: "H" |
| MD_BOOT0 | Input | |
| QSPI0_SPCLK | Output | Clock output to serial flash memory |
| QSPI0_SSL | Output | Slave selection for serial flash memory |
| QSPI0_IO0 | Input/Output | Data 0 pin for serial flash memory |
| QSPI0_IO1 | Input/Output | Data 1 pin for serial flash memory |
| QSPI0_IO2 | Input/Output | Data 2 pin for serial flash memory |
| QSPI0_IO3 | Input/Output | Data 3 pin for serial flash memory |
| RPC_RESET# (Note) | Output | Reset control signal output to serial flash memory |
| P6_0 | Output | LED blinking |
| RxD4 (P9_1) | Input | Serial receive data signal |
| TxD4 (P9_0) | Output | Serial transmit data signal |

Note: The symbol of "#" indicates negative logic (or active low)

# 5. Software

## 5.1 Operation Overview

After reset is canceled, the boot startup on-chip ROM program (hereinafter referred to as Boot Program) stored in on-chip ROM (address: H'FFFF 0000) for RZ/A2M runs. Boot Program, at first, sets up the configuration for accessing serial flash memory under boot mode 3 and then, jumps to the address H'2000 0000 which is the starting address of SPI Multi I/O bus space.

This initialization program consists of the loader program placed in the address H'2000 0000 and application program placed in the address H'2001 0000.

First, the loader program sets up the appropriate configuration for accessing serial flash memory and jumps to the start-up processing implemented in the application program.

Hereafter, it is described that how initialization is performed in the sample program.

In the start-up processing, the following initialization should be carried out and then jumps to resetprg function:

Initialization of stack pointer
MMU setup
FPU setup
Initialization of memory section

In resetprg function, after setting processing of unused channels of RTC and USB, the enablement of L1 and L2 cache and initialization of INTC are carried out. Then, a certain address of the area in large-capacity on-chip RAM is specified for VBAR in order to accelerate interrupt processing. Finally, IRQ and FIQ interrupt are enabled followed by main function call.

In the main function, initial setting processing of CPG, OSTM channel 0, SCIFA channel 4, GPIO is performed. As a result of these initialization processing being executed, the main function outputs the character strings (startup message) to the terminal on the host PC connected with the serial interface and sets OSTM channel 0 timer to interval timer mode to start the timer. OSTM channel 0 generates an interrupt of 500ms cycle, by using this OSTM interrupt, the sample code repeats turning on and off the LED on the CPU board every 500ms by interrupt processing.

For details on the processing implemented in loader program, please refer to the application note "Example of booting from serial flash memory".

Figure 5.1 shows the Sequence Diagram of Initialization until Main Processing Starts.

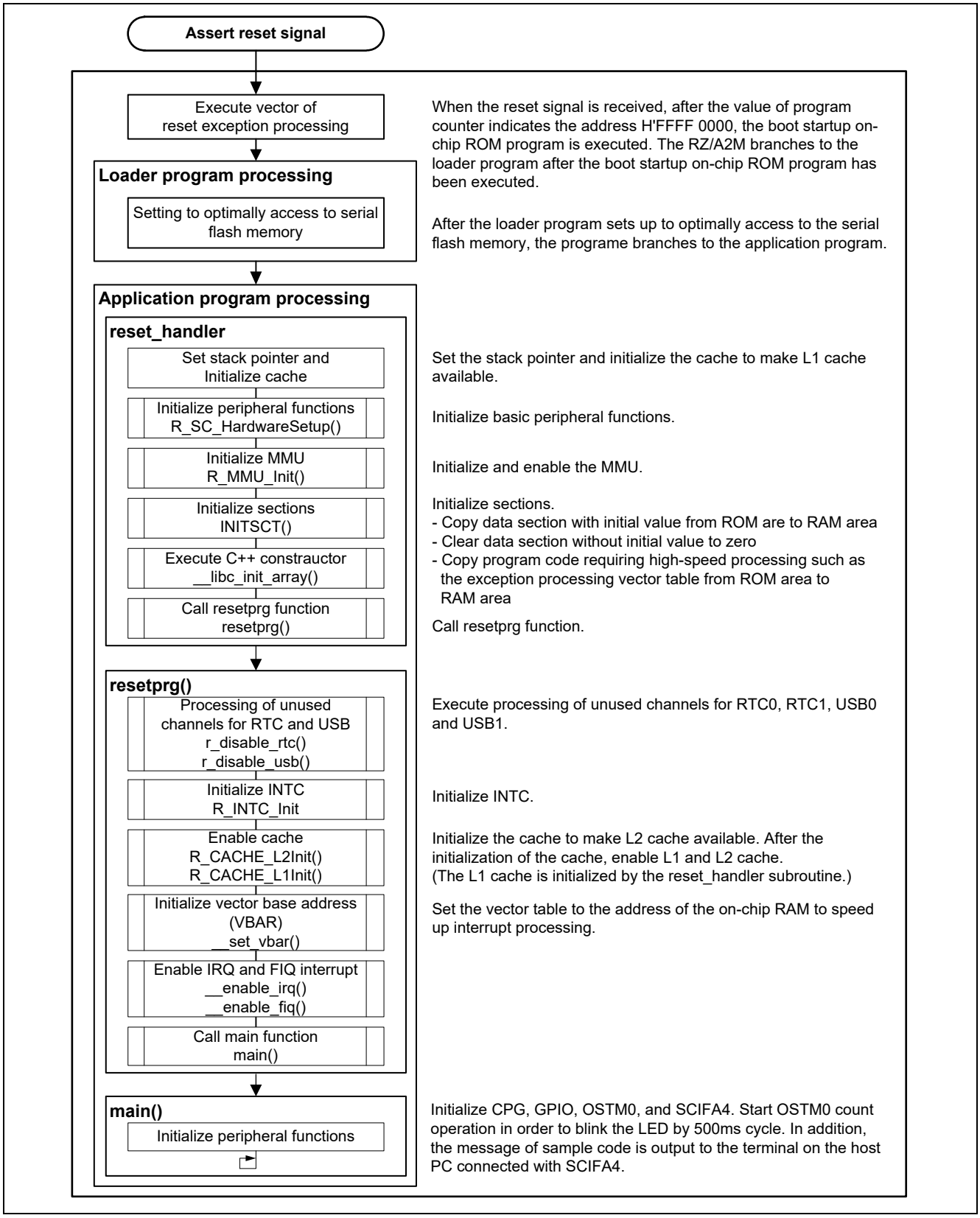



**Figure 5.1 Sequence Diagram of Initialization until Main Processing Starts**

## 5.2 Peripheral Functions and Memory Address Map in Sample Program

### 5.2.1 Setting for Peripheral Functions

Table 5.1 shows the Setting for Peripheral Functions in sample program.

**Table 5.1 Setting for Peripheral Functions**

| Peripheral Functions | Setting |
|---|---|
| CPG | CPU clock: 1/2 of the PLL circuit frequency<br>Internal bus clock: 1/8 of the PLL circuit frequency<br>Peripheral clock 1: 1/16 of the PLL circuit frequency<br><br>Concretely, each clock becomes the following frequency under the Clock Mode 1 (i.e. Frequency division ratio by divider 1 is 1/2 and frequency of the input clock frequency is multiplied by 88) when the input clock frequency is 24MHz:<br>• CPU clock (Iϕ): 528MHz<br>• Image processing clock (Gϕ): 264MHz<br>• Internal bus clock (Bϕ): 132MHz<br>• Peripheral clock 1 (P1ϕ): 66MHz<br>• Peripheral clock 0 (P0ϕ): 33MHz<br>• QSPI0_SPCLK: 66MHz (when selecting Bϕ as output clock)<br>• CKIO clock: 132MHz (when selecting Bϕ as output clock) |
| STB | Make on-chip Data Retention RAM writable and supply the clock to peripheral functions(OSTM0,OSTM2,and SCIF4). |
| GPIO | Configure the multiplexed pin function of PORT6 and PORT9 as follows:<br>• P6_0: LED blinking<br>• P9_1: RxD4, P9_0: TxD4 |
| OSTM | Configure the channel 0 and the channel 2 as interval timer mode:<br>• Channel 0<br>Sets the timer counter to have interrupt request generated every 500ms when P1ϕ is 66MHz. Control the cycle for blinking LED.<br>• Channel 2<br>Sets the timer counter to have interrupt request generated every 1ms when P1ϕ is 66MHz. Use for time management by OS Abstraction Layer. |
| INTC | Initial setup of INTC, interrupt hander registration and invocation<br>Used interrupt factor:<br>• OSTM channel 0 interrupt handler (Interrupt ID: 88)<br>• OSTM channel 2 interrupt handler (Interrupt ID: 90)<br>• SCIFA channel 4 interrupt handler (Interrupt ID: 322 and 323) |
| SCIFA | Configure SCIFA channel 4 as asynchronous communication mode<br>Serial communication setting:<br>• Data length: 8 bits<br>• Stop bits: 1 bit<br>• Parity: None<br>• LSB-first transfer<br>SCIFA is configured as follows in case of setting P1ϕ to 66MHz:<br>• Clock source: No frequency division<br>• Baud rate generator: double-speed mode<br>• Operating on the base clock with 8 times the bit rate<br>• Configures the appropriate bit rate so that the baud rate can become 115200bps. Thus, the bit rate should become -0.53% |

### 5.2.2 Memory Mapping

Figure 5.2 shows RZ/A2M group address space and memory address map of RZ/A2M CPU board.

In the sample program, the code and data using ROM area is placed in the serial flash memory connected to SPI Multi I/O bus space, while the code and data are using RAM area is placed in the large-capacity on-chip RAM.

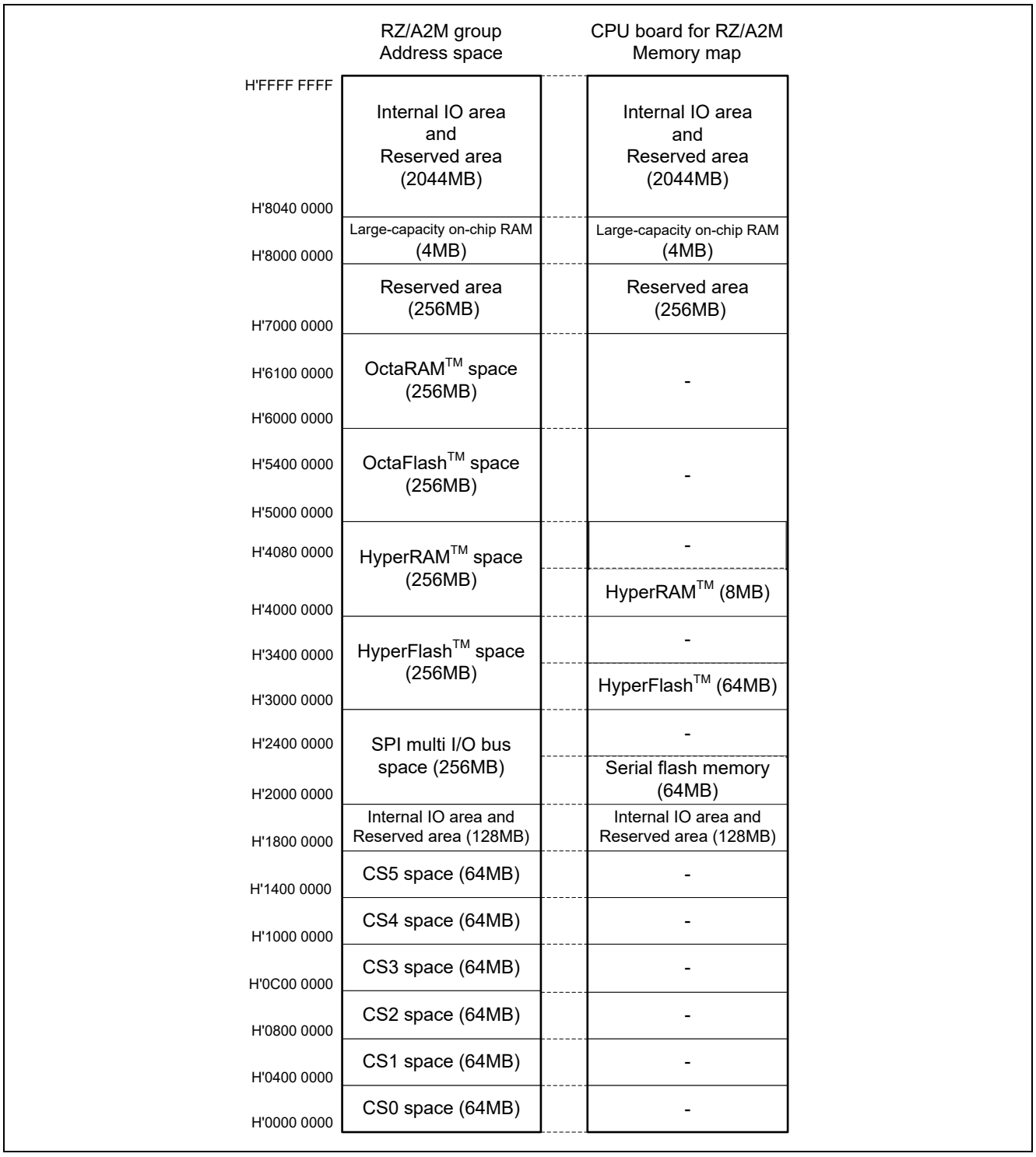



**Figure 5.2 Memory Mapping**

### 5.2.3 MMU Configuration

By using the descriptor type "Section" of first-level descriptors, the MMU is set to manage the 4 GB area in 1MB unit
from the address H'0000 0000 in response to the memory map of the hardware resource used for the RZ/A2M CPU board. Please note that MMU configuration table. (MMU_SC_TABLE[ ]) is defined in the file r_mmu_drv_sc_cfg.h. When customizing MMU configuration for your system, please specify 1MB as minimum unit.

Translation table for 1st level descriptor is configured by the Translation Table Base Control Register (TTBCR) and Translation Table Base Register 0 or 1 (TTBR0 or TTBR1). After all the translation tables are configured, MMU can be enabled by configuring the M bit in System Control Register (SCTLR). For details on MMU, please refer to "ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition Issue C".

- TTBCR
  This register determines the base register (either TTBR0 or TTBR1) used as the base address for the translation table. Also, it determines the size of translation table specified by TTBR0. In the sample program, b'000 is specified for the N[2:0] bits so that TTBR0 can be always used and the size of translation table can become 16KB (i.e. 4096 entries).
  As shown in Figure 5.3, the 1st level translation table and section descriptor format are used as translation table and 1st level descriptor respectively. When using section descriptor format, each entry of translation table becomes 1MB sized memory block, and the translation from virtual address to physical address using translation table is carried out against 4GB (4096 entries x 1MB) sized address space.
- TTBCR
  This register specifies the base address of translation table, cacheable attributes for the region where translation table is located, etc. In the sample code, the base address is "__mmu_page_table_base" defined in the file linker_script.ld, which is the starting address of section area of translation table located in the large-capacity on-chip RAM.

Figure 5.3, Table 5.2 to Table 5.4 show the overview of 1st level descriptor and its configuration in the sample program respectively. Also, Table 5.5 and Table 5.6 show the MMU setting in the sample program.

| b31 20 | 19 | 18 | 17 | 16 | 15 | 14 12 | 11 10 | 9 | 8 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Base address of section PA[31:20] | NS | 0 | nG | S | AP[2] | TEX[2:0] | AP[1:0] | IMP | Domain | XN | C | B | 1 | 0 |

**Figure 5.3 1st Level Descriptor Format when Specifying Section for Descriptor Type**

**Table 5.2 Fields of 1st Level Descriptor**

| Field | Description |
|---|---|
| TEX[2:0], C, B | Memory region attribute bits<br>Please refer to Table 5.3 on the configuration in the sample program |
| XN | Execute-never bit<br>When setting this bit to 1, the instruction shouldn't be fetched from the memory area located in the domain specified as client.<br>In the sample program, Normal region is configured as executable (i.e. XN = 1), while Strongly-ordered region is configured as non-executable (i.e. XN = 0). |
| Domain | 2 types of domain access are supported. One is client mode and the other is manager mode. Domain access type is specified by Domain Access Control Register (DACR).<br>In the sample program, D15 field of DACR is configured as b'01 (i.e. client access mode) and b'1111 (D15 field) is specified for all the spaces. |
| IMP | This is NOT implemented in RZ/A2M and therefore, the configuration of this bit is ignored. |
| AP[2], AP[1:0] | These field and the domain field determine if the access is enabled.<br>In the sample program, the access to reserved area is disabled (i.e. AP[2:0] = b'000), while the access to all the other area is enabled (i.e. AP[2:0] = b'011). |
| S | Sharable bit that can determines if the memory area is sharable.<br>In the sample program, All the Normal regions are configured as no-sharable area (i.e. S=0). |
| nG | The non-Global bit<br>In the sample program, all the regions are configured as the global (i.e. nG = 0) |
| NS (Note) | The Non-Secure bit<br>In the sample program, large-capacity on-chip RAM and external address space are configured as the Non-Secure space (i.e. NS=1), while internal peripheral module space is configured as the Secure space (i.e. NS=0). |
| PA[31:20] | MSB 12-bits of the physical address translated from virtual address |

Note: For RZ/A2M, please configure CPU security state as the "Secure" (i.e. NS bit of Secure Configuration Register sets to 0). By getting CPU to configure "Secure", CPU can access all the area where its physical address is converted into virtual address using MMU regardless of NS bit setting.
In the sample code, CPU accesses the large capacity on-chip RAM and external address space by setting NS bit to 1 on the basis that AXI bus related registers MSTACCCTL0, MSTACCCTL1, MSTACCCTL2, MSTACCCTL3 and MSTACCCTL4 and for on-chip peripheral modules are set as 1 (i.e. Non Secure Access). By default, MSTACCCTL0-4 registers are configured as 1.

**Table 5.3 Memory Attribute Setting in the Sample Program**

| TEX[2:0] | C | B | Memory Type | L1 Cache | L2 Cache |
|---|---|---|---|---|---|
| b'000 | 0 | 0 | Strongly-ordered | - | - |
| b'000 | 0 | 1 | Device (sharable) | - | - |
| b'001 | 1 | 1 | Normal memory | Enabled | Enabled |
| b'100 | 0 | 1 | Normal memory | Enabled | Disabled |
| b'100 | 0 | 0 | Normal memory | Disabled | Disabled |

**Table 5.4 MMU Settings and Its Description**

| Name of MMU settings | Memory type | Cache | NS | AP[2:0] (Access Permission) | XN |
|---|---|---|---|---|---|
| Unused | Strongly-ordered | - | 1 | Read/Write (b'011) | 1 |
| Strongly-ordered (Secure, Never-execute) | Strongly-ordered | - | 0 | Read/Write (b'011) | 1 |
| Strongly-ordered (Non-secure, Never-execute) | Strongly-ordered | - | 1 | Read/Write (b'011) | 1 |
| Strongly-ordered (Non-secure, Executable) | Strongly-ordered | - | 1 | Read/Write (b'011) | 0 |
| Sharable Device (Secure) | Device | - | 0 | Read/Write (b'011) | 1 |
| Sharable Device (Non-secure) | Device | - | 1 | Read/Write (b'011) | 1 |
| Normal (Non-secure, L1 cacheable) | Normal | Only L1 cache is enabled | 1 | Read/Write (b'011) | 0 |
| Normal (Non-secure, L2 cacheable) | Normal | Only L2 cache is enabled | 1 | Read/Write (b'011) | 0 |
| Normal (Non-secure, L1/L2 cacheable) | Normal | Both L1 and L2 are enabled | 1 | Read/Write (b'011) | 0 |
| Normal (Non-secure, Non-cacheable) | Normal | Both L1 and L2 are disabled | 1 | Read/Write (b'011) | 0 |
| Reserved | Strongly-ordered | - | 1 | Access Inhibit (b'000) | 1 |

**Table 5.5 MMU settings (1/2)**

| Virtual address | Physical address | Size | Address area | Name of MMU settings (Note) |
|---|---|---|---|---|
| H'0000 0000<br>H'03FF FFFF | H'0000 0000<br>H'03FF FFFF | 64MB | CS0 area<br>(Unused) | Unused |
| H'0400 0000<br>H'07FF FFFF | H'0400 0000<br>H'07FF FFFF | 64MB | CS1 area<br>(Unused) | Unused |
| H'0800 0000<br>H'0BFF FFFF | H'0800 0000<br>H'0BFF FFFF | 64MB | CS2 area<br>(Unused) | Unused |
| H'0C00 0000<br>H'0FFF FFFF | H'0C00 0000<br>H'0FFF FFFF | 64MB | CS3 area<br>Cacheable | Normal<br>(Non-secure, L1/L2 cacheable) |
| H'1000 0000<br>H'13FF FFFF | H'1000 0000<br>H'13FF FFFF | 64MB | CS4 area<br>(Unused) | Unused |
| H'1400 0000<br>H'17FF FFFF | H'1400 0000<br>H'17FF FFFF | 64MB | CS5 area<br>(Unused) | Unused |
| H'1800 0000<br>H'1EFF FFFF | H'1800 0000<br>H'1EFF FFFF | 112MB | Reserved | Reserved |
| H'1F00 0000<br>H'1FFF FFFF | H'1F00 0000<br>H'1FFF FFFF | 16MB | Internal I/O<br>area | Strongly-ordered<br>(Secure, Never-execute) |
| H'2000 0000<br>H'2FFF FFFF | H'2000 0000<br>H'2FFF FFFF | 256MB | SPI Multi I/O<br>bus area<br>Cacheable | Normal<br>(Non-secure, L1/L2 cacheable) |
| H'3000 0000<br>H'3FFF FFFF | H'3000 0000<br>H'3FFF FFFF | 256MB | HyperFlash<br>Cacheable | Normal<br>(Non-secure, L1/L2 cacheable) |
| H'4000 0000<br>H'4FFF FFFF | H'4000 0000<br>H'4FFF FFFF | 256MB | HyperRAM<br>Cacheable | Normal<br>(Non-secure, L1/L2 cacheable) |
| H'5000 0000<br>H'5FFF FFFF | H'5000 0000<br>H'5FFF FFFF | 256MB | OctaFlash<br>Cacheable | Normal<br>(Non-secure, L1/L2 cacheable) |
| H'6000 0000<br>H'6FFF FFFF | H'6000 0000<br>H'6FFF FFFF | 256MB | OctaRAM<br>Cacheable | Normal<br>(Non-secure, L1/L2 cacheable) |
| H'7000 0000<br>H'7FFF FFFF | H'2000 0000<br>H'2FFF FFFF | 256MB | SPI Multi I/O<br>bus area<br>Non-cacheable | Strongly-ordered<br>(Non-secure, Executable) |
| H'8000 0000<br>H'803F FFFF | H'8000 0000<br>H'803F FFFF | 4MB | Large-capacity<br>on-chip RAM<br>Cacheable | Normal<br>(Non-secure, L1/L2 cacheable) |

**Table 5.6 MMU settings (2/2)**

| Virtual address | Physical address | Size | Address area | Name of MMU settings (Note) |
|---|---|---|---|---|
| H'8040 0000<br>H'81FF FFFF | H'8040 0000 ~<br>H'81FF FFFF | 28MB | Reserved | Unused |
| H'8200 0000<br>H'823F FFFF | H'8000 0000<br>H'803F FFFF | 4MB | Large-capacity on-chip RAM Non-cacheable | Normal (Non-secure, Non-cacheable) |
| H'8240 0000<br>H'87FF FFFF | H'8240 0000<br>H'87FF FFFF | 92MB | Reserved | Unused |
| H'8800 0000<br>H'8BFF FFFF | H'0000 0000<br>H'03FF FFFF | 64MB | CS0 area (Unused) | Unused |
| H'8C00 0000<br>H'8FFF FFFF | H'0400 0000<br>H'07FF FFFF | 64MB | CS1 area (Unused) | Unused |
| H'9000 0000<br>H'93FF FFFF | H'0800 0000<br>H'0BFF FFFF | 64MB | CS2 area (Unused) | Unused |
| H'9400 0000<br>H'97FF FFFF | H'0C00 0000<br>H'0FFF FFFF | 64MB | CS3 area Non-cacheable | Normal (Non-secure, Non-cacheable) |
| H'9800 0000<br>H'9BFF FFFF | H'1000 0000<br>H'13FF FFFF | 64MB | CS4 area (Unused) | Unused |
| H'9C00 0000<br>H'9FFF FFFF | H'1400 0000<br>H'17FF FFFF | 64MB | CS5 area (Unused) | Unused |
| H'A000 0000<br>H'AFFF FFFF | H'3000 0000<br>H'3FFF FFFF | 256MB | HyperFlash Non-cacheable | Strongly-ordered (Non-secure, Executable) |
| H'B000 0000<br>H'BFFF FFFF | H'4000 0000<br>H'4FFF FFFF | 256MB | HyperRAM Non-cacheable | Normal (Non-secure, Non-cacheable) |
| H'C000 0000<br>H'CFFF FFFF | H'5000 0000<br>H'5FFF FFFF | 256MB | OctaFlash Non-cacheable | Strongly-ordered (Non-secure, Non-cacheable) |
| H'D000 0000<br>H'DFFF FFFF | H'6000 0000<br>H'6FFF FFFF | 256MB | OctaRAM Non-cacheable | Normal (Non-secure, Non-cacheable) |
| H'E000 0000<br>H'E7FF FFFF | H'E000 0000<br>H'E7FF FFFF | 128MB | Reserved | Reserved |
| H'E800 0000<br>H'FFFF FFFF | H'E800 0000<br>H'FFFF FFFF | 384MB | Internal I/O area | Strongly-ordered (Secure, Never-execute) |

Notes: 1. Regarding the relationship between MMU settings and the corresponding name, please refer to Table 5.3 and Table 5.4.
2. Physical address is written in red when virtual address is different from the corresponding physical address.

### 5.2.4 Virtual Address Space in Sample Program

Table 5.5 and Table 5.6 show the virtual address space which is configured using MMU and accessible from CPU.

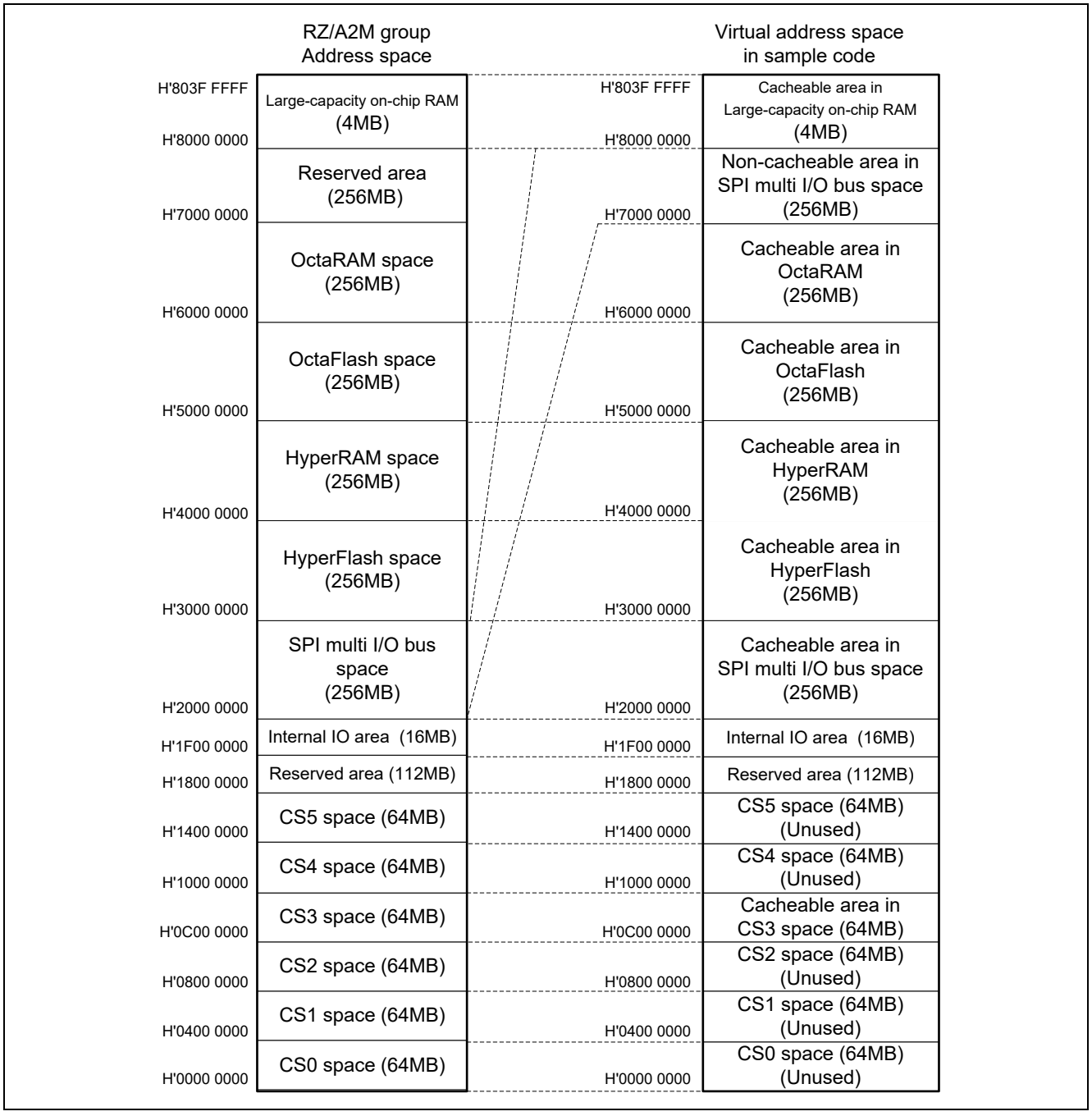
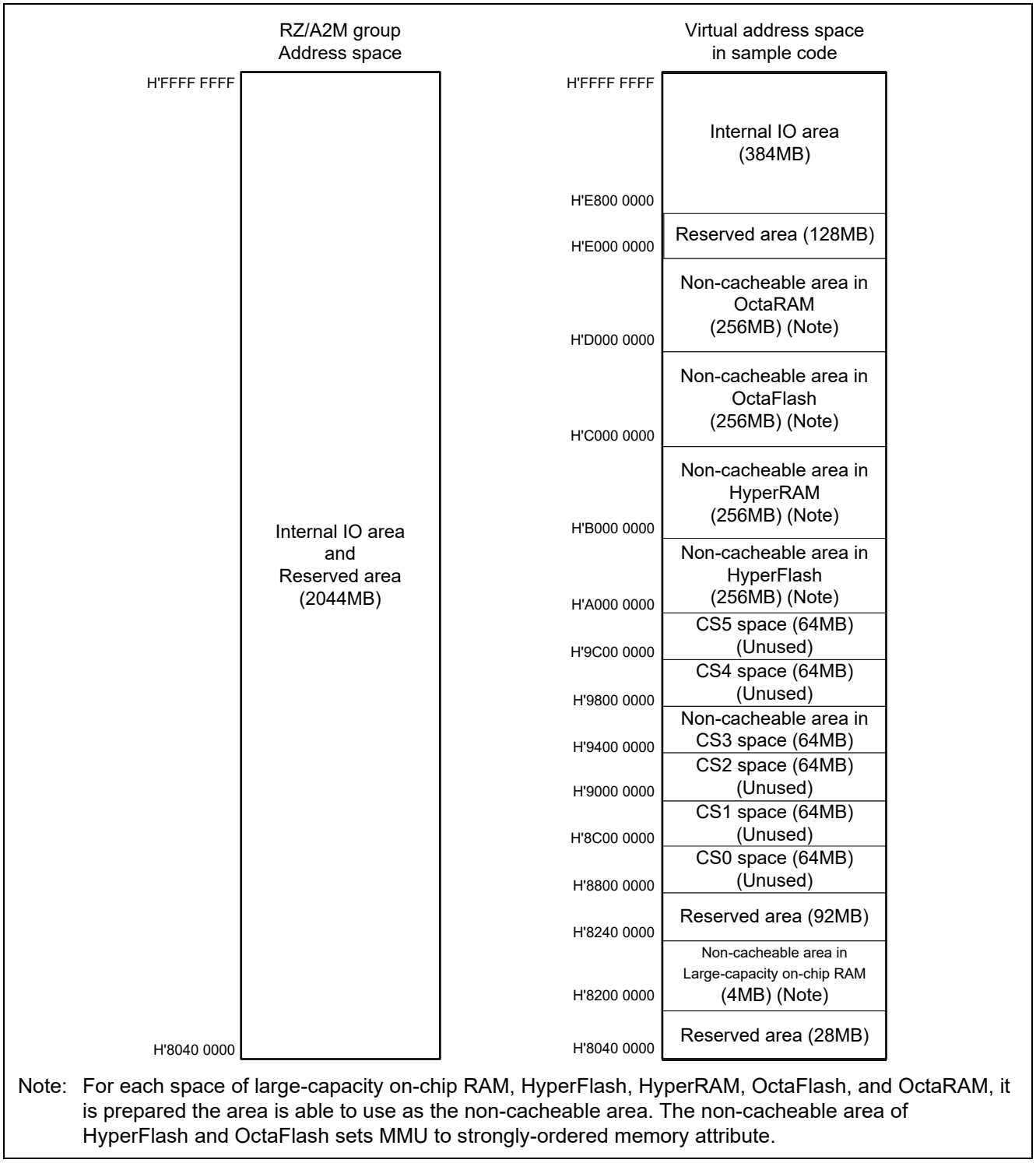



**Figure 5.4 Virtual Address Space in the Sample Program (1/2)**




Note: For each space of large-capacity on-chip RAM, HyperFlash, HyperRAM, OctaFlash, and OctaRAM, it is prepared the area is able to use as the non-cacheable area. The non-cacheable area of HyperFlash and OctaFlash sets MMU to strongly-ordered memory attribute.

**Figure 5.5 Virtual Address Space in the Sample Program (2/2)**

### 5.2.5 Section setup in the sample program

In this sample code, the exception processing vector table and the IRQ interrupt handler are assigned to the large-capacity on-chip RAM, and they are executed in such RAM to speed up the interrupt processing. The transfer processing from the serial flash memory area which is the program code of the exception processing vector table and the IRQ interrupt handler to the large-capacity on-chip RAM area, the clear to zero processing for the data section without initial data, and the initialization for the data section with initial data are executed by INTSCT function. The INITSCT function refer to the table data for section initialization defined in the file section.c and initialize each section. The assignment of program data is described in the linker script (linker_script.ld).

Table 5.7 list the Sections to be Used in the sample code. Figure 5.3 shows the Section Assignment for the initial condition of the sample code and the condition after using INTSCT function.

**Table 5.7 Memory Area to be used**

| Output section Name | Input section Name<br>Input Object Name | Description | Loading Area | Execution Area |
|---|---|---|---|---|
| LOAD_MODULE1 | VECTOR_TABLE | Exception processing vector table | FLASH | FLASH |
| LOAD_MODULE2 | */r_cpg/*.o<br>(.text .rodata .data) | Program code area for CPG | FLASH | LRAM |
| | */rza_io_regrw.o<br>(.text .rodata .data) | Program code area for function of I/O register access | | |
| | */hwsetup*.o<br>(.text .rodata .data) | HardwareSetup setting processing | | |
| LOAD_MODULE3 | */r_cpg/*.o (.bss) | Data area without initial value for CPG | - | LRAM |
| | */rza_io_regrw.o (.bss) | Data area without initial value for function of I/O register access | | |
| LOAD_MODULE4 | RESET_HANDLER | Program code area of reset handler processing | FLASH | FLASH |
| | INIT_SECTION<br>*/sections.o | Program code area of section initialization processing | | |
| .data | VECTOR_MIRROR_TABLE | Exception processing vector table | FLASH | LRAM |
| | */r_intc_*.o<br>(.text .rodata .data) | Program code area for INTC Driver | | |
| | IRQ_FIQ_HANDLER | IRQ/FIQ handler processing | | |
| .bss | none | none | - | LRAM |
| .uncached_RAM | */r_cache_*.o (.bss) | Data area without initial value for setting the L1 and L2 caches (see Note2) | - | LRAM |
| | UNCACHED_BSS | Data area without initial value for buffer. | | |
| .uncached_RAM2 | */r_cache_*.o<br>(.text .rodata .data) | Program code area for setting the L1 and L2 caches (see Note2) | FLASH | LRAM |
| | UNCACHED_DATA | Data area without initial value for buffer. | | |
| .mmu_page_table | none | MMU translation table area | - | LRAM |
| .stack | none | Stack area for system mode | - | LRAM |
| | | Stack area for IRQ mode | | |
| | | Stack area for FIQ mode | | |
| | | Stack area for supervisor(SVC) mode | | |
| | | Stack area for abort(ABT) mode | | |
| .text2 | * (.text .text.*) | Program code area for defaults | FLASH | FLASH |
| | * (.rodata .rodata.*) | Constant data area for defaults | | |
| .data2 | * (.data .data.*) | Data area with initial value for defaults | FLASH | LRAM |
| .bss2 | * (.bss .bss.*)<br>* (COMMON) | Data area without initial value for defaults | - | LRAM |
| .heap | none | Heap area | - | LRAM |

Notes: 1. "FLASH" and "LRAM" shown in "Loading Area" and "Execution Area" indicate the serial flash memory area and the large-capacity on-chip RAM area respectively.
2. This section should be placed in the cache-disabled area.

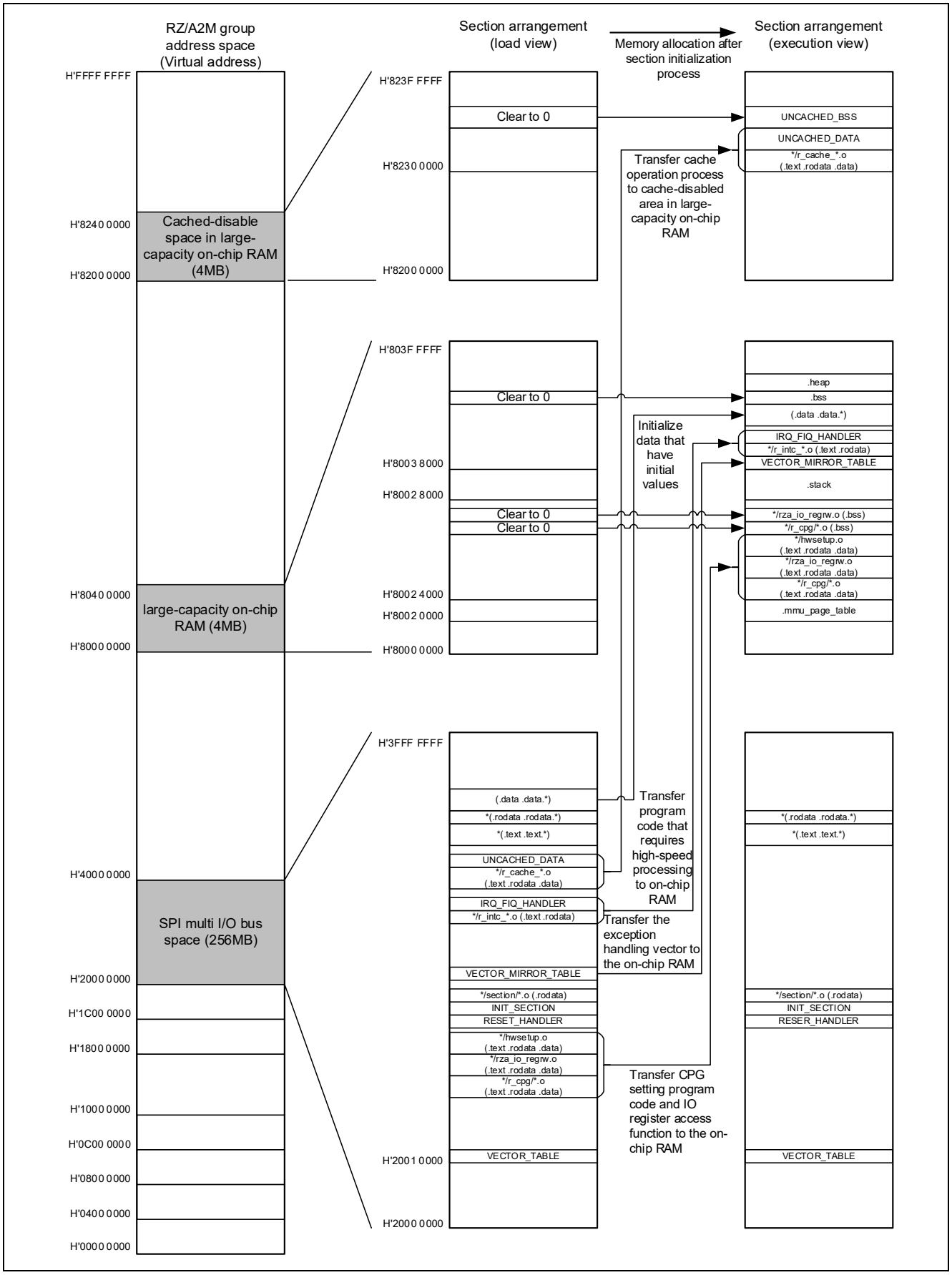
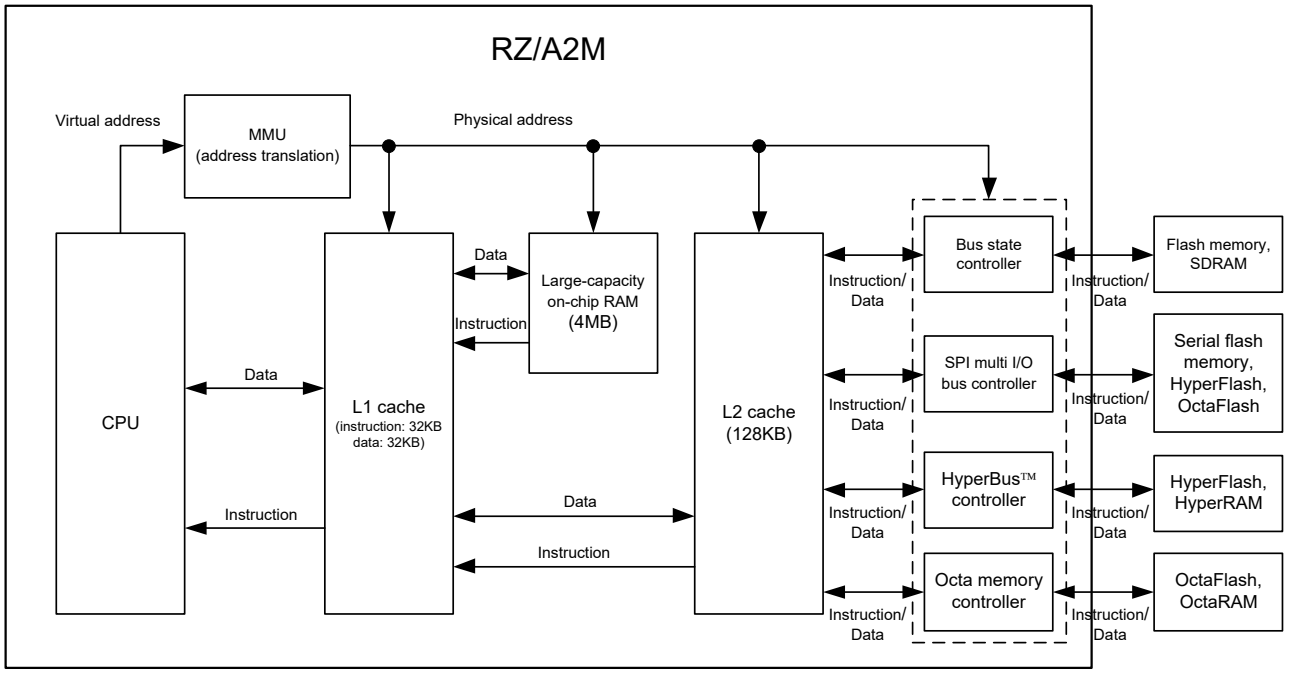



**Figure 5.6 Section Assignment**

### 5.2.6 L1 and L2 Cache Settings

RZ/A2M has two types of cache, L1 cache and L2 cache. L1 cache consists of 32Kbyte instruction cache and 32Kbyte data cache, while L2 cache consists of 128Kbyte cache which is commonly used for instruction and data.

Figure 5.7 shows the Block Diagram of L1 and L2 cache in memory Hierarchy.




Note: The L1 instruction cache is given virtual indexes and physical tags. The L1 data cache is given physical indexes and physical tags.
The figure above shows the memory hierarchy including the RZ/A2M caches. For details about the cache memory operation, refer to "ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition Issue C".

**Figure 5.7 Block Diagram of L1 and L2 cache in memory Hierarchy**

When accessing external memory space or on-chip large-capacity RAM from CPU, the virtual address is translated to the physical address using the 1st level descriptor for MMU translation table. As shown in Figure 5.3, the cache behavior of memory area whose base address and size are determined by the 1st level descriptor should also be specified by the 1st level descriptor. Please note that the size of area should be 1MB when "Section" is specified as 1st level descriptor type. For RZ/A2M, cache behavior of both external memory spaces (CS0 to CS5, SPI Multi I/O Bus space, HyperFlash/HyperRAM space and OctaFlash/OctaRAM space) and on-chip memory space (such as on-chip large-capacity RAM, on-chip peripheral module and reserved area) should be set up by MMU translation table.

Enabling/Disabling cache is controlled by the registers stated below:

- CP15 System Control Register (SCTLR) : I bit (b12), C bit (b2)
  I bit "0" denotes instruction cache is disabled, while "1" denotes instruction cache is enabled
  C bit "0" denotes data cache is disabled, while "1" denotes data cache is enabled
  Please note that SCTLR has M bit (b0) which enables/disables MMU ("1" denotes MMU is enabled)
- Control Register (reg1_control) : L2 Cache enable bit (b0)
  L2 Cache enable bit "0" denotes L2 Cache is disabled, while "1" denotes L2 Cache is enabled
  Please note that reg1_control is the register implemented in CoreLink level2 cache controller (L2C-310).

Notes: 1. Before enabling L1 instruction cache and L1 data cache, it is necessary to invalidate L1 instruction cache, L1 data cache and TLB with MMU, L1 instruction cache and L1 data cache disabled. Before enabling L2 cache, it is necessary to invalidate L2 cache as L1 cache do.
2. Direct Memory Access Controller (DMAC) shouldn't access to memory via L1 cache, while L2 cache is enabled. So, if both CPU and DMAC may access the shared area where L1 cache is enabled, user application should execute appropriate cache operation to care about cache coherency before kicking DMAC.

Figure 5.8 and 5.9 shows how L1 and L2 cache are initialized respectively.

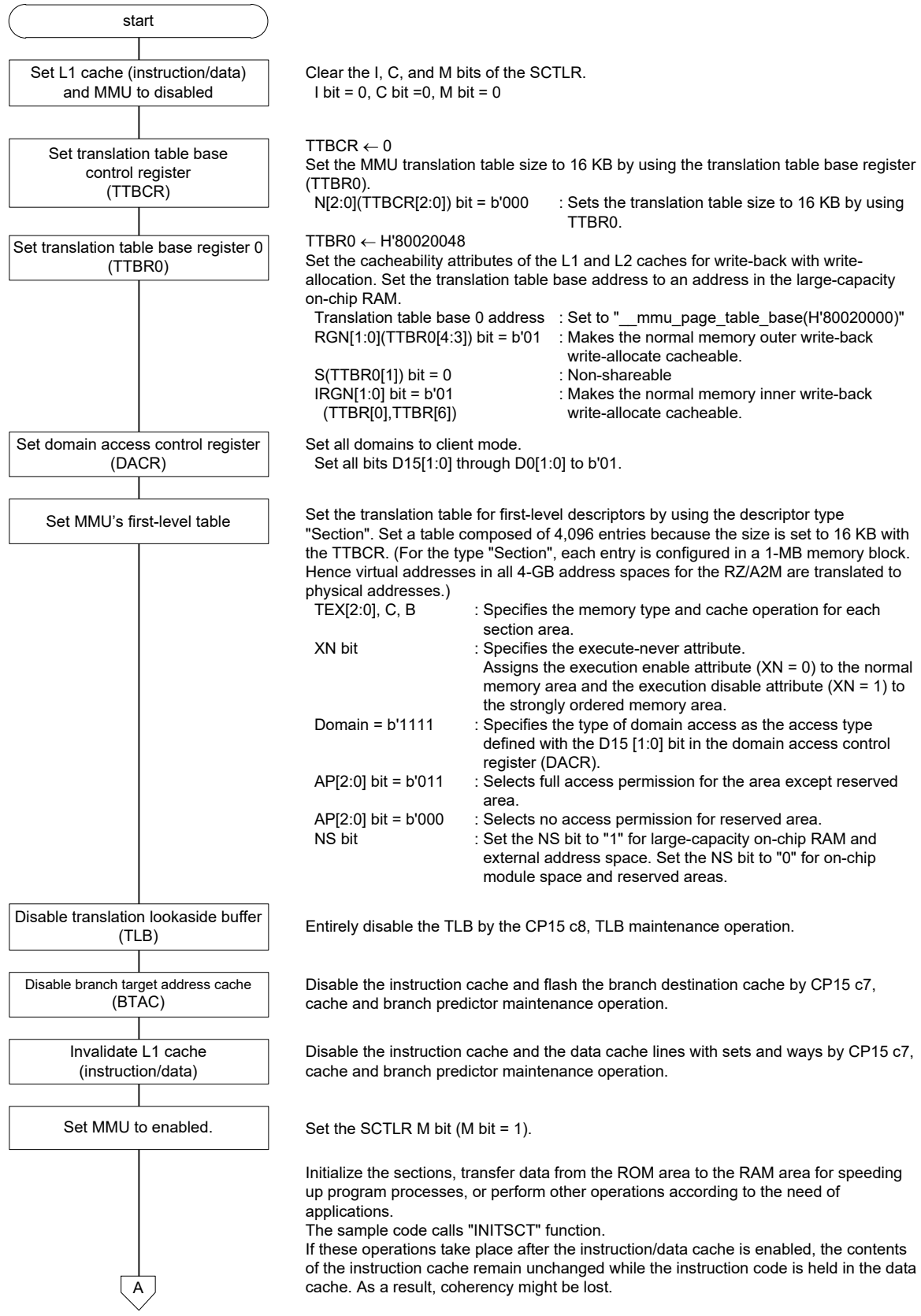



**Figure 5.8 Flow of L1 and L2 Cache Initial Setup (1/2)**

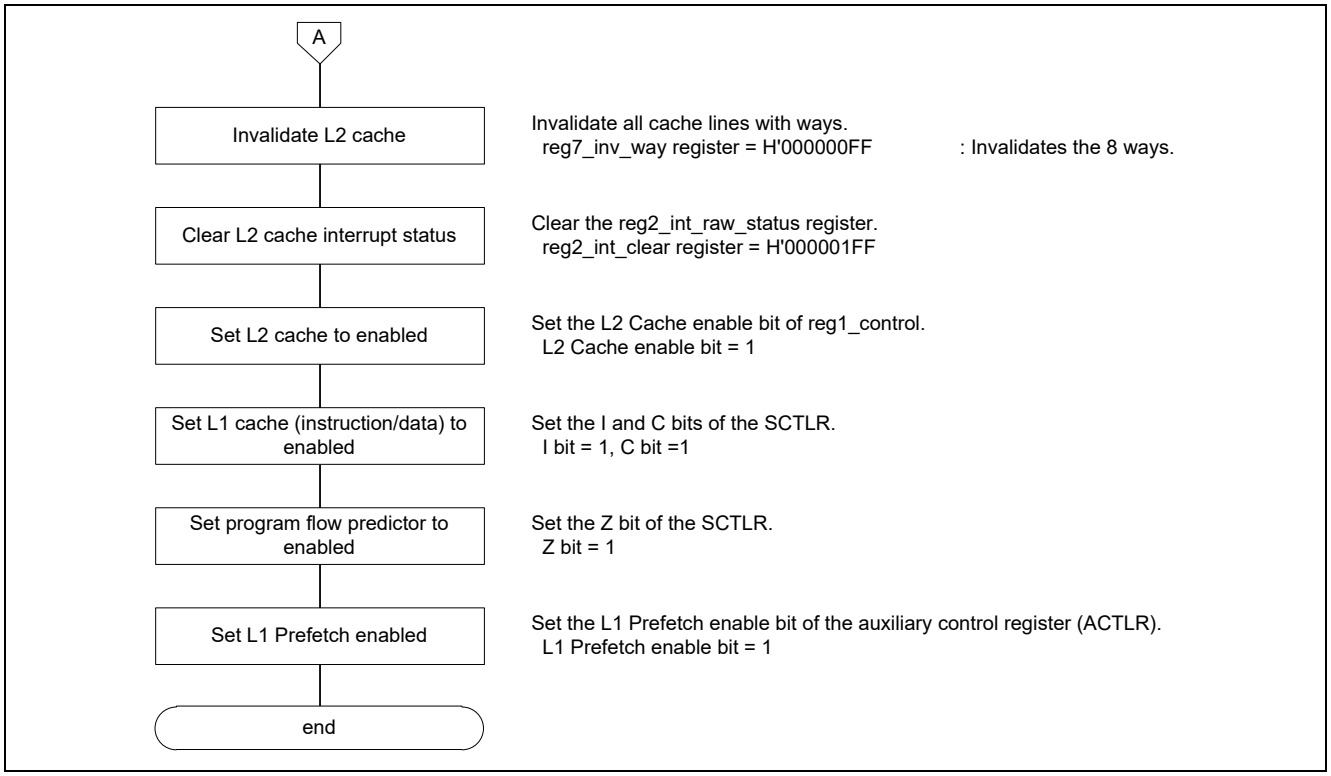



**Figure 5.9 Flow of L1 and L2 Cache Initial Setup (2/2)**

### 5.2.7 Exception Vector Table

On RZ/A2M, the 7 types of exception (i.e. reset exception, undefined instruction exception, software interrupt exception, prefetch abort exception, data abort exception, IRQ exception and FIQ exception) might occur.

In the sample program, the processing of application program sets VBAR to the address indicated by "__vector_mirror_table_base" after switching Low vector, then the exception processing vector table is assigned to the area from the start address "__vector_mirror_table_base" to the area of 32 bytes. And the branch instruction to each exception should be described in the exception vector table.

Figure 5.9 shows the exception vector table implemented in the sample program for your reference.

```
vector_table2:
  LDR pc, =reset_handler     ; +0x0000_0000 : Reset exception
  LDR pc, =undefined_handler ; +0x0000_0004 : Undefined instructions exception
  LDR pc, =svc_handler       ; +0x0000_0008 : Software interrupts exception
  LDR pc, =prefetch_handler  ; +0x0000_000C : Prefetch abort exception
  LDR pc, =abort_handler     ; +0x0000_0010 : Data abort exception
  LDR pc, =reserved_handler  ; +0x0000_0014 : Reserved
  LDR pc, =irq_handler       ; +0x0000_0018 : IRQ exception
  LDR pc, =fiq_handler       ; +0x0000_001C : FIQ exceptoin
```

**Figure 5.10 Example of Exception Vector Table Implementation**

### 5.2.8 Processing for RTC and USB unused channel

In the sample program, the processing for RTC and USB unused channel is implemented at the top of resetprg function to reduce power consumption. The macro definition listed in Table 5.8 determines if the processing for unused processing is carried out.

**Table 5.8 Macro Definition for Selecting RTC and USB Channel to be Used**

| Macro definition | Settings | Description |
|---|---|---|
| STARTUP_CFG_DISABLE_RTC0 | 0 | RTC channel 0 is used |
| | 1 (Default) | RTC channel 0 is NOT used |
| STARTUP_CFG_DISABLE_RTC1 | 0 | RTC channel 1 is used |
| | 1 (Default) | RTC channel 1 is NOT used |
| STARTUP_CFG_DISABLE_USB0 | 0 | USB channel 0 is used |
| | 1 (Default) | USB channel 0 is NOT used |
| STARTUP_CFG_DISABLE_USB1 | 0 | USB channel 1 is used |
| | 1 (Default) | USB channel 1 is NOT used |

Figure 5.11 shows the Processing Flow for RTC Unused Channel.

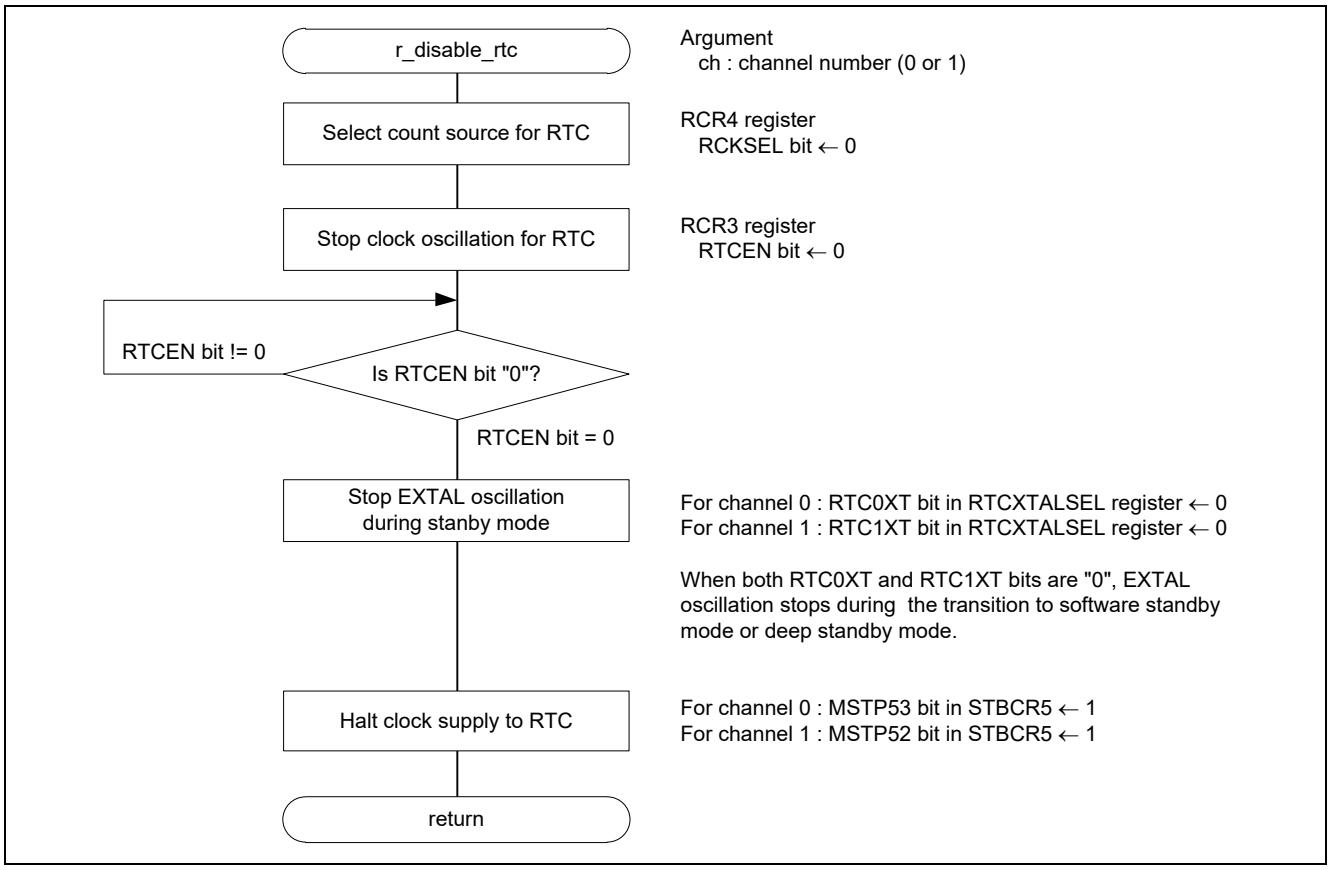



**Figure 5.11 Processing Flow for RTC Unused Channel**

Figure 5.12 shows the Processing Flow for USB Unused Channel.

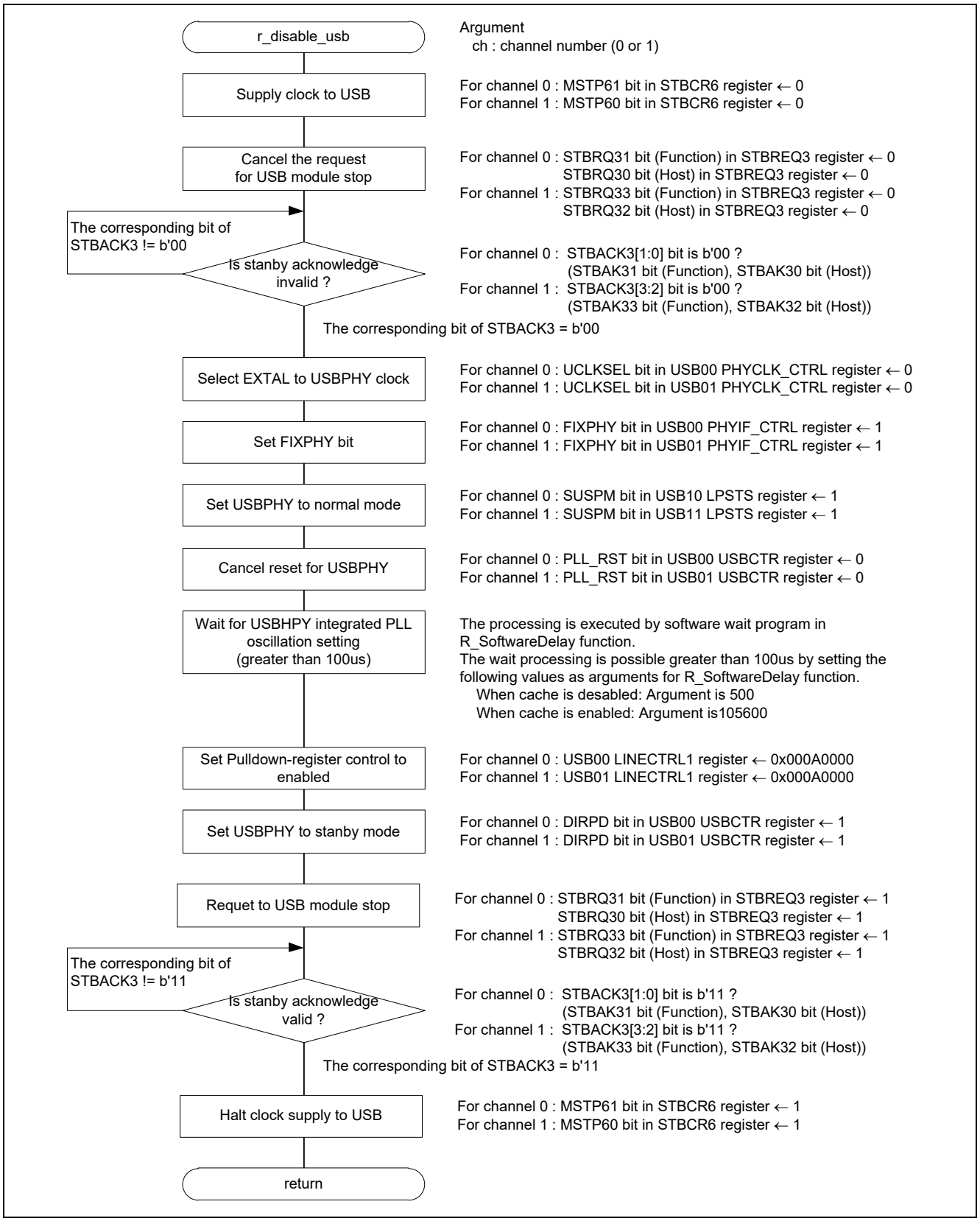



**Figure 5.12 Processing Flow for USB Unused Channel**

## 5.3 Interrupts Used

Table 5.9 shows the Interrupts Used in Sample Program.

**Table 5.9 Interrupts Used in Sample Program**

| Interrupt Source (Interrupt ID) | Priority | Processing Outline |
|---|---|---|
| OSTM0 (88) | 3 | Generate interrupt every 500ms |
| OSTM2 (90) | 30 | Generate interrupt every 1ms |
| RXI4 (322) | 30 | Generate SCIFA's RXI4 interrupt |
| TXI4 (323) | 30 | Generate SCIFA's TXI4 interrupt |

## 5.4 Fixed-Width Integers

Table 5.10 shows the Fixed-Width Integers Used in the Sample Program.

**Table 5.10 Fixed-Width Integers Used in the Sample Program**

| Symbol | Description |
|---|---|
| char_t | 8-bits character |
| bool_t | Boolean type. Allowable value is true (1) or false (0) |
| int_t | High-speed signed integer. In the sample code, its width is 32-bits |
| int8_t | Signed 8-bits integer (declared in the standard library stdint.h) |
| int16_t | Singed 16-bits integer (declared in the standard library stdint.h) |
| int32_t | Singed 32-bits integer (declared in the standard library stdint.h) |
| int64_t | Singed 64-bits integer (declared in the standard library stdint.h) |
| uint8_t | Unsigned 8-bits integer (declared in the standard library stdint.h) |
| uint16_t | Unsinged 16-bits integer (declared in the standard library stdint.h) |
| uint32_t | Unsinged 32-bits integer (declared in the standard library stdint.h) |
| uint64_t | Unsinged 64-bits integer (declared in the standard library stdint.h) |
| float32_t | 32-bit floating point |
| float64_t | 64-bit floating point |
| float128_t | 128-bit floating point |

## 5.5 Constants

Table 5.11 to Table 5.14 shows the Table 5.11 Constants Used in Sample Program.

**Table 5.11 Constants Used in Sample Program**

| Constants Name | Setting Value | Description |
|---|---|---|
| MAIN_PRV_LED_ON | (1) | LED is ON |
| MAIN_PRV_LED_OFF | (0) | LED is OFF |

**Table 5.12 Constants Used by GPIO Driver (1)**

| Constants Name | Setting Value of GPIO | | | | Description |
|---|---|---|---|---|---|
| | PMR | PDR | PODR | PSEL | |
| GPIO_FUNC_HIZ | 0 | b'00 | - | - | Set pin assignment to no use (Hi-Z) |
| GPIO_FUNC_IN | 0 | b'10 | 1 | - | Set pin assignment to general purpose input function |
| GPIO_FUNC_OUT_HIGH | 0 | b'11 | 0 | - | Set pin assignment to general purpose out function(High Level) |
| GPIO_FUNC_OUT_LOW | 0 | b'11 | - | - | Set pin assignment to general purpose out function(Low Level) |
| GPIO_FUNC_OUT_PERIPHERAL0 | 1 | b'00 | - | 0 | Set pin assignment to peripheral function 0 |
| GPIO_FUNC_OUT_PERIPHERAL1 | 1 | b'00 | - | 1 | Set pin assignment to peripheral function 1 |
| GPIO_FUNC_OUT_PERIPHERAL2 | 1 | b'00 | - | 2 | Set pin assignment to peripheral function 2 |
| GPIO_FUNC_OUT_PERIPHERAL3 | 1 | b'00 | - | 3 | Set pin assignment to peripheral function 3 |
| GPIO_FUNC_OUT_PERIPHERAL4 | 1 | b'00 | - | 4 | Set pin assignment to peripheral function 4 |
| GPIO_FUNC_OUT_PERIPHERAL5 | 1 | b'00 | - | 5 | Set pin assignment to peripheral function 5 |
| GPIO_FUNC_OUT_PERIPHERAL6 | 1 | b'00 | - | 6 | Set pin assignment to peripheral function 6 |
| GPIO_FUNC_OUT_PERIPHERAL7 | 1 | b'00 | - | 7 | Set pin assignment to peripheral function 7 |

**Table 5.13 Constants Used by GPIO Driver (2)**

| Constants Name | PmnPFS.ISEL Setting | Description |
|---|---|---|
| GPIO_TINT_DISABLE | 0 | Disable Pin interrupts function |
| GPIO_TINT_ENABLE | 1 | Enable Pin interrupts function |
| GPIO_TINT_RESERVED | - | Macro definition when using pin that can not use pin interrupts function |

**Table 5.14 Constants Used by GPIO Driver (3)**

| Constants Name | DSCR Setting | Description |
|---|---|---|
| GPIO_CURRENT_8mA | 0 | Set the pin drive strength to 8mA |
| GPIO_CURRENT_4mA | 1 | Set the pin drive strength to 4mA |
| GPIO_CURRENT_RESERVED | - | Macro definition when using pin that can not set drive strength |

## 5.6 Functions

The sample code consists of the following functions:

- API function for using peripheral functions
- User Defined Function depending on user system for which user need to prepare (Functions called by API function)
- Sample Function implemented for getting sample program to be worked as a reference

In the sample code, the IO register-write and register-read function is used when accessing peripheral IO registers in bit unit.

Table 5.15 lists the Sample Functions. Table 5.16 and Table 5.17 list the API Functions. Table 5.18 lists the User Defined Functions.

**Table 5.15 Sample Functions**

| Function Name | Description |
|---|---|
| reset_handler | Reset handler processing |
| resetprg | Initial setting of peripheral functions (Initial setting of INTC, L1 and L2 cache) |
| INITSCT | Initialize section |
| R_SC_HardwareSetup | Initial setting of peripheral functions |
| main | Main processing |
| irq_handler | IRQ handler processing |
| INTC_Handler_Interrupt | INTC interrupt handler processing |
| fiq_handler | FIQ handler processing |
| NMI_Handler_Interrupt | NMI interrupt handler processing |
| direct_open | Open the peripheral I/O driver function |
| direct_close | Close the peripheral I/O driver function |
| direct_read | Read the peripheral I/O driver function |
| direct_write | Write the peripheral I/O driver function |
| direct_control | Control the peripheral I/O driver function |
| direct_get_version | Get version information of the peripheral I/O driver |
| Sample_LED_Blink | OSTM channel 0 interrupt processing |
| r_disable_rtc | Setting process of unused channel of RTC |
| r_disable_usb | Setting process of unused channel of USB |

**Table 5.16 API Functions (1/2)**

| Function Name | Description |
|---|---|
| R_MMU_Init | Initial setting of the translation table of the MMU |
| R_MMU_WriteTbl | Configure MMU translation table |
| R_MMU_ReadTbl | Read MMU translation table |
| R_MMU_VAtoPA | Tranlate virtual address to physical address |
| R_CACHE_L1Init | Initialization of the L1 cache |
| R_CACHE_L1InstEnable | Setting the L1 instruction cache to enabled |
| R_CACHE_L1InstDisable | Setting the L1 instruction cache to disabled |
| R_CACHE_L1InstInvalidAll | Invalidate all L1 instruction cache lines |
| R_CACHE_L1DataEnable | Setting the L1 data cache to enabled |
| R_CACHE_L1DataDisable | Setting the L1 data cache to disabled |
| R_CACHE_L1DataInvalidAll | Invalidate all L1 data cache lines |
| R_CACHE_L1DataCleanAll | Clean all L1 data cache lines |
| R_CACHE_L1DataCleanInvalidAll | Clean&Invalidate all L1 data cache lines |
| R_CACHE_L1DataInvalidLine | Invalidate L1 data cache on cache lines (32 byte) basis |
| R_CACHE_L1DataCleanLine | Clean L1 data cache on cache lines (32 byte) basis |
| R_CACHE_L1DataCleanInvalidLine | Clean&Invalidate L1 data cache on cache lines (32 byte) basis |
| R_CACHE_L1BtacEnable | Setting the program predictor to enabled |
| R_CACHE_L1BtacDisable | Setting the program flow predictor to disabled |
| R_CACHE_L1BtacInvalidate | Invalidate all entries of the program flow predictor. |
| R_CACHE_L1PrefetchEnable | Setting the L1 prefetcher to enabled |
| R_CACHE_L1PrefetchDisable | Setting the L1 prefetcher to disabled |
| R_CACHE_L2Init | Initialization of the L2 cache |
| R_CACHE_L2CacheEnable | Setting the L2 cache to enabled |
| R_CACHE_L2CacheDisable | Setting the L2 cache to disabled |
| R_CACHE_L2InvalidAll | Invalidate all L2 caches |
| R_CACHE_L2CleanAll | Clean all L2 cache |
| R_CACHE_L2CleanInvalidAll | Clean&Invalidate all L2 cache |
| R_INTC_Init | Initialization of INTC |
| R_INTC_Enable | INTC interrupt enable |
| R_INTC_Disable | INTC interrupt disable |
| R_INTC_SetPriority | Setting for INTC interrupt priority level |
| R_INTC_SetMaskLevel | Setting for INTC interrupt mask level |
| R_INTC_GetMaskLevel | Obtaining INTC interrupt mask level |
| R_INTC_RegistIntFunc | Registration of INTC interrupt handler function |
| R_STB_StartModule | Canceling Module Standby |
| R_STB_StopModule | Transition to Module Standby |

**Table 5.17 API Functions (2/2)**

| Function Name | Description |
|---|---|
| R_CPG_InitialiseHwIf | Initialization process of CPG |
| R_GPIO_HWInitialise | Initialization process of GPIO |
| R_GPIO_InitByPinList | GPIO setting by pin list |
| R_GPIO_InitByTable | GPIO setting by GPIO configuration table |
| R_GPIO_PinWrite | Setting the pin output level |
| R_GPIO_PinRead | Getting the pin input level |
| RZA_IO_RegWrite_8 | I/O register write function (For I/O register accessible by 8 bits) |
| RZA_IO_RegWrite_16 | I/O register write function (For I/O register accessible by 16 bits) |
| RZA_IO_RegWrite_32 | I/O register write function (For I/O register accessible by 32 bits) |
| RZA_IO_RegRead_8 | I/O register read function (For I/O register accessible by 8 bits) |
| RZA_IO_RegRead_16 | I/O register read function (For I/O register accessible by 16 bits) |
| RZA_IO_RegRead_32 | I/O register read function (For I/O register accessible by 32 bits) |

**Table 5.18 User Defined Functions**

| Function Name | Description |
|---|---|
| Userdef_INTC_Pre_Interrupt | Callback function that before interrupt handler processing is executed |
| Userdef_INTC_Post_Interrupt | Callback function that after interrupt handler processing is executed |
| Userdef_INTC_UndefId | Processing when accepting unsupported interrupt ID of INTC interrupt |
| Userdef_INTC_UnregisteredID | Processing when INTC interrupt handler accepts unregistered ID |
| Userdef_PreHardwareSetup | Processing to be performed before executing hardware initialization |
| Userdef_PostHardwareSetup | Processing to be performed after executing hardware initialization |

## 5.7 Function Specification

This section describes the specification of function in the sample code.

| reset_handler | |
|---|---|
| **Overview** | Reset handler processing |
| **Syntax** | reset_handler FUNCTION {} |
| **Description** | This is the assembler function invoked just after reset is canceled. This function carries out the initial setup of stack pointer, peripherals needed minimally after reset is canceled and MMU and then, resetprg is called. |
| **Parameters** | None |
| **Return value** | None |

| resetprg | |
|---|---|
| **Overview** | Initial setting of peripheral functions (Initial setting of INTC, L1 and L2 cache) |
| **Syntax** | void resetprg(void) |
| **Description** | Initialize INTC and L1/L2 Cache and then, call the function main. |
| **Parameters** | None |
| **Return value** | None |

| INITSCT | |
|---|---|
| **Overview** | Initialize section |
| **Syntax** | INITSCT FUNCTION{} |
| **Description** | Initialize DATA and BSS section in accordance with the specified initialization table |
| **Parameters** | r0 : Pointer to initialization table for DATA section<br>r1 : Pointer to initialization table for BSS section |
| **Return value** | None |

| R_SC_HardwareSetup | |
|---|---|
| **Overview** | Initial setting of peripheral functions |
| **Syntax** | void R_SC_HardwareSetup(void) |
| **Description** | Initialize peripheral functions<br>This function calls Userdef_PreHardwareSetup function for release locked pin when resuming deep standby mode before doing initialization and calls Userdef_PostHardwareSetup function for enabling write operation in each retention RAM area after doing initialization. |
| **Parameters** | None |
| **Return value** | None |
| **Note** | DATA and BSS section shouldn't be initialized at the time this function is called. |

| main | |
|---|---|
| **Overview** | Main processing |
| **Syntax** | int_t main(void) |
| **Description** | This function displays the information about sample code to the terminal running on the host PC connected with RZ/A2M CPU board via serial interface and initializes GPIO connected to on-board LED.<br>Also, OSTM channel 0 is initialized and configured its counter so that the interrupt can be fired every 500ms and then, it is kicked. |
| **Parameters** | None |
| **Return value** | 0 |

| irq_handler | |
|---|---|
| **Overview** | IRQ handler processing |
| **Syntax** | irq_handler |
| **Description** | This function is the assembler function executed when the IRQ interrupt is generated. After saving LR_irq, SPSR_irq and general-purpose register to the stack and obtaining the INTC interrupt source ID, calls the INTC_Handler_Interrupt function<br>written in C language and executes the INTC interrupt handler processing which corresponds to the interrupt source ID.<br>After the INTC interrupt handler processing, restores the general-purpose register from the stack, and also restores LR_irq and SPSR_irq from the stack by executing the RFE instruction.<br>When the interrupt ID indicates 1022 and 1023, the interrupt handler processing is omitted.<br>This function executes the processing for saving and restoring and returns to the processing before the interrupt is generated. |
| **Parameters** | None |
| **Return value** | None |

| INTC_Handler_Interrupt | |
|---|---|
| **Overview** | INTC interrupt handler processing |
| **Syntax** | void INTC_Handler_Interrupt(uint32_t icciar) |
| **Description** | This is the INTC interrupt handler processing called by the irq_handler.<br>Executes the handler processing function (function registered in g_intc_func_table [int_id]) corresponding to the interrupt ID specified by icciar.<br>If the interrupt handler function is not registered in g_intc_func_table [int_id], execute the Userdef_INTC_UnregisteredID function.<br>Execute the Userdef_INTC_Pre_Interrupt function and Userdef_INTC_Post_Interrupt function before and after the interrupt handler processing function. If the interrupt ID is 512 or more, execute the Userdef_INTC_UndefId function.<br>If the interrupt handler function is not registered in g_intc_func_table [int_id], execute the Userdef_INTC_UnregisteredID function. Execute the Userdef_INTC_Pre_Interrupt function and Userdef_INTC_Post_Interrupt function before and after the interrupt handler processing function. If the interrupt ID is 512 or more, execute the Userdef_INTC_UndefId function. |
| **Parameters** | uint32_t icciar : Interrupt ID (0 to 511) |
| **Return value** | None |

| fiq_handler | |
|---|---|
| **Overview** | FIQ handler processing |
| **Syntax** | fiq_handler |
| **Description** | This function is the assembler function executed when the FIQ interrupt is generated.<br>After saving LR_irq, SPSR_irq and general-purpose register to the stack, calls the NMI_Handler_Interrupt function written in C language and executes the NMI handler processing.<br>After the NMI handler processing, restores the general-purpose register from the stack, and also restores LR_irq and SPSR_irq from the stack by executing the RFE instruction. Then returns to the processing before the interrupt was generated from the FIQ interrupt processing. |
| **Parameters** | None |
| **Return value** | None |

| NMI_Handler_Interrupt | |
|---|---|
| **Overview** | NMI interrupt handler processing |
| **Syntax** | void NMI_Handler_Interrupt(void) |
| **Description** | This is the NMI interrupt handler processing called by the fiq_handler.<br>Executes the handler processing function (registered function in g_intc_func_table[512]) when the interrupt ID is 512.<br>If the interrupt handler function is not registered in g_intc_func_table[512], executes the Userdef_INTC_UnregisteredID function. |
| **Parameters** | None |
| **Return value** | None |

| direct_open | |
|---|---|
| **Overview** | Open the peripheral I/O driver function |
| **Syntax** | int_t direct_open (char_t *p_driver_name, int_t param) |
| **Description** | Calls the open function of the peripheral IO driver with the configuration name specified by *p_driver_name. (In other words, calls the Open function related to the peripheral IO driver of the configuration name registered in gs_mount_table.)<br>Close, read, write, and control of the peripheral IO driver functions are performed using the handle number which is the return value of this function. In the sample code, this function is used to open the driver functions of CPG, OSTM0, SCIFA4, and GPIO, and initialize them. |
| **Parameters** | char_t *p_driver_name : Configuration name<br>int_t param : Open attribute |
| **Return value** | 0 or more : Success<br>-1 : Error |

| direct_close | |
|---|---|
| **Overview** | Close the peripheral I/O driver function |
| **Syntax** | int_t direct_close (int_t handle) |
| **Description** | Calls the close function of the peripheral IO driver function with the handle number. In the sample code, the driver functions of OSTM0 and SCIFA4 are closed. |
| **Parameters** | int_t handle : Handle number |
| **Return value** | 0 or more : Success<br>-1 : Error |

| direct_read | |
|---|---|
| **Overview** | Read the peripheral I/O driver function |
| **Syntax** | int_t direct_read (int handle, uint8_t *buff_ptr, uint32_t count) |
| **Description** | Calls the read function of the peripheral IO driver function with the handle number. When calling the read function, pass the pointer of the read data storage buffer specified by *buff_ptr and the number of bytes of the data to be read specified by count. The sample code calls the read function of OSTM0 and SCIFA4. |
| **Parameters** | int handle : Handle number<br>uint8_t *buff_ptr : Pointer of read data storage buffer<br>uint32_t count : Number of bytes of read data |
| **Return value** | 0 or more : Number of bytes of read data<br>-1 : Error |

| direct_write | |
|---|---|
| **Overview** | Write the peripheral I/O driver function |
| **Syntax** | int_t direct_write (int handle, uint8_t *buff_ptr, uint32_t count) |
| **Description** | Calls the write function of the peripheral IO driver function with the handle number. When calling the write function, pass the pointer of the write data storage buffer that specified by *buff_ptr and the number of bytes of the data to be write that specified by count.<br>The sample code calls the write function of SCIFA4. |
| **Parameters** | int handle : Handle number<br>uint8_t *buff_ptr : Pointer of write data storage buffer<br>uint32_t count : Number of bytes of write data |
| **Return value** | 0 or more : Number of bytes of write data<br>-1 : Error |

| direct_control | |
|---|---|
| **Overview** | Control the peripheral I/O driver function |
| **Syntax** | int_t direct_control (int handle, uint32_t ctlCode, void *pCtlStruct) |
| **Description** | Calls the control function of the peripheral IO driver function with the handle number. When calling the control function, pass the control code that specified by ctlCode and the pointer of a structure related to the control code that specified by *pCtlStruct.<br>The sample code calls the control function of "Start counting of OSTM0 timer" and "GPIO setting of LED connection port". |
| **Parameters** | int handle : Handle number<br>uint32_t ctlCode : Control code<br>void *pCtlStruct : Pointer of a structure related to the control code |
| **Return value** | 0 or more : Success<br>-1 : Error |

| direct_get_version | |
|---|---|
| **Overview** | Get version information of the peripheral I/O driver |
| **Syntax** | int_t direct_get_version(char *p_driver_name, st_ver_info_t *info) |
| **Description** | This function returns the version information of the peripheral IO driver corresponding to the configuration name specified by the argument * p_driver_name. |
| **Parameters** | char_t *p_driver_name : Configuration name<br>st_ver_info_t *info : Pointer to the variable that stores the version information |
| **Return value** | 0 or more : Success<br>-1 : Error |

| Sample_LED_Blink | |
|---|---|
| **Overview** | OSTM channel 0 interrupt processing |
| **Syntax** | void Sample_LED_Blink (uint32_t int_sense) |
| **Description** | This function is executed when the OSTM0 interrupt is accepted.<br>In this sample code, this function executes the processing to blink the LEDs on the RZ/A2M CPU board every 500ms. |
| **Parameters** | uint32_t int_sense : Detection method for interrupts<br>INTC_LEVEL_SENSITIVE : Level sense<br>INTC_EDGE_TRIGGER : Edge trigger |
| **Return value** | None |

| r_disable_rtc | |
|---|---|
| **Overview** | Setting process of unused channel of RTC |
| **Syntax** | static void r_disable_rtc (uint32_t ch) |
| **Description** | Perform processing for setting the unused channel of RTC described in "Figure 5.11 Processing Flow for RTC Unused ". |
| **Parameters** | uint32_t ch : RTC channel number (0 or 1) |
| **Return value** | None |

| r_disable_usb | |
|---|---|
| **Overview** | Setting process of unused channel of USB |
| **Syntax** | static void r_disable_usb (uint32_t ch) |
| **Description** | Perform processing for setting the unused channel of RTC described in "Figure 5.12 Processing Flow for USB Unused ". |
| **Parameters** | uint32_t ch : USB channel number (0 or 1) |
| **Return value** | None |

| R_MMU_Init | |
|---|---|
| **Overview** | Initial setting of the translation table of the MMU |
| **Syntax** | void R_MMU_Init(void) |
| **Description** | Initialize the translation table of the MMU.<br>Set the translation tables for first-level descriptors with the configuration table that specified by MMU_SC_TABLE. |
| **Parameters** | None |
| **Return value** | None |

| R_MMU_WriteTbl | | |
|---|---|---|
| **Overview** | Configure MMU translation table | |
| **Syntax** | void R_MMU_WriteTbl<br>( uint32_t vaddress, uint32_t paddress, uint32_t size, uint32_t entry ) | |
| **Description** | This function assigns physical address to the 1st level MMU translation table entry for virtual address space specified by the parameter vaddress, and set up its attribute. Table 5.5 and Table 5.6 show the 1st level descriptor that can be specified for translation table. | |
| **Parameters** | uint32_t vaddress | Starting address of virtual address area to be specified (Only b31-b20 is valid) |
| | uint32_t paddress | Starting address of physical address area to be assigned (Only b31-b20 is valid) |
| | uint32_t size | Number of entry to be specified (1~4096) |
| | uint32_t entry | Attribute to be specified (Only b19-b0 is valid) |
| **Return value** | MMU_SUCCESS | : Success |
| | MMU_ERR_OVERFLOW | : Size overflow |
| **Notes** | This function must be invoked after R_MMU_Init is terminated | |

| R_MMU_Read | |
|---|---|
| **Overview** | Read MMU translation table |
| **Syntax** | uint32_t R_MMU_ReadTbl<br>( uint32_t address ) |
| **Description** | Read the entry data (1st level MMU descriptors format) corresponding to the virtual address specified by the argument. |
| **Parameters** | uint32_t address : Virtual address (valid only for b31 - b20) |
| **Return value** | Entry data corresponding to the virtual address specified by the argument |
| **Notes** | This function must be invoked after R_MMU_Init is terminated |

| R_MMU_VAtoPA | |
|---|---|
| **Overview** | Translate virtual address to physical address |
| **Syntax** | Translate virtual address specified by argument to physical address. |
| **Description** | e_mmu_err_t R_MMU_VAtoPA<br>( uint32_t vaddress, uint32_t *paddress ) |
| **Parameters** | uint32_t vaddress : virtual address<br>uint32_t *paddress : storage area of physical address |
| **Return value** | MMU_SUCCESS : Success<br>MMU_ERR_TRANSLATION : Translation Error |
| **Notes** | This function must be invoked after R_MMU_Init is terminated |

| R_CACHE_L1Init | |
|---|---|
| **Overview** | Initialization of the L1 cache |
| **Syntax** | void R_CACHE_L1Init(void) |
| **Description** | In this sample code, the L1 cache, program flow predictor, and L1 prefetcher are set to enabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1InstEnable | |
|---|---|
| **Overview** | Setting the L1 instruction cache to enabled |
| **Syntax** | void R_CACHE_L1InstEnable(void) |
| **Description** | Sets the L1 instruction cache to enabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1InstDisable | |
|---|---|
| **Overview** | Setting the L1 instruction cache to disabled |
| **Syntax** | void R_CACHE_L1InstDisable(void) |
| **Description** | Sets the L1 instruction cache to disabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1InstInvalidAll | |
|---|---|
| **Overview** | Invalidate all L1 instruction cache lines |
| **Syntax** | void R_CACHE_L1InstInvalidAll(void) |
| **Description** | This function performs invalidate operation to all cache lines of the L1 instruction cache. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1DataEnable | |
|---|---|
| **Overview** | Setting the L1 data cache to enabled |
| **Syntax** | void R_CACHE_L1DataEnable(void) |
| **Description** | Sets the L1 data cache to enabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1DataDisable | |
|---|---|
| **Overview** | Setting the L1 data cache to disabled |
| **Syntax** | void R_CACHE_L1DataDisable(void) |
| **Description** | Sets the L1 data cache to disabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1DataInvalidAll | |
|---|---|
| **Overview** | Invalidate all L1 data cache lines |
| **Syntax** | void R_CACHE_L1DataInvalidAll(void) |
| **Description** | This function performs invalidate operation to all cache lines of the L1 data cache. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1DataCleanAll | |
|---|---|
| **Overview** | Clean all L1 data cache lines |
| **Syntax** | void R_CACHE_L1DataCleanAll(void) |
| **Description** | This function performs clean operation to all cache lines of the L1 data cache. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1DataCleanInvalidAll | |
|---|---|
| **Overview** | Clean&Invalidate all L1 data cache lines |
| **Syntax** | void R_CACHE_L1DataCleanInvalidAll(void) |
| **Description** | This function performs combination of clean and invalidate operations to all cache lines of the L1 data cache. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1DataInvalidLine | |
|---|---|
| **Overview** | Invalidate L1 data cache on cache lines (32 byte) basis |
| **Syntax** | e_err_code_t R_CACHE_L1DataInvalidLine(void* line_addr, uint32_t size) |
| **Description** | Invalidate the L1 data cache of each cache line from the address that specified by in_addr to the region they specified by size. |
| **Parameters** | void* line_addr : Start address of invalidate operation target region<br>uint32_t size : Number of bytes to invalidate |
| **Return value** | DRV_SUCCESS : Success<br>DRV_ERROR: Error |

| R_CACHE_L1DataCleanLine | |
|---|---|
| **Overview** | Clean L1 data cache on cache lines (32 byte) basis |
| **Syntax** | e_err_code_t R_CACHE_L1DataCleanLine(void* line_addr, uint32_t size) |
| **Description** | Clean the L1 data cache of each cache line from the address that specified by in_addr to the region they specified by size. |
| **Parameters** | void* line_addr : Start address of clean operation target region<br>uint32_t size : Number of bytes to clean |
| **Return value** | DRV_SUCCESS: Success<br>DRV_ERROR: Error |

| R_CACHE_L1DataCleanInvalidLine | |
|---|---|
| **Overview** | Clean&Invalidate L1 data cache on cache lines (32 byte) basis |
| **Syntax** | e_err_code_t R_CACHE_L1DataCleanInvalidLine(void* line_addr, uint32_t size) |
| **Description** | Clean&Invalidate the L1 data cache of each cache line from the address that specified by in_addr to the region they specified by size. |
| **Parameters** | void* line_addr : Start address of clean&invalidate operation target region<br>uint32_t size : Number of bytes to clean&invalidate |
| **Return value** | DRV_SUCCESS: Success<br>DRV_ERROR: Error |

| R_CACHE_L1BtacEnable | |
|---|---|
| **Overview** | Setting the program predictor to enabled |
| **Syntax** | void R_CACHE_L1BtacEnable(void) |
| **Description** | Sets the program flow predictor to enabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1BtacDisable | |
|---|---|
| **Overview** | Setting the program flow predictor to disabled |
| **Syntax** | void R_CACHE_L1BtacDisable(void) |
| **Description** | Sets the program flow predictor to disabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1BtacInvalidate | |
|---|---|
| **Overview** | Invalidate all entries of the program flow predictor. |
| **Syntax** | void R_CACHE_L1BtacInvalidate(void) |
| **Description** | This function performs invalidate operation to all entries of the program flow predictor. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1PrefetchEnable | |
|---|---|
| **Overview** | Setting the L1 prefetcher to enabled |
| **Syntax** | void R_CACHE_L1PrefetchEnable(void) |
| **Description** | Sets the L1 prefetcher to enabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L1PrefetchDisable | |
|---|---|
| **Overview** | Setting the L1 prefetcher to disabled |
| **Syntax** | void R_CACHE_L1PrefetchDisable(void) |
| **Description** | Sets the L1 prefetcher to disabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L2Init | |
|---|---|
| **Overview** | Initialization of the L2 cache |
| **Syntax** | void R_CACHE_L2Init(void) |
| **Description** | This function initializes the L2 cache in the following procedure.<br>1. Sets the L2 cache to disabled.<br>2. Invalidate all L2 cache lines.<br>3. Clear the all interrupt status of the L2 cache.<br>4. Sets the L2 cache to enabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L2CacheEnable | |
|---|---|
| **Overview** | Setting the L2 cache to enabled |
| **Syntax** | void R_CACHE_L2CacheEnable(void) |
| **Description** | Sets the L2 cache to enabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L2_CacheDisable | |
|---|---|
| **Overview** | Setting the L2 cache to disabled |
| **Syntax** | void R_CACHE_L2CacheDisable(void) |
| **Description** | Sets the L2 cache to disabled. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L2InvalidAll | |
|---|---|
| **Overview** | Invalidate all L2 caches |
| **Syntax** | void R_CACHE_L2InvalidAll(void) |
| **Description** | In this sample code, all L2 cache lines are invalidated by way-based operation. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L2CleanAll | |
|---|---|
| **Overview** | Clean all L2 cache |
| **Syntax** | void R_CACHE_L2CleanAll(void) |
| **Description** | In this sample code, all L2 cache lines are cleaned by way-based operation. |
| **Parameters** | None |
| **Return value** | None |

| R_CACHE_L2CleanInvalidAll | |
|---|---|
| **Overview** | Clean&Invalidate all L2 cache |
| **Syntax** | void R_CACHE_L2CleanInvalidAll(void) |
| **Description** | In this sample code, all L2 cache lines are clean&invalidated by way-based operation. |
| **Parameters** | None |
| **Return value** | None |

| R_INTC_Init | |
|---|---|
| **Overview** | Initialization of INTC |
| **Syntax** | e_r_drv_intc_err_t R_INTC_Init(void) |
| **Description** | Initialize the GIC interrupt controller.<br>The interrupts with the interrupt priority level from 0 to 30 are accepted. |
| **Parameters** | None |
| **Return value** | INTC_SUCCESS : Success |

| R_INTC_Enable | |
|---|---|
| **Overview** | INTC interrupt enable |
| **Syntax** | e_r_drv_intc_err_t R_INTC_Enable(e_r_drv_intc_intid_t int_id) |
| **Description** | Enables the interrupts of the ID specified by the int_id. |
| **Parameters** | e_r_drv_intc_intid_t int_id : Interrupt ID (0 to 511) |
| **Return value** | INTC_SUCCESS : Success<br>INTC_ERR_INVALID_ID : Invalid interrupt ID |

| R_INTC_Disable | |
|---|---|
| **Overview** | INTC interrupt disable |
| **Syntax** | e_r_drv_intc_err_t R_INTC_Disable(e_r_drv_intc_intid_t int_id) |
| **Description** | Disables the interrupts of the ID specified by the int_id. |
| **Parameters** | e_r_drv_intc_intid_t int_id : Interrupt ID (0 to 511) |
| **Return value** | INTC_SUCCESS : Success<br>INTC_ERR_INVALID_ID : Invalid interrupt ID |

| R_INTC_SetPriority | |
|---|---|
| **Overview** | Setting for INTC interrupt priority level |
| **Syntax** | e_r_drv_intc_err_t R_INTC_SetPriority(e_r_drv_intc_intid_t int_id, intc_priority_t priority) |
| **Description** | Sets the interrupt priority level for the ID specified by the int_id to the priority level specified by the priority. |
| **Parameters** | e_r_drv_intc_intid_t int_id : Interrupt ID (0 to 511)<br>intc_priority_t priority : Interrupt priority level (0 to 31 (0 : Highest level, 31 : Lowest level )) |
| **Return value** | INTC_SUCCESS : Success<br>INTC_ERR_INVALID_ID : Invalid interrupt ID<br>INTC_ERR_INVALID_PRIORITY : Invalid interrupt priority level |
| **Notes** | If 31 (INTC_PRIORITY_LOWEST or INTC_PRIORITY_31) is specified as the interrupt priority level for this function, no interrupt is generated for the specified ID. |

| R_INTC_SetMaskLevel | |
|---|---|
| **Overview** | Setting for INTC interrupt mask level |
| **Syntax** | e_r_drv_intc_err_t R_INTC_SetMaskLevel(e_r_drv_intc_priority_t mask_level) |
| **Description** | Sets the interrupt mask level specified by the mask_level. |
| **Parameters** | e_r_drv_intc_priority_t mask_level : Interrupt priority mask level (0 to 31 (0 : Highest level, 31 : Lowest level )) |
| **Return value** | INTC_SUCCESS : Success<br>INTC_ERR_INVALID_PRIORITY : Invalid interrupt priority mask level |
| **Notes** | Masks the generation of an interrupt whose priority level is lower than the mask level specified for this function. (Interrupts are masked if the priority level values are equal or higher) |

| R_INTC_GetMaskLevel | |
|---|---|
| **Overview** | Obtaining INTC interrupt mask level |
| **Syntax** | e_r_drv_intc_err_t R_INTC_GetMaskLevel(e_r_drv_intc_priority_t *mask_level) |
| **Description** | Obtains the setting value of the interrupt mask level and returns the obtained value to the mask_level. |
| **Parameters** | e_r_drv_intc_priority_t *mask_level : Pointer to the variable that stores the interrupt mask level (0 to 31 (0 : Highest level, 31 : Lowest level )) |
| **Return value** | INTC_SUCCESS : Success |

| R_INTC_RegistIntFunc | |
|---|---|
| **Overview** | Registration of INTC interrupt handler function |
| **Syntax** | e_r_drv_intc_err_t R_INTC_RegistIntFunc(e_r_drv_intc_intid_t int_id, void (* func)(uint32_t int_sense)) |
| **Description** | Register the function specified by func in the interrupt handler function table as the component specified by int_id. (Interrupt handler is registered to g_intc_func_table[int_id] by this function)<br>When 512 is specified as int_id, this function registers the handler of NMI. (NMI handler is also registered to g_intc_func_table[int_id] by this function)<br>If an invalid interrupt ID outside the range is specified, this function does not register an interrupt handler. |
| **Parameters** | e_r_drv_intc_intid_t int_id : Interrupt ID (0 to 512)<br>void (* func) (uint32_t int_sense) : Pointer to interrupt handler function (A function with "uint32_t int_sense" as an argument is required) |
| **Return value** | INTC_SUCCESS : Success<br>INTC_ERR_INVALID : Error |

| R_STB_StartModule | |
|---|---|
| **Overview** | Canceling Module Standby |
| **Syntax** | int_t R_STB_StartModule(e_stb_module_t module) |
| **Description** | Cancels the module standby state of the module specified by the argument "module".<br>By calling this function, clock is supplied to the specified module. |
| **Parameters** | e_stb_module_t module : Number of the module |
| **Return value** | DRV_SUCCESS : Success<br>DRV_ERROR : Error |

| R_STB_StopModule | |
|---|---|
| **Overview** | Transition to Module Standby |
| **Syntax** | int_t R_STB_StopModule(e_stb_module_t module) |
| **Description** | Transitions the module that specified by the argument "module" to the module standby state.<br>By calling this function, halts clock supply to the specified module. |
| **Parameters** | e_stb_module_t module : Number of the module |
| **Return value** | DRV_SUCCESS : Success<br>DRV_ERROR : Error |

| R_CPG_InitialiseHwIf | |
|---|---|
| **Overview** | Initialization process of CPG |
| **Syntax** | int_t R_CPG_InitialiseHwIf( void ) |
| **Description** | This function sets CPG registers (FRQCR, CKIOSEL, SCLKSEL) by using CPG configuration data of r_cpg_drv_sc_cfg.h.<br>In the sample code, this function is called via the direct_open function and becomes the operating frequency described in "Table 2.1 Operation Conformation Condition (1/2)". |
| **Parameters** | None |
| **Return value** | DRV_SUCCESS : Success<br>DRV_ERROR : CPG configuration data of r_cpg_drv_sc_cfg.h is invalid. |

| R_GPIO_HWInitialise | |
|---|---|
| **Overview** | Initialization process of GPIO |
| **Syntax** | int_t R_GPIO_HWInitialise(void) |
| **Description** | Initialize the GPIO driver. Also, after initializing the GPIO driver, this function refers to the data of GPIO_SC_TABLE_INIT[] in r_gpio_drv_sc_cfg.h and execute the pin setting processing.<br>In the sample code, this function is called by the direct_open function when opening the GPIO driver function, but since the element of GPIO_SC_TABLE_INIT[] is not defined, the setting processing of the pins by this function is not performed. |
| **Parameters** | None |
| **Return value** | DRV_SUCCESS : Success |
| **Notes** | The GPIO driver manages the pin to which the pin function is assigned by this function as "in use" state.<br>In the GPIO driver, pins that are managed as "in use" state can not be reset to other pin functions. To reconfigure the pin function, set the pin to "unused" state by using the R_GPIO_ClearByPinList function beforehand. |

| R_GPIO_InitByPinList | |
|---|---|
| **Overview** | GPIO setting by pin list |
| **Syntax** | e_r_drv_gpio_err_t R_GPIO_PinInitByPinList(const r_gpio_port_pin_t * p_pin_list, uint32_t count) |
| **Description** | Configure GPIO with pin list.<br>In the sample code, this function is called by the direct_control function, and P6_0 is set to be able to turn on and off the LED on the RZ/A2M CPU board. |
| **Parameters** | const r_gpio_port_pin_t * p_pin_list : Pointer of pin list<br>uint32_t count : Number of pin to set |
| **Return value** | GPIO_SUCCESS : Success<br>Other than GPIO_SUCCESS : Error |
| **Notes** | The GPIO driver manages the pin to which the pin function is assigned by this function as "in use" state.<br>In the GPIO driver, pins that are managed as "in use" state can not be reset to other pin functions. To reconfigure the pin function, set the pin to "unused" state by using the R_GPIO_ClearByPinList function beforehand. |

| R_GPIO_InitByTable | |
|---|---|
| **Overview** | GPIO setting by GPIO configuration table |
| **Syntax** | e_r_drv_gpio_err_t R_GPIO_PinInitByTable(const st_r_drv_gpio_sc_config_t * p_table, uint32_t count) |
| **Description** | Specify GPIO configuration table with argument p_table and perform pin setting processing.<br>In the sample code, the this function is called when openning the SCIFA driver, and the P9_0 and P9_1 pins are set to TxD4 and RxD4 functions for SCIFA. |
| **Parameters** | const st_r_drv_gpio_sc_config_t * p_table : Pointer of GPIO configuration table<br>uint32_t count : Number of pins to set |
| **Return value** | GPIO_SUCCESS : Success<br>Other than GPIO_SUCCESS : Error |
| **Notes** | The GPIO driver manages the pin to which the pin function is assigned by this function as "in use" state.<br>In the GPIO driver, pins that are managed as "in use" state can not be reset to other pin functions. To reconfigure the pin function, set the pin to "unused" state by using the R_GPIO_ClearByPinList function beforehand. |

| R_GPIO_PinWrite | |
|---|---|
| **Overview** | Setting the pin output level |
| **Syntax** | e_r_drv_gpio_err_t R_GPIO_PinWrite(e_r_drv_gpio_pin_t pin, e_r_drv_gpio_level_t level) |
| **Description** | When the pin specified by the argument "pin" is set to general I/O(OUTPUT),<br>set the output level of the pin specified by "pin" to the value specified by "level". |
| **Parameters** | e_r_drv_gpio_pin_t pin : Pin number<br>e_r_drv_gpio_level_t level : Output level<br>GPIO_LEVEL_LOW : 1<br>GPIO_LEVEL_HIGH : 2 |
| **Return value** | GPIO_SUCCESS : Success<br>Other than GPIO_SUCCESS : Error |

| R_GPIO_PinRead | |
|---|---|
| **Overview** | Getting the pin input level |
| **Syntax** | e_r_drv_gpio_err_t R_GPIO_PinRead(e_r_drv_gpio_pin_t pin, e_r_drv_gpio_level_t *p_level) |
| **Description** | When the pin specified by the argument "pin" is set to general I/O(INPUT),<br>acquires the input level of the pin specified as "pin" and stores it in p_level. |
| **Parameters** | e_r_drv_gpio_pin_t pin : Pin number<br>e_r_drv_gpio_level_t *p_level : Pointer of structure that stores input level<br>GPIO_LEVEL_LOW : 1<br>GPIO_LEVEL_HIGH : 2 |
| **Return value** | GPIO_SUCCESS : Success<br>Other than GPIO_SUCCESS : Error |

| RZA_IO_RegWrite_8 | |
|---|---|
| **Overview** | I/O register write function (For I/O register accessible by 8 bits) |
| **Syntax** | void RZA_IO_RegWrite_8(volatile uint8_t * ioreg, uint8_t write_value,<br>uint8_t shift, uint32_t mask) |
| **Description** | Executes write processing for the on-chip peripheral I/O register which can be accessed by 8 bits. |
| **Parameters** | volatile uint8_t *ioreg : I/O register to be written<br>The I/O register defined in the directory under the iodefines should be specified.<br>uint8_t write_value : Write value to the I/O register<br>uint8_t shift : Left shift value to the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a shift value in the directory under the iobitmasks. The shift value to the defined I/O register should be specified.<br>uint32_t mask : Mask value of the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a mask value in the directory under the iobitmasks. The mask value of the defined I/O register should be specified. When "IOREG_NONMASK_ACCESS" is specified, write to the I/O register directly without performing mask processing (read-modify-write processing). |
| **Return value** | None |
| **Notes** | When transferring this function to the RAM area, do not call this function before the section is transferred from the ROM area to the RAM area. |

| RZA_IO_RegWrite_16 | |
|---|---|
| **Overview** | I/O register write function (For I/O register accessible by 16 bits) |
| **Syntax** | void RZA_IO_RegWrite_16(volatile uint16_t * ioreg, uint16_t write_value,<br>uint16_t shift, uint32_t mask) |
| **Description** | Executes write processing for the on-chip peripheral I/O register which can be accessed by 16 bits. |
| **Parameters** | volatile uint16_t *ioreg : I/O register to be written<br>The I/O register defined in the directory under the iodefines should be specified.<br>uint16_t write_value : Write value to the I/O register<br>uint16_t shift : Left shift value to the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a shift value in the directory under the iobitmasks. The shift value to the defined I/O register should be specified.<br>uint32_t mask : Mask value of the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a mask value in the directory under the iobitmasks. The mask value of the defined I/O register should be specified. When "IOREG_NONMASK_ACCESS" is specified, write to the I/O register directly without performing mask processing (read-modify-write processing). |
| **Return value** | None |
| **Notes** | When transferring this function to the RAM area, do not call this function before the section is transferred from the ROM area to the RAM area. |

| RZA_IO_RegWrite_32 | |
|---|---|
| **Overview** | I/O register write function (For I/O register accessible by 32 bits) |
| **Syntax** | void RZA_IO_RegWrite_32(volatile uint32_t * ioreg, uint32_t write_value,<br>uint32_t shift, uint32_t mask) |
| **Description** | Executes write processing for the on-chip peripheral I/O register which can be accessed by 32 bits. |
| **Parameters** | volatile uint32_t *ioreg : I/O register to be written<br>The I/O register defined in the directory under the iodefines should be specified.<br>uint32_t write_value : Write value to the I/O register<br>uint32_t shift : Left shift value to the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a shift value in the directory under the iobitmasks. The shift value to the defined I/O register should be specified.<br>uint32_t mask : Mask value of the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a mask value in the directory under the iobitmasks. The mask value of the defined I/O register should be specified. When "IOREG_NONMASK_ACCESS" is specified, write to the I/O register directly without performing mask processing (read-modify-write processing). |
| **Return value** | None |
| **Notes** | When transferring this function to the RAM area, do not call this function before the section is transferred from the ROM area to the RAM area. |

| RZA_IO_RegRead_8 | |
|---|---|
| **Overview** | I/O register read function (For I/O register accessible by 8 bits) |
| **Syntax** | uint8_t RZA_IO_RegRead_8(volatile uint8_t * ioreg, uint8_t shift, uint32_t mask) |
| **Description** | Executes read processing for the on-chip peripheral I/O register which can be accessed by 8 bits. |
| **Parameters** | volatile uint8_t *ioreg : I/O register to be read<br>The I/O register defined in the directory under the iodefines should be specified. |
| | uint8_t shift : Left shift value to the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a shift value in the directory under the iobitmasks. The shift value to the defined I/O register should be specified. |
| | uint32_t mask : Mask value of the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a mask value in the directory under the iobitmasks. The mask value of the defined I/O register should be specified. When "IOREG_NONMASK_ACCESS" is specified, the I/O register is directly read without mask processing. |
| **Return value** | Read value of the specified I/O register |
| **Notes** | When transferring this function to the RAM area, do not call this function before the section is transferred from the ROM area to the RAM area. |

| RZA_IO_RegRead_16 | |
|---|---|
| **Overview** | I/O register read function (For I/O register accessible by 16 bits) |
| **Syntax** | uint16_t RZA_IO_RegRead_16(volatile uint16_t * ioreg, uint16_t shift, uint32_t mask) |
| **Description** | Executes read processing for the on-chip peripheral I/O register which can be accessed by 16 bits. |
| **Parameters** | volatile uint16_t *ioreg : I/O register to be read<br>The I/O register defined in the directory under the iodefines should be specified.<br>uint16_t shift : Left shift value to the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a shift value in the directory under the iobitmasks. The shift value to the defined I/O register should be specified.<br>uint32_t mask : Mask value of the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a mask value in the directory under the iobitmasks. The mask value of the defined I/O register should be specified. When "IOREG_NONMASK_ACCESS" is specified, the I/O register is directly read without mask processing. |
| **Return value** | Read value of the specified I/O register |
| **Notes** | When transferring this function to the RAM area, do not call this function before the section is transferred from the ROM area to the RAM area. |

| RZA_IO_RegRead_32 | |
|---|---|
| **Overview** | I/O register read function (For I/O register accessible by 32 bits) |
| **Syntax** | uint32_t RZA_IO_RegRead_32(volatile uint32_t * ioreg, uint32_t shift, uint32_t mask) |
| **Description** | Executes read processing for the on-chip peripheral I/O register which can be accessed by 32 bits. |
| **Parameters** | volatile uint32_t *ioreg : I/O register to be read<br>The I/O register defined in the directory under the iodefines should be specified.<br>uint32_t shift : Left shift value to the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a shift value in the directory under the iobitmasks. The shift value to the defined I/O register should be specified.<br>uint32_t mask : Mask value of the bit for I/O register<br>Defines the value to access to the bit position within the I/O register as a mask value in the directory under the iobitmasks. The mask value of the defined I/O register should be specified. When "IOREG_NONMASK_ACCESS" is specified, the I/O register is directly read without mask processing. |
| **Return value** | Read value of the specified I/O register |
| **Notes** | When transferring this function to the RAM area, do not call this function before the section is transferred from the ROM area to the RAM area. |

| Userdef_INTC_Pre_Interrupt | |
|---|---|
| **Overview** | Callback function that before interrupt handler processing is executed |
| **Syntax** | e_r_drv_intc_err_t Userdef_INTC_Pre_Interrupt(e_r_drv_intc_intid_t int_id) |
| **Description** | This is the user-defined function called by the INTC_Handler_Interrupt.<br>Implement the necessary processing before calling each interrupt handler.<br>In the sample code implementation, this function returns without doing anything. |
| **Parameters** | e_r_drv_intc_intid_t int_id : Interrupt ID (0 to 511) |
| **Return value** | INTC_SUCCESS : Success |

| Userdef_INTC_Post_Interrupt | |
|---|---|
| **Overview** | Callback function that after interrupt handler processing is executed |
| **Syntax** | e_r_drv_intc_err_t Userdef_INTC_Post_Interrupt(e_r_drv_intc_intid_t int_id) |
| **Description** | This is the user-defined function called by the INTC_Handler_Interrupt.<br>Implement the necessary processing after returning from each interrupt handler.<br>In the sample code implementation, this function returns without doing anything. |
| **Parameters** | e_r_drv_intc_intid_t int_id : Interrupt ID (0 to 511) |
| **Return value** | INTC_SUCCESS : Success |

| Userdef_INTC_UndefId | |
|---|---|
| **Overview** | Processing when accepting unsupported interrupt ID of INTC interrupt |
| **Syntax** | e_r_drv_intc_err_t Userdef_INTC_UndefId(e_r_drv_intc_intid_t int_id) |
| **Description** | This is the user-defined function called by the INTC_Handler_Interrupt.<br>Implement the function to be executed when the interrupts which have unsupported interrupt IDs with 512 and above are accepted. |
| **Parameters** | e_r_drv_intc_intid_t int_id : Interrupt ID (512 to 1021) |
| **Return value** | INTC_SUCCESS : Success |

| Userdef_INTC_UnregisteredID | |
|---|---|
| **Overview** | Processing when INTC interrupt handler accepts unregistered ID |
| **Syntax** | e_r_drv_intc_err_t Userdef_INTC_UnregisteredID(e_r_drv_intc_intid_t int_id) |
| **Description** | This is the user-defined function called by the INTC_Handler_Interrupt.<br>Implement the function to be executed when the interrupt ID with an unregistered user interrupt handler function is accepted.<br>This sample code disables the IRQ interrupt and executes the processing for infinite loop. |
| **Parameters** | e_r_drv_intc_intid_t int_id : Interrupt ID (0 to 511) |
| **Return value** | INTC_SUCCESS : Success |

| Userdef_PreHardwareSetup | |
|---|---|
| **Overview** | Processing to be performed before executing hardware initialization |
| **Syntax** | void Userdef_PreHardwareSetup (void) |
| **Description** | This is user-defined function for implementing additional process before executing hardware initialization and called in beginning of R_SC_HardwareSetup function.<br>In application program, clear IOKEEP bit to release retain of pin states after canceling deep standby mode. |
| **Parameters** | None |
| **Return value** | None |

| Userdef_PostHardwareSetup | |
|---|---|
| **Overview** | Processing to be performed after executing hardware initialization |
| **Syntax** | void Userdef_PostHardwareSetup (void) |
| **Description** | This is user-defined function for implementing additional process before executing hardware initialization and called in end of R_SC_HardwareSetup function.<br>In application program, enables writing to page in each on-chip data-retention RAM. |
| **Parameters** | None |
| **Return value** | None |

## 5.8 Flowcharts

### 5.8.1 Reset Handler Processing

Figure 5.13 shows the flowchart of Reset Handler Processing.

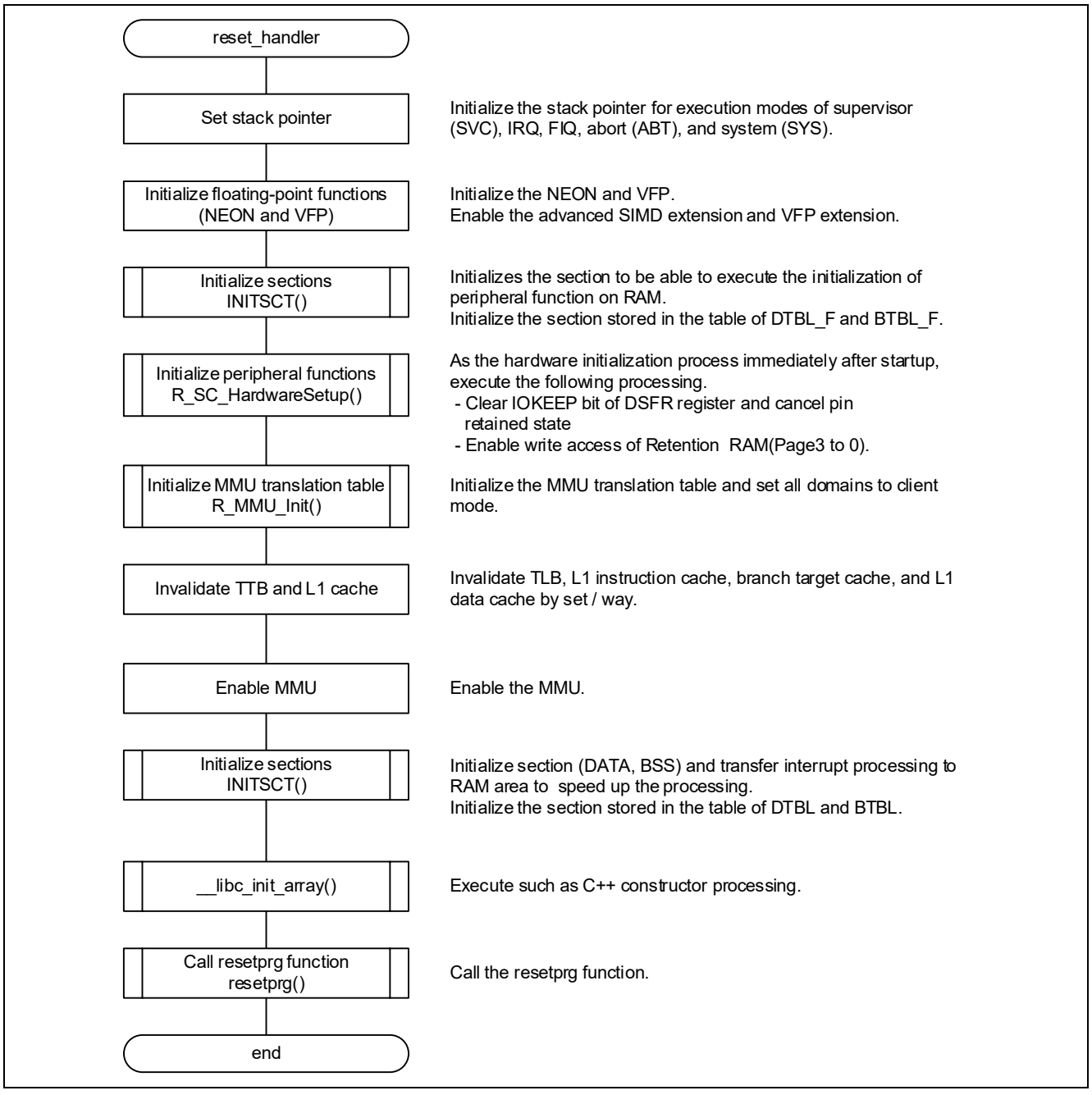
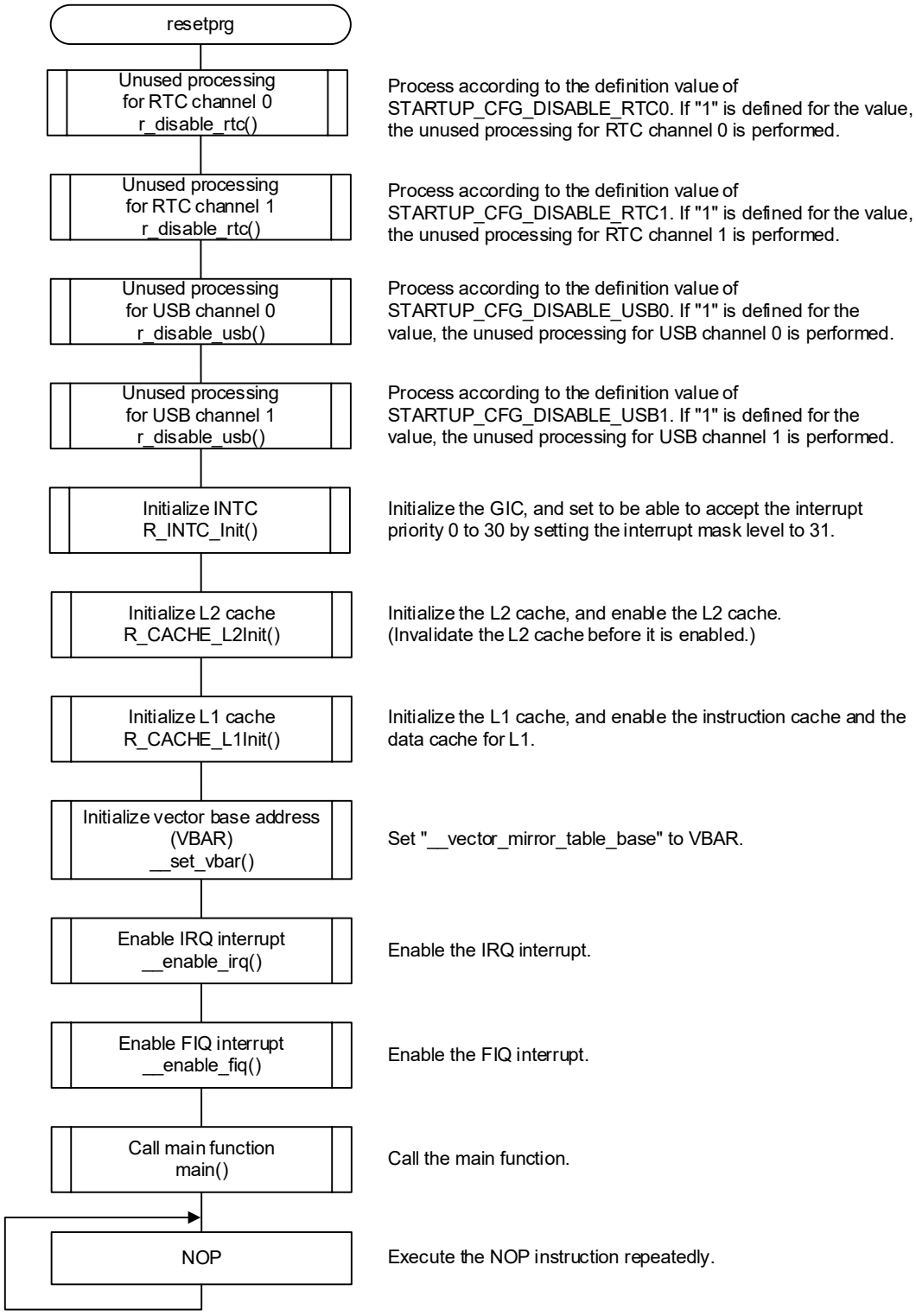



**Figure 5.13 Reset Handler Processing**

### 5.8.2 resetprg Function

Figure 5.14 shows the flowchart of resetprg Function.




Note; Before enabling L1 instruction cache and L1 data cache, it is necessary to invalidate L1 instruction cache, L1 data cache and TLB with MMU, L1 instruction cache and L1 data cache disabled. Before enabling L2 cache, it is necessary to invalidate L2 cache as L1 cache do.

**Figure 5.14 resetprg Function**

### 5.8.3 Main Processing

Figure 5.15 and Figure 5.16 show the flowchart of Main Processing.

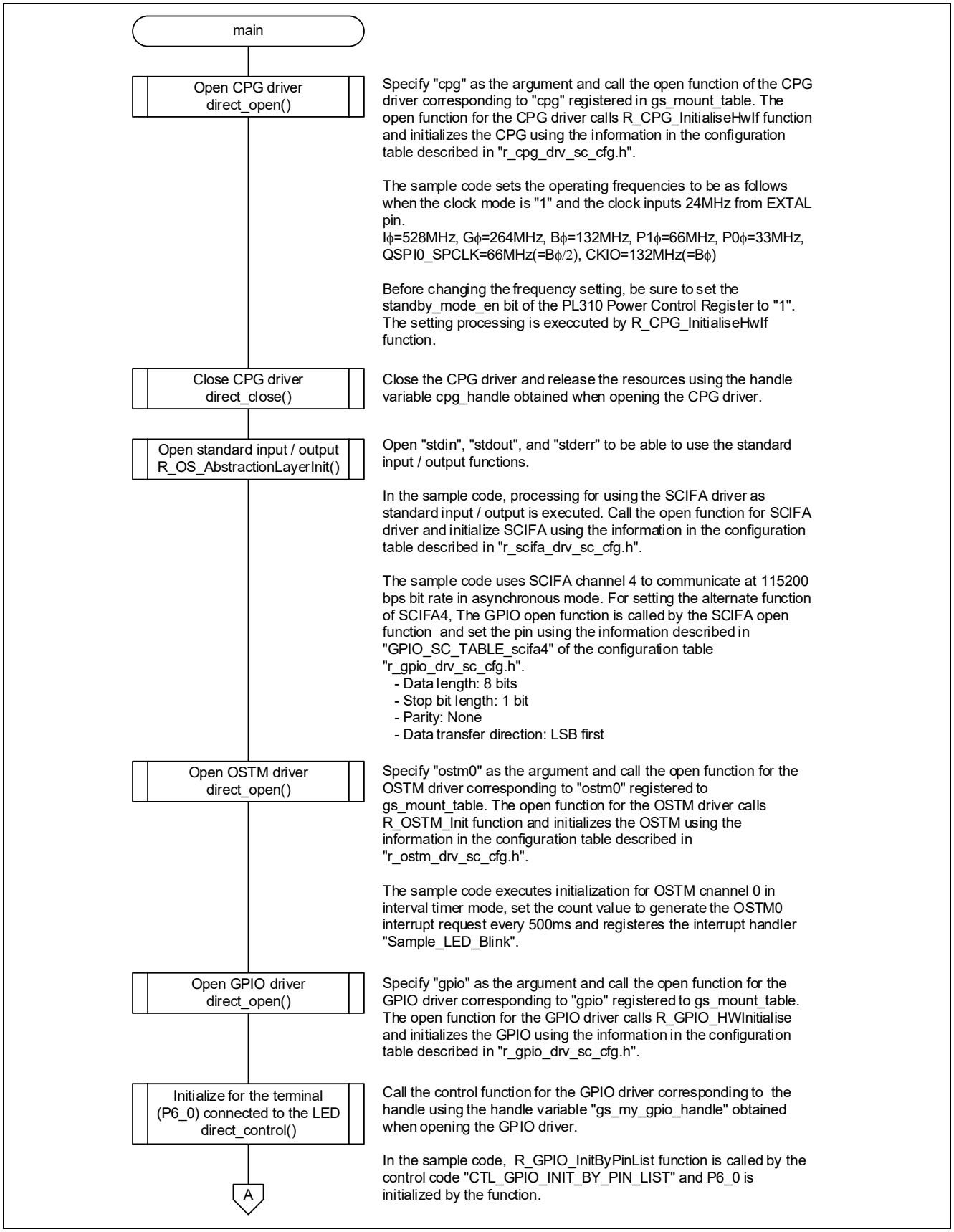



**Figure 5.15 Main Processing (1/2)**

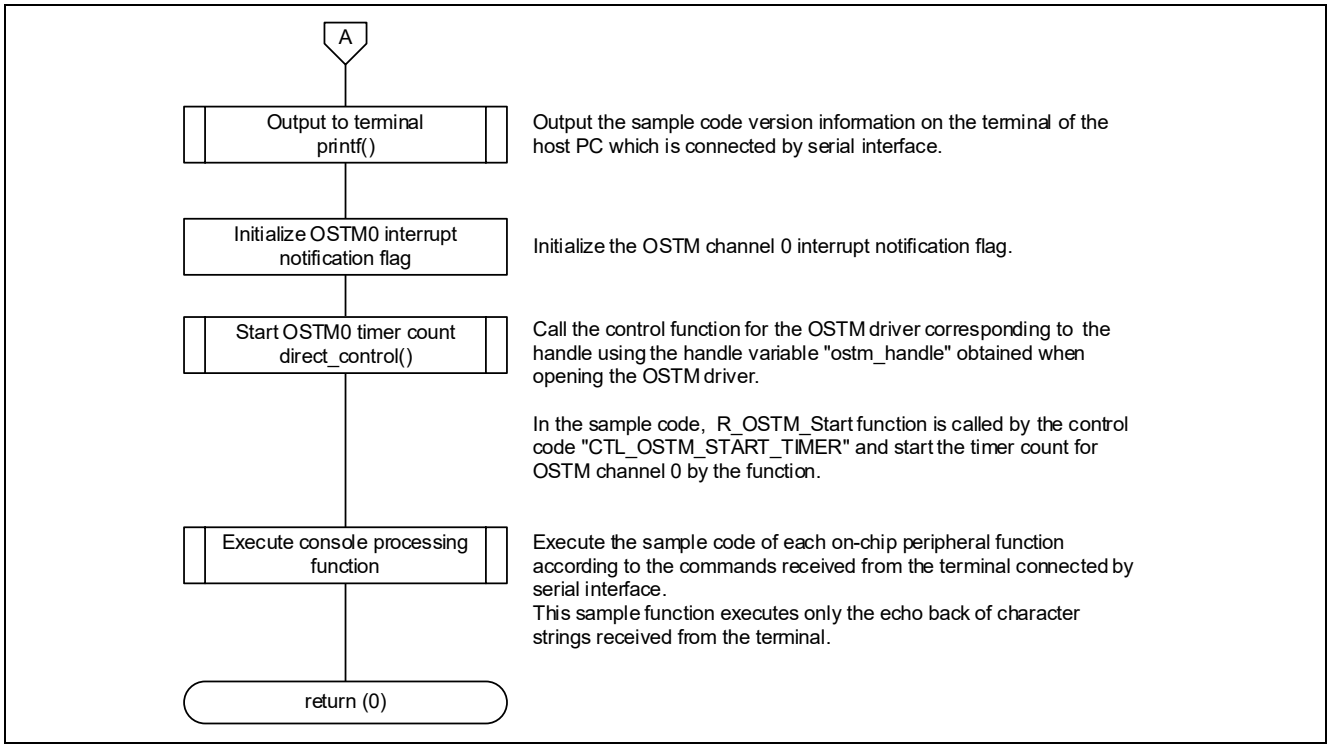



**Figure 5.16 Main Processing (2/2)**

### 5.8.4 L2 Cache Initialization Function

Figure 5.17 shows the flowchart of L2 Cache Initialization Function.

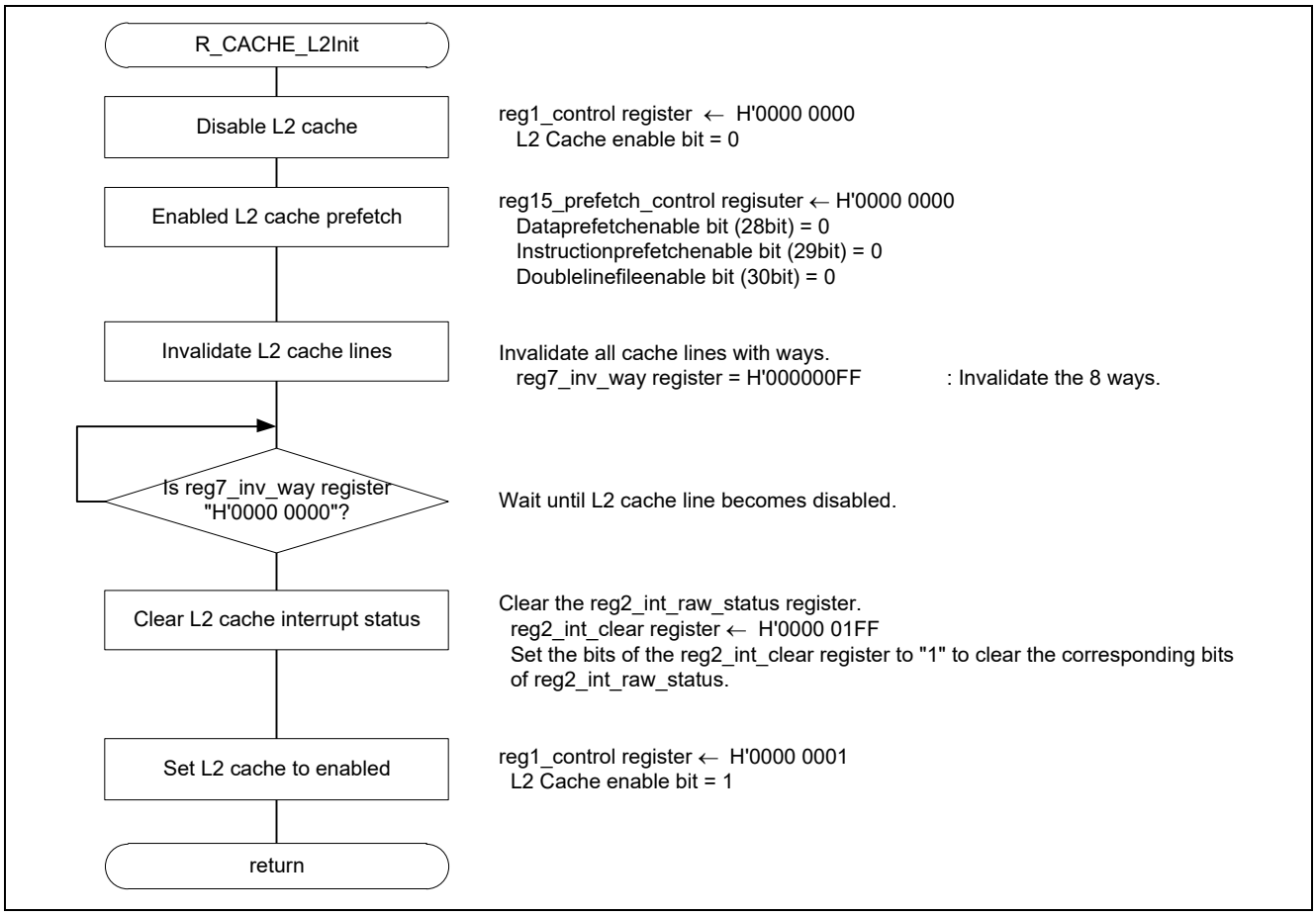



**Figure 5.17 L2 Cache Initialization Function**

### 5.8.5 Initialization Function for MMU

Figure 5.18 shows the flowchart of Initialization Function for MMU.

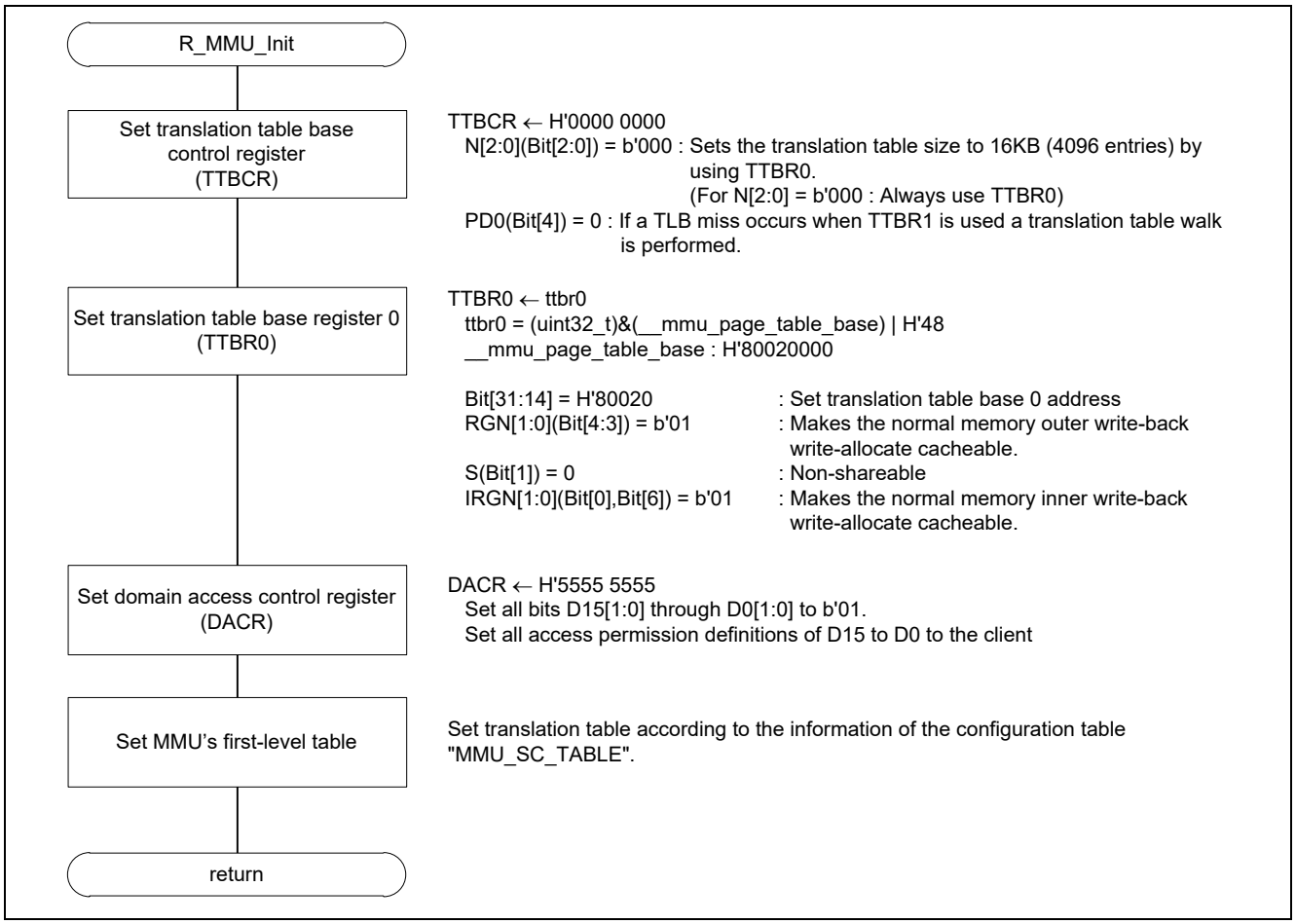



**Figure 5.18 Initialization Function for MMU**

### 5.8.6 Setting Function for MMU Translation Table

Figure 5.19 shows the flowchart of Setting Function for MMU Translation Table.

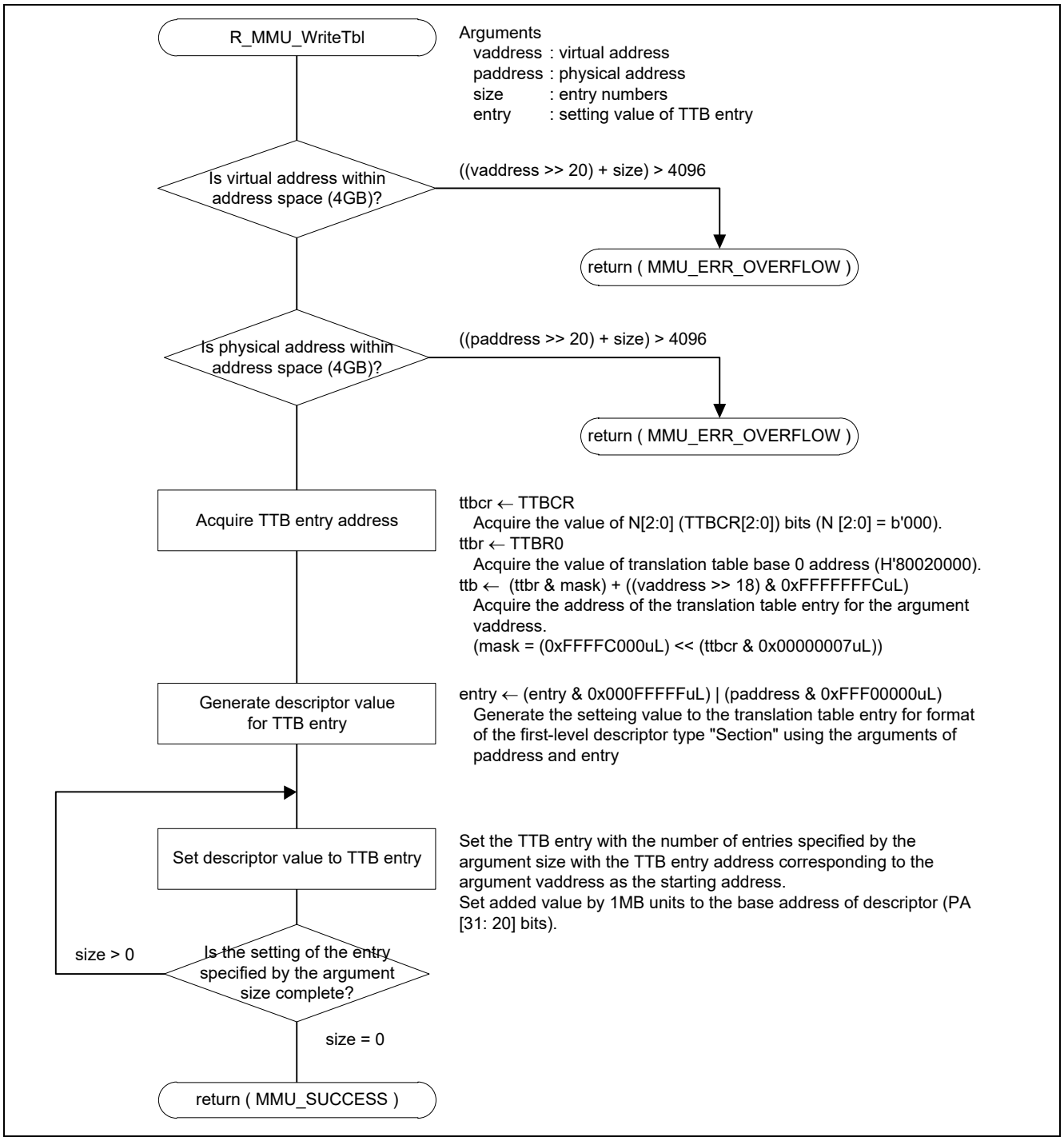



**Figure 5.19 Setting Function for MMU Translation Table**

### 5.8.7 INTC Initialization Function

Figure 5.20 shows the flowchart of INTC Initialization Function.

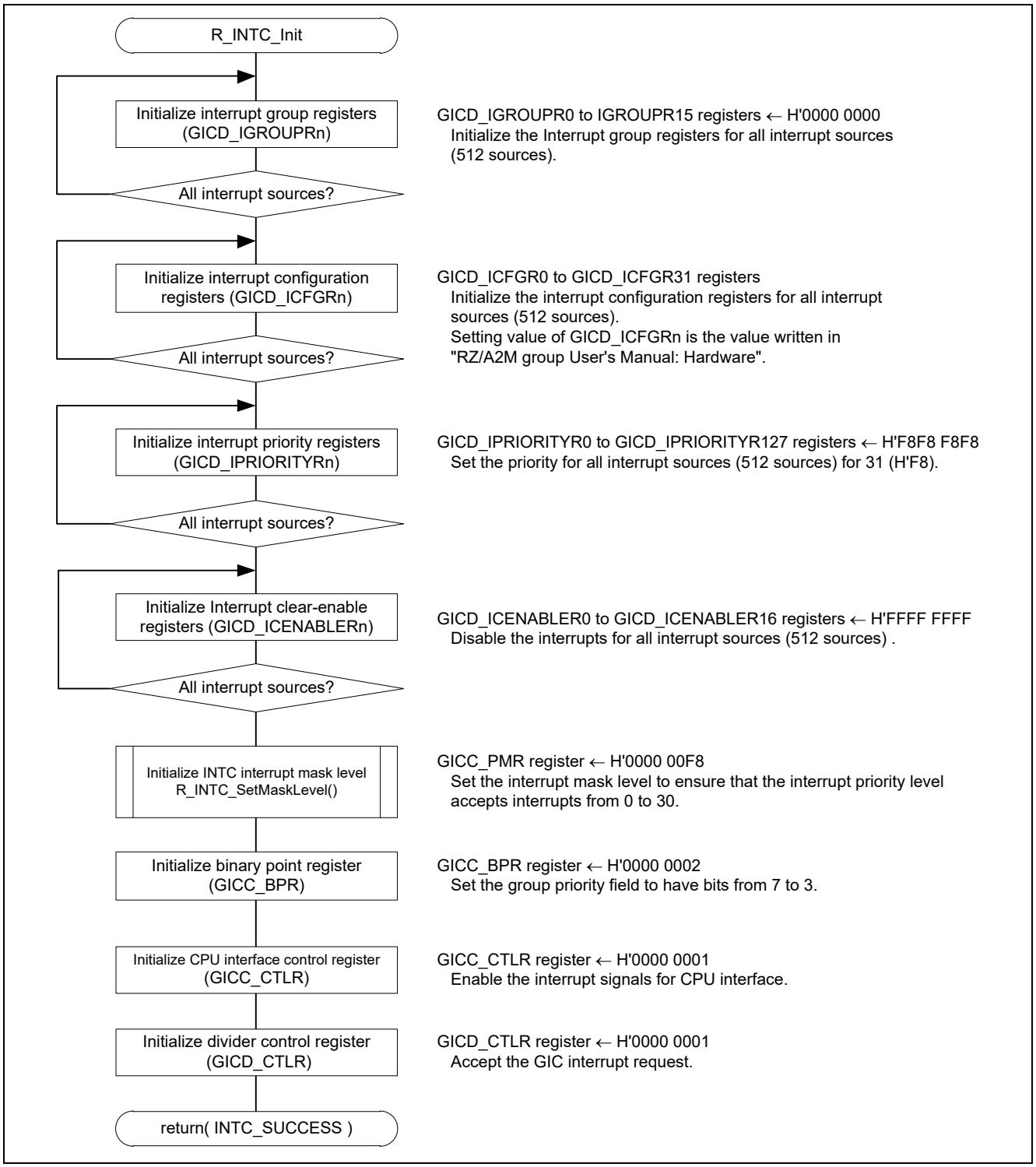



**Figure 5.20 INTC Initialization Function**

### 5.8.8 INTC Interrupt Enable Function

Figure 5.21 shows the flowchart of INTC Interrupt Enable Function.

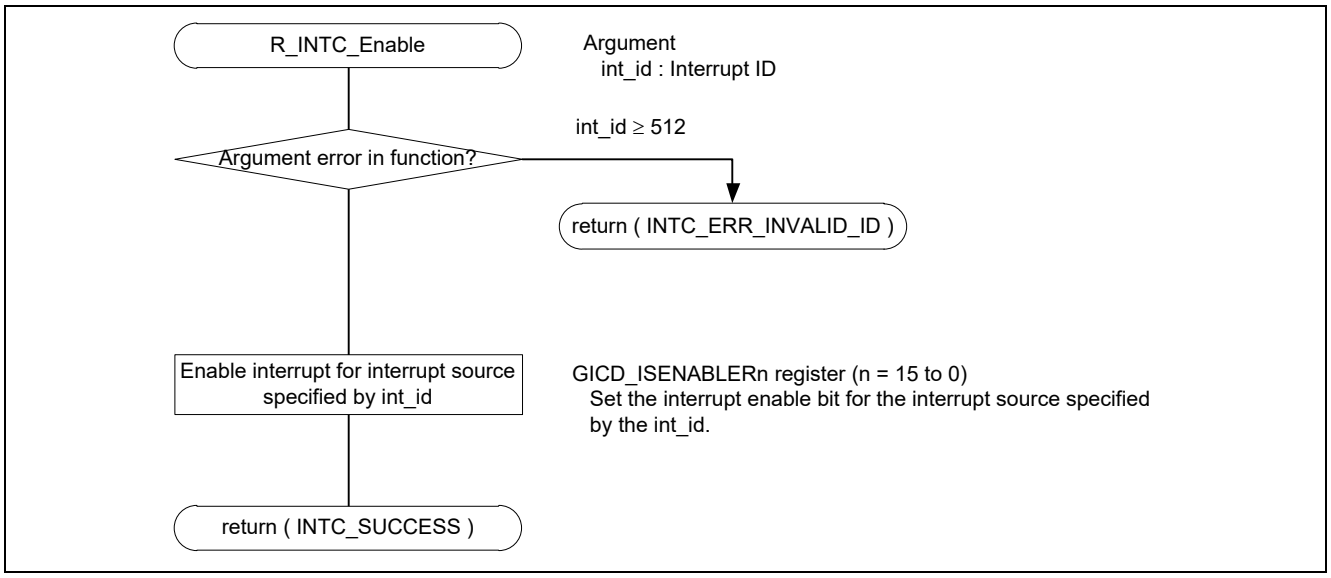



**Figure 5.21 INTC Interrupt Enable Function**

### 5.8.9 INTC Interrupt Disable Function

Figure 5.22 shows the flowchart of INTC Interrupt Disable Function.

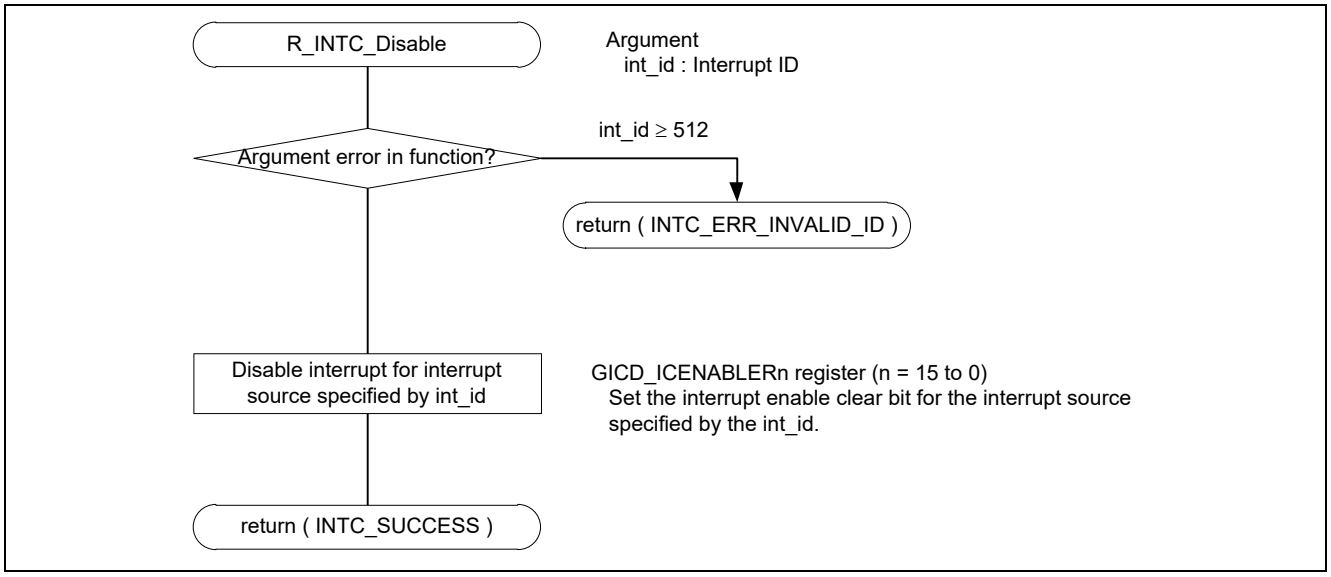



**Figure 5.22 INTC Interrupt Disable Function**

### 5.8.10 Setting Function for INTC Interrupt Priority Level

Figure 5.23 shows the flowchart of Setting Function for INTC Interrupt Priority Level.

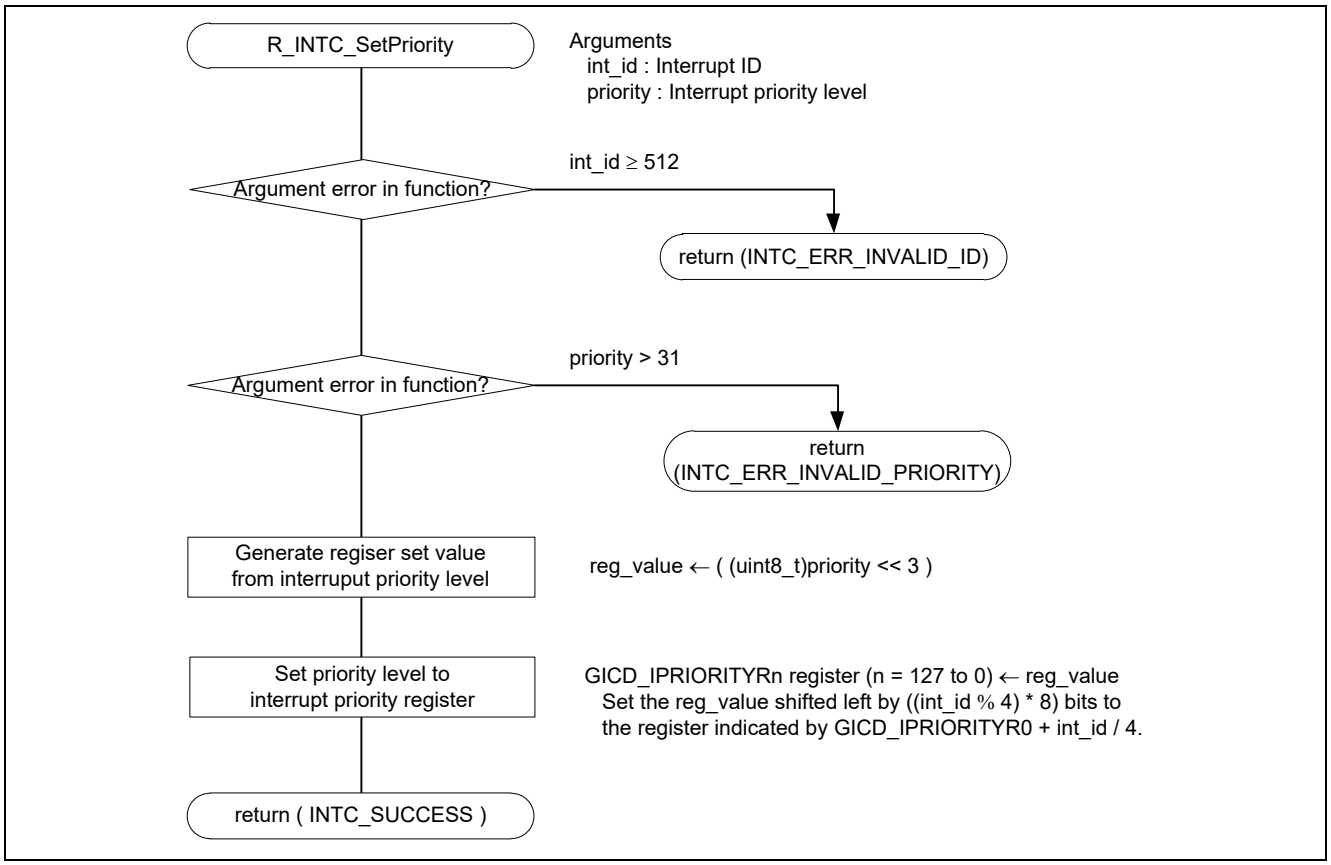



**Figure 5.23 Setting Function for INTC Interrupt Priority Level**

### 5.8.11 Setting Function for INTC Interrupt Mask Level

Figure 5.24 shows the flowchart of Setting Function for INTC Interrupt Mask Level.

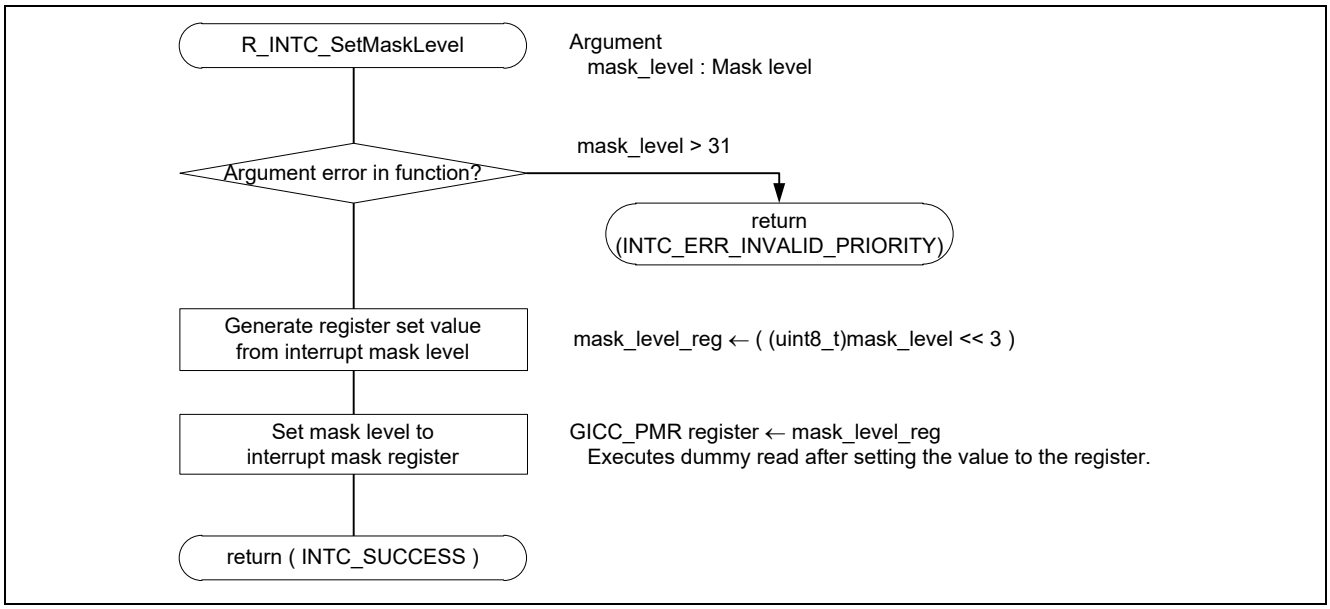



**Figure 5.24 Setting Function for INTC Interrupt Mask Level**

### 5.8.12 Get Function for INTC Interrupt Mask Level

Figure 5.25 shows the flowchart of Get Function for INTC Interrupt Mask Level.

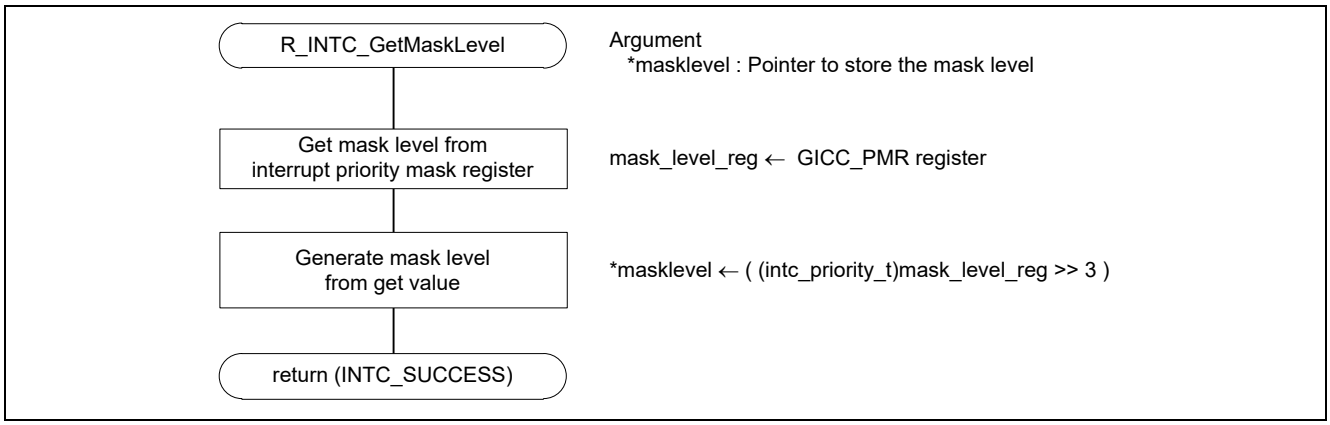



**Figure 5.25 Get Function for INTC Interrupt Mask Level**

### 5.8.13 Registration Function of INTC Interrupt Handler Function

Figure 5.26 shows the flowchart of Registration Function of INTC Interrupt Handler Function.

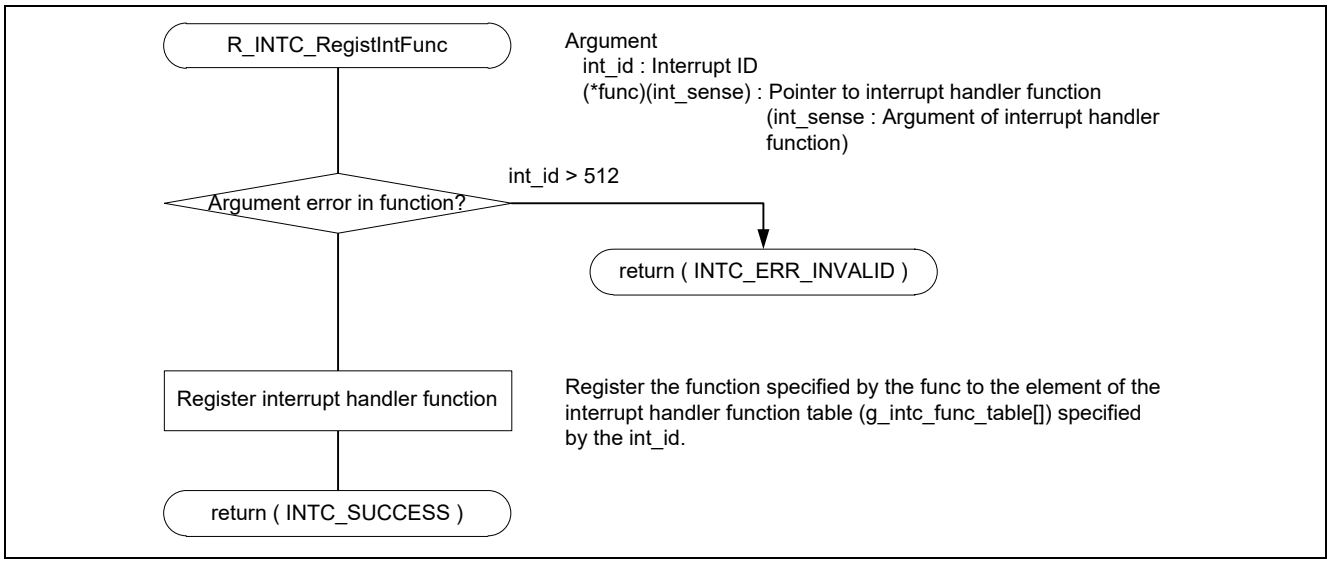



**Figure 5.26 Registration Function of INTC Interrupt Handler Function**

### 5.8.14 IRQ Handler Processing

Figure 5.27 shows the flowchart of IRQ Handler Processing.

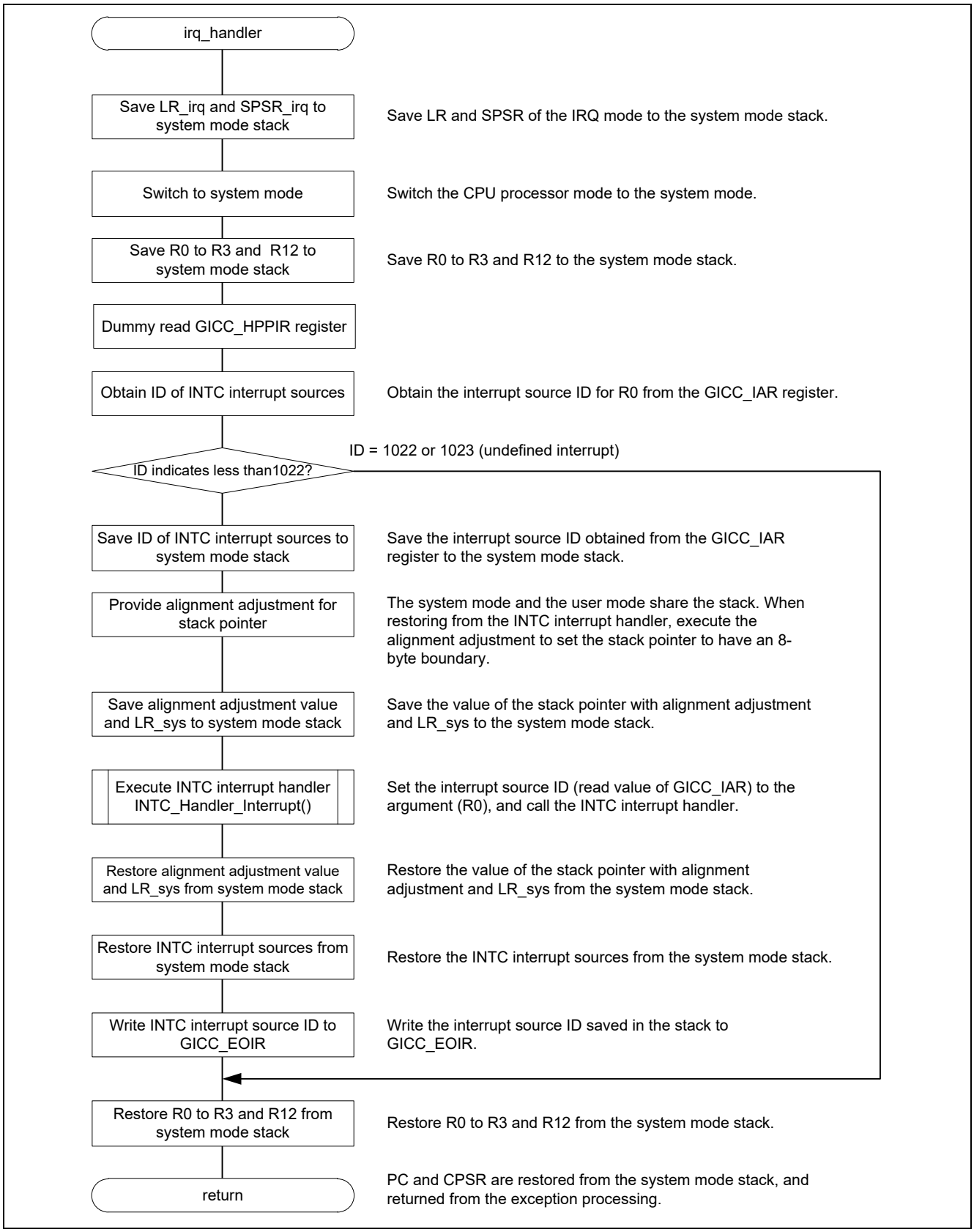



**Figure 5.27 IRQ Handler Processing**

### 5.8.15 INTC Interrupt Handler Processing

Figure 5.28 shows the flowchart of INTC Interrupt Handler Processing.

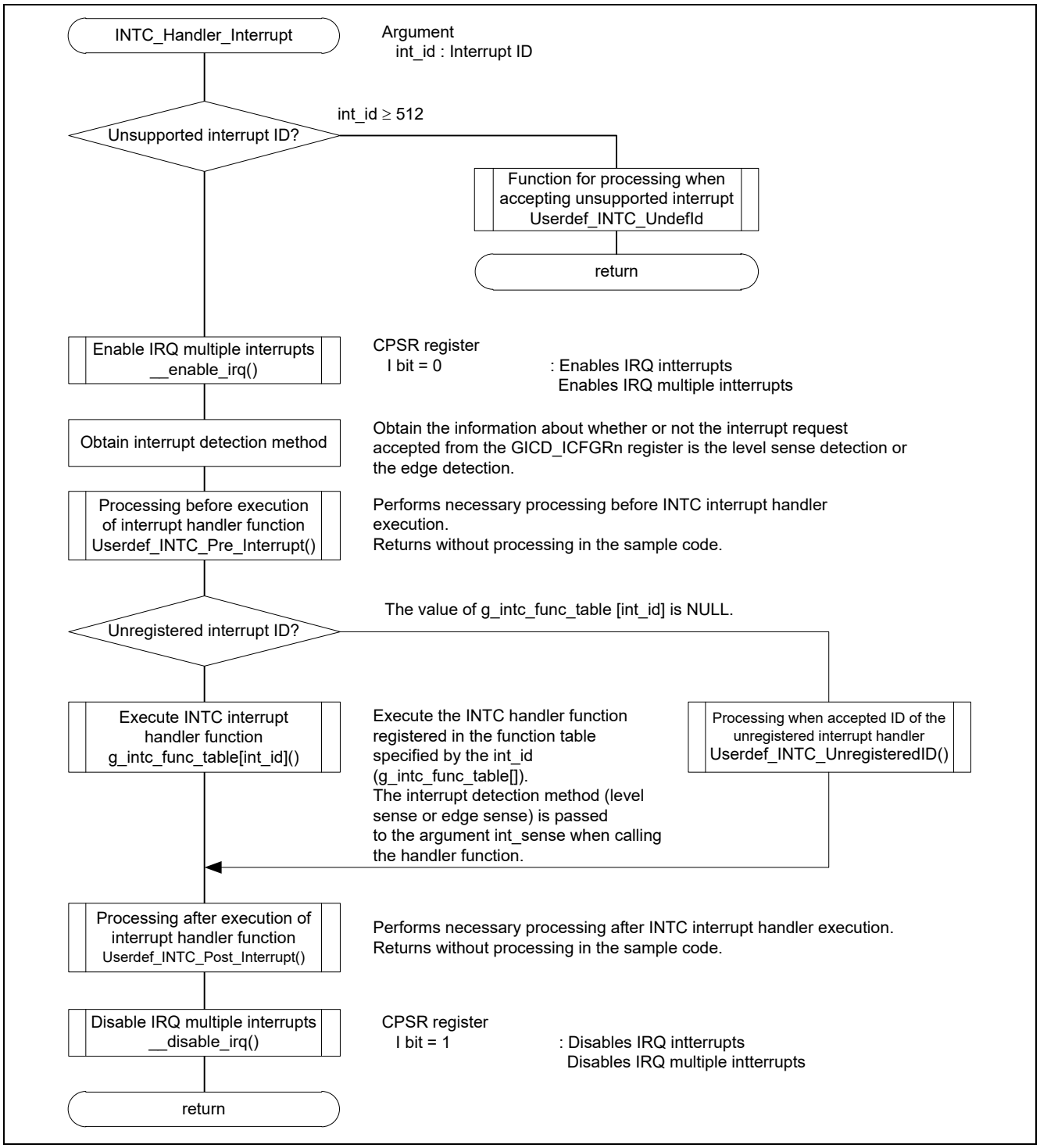



**Figure 5.28 INTC Interrupt Handler Processing**

## 6. Sample Code

Sample code can be downloaded from the Renesas Electronics website.

## 7. Reference Documents

User's Manual: Hardware

RZ/A2M Group User's Manual: Hardware
The latest version can be downloaded from the Renesas Electronics website.

RTK7921053C00000BE (RZ/A2M CPU board) User's Manual
The latest version can be downloaded from the Renesas Electronics website.

RTK79210XXB00000BE (RZ/A2M SUB board) User's Manual
The latest version can be downloaded from the Renesas Electronics website.

ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition Issue C
The latest version can be downloaded from the ARM website.

ARM Cortex™-A9 Technical Reference Manual Revision: r4p1
The latest version can be downloaded from the ARM website.

ARM Generic Interrupt Controller Architecture Specification - Architecture version 2.0
The latest version can be downloaded from the ARM website.

ARM CoreLink™ Level 2 Cache Controller L2C-310 Technical Reference Manual Revision: r3p3
The latest version can be downloaded from the ARM website.

Technical Update/Technical News

The latest information can be downloaded from the Renesas Electronics website.

User's Manual: Development Tools

Integrated development environment e2studio User's Manual can be downloaded from the Renesas Electronics website.
The latest version can be downloaded from the Renesas Electronics website.

## Revision History

| Rev. | Date | Description | |
|---|---|---|---|
| | | Page | Summary |
| Rev.1.00 | Oct.04.18 | — | First edition issued |
| Rev.1.10 | Dec.28.18 | P4 | Table 1.1 Peripheral Function to be Used<br>Added OSTM2. |
| | | P12 | Table 5.1 Setting for Peripheral Functions<br>Added OSTM2. |
| | | P18 | Table 5.6 MMU settings (2/2)<br>Changed "CS3 space non-cacheable area", "HyperRAM non-cacheable area", "OctaRAM non-cacheable area" setting. |
| | | P22 | Table 5.7 Memory Area to be used<br>• Moved section for IO register access.<br>• Added UNCACHED_DATA section. |
| | | P23 | Figure 5.6 Section Assignment<br>• Moved section for IO register access.<br>• Added UNCACHED_DATA section. |
| | | P31 | Table 5.9 Interrupts Used in Sample Program<br>Added OSTM2 interrupt. |
| | | P32 | Added Table 5.12 Constants Used by GPIO Driver (1) |
| | | P32 | Added Table 5.13 Constants Used by GPIO Driver (2) |
| | | P32 | Added Table 5.14 Constants Used by GPIO Driver (3) |
| | | P34 | Table 5.16 API Functions (1/2)<br>Added R_MMU_ReadTbl(), R_MMU_VAtoPA(). |
| | | P43 to 44 | 5.7 Function Specification<br>Added R_MMU_ReadTbl(), R_MMU_VAtoPA(). |
| | | P53 to 54 | 5.7 Function Specification<br>Add notes to R_GPIO_HWInitialise(), R_GPIO_InitByPinList(), and, R_GPIO_InitByTable(). |
| | | P63 | Figure 5.13 Reset Handler Processing<br>Added write access permission processing to Retention RAM (pages 3 to 0). |
| Rev.1.20 | Apr.15.19 | – | Modified the sample code to be able to output CKIO |
| | | P25 | 5.2.6 L1 and L2 Cache Settings<br>Added notes on enabling cache. |
| | | P63 | Figure 5.13 Reset Handler Processing<br>• Changed the setting procedure of NEON and VFP in initial setting process.<br>• Added processing to clear IOKEEP bit of DSFR register. |
| | | P64 | Figure 5.14 resetprg Function<br>Added notes on enabling cache. |
| | | P65 | Figure 5.15 Main Processing (1/2)<br>Changed the description when opening the CPG driver. |

| Rev. | Date | Description | |
|---|---|---|---|
| | | **Page** | **Summary** |
| Rev.1.30 | May.17.19 | P6 | Table 2.2 Operation Conformation Condition (2/2)<br>Remove compiler option "-mthumb-interwork" |
| Rev.1.40 | Nov.20.19 | P22<br>P23 | Changed the following table and figure because an input section for processing of the R_SC_HardwareSetup function was added.<br>• Table 5.7 Memory Area to be used<br>• Figure 5.6 Section Assignment |
| | | P35<br>P62 | Added to functions and function specification because of addition the Userdef_PreHardwareSetup function and the Userdef_PostHardwareSetup function<br>• Table 5.18 User Defined Functions<br>• Funtion Specification |
| | | P63 | Figure 5.13 Reset Handler Processing<br>Fixed the figure due to a clerical error of the processing order about "Initialize floating-point functions" and "Initialize sections". |
| | | P65 | Figure 5.15 Main Processing (1/2)<br>Fixed a crelical error of function name (corrected from initialise_monitor_handles function to R_OS_AbstractionLayerInit function). |
| | | | |

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

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

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

# Notice

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

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

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.
(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

## Corporate Headquarters

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

## Trademarks

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

## Contact information

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