RX Family

Flash Module Using Firmware Integration Technology

Introduction

This application note describes a flash module which uses Firmware Integration Technology (FIT)[*1].
This module has been developed to allow users of supported devices to easily integrate reprogramming of internal flash memory into their applications using self-programming[*2].
This application note focuses on using this module and integrating it with your application program.

[*1] This module is different from the “Simple Flash API for RX (R01AN0544)”.
[*2] Self-programming is a method of reprogramming flash memory using user applications.

Target Devices

- RX110 Group
- RX111 Group
- RX113 Group
- RX130 Group
- RX13T Group
- RX140 Group
- RX230, RX231 Groups
- RX23E-A Group
- RX23E-B Group
- RX23T Group
- RX23W Group
- RX24T Group
- RX24U Group
- RX260, RX261 Group
- RX26T Group
- RX64M Group
- RX65N, RX651 Groups
- RX660 Group
- RX66N Group
- RX66T Group
- RX671 Group
- RX71M Group
- RX72M Group
- RX72N Group
- RX72T Group

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

Target Compilers

- Renesas Electronics C/C++ Compiler Package for RX Family
- GCC for Renesas RX
- IAR C/C++ Compiler for Renesas RX

For details of the confirmed operation contents of each compiler, refer to “5.1 Confirmed Operation Environment”.

Related Documents

- Firmware Integration Technology User’s Manual (R01AN1833)
- Board Support Package Firmware Integration Technology Module (R01AN1685)

Contents

1. Overview

1.1 Flash Module Overview

This module was designed so that the flash memory (code flash memory and data flash memory) embedded in the MCU can be reprogrammed.

An API function used to reprogram flash memory is provided with this module.

1.1.1 Flash Types Overview

Flash memory is categorized by the features supported by MCU. Table 1.1 summarizes the categories relevant to this module.

Table 1.1 Supported MCU Groups by Flash Type

Flash Type	Supported MCU Groups
1	RX110[*1], RX111, RX113, RX130, RX13T, RX140 RX230, RX231, RX23E-A, RX23E-B, RX23T[*1], RX23W, RX24T, RX24U RX260, RX261
3	RX64M, RX660, RX66T, RX71M, RX72T
4	RX651[*2], RX65N[*2], RX66N, RX671, RX72M, RX72N
5	RX26T

[*1] No data flash memory.

[*2] No data flash memory in products 1 Mbyte or less of code flash memory.

1.1.2 Supported Features

Table 1.2 describes the flash types that are required for the features supported by this module.

Table 1.2 Supported Features by Flash Type

Functionality	Overview	Flash Type			
		1	3	4	5
Program	Programs the specified region.	✔	✔	✔	✔
Erase	Erases the specified region.	✔	✔	✔	✔
Blank check	Checks that a specified region is not programmed.	✔ *1	✔ *1	✔ *1	✔ *1
Access window	Sets only specified regions as reprogrammable so as to protect other regions.	✔ *2	—	✔ *2	✔ *2
Startup program protection	Swaps the region containing the startup program after a reset to protect the startup region.	✔ *3	—	✔	✔
Lockbit	Enables/disables a specified region as reprogrammable to protect the specified regions.	—	✔*4	—	—
ROM cache	Enables/disables the code flash memory cache.	✔ *5	✔ *6	✔	—
Disable cache	Sets regions for which cache is disabled.	—	✔ *6	✔ *7	—
Dual bank	Swaps the startup bank.	—	—	✔ *8	✔ *9
Flash sequencer reset	Resets the flash sequencer.	✔	✔	✔	✔
Flash sequencer usage frequency notification	Provides notification of the frequency used by the flash sequencer.	—	✔	✔	✔

*1 Only Flash Type 1 supports blank checks on code flash memory.
*2 Access window can only be used on code flash memory.
*3 Only supported on products with at least 32 Kbytes of code flash memory.
*4 Lockbit can only be used on code flash memory.
*5 Supported by RX24T and RX24U only.
*6 Supported by RX66T and RX72T only.
*7 Supported by RX66N, RX671, RX72M, and RX72N only.
*8 Only supported on products with at least 1 Mbytes of code flash memory.
*9 Only supported on products with at least 512 Kbytes of code flash memory.

1.2 API Overview

Table 1.3 describes information on the API information embedded in this module.

Table 1.3 API Functions

Function	Description of Function
R_FLASH_Open()	Initializes this module.
R_FLASH_Close()	Closes this module.
R_FLASH_Erase()	Erases specified blocks in data flash memory or code flash memory.
R_FLASH_BlankCheck()	Checks that specified regions in data flash memory or code flash memory have not been programmed.
R_FLASH_Write()	Programs specific data into specified regions in data flash memory or code flash memory.
R_FLASH_Control()	Performs functionality other than programming, erasing, and blank check.
R_FLASH_GetVersion()	Returns the current version of this module.

1.3 Limitations

1.3.1 Flash Memory Access Restrictions

The flash sequencer has a read mode for reading the flash memory and a P/E mode for reprogramming the flash memory.

Table 1.4 describes the regions that can and cannot be read during P/E mode.

Table 1.4 Regions With/Without Read Access During P/E Mode

Region Accessed During P/E Mode	Regions Without Read Access	Regions With Read Access*1
Code flash memory	Code flash memory	Data flash memory RAM External memory Other code flash memory*2
Data flash memory	Data flash memory	Code flash memory RAM External memory

*1 Excluding data flash memory, reprogramming code and interrupt vector tables should be allocated in regions with read access.

*2 Products with multiple regions of code flash memory.

Refer to section 2.16.1 for more information on running reprogramming code from RAM.

Refer to section 2.16.2 for more information on reprogramming code flash memory with data in other code flash memory.

It is necessary to reallocate interrupt vector tables and interrupt handlers to the RAM for interrupts that may occur while the code flash memory is being reprogrammed. Refer to Example 1 in section 3.6 for a usage example.

1.3.2 RAM Allocation Restrictions

With FIT, configuring pointer arguments of API functions with values equivalent to NULL values results in parameter checks sometimes producing return errors. As such, do not set values of pointer arguments passed to API functions to values equivalent to NULL values.

The NULL value is defined in standard library specifications as zero (0). As such, the issue above will occur if variables and functions passed to API function pointer arguments are stored in starting addresses (0x0 addresses) in RAM. In this case, change the configuration of sections or create dummy variables to be stored at the beginning of RAM to prevent variables and functions passed to API function pointer arguments from being stored at 0x0 addresses.

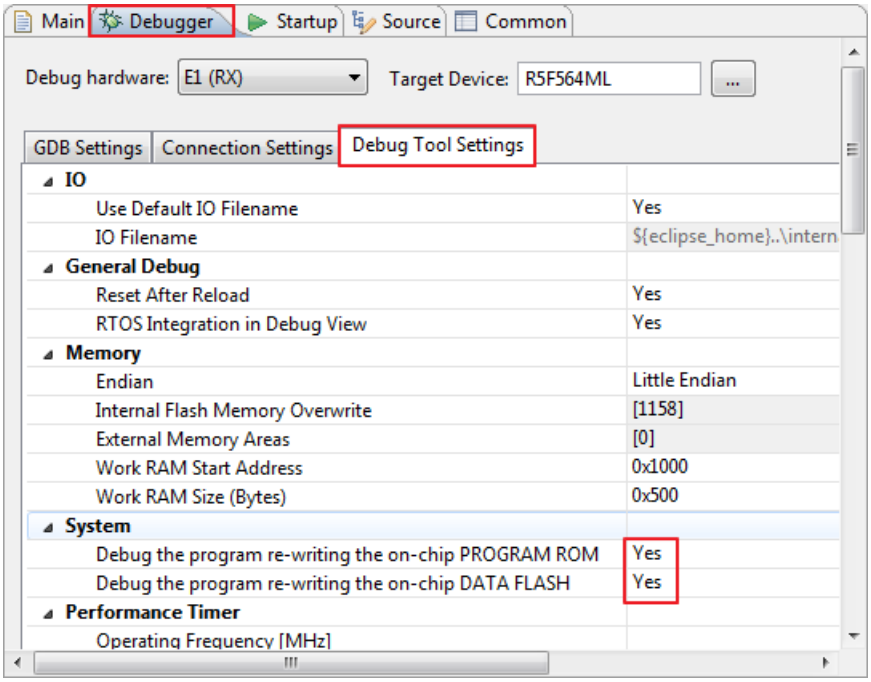
CCRX projects (e^2 studio V7.5.0) are configured so that 0x4 is the starting RAM address to prevent variables from being stored at the 0x0 address. This issue must be prevented in case of GCC projects (e^2 studio V7.5.0) and IAR projects (EWRX V4.12.1) because the starting RAM address is set to 0x0.

Default section settings may need to be changed whenever the IDE is upgraded. Make sure to always check section settings before using the latest version of your IDE.

1.3.3 Emulator Debug Configuration Restrictions

To confirm the data written to code flash memory and data flash memory during debug, change the Debug Tool Settings of the debug configuration as follows.

1. In Project Explorer, click the project you want to debug.
2. Click Execute -> Debug Configuration to open the Debug Configuration window.
3. On the Debug Configuration window, expand the display of the “Renesas GDB Hardware Debugging” debug configuration and click the debug configuration you want to debug.
4. Switch to the “Debugger” tab, click the “Debug Tool Settings” in the “Debugger” tab and make the following settings.
   - System
     - Debug the program re-writing the on-chip PROGRAM ROM = “Yes”
     - Debug the program re-writing the on-chip DATA FLASH = “Yes”



2. API Information

This module has been confirmed to operate under the following conditions.

2.1 Hardware Requirements

This driver requires that your MCU supports the following peripheral(s):

- Flash memory (code flash memory and data flash memory)

2.2 Software Requirements

The driver is dependent on the following FIT module.

- Board Support Package (r_bsp) v5.20 or later

2.3 Supported Toolchains

This module has been confirmed to work with the toolchain listed in 5.1 Confirmed Operation Environment.

2.4 Interrupt Vector

When the FLASH_CFG_DATA_FLASH_BGO or FLASH_CFG_CODE_FLASH_BGO configuration option (see section 2.7) is 1, the interrupts shown in Table 2.1 below are enabled.

Table 2.1 Interrupt Vectors Used in this Module

Flash Type	Interrupt Vector
1	FRDYI interrupt (vector no.: 23)
3, 4, 5	FRDYI interrupt (vector no.: 23), FIFERR interrupt (vector no.: 21)

2.5 Header Files

All API calls and their supporting interface definitions are located in “r_flash_rx_if.h”. This file should be included by all files which utilize the Flash Module.

The configuration options that can be set at build time are defined in the “r_flash_rx_config.h” file.

2.6 Integer Types

This project uses ANSI C99 “Exact width integer types” in order to make the code clearer and more portable. These types are defined in stdint.h.

2.7 Configuration Overview

Configuring this module is done through the supplied r_flash_rx_config.h header file. Each configuration item is represented by a macro definition in this file. Each configurable item is detailed in the table below.

Configuration options in r_flash_rx_config.h	
FLASH_CFG_PARAM_CHECKING_ENABLE *Default value is “1”.	Enables/disables the inclusion of parameter check processing into the code. A value of “0” omits parameter check processing from the code. A value of “1” includes parameter check processing in the code.
FLASH_CFG_CODE_FLASH_ENABLE *Default value is “0”.	Enables/disables the inclusion of code used to program code flash memory regions. A value of “0” includes code used to program data flash memory regions only (no code flash memory regions). A value of “1” includes code used to program both code flash memory regions and data flash memory regions.
FLASH_CFG_DATA_FLASH_BGO *Default value is “0”.	Specifies the processing method for data flash memory. A value of “0” processes data flash memory in blocking mode. A value of “1” processes data flash memory in non-blocking mode. When FLASH_CFG_CODE_FLASH_ENABLE is set to “1”, make the same setting as FLASH_CFG_CODE_FLASH_BGO. Refer to 2.13 for details on blocking mode and non-blocking mode.
FLASH_CFG_CODE_FLASH_BGO *Default value is “0”.	Specifies the processing method for code flash memory. A value of “0” processes code flash memory in blocking mode. A value of “1” processes code flash memory in non-blocking mode. When FLASH_CFG_CODE_FLASH_ENABLE is set to “1”, make the same setting as FLASH_CFG_DATA_FLASH_BGO. Refer to 2.13 for details on blocking mode and non-blocking mode.
FLASH_CFG_CODE_FLASH_RUN_FROM_ROM*1 *Default value is “0”.	Specifies the code location for running the program and erase features on flash memory. This option is enabled only when FLASH_CFG_CODE_FLASH_ENABLE is set to “1”. If set to “0”, the code for running the program and erase features on flash memory is stored and ran in RAM. Refer to section 2.16.1 for details. If set to “1”, the code for running the program and erase features on flash memory is allocated and ran in code flash memory. Refer to section 2.16.2 for details.

*1 Supported only in products with multiple regions of code flash memory.

2.8 Code Size

The ROM size, RAM size, and the maximum stack size of this module are described in the following table. Separate examples are given for each type of product: Flash Type 1 with data flash memory, Flash Type 1 without data flash memory, Flash Type 3, 4, and Flash Type 5.

The ROM (code and constants) and RAM (global data) sizes are determined by the build-time configuration options set in the module configuration header file.

The values in the table below are confirmed under the following conditions.

Module Revision:	r_flash_rx Rev.5.00
Compiler Version:	Renesas Electronics C/C++ Compiler Package for RX Family V3.05.00 (The option of “-lang = c99” is added to the default settings of the integrated development environment.) GCC for Renesas RX 8.03.00.202204 (The option of “-std = gnu99” is added to the default settings of the integrated development environment.) IAR C/C++ Compiler for Renesas RX version 4.20.3 (The default settings of the integrated development environment.)
Configuration Options:	The setting of configuration options that are different is described in each table. Other configuration options are default settings.

Flash Type 1: ROM, RAM and Stack Code Sizes (Maximum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX130	ROM	3510 bytes	3148 bytes	5092 bytes	4516 bytes	5416 bytes	4912 bytes
	RAM	3026 bytes		4480 bytes		4785 bytes	
	STACK	112 bytes		-		100 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 1 FLASH_CFG_DATA_FLASH_BGO 1 FLASH_CFG_CODE_FLASH_BGO 1							

Flash Type 1: ROM, RAM and Stack Code Sizes (Minimum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX130	ROM	1843 bytes	1681 bytes	2544 bytes	2320 bytes	2583 bytes	2368 bytes
	RAM	61 bytes		60 bytes		41 bytes	
	STACK	52 bytes		-		44 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 0 FLASH_CFG_DATA_FLASH_BGO 0 FLASH_CFG_CODE_FLASH_BGO 0							

Flash Type 1: ROM, RAM and Stack Code Sizes (Maximum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX140	ROM	3442 bytes	3067 bytes	4996 bytes	4404 bytes	5392 bytes	4878 bytes
	RAM	2939 bytes		4352 bytes		4728 bytes	
	STACK	108 bytes		-		100 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 1 FLASH_CFG_DATA_FLASH_BGO 1 FLASH_CFG_CODE_FLASH_BGO 1							

Flash Type 1: ROM, RAM and Stack Code Sizes (Minimum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX140	ROM	1861 bytes	1686 bytes	2544 bytes	2304 bytes	2615 bytes	2391 bytes
	RAM	61 bytes		0 bytes		41 bytes	
	STACK	52 bytes		-		44 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 0 FLASH_CFG_DATA_FLASH_BGO 0 FLASH_CFG_CODE_FLASH_BGO 0							

Flash Type 1: ROM, RAM and Stack Code Sizes (Maximum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX23T*	ROM	2994 bytes	2687 bytes	4208 bytes	3704 bytes	4472 bytes	4048 bytes
	RAM	2740 bytes		3840 bytes		4185 bytes	
	STACK	112 bytes		-		100 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 1 FLASH_CFG_DATA_FLASH_BGO 1 FLASH_CFG_CODE_FLASH_BGO 1							

*Device without data flash

Flash Type 1: ROM, RAM and Stack Code Sizes (Minimum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX23T*	ROM	2654 bytes	2360 bytes	3624 bytes	3176 bytes	3808 bytes	3392 bytes
	RAM	2405 bytes		3328 bytes		3520 bytes	
	STACK	52 bytes		-		44 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 1 FLASH_CFG_DATA_FLASH_BGO 0 FLASH_CFG_CODE_FLASH_BGO 0							

*Device without data flash

Flash Type 3: ROM, RAM and Stack Code Sizes (Maximum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX64M	ROM	3626 bytes	3154 bytes	5240 bytes	4600 bytes	5680 bytes	5076 bytes
	RAM	3181 bytes		4608 bytes		5053 bytes	
	STACK	220 bytes		-		176 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 1 FLASH_CFG_DATA_FLASH_BGO 1 FLASH_CFG_CODE_FLASH_BGO 1							

Flash Type 3: ROM, RAM and Stack Code Sizes (Minimum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX64M	ROM	2237 bytes	2048 bytes	3128 bytes	2864 bytes	3279 bytes	3030 bytes
	RAM	65 bytes		0 bytes		48 bytes	
	STACK	76 bytes		-		56 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 0 FLASH_CFG_DATA_FLASH_BGO 0 FLASH_CFG_CODE_FLASH_BGO 0							

Flash Type 4: ROM, RAM and Stack Code Sizes (Maximum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX65N	ROM	3674 bytes	3190 bytes	5160 bytes	4464 bytes	5636 bytes	5004 bytes
	RAM	3053 bytes		4480 bytes		4815 bytes	
	STACK	204 bytes		-		172 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 1 FLASH_CFG_DATA_FLASH_BGO 1 FLASH_CFG_CODE_FLASH_BGO 1							

Flash Type 4: ROM, RAM and Stack Code Sizes (Minimum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX65N	ROM	2085 bytes	1896 bytes	2896 bytes	2616 bytes	3080 bytes	2831 bytes
	RAM	61 bytes		128 bytes		47 bytes	
	STACK	80 bytes		-		56 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 0 FLASH_CFG_DATA_FLASH_BGO 0 FLASH_CFG_CODE_FLASH_BGO 0							

Flash Type 5: ROM, RAM and Stack Code Sizes (Maximum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX26T	ROM	3583 bytes	3099 bytes	5032 bytes	4328 bytes	5464 bytes	4824 bytes
	RAM	2973 bytes		4352 bytes		4653 bytes	
	STACK	240 bytes		-		184 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 1 FLASH_CFG_DATA_FLASH_BGO 1 FLASH_CFG_CODE_FLASH_BGO 1							

Flash Type 5: ROM, RAM and Stack Code Sizes (Minimum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX26T	ROM	2068 bytes	1879 bytes	2860 bytes	2588 bytes	3049 bytes	2800 bytes
	RAM	61 bytes		128 bytes		47 bytes	
	STACK	76 bytes		-		56 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE_FLASH_ENABLE 0 FLASH_CFG_DATA_FLASH_BGO 0 FLASH_CFG_CODE_FLASH_BGO 0							

2.9 Parameters

This section defines the structure and enumeration used for API function arguments. This section provides common module definitions and definitions that vary depending on flash memory functionality and capacity.

2.9.1 Definitions of Common Arguments

Structures and enumerations commonly used as module arguments are defined in “r_flash_rx_if.h”.

```
/* Callback function event type */
typedef enum _flash_interrupt_event
{
    FLASH_INT_EVENT_INITIALIZED,               // No value is returned
    FLASH_INT_EVENT_ERASE_COMPLETE,            // Completion of erase process
    FLASH_INT_EVENT_WRITE_COMPLETE,            // Completion of program process
    FLASH_INT_EVENT_BLANK,                     // Blank check result - blank
    FLASH_INT_EVENT_NOT_BLANK,                 // Blank check result - not blank
    FLASH_INT_EVENT_TOGGLE_STARTUPAREA,        // Swapping of the startup region
    FLASH_INT_EVENT_SET_ACCESSWINDOW,          // Configuration of access window
    FLASH_INT_EVENT_LOCKBIT_WRITTEN,           // Setting of lockbit
    FLASH_INT_EVENT_LOCKBIT_PROTECTED,         // Enabling of lockbit protection
    FLASH_INT_EVENT_LOCKBIT_NON_PROTECTED,     // Disabling of lockbit protection
    FLASH_INT_EVENT_ERR_DF_ACCESS,             // Data flash memory access violation
    FLASH_INT_EVENT_ERR_CF_ACCESS,             // Code flash memory access violation
    FLASH_INT_EVENT_ERR_SECURITY,       // Access window write protection violation
    FLASH_INT_EVENT_ERR_CMD_LOCKED,     // Command is locked
    FLASH_INT_EVENT_ERR_LOCKBIT_SET,    // Error due in region protected by lockbit
    FLASH_INT_EVENT_ERR_FAILURE,        // Error during program or erase process
    FLASH_INT_EVENT_TOGGLE_BANK,        // Swapping of startup bank
    FLASH_INT_EVENT_END_ENUM            // No value is returned
} flash_interrupt_event_t;
```

```
/* Definitions used for registration of callback function */
typedef struct _flash_interrupt_config
{
    void    (*pcallback)(void *);              // Callback function pointer
    uint8_t int_priority;                      // Interrupt priority
} flash_interrupt_config_t;
```

```
/* Definitions used as the callback function arguments */
typedef struct
{
    flash_interrupt_event_t event;             // Interrupt-causing event
} flash_int_cb_args_t;
```

```
/* R_FLASH_Control Function command definitions */
typedef enum _flash_cmd
{
    FLASH_CMD_RESET,              // Resets the flash sequencer
    FLASH_CMD_STATUS_GET,         // Retrieves the status of the FLASH FIT module API
    FLASH_CMD_SET_BGO_CALLBACK,   // Registers the callback function
    FLASH_CMD_SWAPFLAG_GET,     // Retrieves configuration of the current startup region
    FLASH_CMD_SWAPFLAG_TOGGLE,    // Swaps the startup region
    FLASH_CMD_SWAPSTATE_GET,    // Retrieves setting of the startup region selection bit
    FLASH_CMD_SWAPSTATE_SET,      // Sets the startup region selection bit
    FLASH_CMD_ACCESSWINDOW_SET,   // Sets the access window boundary
    FLASH_CMD_ACCESSWINDOW_GET,   // Retrieves the access window boundary
    FLASH_CMD_LOCKBIT_READ,     // Retrieves lockbit information for the specified block
    FLASH_CMD_LOCKBIT_WRITE,      // Sets the lockbit for the specified block
    FLASH_CMD_LOCKBIT_ENABLE,     // Enables lockbit protection
    FLASH_CMD_LOCKBIT_DISABLE,    // Disables lockbit protection
    FLASH_CMD_CONFIG_CLOCK,       // Provides notification of operating frequency to the
flash sequencer
    FLASH_CMD_ROM_CACHE_ENABLE,   // Enables ROM cache
    FLASH_CMD_ROM_CACHE_DISABLE,  // Disables ROM cache
    FLASH_CMD_ROM_CACHE_STATUS,   // Retrieves ROM cache status (enabled/disabled)
    FLASH_CMD_SET_NON_CACHED_RANGE0, // Sets the range of RANGE0 where cache is disabled
    FLASH_CMD_SET_NON_CACHED_RANGE1, // Sets the range of RANGE1 where cache is disabled
    FLASH_CMD_GET_NON_CACHED_RANGE0, // Retrieves the setting of RANGE0 where cache is
disabled
    FLASH_CMD_GET_NON_CACHED_RANGE1, // Retrieves the setting of RANGE1 where cache is
disabled
    FLASH_CMD_BANK_TOGGLE,  // Swaps the startup bank
    FLASH_CMD_BANK_GET,     // Retrieves settings of the current bank selection register
    FLASH_CMD_END_ENUM      // This definition is not used
} flash_cmd_t;
```

```
/* Definitions of R_FLASH_Control and R_FLASH_BlankCheck function results
typedef enum _flash_res
{
    FLASH_RES_LOCKBIT_STATE_PROTECTED,  // FLASH_CMD_LOCKBIT_READ result - protected
    FLASH_RES_LOCKBIT_STATE_NON_PROTECTED, // FLASH_CMD_LOCKBIT_READ result - not
protected
    FLASH_RES_BLANK,                    // R_FLASH_BlankCheck result - blank
    FLASH_RES_NOT_BLANK                 // R_FLASH_BlankCheck result - not blank
} flash_res_t;
```

```
/* Definitions used with FLASH_CMD_BANK_GET command in R_FLASH_Control function */
typedef enum _flash_bank
{
    FLASH_BANK1          = 0,                  // BANKSEL.BANKSWP is 000
    FLASH_BANK0          = 1,                  // BANKSEL.BANKSWP is 111
    FLASH_BANK0_FFE00000 = 0,                  // BANKSEL.BANKSWP is 000
    FLASH_BANK1_FFF00000 = 0,                  // BANKSEL.BANKSWP is 000
    FLASH_BANK0_FFF00000 = 1,                  // BANKSEL.BANKSWP is 111
    FLASH_BANK1_FFE00000 = 1                   // BANKSEL.BANKSWP is 111
} flash_bank_t;
```

```
/* Definitions used with FLASH_CMD_ACCESSWINDOW_SET/GET commands in R_FLASH_Control
function */
typedef struct _flash_access_window_config
{
    uint32_t start_addr;                        // Start address of access window
    uint32_t end_addr;                          // End address of access window
} flash_access_window_config_t;
```

```
/* Definitions used with FLASH_CMD_LOCKBIT_READ/WRITE commands in R_FLASH_Control
function */
typedef struct _flash_lockbit_config
{
    flash_block_address_t     block_start_address;   // Start address*1
    flash_res_t               result; // Retrieval result of lockbit information*2
    uint32_t                  num_blocks; // Number of blocks to have lockbit set*3
} flash_lockbit_config_t;
```

*1. The actual definition of flash_block_address_t varies depending on the MCU.

*2 Used when using the FLASH_CMD_LOCKBIT_READ command.

*3 Used when using the FLASH_CMD_LOCKBIT_WRITE command.

```
/* Definitions used for specifying sizes of caches being disabled */
typedef enum _flash_no_cache_size
{
    FLASH_NON_CACHED_16_BYTES   = 0x10,                 // 16 bytes
    FLASH_NON_CACHED_32_BYTES   = 0x20,                 // 32 bytes
    FLASH_NON_CACHED_64_BYTES   = 0x40,                 // 64 bytes
    FLASH_NON_CACHED_128_BYTES  = 0x80,                 // 128 bytes
    FLASH_NON_CACHED_256_BYTES  = 0x100,                // 256 bytes
    FLASH_NON_CACHED_512_BYTES  = 0x200,                // 512 bytes
    FLASH_NON_CACHED_1_KBYTE    = 0x400,                // 1 Kbyte
    FLASH_NON_CACHED_2_KBYTES   = 0x800,                // 2 Kbytes
    FLASH_NON_CACHED_4_KBYTES   = 0x1000,               // 4 Kbytes
    FLASH_NON_CACHED_8_KBYTES   = 0x2000,               // 8 Kbytes
    FLASH_NON_CACHED_16_KBYTES  = 0x4000,               // 16 Kbytes
    FLASH_NON_CACHED_32_KBYTES  = 0x8000,               // 32 Kbytes
    FLASH_NON_CACHED_64_KBYTES  = 0x10000,              // 64 Kbytes
    FLASH_NON_CACHED_128_KBYTES = 0x20000,              // 128 Kbytes
    FLASH_NON_CACHED_256_KBYTES = 0x40000,              // 256 Kbytes
    FLASH_NON_CACHED_512_KBYTES = 0x80000,              // 512 Kbytes
    FLASH_NON_CACHED_1_MBYTE    = 0x100000,             // 1 Mbyte
    FLASH_NON_CACHED_2_MBYTE    = 0x200000              // 2 Mbytes
} flash_non_cached_size_t;
```

```
/* Definitions used with FLASH_CMD_SET_NON_CACHED_RANGE0/RANGE1 and
FLASH_CMD_GET_NON_CACHED_RANGE0/RANGE1 commands in R_FLASH_Control function */

typedef struct _flash_non_cached
{
    uint32_t                type_mask;      // Type of cache being disabled
    uint32_t                start_addr;     // Start address of cache being disabled
    flash_non_cached_size_t  size;          // Size of cache being disabled
} flash_non_cached_t;
```

2.9.2 Definitions of Arguments that Vary Depending on Flash Memory Functionality and Capacity

The actual definitions of some arguments vary depending on flash memory functionality and capacity.

Argument definitions that are applicable to the RX231, RX64M, and RX72M MCUs are presented here as examples.

File name: r_flash_rx\src\targets\rx231\r_flash_rx231.h

```
/* Definitions related to flash memory block counts, block sizes, minimum programming
sizes, block numbers, and addresses */

- omitted -

#define FLASH_NUM_BLOCKS_DF           (8)
#define FLASH_DF_MIN_PGM_SIZE         (1)
#define FLASH_CF_MIN_PGM_SIZE         (8)

#define FLASH_CF_BLOCK_SIZE           (2048)
#define FLASH_DF_BLOCK_SIZE           (1024)
#define FLASH_DF_FULL_SIZE            (FLASH_NUM_BLOCKS_DF*FLASH_DF_BLOCK_SIZE)
#define FLASH_DF_FULL_PGM_SIZE        (FLASH_DF_FULL_SIZE-FLASH_DF_MIN_PGM_SIZE)
#define FLASH_DF_LAST_VALID_ADDR      (FLASH_DF_BLOCK_INVALID-1)
#define FLASH_DF_HIGHEST_VALID_BLOCK  (FLASH_DF_BLOCK_INVALID-FLASH_DF_BLOCK_SIZE)


#define FLASH_NUM_BLOCKS_CF           (MCU_ROM_SIZE_BYTES / FLASH_CF_BLOCK_SIZE)
#define FLASH_CF_FULL_SIZE            (FLASH_NUM_BLOCKS_CF*FLASH_CF_BLOCK_SIZE)
#define FLASH_CF_LOWEST_VALID_BLOCK   (FLASH_CF_BLOCK_INVALID + 1)
#define FLASH_CF_LAST_VALID_ADDR      (FLASH_CF_LOWEST_VALID_BLOCK)

- omitted -

typedef enum _flash_block_address
{
    FLASH_CF_BLOCK_END     = 0xFFFFFFFF,    /*   Top  of the CS */
    FLASH_CF_BLOCK_0       = 0xFFFFF800,    /*   2KB: 0xFFFFF800 - 0xFFFFFFFF */

- omitted -

    FLASH_CF_BLOCK_255     = 0xFFF80000,    /*   2KB: 0xFFF80000 - 0xFFF807FF */
    FLASH_CF_BLOCK_INVALID = (FLASH_CF_BLOCK_255 - 1),
#endif

- omitted -

    FLASH_DF_BLOCK_0       = 0x00100000,    /*   1KB: 0x00100000 - 0x001003ff */

- omitted -

    FLASH_DF_BLOCK_7       = 0x00101C00,    /* 1KB: 0x00101C00 - 0x00101fff */
    FLASH_DF_BLOCK_INVALID = 0x00102000     /* 1KB: Can't write beyond 0x00101fff */
} flash_block_address_t;

- omitted -
```

File name: r_flash_rx\src\targets\rx64m\r_flash_rx64m.h

```
/* Definitions related to flash memory block counts, block sizes, minimum programming
sizes, block numbers, and addresses */

- omitted -

#if (MCU_CFG_PART_MEMORY_SIZE == 0x15 )
 #define FLASH_NUM_BLOCKS_CF (134)
#elif (MCU_CFG_PART_MEMORY_SIZE == 0x13 )
 #define FLASH_NUM_BLOCKS_CF (102)
#elif (MCU_CFG_PART_MEMORY_SIZE == 0x10 )
 #define FLASH_NUM_BLOCKS_CF (86)
#elif (MCU_CFG_PART_MEMORY_SIZE == 0xF )
 #define FLASH_NUM_BLOCKS_CF (70)
#endif

#define FLASH_NUM_BLOCKS_DF        (1024)
#define FLASH_DF_MIN_PGM_SIZE      (4)
#define FLASH_CF_MIN_PGM_SIZE      (256)

#define FLASH_CF_SMALL_BLOCK_SIZE  (8192)
#define FLASH_CF_MEDIUM_BLOCK_SIZE (32768)
#define FLASH_DF_BLOCK_SIZE        (64)
#define FLASH_DF_HIGHEST_VALID_BLOCK    (FLASH_DF_BLOCK_INVALID - FLASH_DF_BLOCK_SIZE)

- omitted -

typedef enum _flash_block_address
{
    FLASH_CF_BLOCK_END     = 0xFFFFFFFF,    /*   End of Code Flash Area       */
    FLASH_CF_BLOCK_0       = 0xFFFFE000,    /*   8KB: 0xFFFFE000 - 0xFFFFFFFF */

- omitted -

    FLASH_CF_BLOCK_133     = 0xFFC00000,    /*  32KB: 0xFFC00000 - 0xFFC07FFF */
    FLASH_CF_BLOCK_INVALID = (FLASH_CF_BLOCK_133 - 1),  // 0x15 parts 4M ROM
#endif

- omitted -

    FLASH_DF_BLOCK_0       = 0x00100000,    /*   64B: 0x00100000 - 0x0010003F */

- omitted -

    FLASH_DF_BLOCK_1023    = 0x0010FFC0,    /*   64B: 0x0010FFC0 - 0x0010FFFF */
    FLASH_DF_BLOCK_INVALID = 0x00110000     /*   Block 1023 + 64 bytes */}
flash_block_address_t;

- omitted -
```

File name: r_flash_rx\src\targets\rx72m\r_flash_rx72m.h

```
/* Definitions related to flash memory block counts, block sizes, minimum programming
sizes, block numbers, and addresses */

- omitted -

#if (MCU_CFG_PART_MEMORY_SIZE == 0xD)
    #if FLASH_IN_DUAL_BANK_MODE
        #define FLASH_NUM_BLOCKS_CF (30+8)  // 1 Mb per bank dual mode
    #else
        #define FLASH_NUM_BLOCKS_CF (62+8)  // 2 Mb linear mode
    #endif
#elif (MCU_CFG_PART_MEMORY_SIZE == 0x17)
    #if FLASH_IN_DUAL_BANK_MODE
        #define FLASH_NUM_BLOCKS_CF (62+8)  // 2 Mb per bank dual mode
    #else
        #define FLASH_NUM_BLOCKS_CF (126+8) // 4 Mb linear mode
    #endif
#endif


#define FLASH_NUM_BLOCKS_DF                (512)
#define FLASH_DF_MIN_PGM_SIZE              (4)
#define FLASH_CF_MIN_PGM_SIZE              (128)

#define FLASH_CF_SMALL_BLOCK_SIZE          (8192)
#define FLASH_CF_MEDIUM_BLOCK_SIZE         (32768)
#define FLASH_CF_LO_BANK_SMALL_BLOCK_ADDR  (FLASH_CF_BLOCK_77)
#define FLASH_CF_LOWEST_VALID_BLOCK        (FLASH_CF_BLOCK_INVALID + 1)
#define FLASH_DF_BLOCK_SIZE                (64)
#define FLASH_DF_HIGHEST_VALID_BLOCK       (FLASH_DF_BLOCK_INVALID - FLASH_DF_BLOCK_SIZE)

- omitted -
```

```
- omitted -

typedef enum _flash_block_address
{
#ifndef FLASH_IN_DUAL_BANK_MODE             /* LINEAR MODE */
    FLASH_CF_BLOCK_END     = 0xFFFFFFFF,    /*   End of Code Flash Area   */
    FLASH_CF_BLOCK_0       = 0xFFFFE000,    /*   8KB: 0xFFFFE000 - 0xFFFFFFFF   */

- omitted -

    FLASH_CF_BLOCK_69   = 0xFFE00000,      /*  32KB: 0xFFE00000 - 0xFFE07FFF */
#if   MCU_CFG_PART_MEMORY_SIZE == 0x0D     /*   'D' parts 2 Mb ROM */
    FLASH_CF_BLOCK_INVALID   = (FLASH_CF_BLOCK_69 - 1),
#else
    FLASH_CF_BLOCK_70   = 0xFFDF8000,      /*  32KB: 0xFFDF8000 - 0xFFDFFFFF */

- omitted -

    FLASH_CF_BLOCK_133  = 0xFFC00000,      /*  32KB: 0xFFC00000 - 0xFFC07FFF */
    FLASH_CF_BLOCK_INVALID  = (FLASH_CF_BLOCK_133 - 1), /* 'N' parts 4 Mb ROM */
#endif // > 2M

#else                                      /* DUAL MODE */
    FLASH_CF_BLOCK_END  = 0xFFFFFFFF,      /*   End of Code Flash Area       */
    FLASH_CF_HI_BANK_HI_ADDR = FLASH_CF_BLOCK_END,
    FLASH_CF_BLOCK_0    = 0xFFFFE000,      /*   8KB: 0xFFFFE000 - 0xFFFFFFFF */

- omitted -

    FLASH_CF_BLOCK_69   = 0xFFE00000,      /*  32KB: 0xFFE00000 - 0xFFE07FFF */
    FLASH_CF_HI_BANK_LO_ADDR = FLASH_CF_BLOCK_69,
#endif
    FLASH_CF_LO_BANK_HI_ADDR = 0xFFDFFFFF,  /* START OF NEXT BANK */

    FLASH_CF_BLOCK_70        = 0xFFDFE000,  /*   8KB: 0xFFDFE000 - 0xFFDFFFFF */

- omitted -

    FLASH_CF_BLOCK_139     = 0xFFC00000,    /*  32KB: 0xFFC00000 - 0xFFC07FFF   */
    FLASH_CF_LO_BANK_LO_ADDR     = FLASH_CF_BLOCK_139,
    FLASH_CF_BLOCK_INVALID = (FLASH_CF_BLOCK_139 - 1),
#endif // 32 blocks for 4M only
#endif // DUAL MODE

    FLASH_DF_BLOCK_0       = 0x00100000,    /*   64B: 0x00100000 - 0x0010003F   */

- omitted -

    FLASH_DF_BLOCK_511     = 0x00107FC0,    /*   64B: 0x00107FC0 - 0x00107FFF   */
    FLASH_DF_BLOCK_INVALID = 0x00108000     /*   Block 511 + 64 bytes     */
} flash_block_address_t;

- omitted -
```

These definitions are used as the arguments of module API functions. Refer to the descriptions and examples of API functions in section 3 for details on actual usage.

2.10 Return Values

This shows the different values API functions can return. This enumeration is described in the API function prototype declarations as well as in “r_flash_rx_if.h”.

```
/* FLASH FIT module return value definitions */
typedef enum _flash_err
{
    FLASH_SUCCESS = 0,
    FLASH_ERR_BUSY,         // Flash module is in busy state
    FLASH_ERR_ACCESSW,      // Access window error
    FLASH_ERR_FAILURE,      // Flash operation, program, erase process, or other error
    FLASH_ERR_CMD_LOCKED,   // Flash module is in command lock state
    FLASH_ERR_LOCKBIT_SET,  // Error during program or erase process due to lockbit
    FLASH_ERR_FREQUENCY,    // Illegal frequency specified
    FLASH_ERR_BYTES,        // Invalid number of bytes specified
    FLASH_ERR_ADDRESS,      // Invalid address or non-program boundary address specified
    FLASH_ERR_BLOCKS,       // The “number of blocks” argument is invalid
    FLASH_ERR_PARAM,        // Illegal parameter specified
    FLASH_ERR_NULL_PTR,     // NULL specified
    FLASH_ERR_UNSUPPORTED,  // Unsupported command specified
    FLASH_ERR_SECURITY,     // Error caused by access window protection
    FLASH_ERR_TIMEOUT,      // Timeout occurred
    FLASH_ERR_ALREADY_OPEN, // Open() called twice without calling Close().
    FLASH_ERR_HOCO          // The HOCO is not running.
} flash_err_t;
```

2.11 Callback Function

This module calls the callback function specified by the user at timings of FRDYI and FIFERR interrupt generations.

The callback function is configured by storing the address of the user's function in the "pcallback" structure member as described in "2.9 Parameters". When the callback function is called, variables storing the constants described in Table 2.2 through Table 2.5 are passed as arguments.

Use a void pointer variable as the argument of the callback function as arguments are passed as void pointers.

Use values inside the callback function by casting them.

Refer to Example 1 in section 3.6 for example implementations of the callback function.

Table 2.2 Flash Type 1 Callback Function Arguments (enum flash_interrupt_event_t)

Constant Definitions	Description
FLASH_INT_EVENT_ERASE_COMPLETE	Called by the FRDYI interrupt processing and indicates completion of the erase process.
FLASH_INT_EVENT_WRITE_COMPLETE	Called by the FRDYI interrupt processing and indicates completion of the program process.
FLASH_INT_EVENT_BLANK	Called by the FRDYI interrupt processing and indicates that the blank check resulted in a blank state.
FLASH_INT_EVENT_NOT_BLANK	Called by the FRDYI interrupt processing and indicates that the blank check resulted in a non-blank state.
FLASH_INT_EVENT_TOGGLE_STARTUPAREA	Called by the FRDYI interrupt processing and indicates completion of swapping the startup region.
FLASH_INT_EVENT_SET_ACCESSWINDOW	Called by the FRDYI interrupt processing and indicates completion of configuring the access window.
FLASH_INT_EVENT_ERR_FAILURE	Called by the FRDYI interrupt processing and indicates an error occurred during the program or erase process.

Table 2.3 Flash Type 3 Callback Function Arguments (enum flash_interrupt_event_t)

Constant Definitions	Description
FLASH_INT_EVENT_ERASE_COMPLETE	Called by the FRDYI interrupt processing and indicates completion of the erase process.
FLASH_INT_EVENT_WRITE_COMPLETE	Called by the FRDYI interrupt processing and indicates completion of the program process.
FLASH_INT_EVENT_BLANK*1	Called by the FRDYI interrupt processing and indicates that the blank check resulted in a blank state.
FLASH_INT_EVENT_NOT_BLANK*1	Called by the FRDYI interrupt processing and indicates that the blank check resulted in a non-blank state.
FLASH_INT_EVENT_LOCKBIT_WRITTEN	Called by the FRDYI interrupt processing and indicates the setting of lockbit.
FLASH_INT_EVENT_LOCKBIT_PROTECTED	Called by the FRDYI interrupt processing and indicates that lockbit protection is enabled.
FLASH_INT_EVENT_LOCKBIT_NON_PROTECTED	Called by the FRDYI interrupt processing and indicates that lockbit protection is disabled.
FLASH_INT_EVENT_ERR_DF_ACCESS	Called by the FIFERR interrupt processing and indicates an access violation of data flash memory.
FLASH_INT_EVENT_ERR_CF_ACCESS	Called by the FIFERR interrupt processing and indicates an access violation of code flash memory.
FLASH_INT_EVENT_ERR_CMD_LOCKED	Called by the FIFERR interrupt processing and indicates that commands are locked.
FLASH_INT_EVENT_ERR_LOCKBIT_SET	Called by the FIFERR interrupt processing and indicates an error in a region with lockbit protection.
FLASH_INT_EVENT_ERR_FAILURE	Called by the FIFERR interrupt processing and indicates an error occurred during the program or erase process.

*1 The blank check process is only performed on data flash memory.

Table 2.4 Flash Type 4 Callback Function Arguments (enum flash_interrupt_event_t)

Constant Definitions	Description
FLASH_INT_EVENT_ERASE_COMPLETE	Called by the FRDYI interrupt processing and indicates completion of the erase process.
FLASH_INT_EVENT_WRITE_COMPLETE	Called by the FRDYI interrupt processing and indicates completion of the program process.
FLASH_INT_EVENT_BLANK*1	Called by the FRDYI interrupt processing and indicates that the blank check resulted in a blank state.
FLASH_INT_EVENT_NOT_BLANK*1	Called by the FRDYI interrupt processing and indicates that the blank check resulted in a non-blank state.
FLASH_INT_EVENT_TOGGLE_STARTUPAREA	Called by the FRDYI interrupt processing and indicates completion of swapping the startup region.
FLASH_INT_EVENT_SET_ACCESSWINDOW	Called by the FRDYI interrupt processing and indicates completion of configuring the access window.
FLASH_INT_EVENT_TOGGLE_BANK	Called by the FRDYI interrupt processing and indicates completion of swapping of the startup bank.
FLASH_INT_EVENT_ERR_DF_ACCESS	Called by the FIFERR interrupt processing and indicates an access violation of data flash memory.
FLASH_INT_EVENT_ERR_CF_ACCESS	Called by the FIFERR interrupt processing and indicates an access violation of code flash memory.
FLASH_INT_EVENT_ERR_SECURITY	Called by the FIFERR interrupt processing and indicates a reprogramming of a write-protected region of an access window.
FLASH_INT_EVENT_ERR_CMD_LOCKED	Called by the FIFERR interrupt processing and indicates that commands are locked.
FLASH_INT_EVENT_ERR_FAILURE	Called by the FIFERR interrupt processing and indicates an error occurred during the program or erase process.

*1 The blank check process is only performed on data flash memory.

Table 2.5 Flash Type 5 Callback Function Arguments (enum flash_interrupt_event_t)

Constant Definitions	Description
FLASH_INT_EVENT_ERASE_COMPLETE	Called by the FRDYI interrupt processing and indicates completion of the erase process.
FLASH_INT_EVENT_WRITE_COMPLETE	Called by the FRDYI interrupt processing and indicates completion of the program process.
FLASH_INT_EVENT_BLANK*1	Called by the FRDYI interrupt processing and indicates that the blank check resulted in a blank state.
FLASH_INT_EVENT_NOT_BLANK*1	Called by the FRDYI interrupt processing and indicates that the blank check resulted in a non-blank state.
FLASH_INT_EVENT_TOGGLE_STARTUPAREA	Called by the FRDYI interrupt processing and indicates completion of swapping the startup region.
FLASH_INT_EVENT_SET_ACCESSWINDOW	Called by the FRDYI interrupt processing and indicates completion of configuring the access window.
FLASH_INT_EVENT_TOGGLE_BANK	Called by the FRDYI interrupt processing and indicates completion of swapping of the startup bank.
FLASH_INT_EVENT_ERR_DF_ACCESS	Called by the FIFERR interrupt processing and indicates an access violation of data flash memory.
FLASH_INT_EVENT_ERR_CF_ACCESS	Called by the FIFERR interrupt processing and indicates an access violation of code flash memory.
FLASH_INT_EVENT_ERR_CMD_LOCKED	Called by the FIFERR interrupt processing and indicates that commands are locked.
FLASH_INT_EVENT_ERR_FAILURE	Called by the FIFERR interrupt processing and indicates an error occurred during the program or erase process.

*1 The blank check process is only performed on data flash memory.

2.12 Adding the FIT Module to Your Project

This module must be added to each project in which it is used. Renesas recommends the method using the Smart Configurator described in (1), (3) or (5) below. However, the Smart Configurator only supports some RX devices. Please use the methods of (2) or (4) for RX devices that are not supported by the Smart Configurator.

(1) Adding the FIT module to your project using Smart Configurator in e^2 studio
By using the Smart Configurator in e^2 studio, the FIT module is automatically added to your project. Refer to “RX Smart Configurator User’s Guide: e^2 studio (R20AN0451)” for details.

(2) Adding the FIT module to your project using FIT Configurator in e^2 studio
By using the FIT Configurator in e^2 studio, the FIT module can be automatically added to your project. Refer to “Adding Firmware Integration Technology Modules to Projects (R01AN1723)” for details.

(3) Adding the FIT module to your project using Smart Configurator in CS+
By using the Smart Configurator Standalone version in CS+, the FIT module is automatically added to your project. Refer to “RX Smart Configurator User’s Guide: CS+ (R20AN0470)” for details.

(4) Adding the FIT module to your project in CS+
In CS+, manually add the FIT module to your project. Refer to “Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)” for details.

(5) Adding the FIT module to your project using Smart Configurator in IAREW
By using the Smart Configurator Standalone version, the FIT module is automatically added to your project. Refer to “RX Smart Configurator User Guide: IAREW (R20AN0535)” for details.

2.13 “for”, “while” and “do while” statements

In this module, “for”, “while” and “do while” statements (loop processing) are used in processing to wait for register to be reflected and so on. For these loop processing, comments with “WAIT_LOOP” as a keyword are described. Therefore, if user incorporates fail-safe processing into loop processing, user can search the corresponding processing with “WAIT_LOOP”.

The following shows example of description.

```
while statement example :
/* WAIT_LOOP */
while(0 == SYSTEM.OSCOVFSR.BIT.PLOVF)
{
    /* The delay period needed is to make sure that the PLL has stabilized. */
}

for statement example :
/* Initialize reference counters to 0. */
/* WAIT_LOOP */
for (i = 0; i < BSP_REG_PROTECT_TOTAL_ITEMS; i++)
{
    g_protect_counters[i] = 0;
}

do while statement example :
/* Reset completion waiting */
do
{
    reg = phy_read(ether_channel, PHY_REG_CONTROL);
    count++;
} while ((reg & PHY_CONTROL_RESET) && (count < ETHER_CFG_PHY_DELAY_RESET)); /* WAIT_LOOP */
```

2.14 Blocking Mode and Non-blocking Mode

API functions in this module operate in blocking and non-blocking modes.

Blocking mode does not return until the API function has finished processing the flash memory.

Non-blocking mode returns without waiting for the API function to finish processing the flash memory.

2.14.1 Using in Blocking Mode

When using this module in blocking mode, set configuration options as shown below. Set FLASH_CFG_DATA_FLASH_BGO and FLASH_CFG_CODE_FLASH_BGO to the same value.

- FLASH_CFG_DATA_FLASH_BGO: 0
- FLASH_CFG_CODE_FLASH_BGO: 0

2.14.2 Using in Non-blocking Mode

When using this module in non-blocking mode, set configuration options as shown below. Set FLASH_CFG_DATA_FLASH_BGO and FLASH_CFG_CODE_FLASH_BGO to the same value.

- FLASH_CFG_DATA_FLASH_BGO: 1
- FLASH_CFG_CODE_FLASH_BGO: 1

Users should not access flash memory regions until flash memory process is complete. If accessed, the flash sequencer generates an error preventing processing from completing properly.

Notification of the result of flash memory processing is sent via the callback function. Register the callback function in advance by executing R_FLASH_Open() and specifying the FLASH_CMD_SET_BGO_CALLBACK command for the argument of R_FLASH_Control(). (Refer to section 3.6 for details.)
Table 2.6 describes the API functions that send notification of processing results via the callback function.

Table 2.6 API Functions that Send Notifications of Processing Results via the Callback Function

API Function	Processing Result Notification via the Callback Function
R_FLASH_Open(), R_FLASH_Close(), R_FLASH_GetVersion()	Does not send notifications
R_FLASH_Erase(), R_FLASH_BLankCheck(), R_FLASH_Write()	Sends notifications
R_FLASH_Control()	Sends notifications for the following commands: • FLASH_CMD_SWAPFLAG_TOGGLE • FLASH_CMD_ACCESSWINDOW_SET • FLASH_CMD_LOCKBIT_READ • FLASH_CMD_LOCKBIT_WRITE • FLASH_CMD_BANK_TOGGLE

A FRDYI or FIFERR interrupt occurs when flash memory processing completes. The callback functions registered by each interrupt are called. Events indicating the completion status are passed to the callback function. Refer to section 2.11 for details on callback functions.

2.15 Region Protection via Access Windows and Lockbits

Regions of each MCU flash memory can be protected by using the access window or lockbit to prevent regions of code flash memory from being unintentionally rewritten. API functions in this module support the following features.

2.15.1 Access Window-based Region Protection

Regions can be protected by using access windows in Flash Type 1, 4 and 5 products.

Access window configurations include specification of the start and end addresses of the blocks defining the region to which the access window is applied.

The region defined by the start and end addresses of the blocks configuring the region to which an access window is applied can be reprogrammed. Make sure to note that it is the other regions that will be write-protected.

All regions are reprogrammable at the time of shipment as access windows are not set by default.

Use R_FLASH_Control() to configure access windows. Refer to section 3.6 for details.

2.15.2 Lockbit-based Region Protection

Regions can be protected by using lockbits in Flash Type 3 products.

Lockbit configurations include the start address of the blocks defining the region to which lockbit is applied, the number of blocks, and specification of whether lockbit protection is enabled or disabled.

The region defined by the number of specified blocks starting from the specified address configuring the region to which lockbit is applied will be write-protected. Make sure to note that other regions will not be write-protected.

All regions are reprogrammable at the time of shipment as lockbit is not configured by default.

Use R_FLASH_Control() to configure lockbits. Refer to section 3.6 for details.

2.16 Usage Combined with Existing User Projects

Using the BSP startup disable function, this module can be used in combination with existing user projects.

The BSP startup disable function is a function to add and use this module and other peripheral FIT modules to an existing user project without creating a new project.

BSP and this module (if necessary, other peripheral FIT modules) are incorporated into the existing user project. Even though it is necessary to incorporate BSP, since all startup processing performed by the BSP become disabled, this module and other peripheral FIT modules can be used in combination with startup processing of the existing user project.

There are some settings and notes for using the BSP startup disable function. Refer to “RX Family Board Support Package Module Using Firmware Integration Technology (R01AN1685)” for details.

2.17 Reprogramming Flash Memory

Code required to perform flash memory reprogramming is allocated in code flash memory as shown in Figure 2.1 (left figure). As shown in Figure 2.1 (right figure), running this code in code flash memory enables reprogramming of the target regions in flash memory.

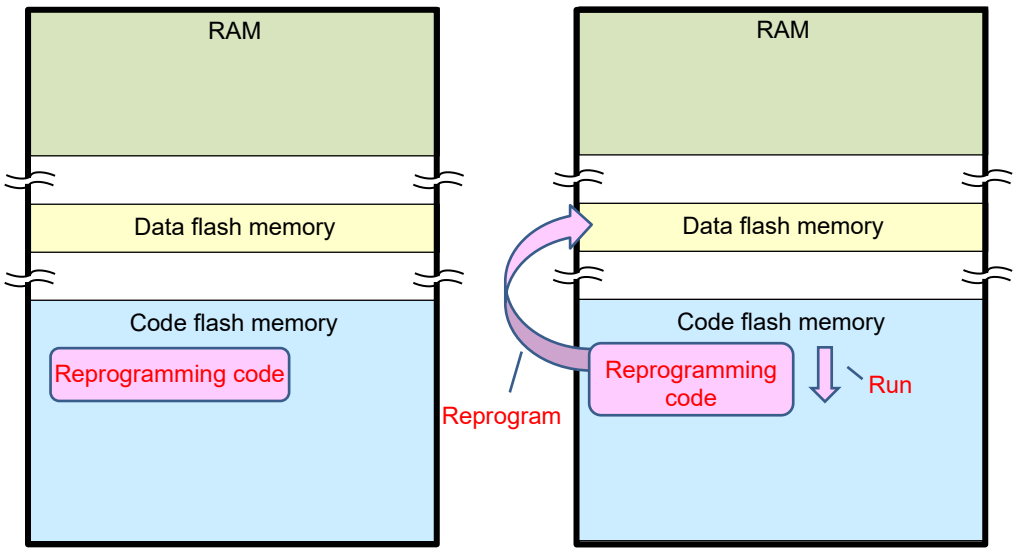


Figure 2.1 Location of Code Required to Perform Flash Memory Reprogramming and Reprogramming Process

Note that, as shown in Figure 2.2, the region containing the code required to perform flash memory reprogramming cannot be reprogrammed.

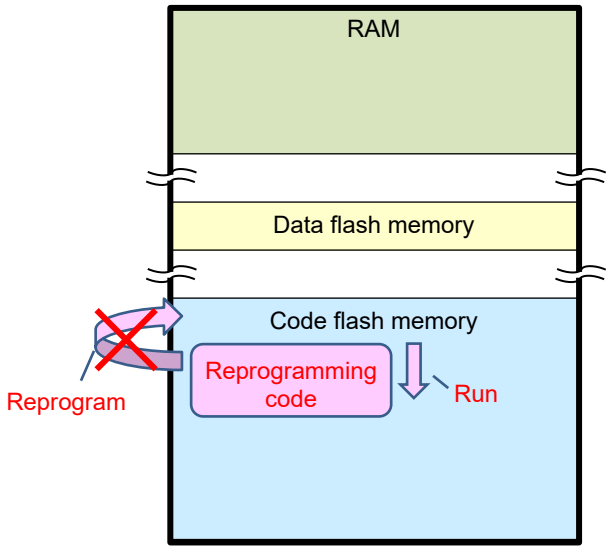


Figure 2.2 Reprogramming of Region Containing Code Required to Perform Flash Memory Reprogramming

2.16.1 through 2.16.3 describe the available methods of reprogramming code flash memory.

2.17.1 Reprogramming Code Flash Memory by Running Code on the RAM

As shown in Figure 2.3, copying to and then running the code required to reprogram flash memory in RAM enables reprogramming of regions in code flash memory.*1*2

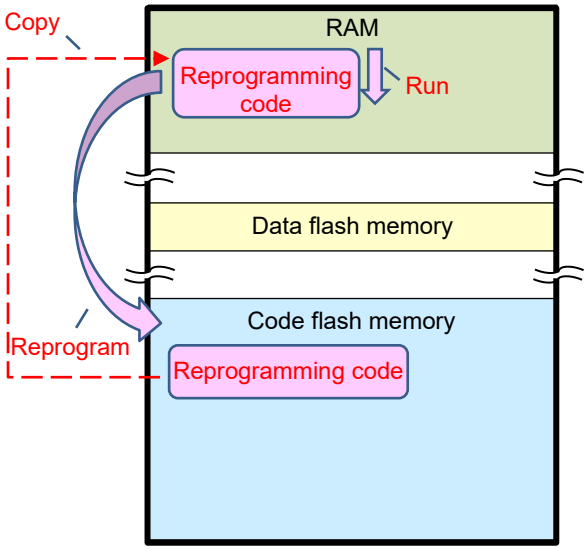


Figure 2.3 Reprogramming Code Flash Memory by Running Code on the RAM

Configure the configuration options of this module as follows.

- FLASH_CFG_CODE_FLASH_ENABLE: 1
- FLASH_CFG_CODE_FLASH_RUN_FROM_ROM: 0

This module of Rev. 4.00 or later supports multiple compilers. To use this module, different settings are required for each compiler. For details of the settings appropriate for the compiler to be used, refer to section 5.3.

*1 The code required to perform flash memory reprogramming is copied to RAM using the R_FLASH_Open() function of this module.
It is necessary to reallocate interrupt vector tables and interrupt handlers to the RAM for interrupts that may occur while the code flash memory is being reprogrammed. For details, refer to Example 1 in section 3.6.

*2 Products with multiple regions of code flash memory can reprogram code flash memory without using RAM. Refer to section 2.16.2 for details.

2.17.2 Reprogramming Code Flash Memory by Running Code on the Code Flash Memory

Table 2.7 shows the products that support reprogramming of code flash memory by running code on the code flash memory. These products support this capability by having multiple regions of code flash memory.

Table 2.7 Products with Multiple Regions of Code Flash Memory

Flash Type	Products with Multiple Regions of Code Flash Memory
3	RX64M*1, RX71M*1
4	RX651*2, RX65N*2, RX66N, RX671, RX72M, RX72N
5	RX26T*3

*1 Products with at least 2.5 Mbytes of code flash memory
*2 Products with at least 1.5 Mbytes of code flash memory
*3 Products with at least 512 Kbytes of code flash memory

The capacity of code flash memory regions varies depending on the MCU. As such, the size and boundaries of code flash memory regions are dependent on the MCU. Refer to the hardware section of the applicable user's manual for details.

As shown in Figure 2.4, code flash memory can be reprogrammed in products with multiple regions of code flash memory as long as the region is not the region containing the code required to perform flash memory reprogramming.

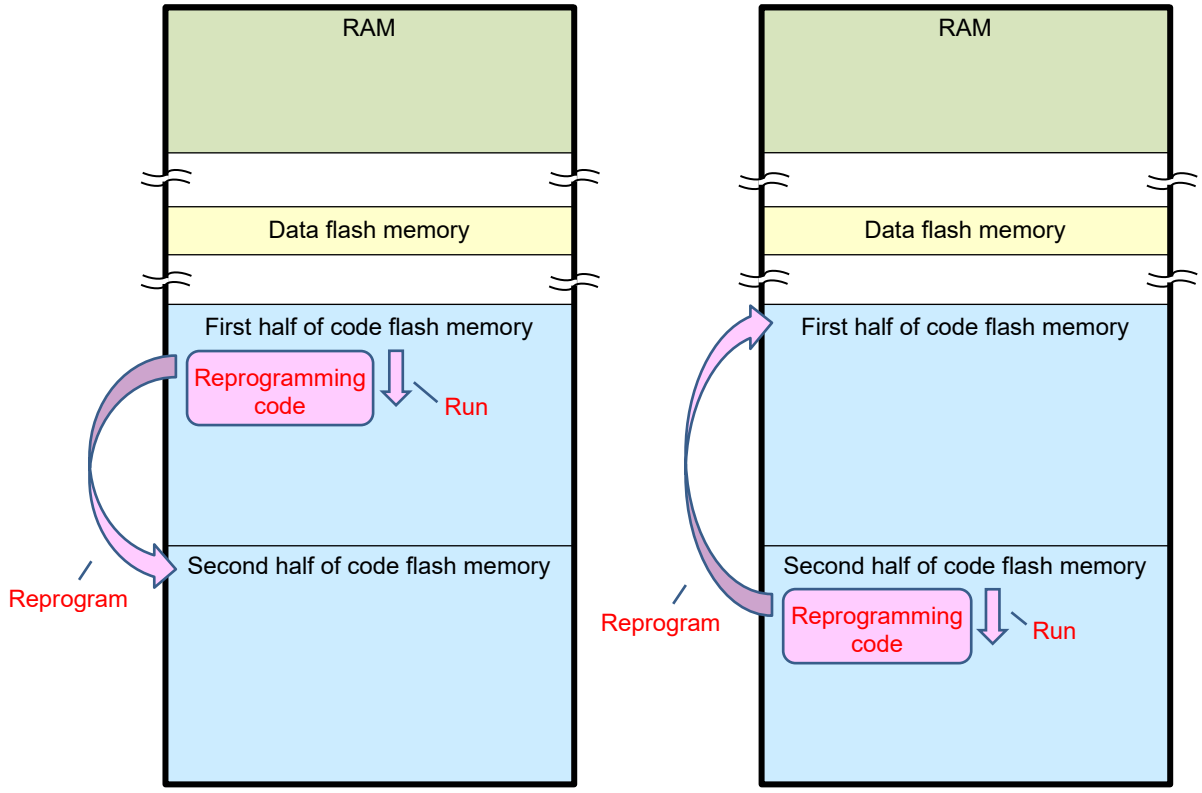


Figure 2.4 Reprogramming Code Flash Memory by Running Code on the Code Flash Memory

Configure the configuration options of this module as follows.

- FLASH_CFG_CODE_FLASH_ENABLE: 1
- FLASH_CFG_CODE_FLASH_RUN_FROM_ROM: 1

2.17.3 Reprogramming Code Flash Memory by Utilizing the Dual Bank Function

Flash Type 4 products with at least 1.5 Mbytes of code flash memory and Flash Type 5 products with at least 512 Kbytes of code flash memory have the dual bank function.

The dual bank function includes bank mode swapping and the startup bank selection function so that programs can be updated while user programs are still running.

Bank mode swapping features a linear mode where the user region in code flash memory is handled as one and a dual mode where it is handled as two bank regions.

To use dual banks, it is necessary to change the constant defined in the configuration file (r_bsp_config.h) of BSP as follows.

- BSP_CFG_CODE_FLASH_BANK_MODE: 1 → 0
  The default setting is “1”. To operate in dual bank mode, set this constant to “0”.

The startup bank selection function is a function to select the bank region used to start program when operating in dual mode.

It is defined in the configuration file (r_bsp_config.h) of BSP as follows.

- BSP_CFG_CODE_FLASH_START_BANK: 0
  The default setting is “0”. The bank 0 is selected as the startup bank.

You can change the startup bank in dual mode by specifying the FLASH_CMD_BANK_TOGGLE command as the first argument of the R_FLASH_Control() function. Note that the swap of the startup bank does not take effect until the next MCU reset.

Sections 2.16.3.1 to 2.16.3.2 show how to reprogramming the code flash memory the configuration option of this module in combination with the dual bank function.

2.17.3.1 Reprogramming the other Bank different from the Startup Bank by Running Code on the Startup Bank

As shown in Figure 2.5 (left figure), the startup bank (bank 0) region containing the code required to perform flash memory reprogramming cannot be reprogrammed, but the other bank (bank 1) region can. By swapping banks, the other bank (bank 0) region can now be reprogrammed as the startup bank (bank 1) region now contains the code required to perform flash memory reprogramming as shown in Figure 2.5 (right figure).

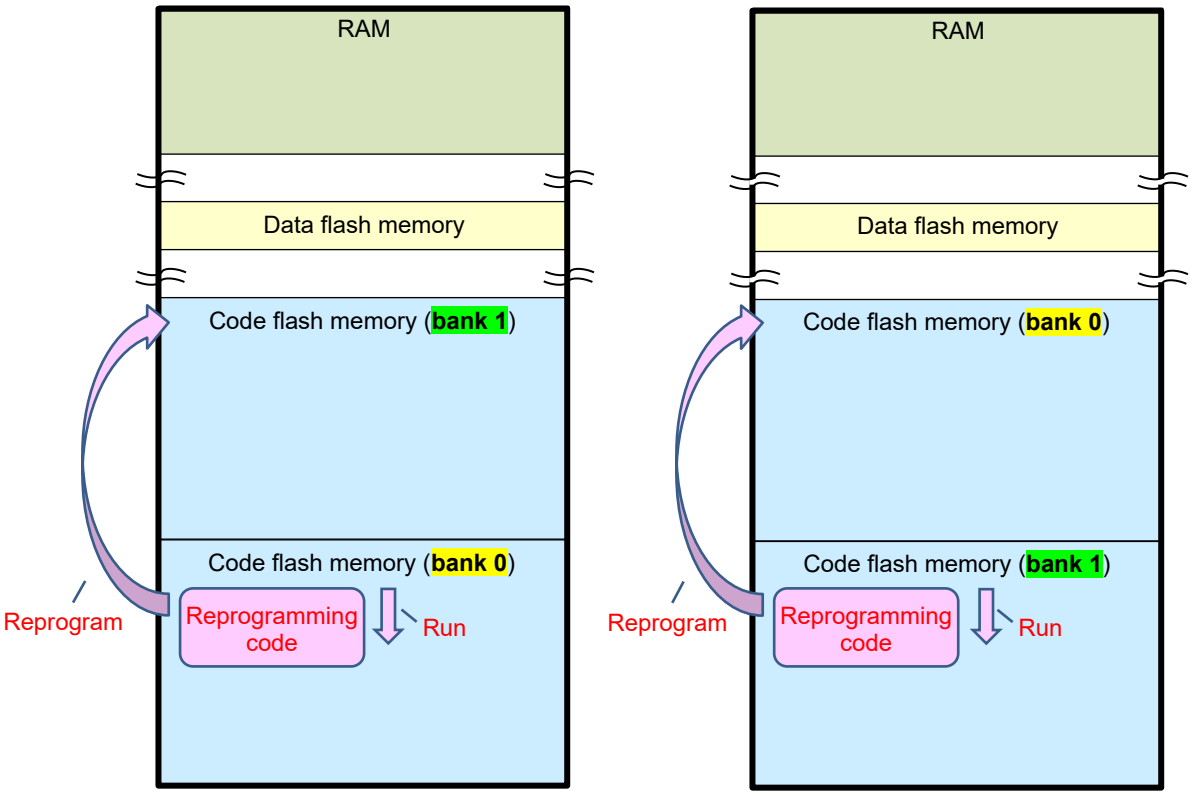


Figure 2.5 Reprogramming the other Bank different from the Startup Bank by Running Code on the Startup Bank

Configure the configuration options of this module as follows.

- FLASH_CFG_CODE_FLASH_ENABLE: 1
- FLASH_CFG_CODE_FLASH_RUN_FROM_ROM: 1

This module of Rev. 4.00 or later supports multiple compilers. To use this module, different settings are required for each compiler. For details of the settings appropriate for the compiler to be used, refer to section 5.3.

2.17.3.2 Reprogramming the Startup Bank and the other Bank by Running Code on the RAM

As shown in Figure 2.6 (left figure), copying to and then running the code required to reprogram flash memory in RAM enables reprogramming of regions in startup bank (bank 0) and the other bank (bank 1). By swapping banks, copying to and then running the code required to reprogram flash memory in RAM enables reprogramming of regions in startup bank (bank 1) and the other bank (bank 0) as shown in Figure 2.6 (right figure).

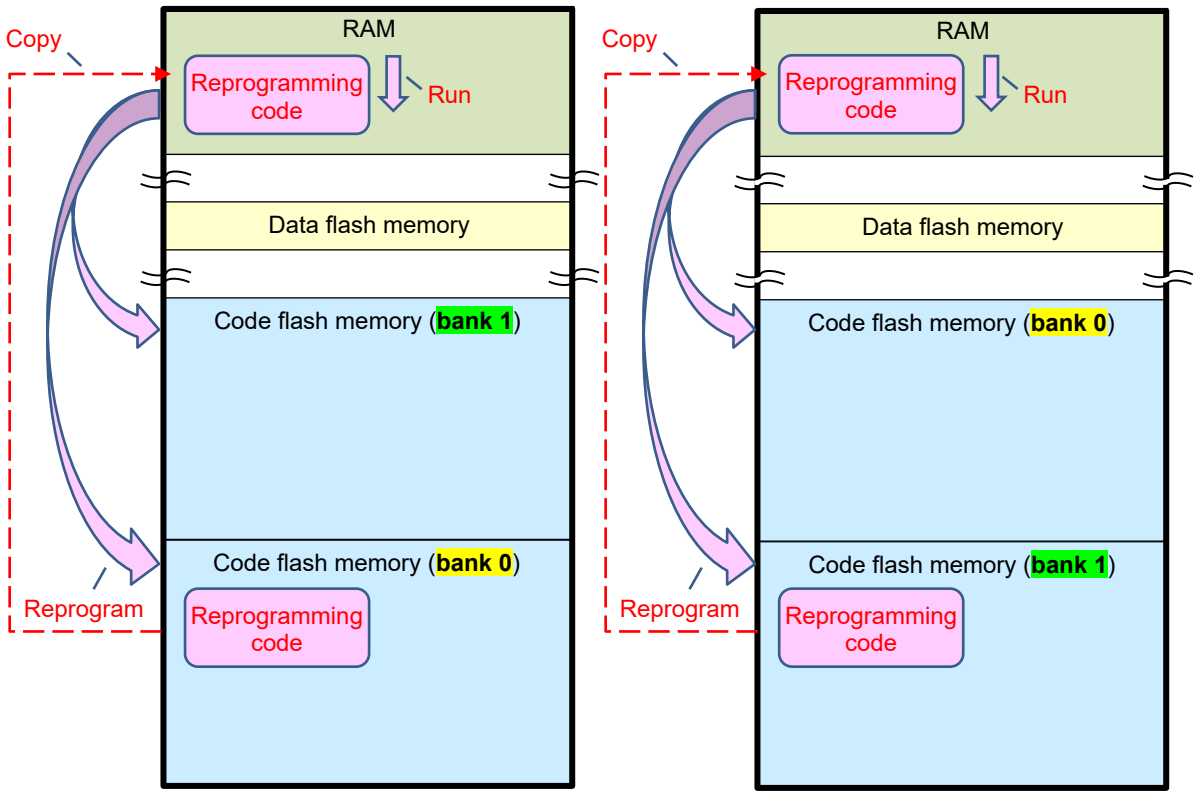


Figure 2.6 Reprogramming the Startup Bank and the other Bank by Running Code on the RAM

Configure the configuration options of this module as follows.

- FLASH_CFG_CODE_FLASH_ENABLE: 1
- FLASH_CFG_CODE_FLASH_RUN_FROM_ROM: 0

This module of Rev. 4.00 or later supports multiple compilers. To use this module, different settings are required for each compiler. For details of the settings appropriate for the compiler to be used, refer to section 5.3.

3. API Functions

3.1 R_FLASH_Open()

This API function initializes flash modules. Note that this function must be called before any other API function.

Format

```
flash_err_t R_FLASH_Open(void)
```

Parameters

None

Return Values

FLASH_SUCCESS	*/* Successfully initialized. */*
FLASH_ERR_BUSY	*/* A different flash memory process is being executed, try again later. */*
FLASH_ERR_ALREADY_OPEN	*/* Already open. Run R_FLASH_Close(). */*
FLASH_ERR_FREQUENCY	*/* The frequency setting of the Flash clock (FCLK) is invalid. */*
FLASH_ERR_HOCO	*/* The HOCO is not running. */*

Properties

Prototyped in file “r_flash_rx_if.h”.

Description

This API function performs the following processing.

1. Preparing the code required to perform flash memory reprogramming
   The code required to perform flash memory reprogramming is allocated depending on the configuration of configuration options as described in Table 3.1.

Table 3.1 Code Allocations in Relation to Configuration of Configuration Options

Configuration Option	Setting	Code Allocation
FLASH_CFG_CODE_FLASH_ENABLE	0	Code that processes flash memory is allocated in code flash memory. However, this code does not include code that processes code flash memory.
FLASH_CFG_CODE_FLASH_ENABLE	1	Code that processes flash memory is copied into RAM. *1
FLASH_CFG_CODE_FLASH_RUN_FROM_ROM	0	
BSP_CFG_CODE_FLASH_BANK_MODE	1	
FLASH_CFG_CODE_FLASH_ENABLE	1	Code that processes flash memory is allocated in code flash memory.
FLASH_CFG_CODE_FLASH_RUN_FROM_ROM	1	
BSP_CFG_CODE_FLASH_BANK_MODE	1	
FLASH_CFG_CODE_FLASH_ENABLE	1	Code that processes flash memory is copied into RAM. *1 Code that provides the dual bank and access window is copied into RAM. *1
FLASH_CFG_CODE_FLASH_RUN_FROM_ROM	0	
BSP_CFG_CODE_FLASH_BANK_MODE	0	
FLASH_CFG_CODE_FLASH_ENABLE	1	Code that processes flash memory is allocated in code flash memory. Code that provides the dual bank and access window is copied into RAM. *1
FLASH_CFG_CODE_FLASH_RUN_FROM_ROM	1	
BSP_CFG_CODE_FLASH_BANK_MODE	0	

*1 The functionality to reallocate interrupt vector tables or interrupt processing is not included in this API function.

2. Default flash sequencer configuration
   For Flash Type 3, 4 and 5 products, the flash sequencer processing clock notification register (FPCKAR) is set with the value of the BSP configuration option (BSP_FCLK_HZ) as the flash sequencer configuration.
   The data flash memory access frequency setting register (EEPFCLK) is also configured in the same manner for Flash Type 4 products with at least 1.5 Mbytes of code flash memory.
   For the RX64M and RX71M, FCU firmware required to use the flash sequencer is also copied to dedicated RAM (FCURAM).
3. Default interrupt configuration
   The interrupts described in section 2.4 are prohibited.

Reentrant

- Not allowed

Example

```
flash_err_t err;

/* Initialize the API. */
err = R_FLASH_Open();

/* Check for errors. */
if (FLASH_SUCCESS != err)
{
    . . .
}
```

Special Notes:

None

3.2 R_FLASH_Close()

This API function terminates flash module processing.

Format

```
flash_err_t R_FLASH_Close(void)
```

Parameters

None

Return Values

```
FLASH_SUCCESS       /* Successful termination of flash module processing. */
FLASH_ERR_BUSY      /* A different flash memory process is being executed,
                       try again later. */
```

Properties

Prototyped in file “r_flash_rx_if.h”.

Description

This API function terminates flash module processing by prohibiting the interrupt described in section 2.4 and setting the module to an uninitialized state.

Reentrant

- Not allowed

Example

```
flash_err_t err;

/* Close the driver */
err = R_FLASH_Close();

/* Check for errors. */
if (FLASH_SUCCESS != err)
{
    . . .
}
```

Special Notes:

None

3.3 R_FLASH_Erase()

This API function erases specified blocks in code flash memory or data flash memory.

Format

```
flash_err_t R_FLASH_Erase(
      flash_block_address_t   block_start_address,
      uint32_t                num_blocks
)
```

Parameters

block_start_address

Specifies the start address of the blocks to be erased.

"flash_block_address_t" defines the starting block address and block number.

"flash_block_address_t" is defined in "r_flash_rx\src\targets\<mcu>\r_flash_<mcu>.h".

num_blocks

Specifies the number of blocks to be erased.

With RX111, RX113, and RX130 products, make sure that regions specified by "block_start_address" and "num_blocks" are not larger than 256 Kbytes.

Return Values

FLASH_SUCCESS	*/* Successful completion of erase processing. In non-blocking mode, this indicates that erase processing has started. */*
FLASH_ERR_BLOCKS	*/* Specified number of blocks is invalid. */*
FLASH_ERR_ADDRESS	*/* Specified address is invalid. */*
FLASH_ERR_BUSY	*/* A different flash memory process is being executed, or the module is not initialized. */*
FLASH_ERR_FAILURE	*/* Erase processing failure. In non-blocking mode, the callback function is not registered. */*

Properties

Prototyped in file "r_flash_rx_if.h".

Description

Code flash memory and data flash memory is erased in blocks.

Table 3.2 describes the difference in block sizes by MCU group.

Table 3.2 Block Sizes by MCU Group

MCU Group	Code Flash Memory	Data Flash Memory*4
RX110	1 Kbyte*1	—*5
RX111	1 Kbyte*1	1 Kbyte
RX113	1 Kbyte*1	1 Kbyte
RX130	1 Kbyte*1	1 Kbyte
RX13T	1 Kbyte*1	1 Kbyte
RX140	2 Kbytes	256 bytes
RX230, RX231	2 Kbytes*1	1 Kbyte
RX23E-A, RX23E-B	2 Kbytes*1	1 Kbyte
RX23T	2 Kbytes*1	—*4
RX23W	2 Kbytes*1	1 Kbyte
RX24T	2 Kbytes*1	1 Kbyte
RX24U	2 Kbytes*1	1 Kbyte
RX260, RX261	2 Kbytes	256 bytes
RX26T	4 Kbytes, 16 Kbytes*2	64 bytes
RX64M	8 Kbytes, 32 Kbytes*3	64 bytes
RX65N, RX651	8 Kbytes, 32 Kbytes*3	64 bytes*6
RX660	8 Kbytes, 32 Kbytes*3	64 bytes
RX66N	8 Kbytes, 32 Kbytes*3	64 bytes
RX66T	8 Kbytes, 32 Kbytes*3	64 bytes
RX671	8 Kbytes, 32 Kbytes*3	64 bytes
RX71M	8 Kbytes, 32 Kbytes*3	64 bytes
RX72M	8 Kbytes, 32 Kbytes*3	64 bytes
RX72N	8 Kbytes, 32 Kbytes*3	64 bytes
RX72T	8 Kbytes, 32 Kbytes*3	64 bytes

*1 Defined as FLASH_CF_BLOCK_SIZE in the specific MCU definitions file (“r_flash_rx\src\targets\<mcu>\r_flash_<mcu>.h”).

*2 Contains both 4-Kb and 16-Kb blocks.
4-Kbytes blocks are defined as FLASH_CF_SMALL_BLOCK_SIZE while 16-Kbytes blocks are defined as FLASH_CF_MEDIUM_BLOCK_SIZE in the specific MCU definitions file (“r_flash_rx\src\targets\<mcu>\r_flash_<mcu>.h”).

*3 Contains both 8-Kb and 32-Kb blocks.
8-Kbytes blocks are defined as FLASH_CF_SMALL_BLOCK_SIZE while 32-Kbytes blocks are defined as FLASH_CF_MEDIUM_BLOCK_SIZE in the specific MCU definitions file (“r_flash_rx\src\targets\<mcu>\r_flash_<mcu>.h”).

*4 Defined as FLASH_DF_BLOCK_SIZE in the specific MCU definitions file (“r_flash_rx\src\targets\<mcu>\r_flash_<mcu>.h”).

*5 Does not contain any data flash memory.

*6 Products with no more than 1 Mbyte of code flash memory do not have data flash memory.

When this API function is used in non-blocking mode, FRDYI interrupt occurs after blocks for the specified number are erased, and then the callback function is called.

Reentrant

- Not allowed

Example

The first argument specifies the starting block address for the erase process.

The second argument specifies the number of blocks to be erased starting from the starting block address for the erase process.

The following code examples shows erase processing for flash memory with multiple blocks specified.

Note that the direction in which blocks are erased varies depending on whether erasing data flash memory or code flash memory and on differences in flash types.

```
flash_err_t err;

/* Common for Flash Type 1, 3, 4, and 5 products. */
/* Erases data flash memory blocks in order from smaller to larger block numbers starting
from block 5. */
/* The following code causes blocks 5 and 6 in data flash memory to be erased. */
err = R_FLASH_Erase(FLASH_DF_BLOCK_5, 2);

/* Check for errors. */
if (FLASH_SUCCESS != err)
{
    . . .
}

/* For Flash Type 1 products */
/* Erases code flash memory blocks in order from larger to smaller block numbers starting
from block 5. */
/* The following code causes blocks 4 and 5 in code flash memory to be erased. */
err = R_FLASH_Erase(FLASH_CF_BLOCK_5, 2);

/* Check for errors. */
if (FLASH_SUCCESS != err)
{
    . . .
}

/* For Flash Type 3, 4 and 5 products */
/* Erases code flash memory blocks in order from smaller to larger block numbers starting
from block 5. */
/* The following code causes blocks 5 and 6 in code flash memory to be erased. */
err = R_FLASH_Erase(FLASH_CF_BLOCK_5, 2);

/* Check for errors. */
if (FLASH_SUCCESS != err)
{
    . . .
}
```

Special Notes:

None

3.4 R_FLASH_BlankCheck()

This API function determines if specified code flash memory or data flash memory blocks are blank.

Format

```
flash_err_t R_FLASH_BlankCheck(
        uint32_t        address,
        uint32_t        num_bytes,
        flash_res_t     *blank_check_result
)
```

Parameters

address

Specifies the start address of the region to be processed by the blank check feature.
This parameter must specify a multiple of the minimum programming size of the target flash memory region.

num_bytes

Specifies the number of bytes subject to the blank check.
This parameter must specify a multiple of the minimum programming size of the target flash memory region. For RX111, RX113, and RX130 products, make sure that regions specified by “address” and “num_bytes” are not larger than 256 Kbytes.

**blank_check_result*

Specifies the memory address storing the blank check result when using blocking mode.
The following are stored as the blank check results.

- FLASH_RES_BLANK: Blank
- FLASH_RES_NOT_BLANK: Not blank

In non-blocking mode, specify any value since this parameter is not used.

Return Values

FLASH_SUCCESS	*/* Successful completion of blank check processing. In non-blocking mode, this indicates that blank check processing has started. */*
FLASH_ERR_FAILURE	*/* Blank check processing failure. In non-blocking mode, the callback function is not registered.*
FLASH_ERR_BUSY	*/* A different flash memory process is being executed, or the module is not initialized. */*
FLASH_ERR_BYTES	*/* “num_bytes” was either too large, not a multiple of the minimum programming size, or exceeded the maximum range. */*
FLASH_ERR_ADDRESS	*/* Invalid address was specified. */* */* Address is not a multiple of the minimum programming size or a flash type not supported for blank check was specified. */*
FLASH_ERR_NULL_PTR	*/* “blank_check_result” for storing blank check results was NULL.*/*

Properties

Prototyped in file “r_flash_rx_if.h”.

Description

Table 3.3 describes the MCU groups that support blank check.

Table 3.3 MCU Groups Supporting Blank Check

MCU Group	Code Flash Memory	Data Flash Memory
RX110	●	—*1
RX111	●	●
RX113	●	●
RX130	●	●
RX13T	●	●
RX140	●	●
RX230, RX231	●	●
RX23E-A, RX23E-B	●	●
RX23T	●	—*1
RX23W	●	●
RX24T	●	●
RX24U	●	●
RX260, RX261	●	●
RX26T	—	●
RX64M	—	●
RX65N, RX651	—	●*2
RX660	—	●
RX66N	—	●
RX66T	—	●
RX671	—	●
RX71M	—	●
RX72M	—	●
RX72N	—	●
RX72T	—	●

●: Supported, —: Unsupported

*1 Does not contain any data flash memory.

*2 Products with no more than 1 Mbyte of code flash memory do not have data flash memory.

The address specified by the first argument and the number of bytes specified by the second argument of this API function must be in multiples of the minimum programming size. The minimum programming size varies depending on the type of both the MCU and flash memory. Refer to Table 3.4 in section 3.5 for details.

If this API function is used in non-blocking mode, the result of the blank check is passed as the argument of the callback function after the blank check is complete.

Reentrant

- Not allowed

Example
The first argument specifies the start address to be processed by the blank check feature.
The second argument specifies the number of bytes subject to the blank check.
Both of these arguments must be expressed in multiples of the minimum programming size.

```
flash_err_t err;
flash_res_t result;

/* Run the blank check on the first 64 bytes in block 0 of data flash memory. */
err = R_FLASH_BlankCheck((uint32_t)FLASH_DF_BLOCK_0, 64, &result);
if (FLASH_SUCCESS != err)
{
    /* Error processing */
}
else
{
    /* Check result */
    if (FLASH_RES_NOT_BLANK == result)
    {
        /* Processing when block is not blank */
        . . .
    }
    else if (FLASH_RES_BLANK == ret)
    {
        /* Processing when block is blank */
        . . .
    }
}
```

Special Notes:
None

3.5 R_FLASH_Write()

This API function reprograms code flash memory or data flash memory.

Format

```
flash_err_t R_FLASH_Write(
        uint32_t  src_address,
        uint32_t  dest_address,
        uint32_t  num_bytes
)
```

Parameters

src_address

Specifies the start address of the buffer storing the data to be written in flash memory.

dest_address

Specifies the start address of the region in flash memory to be reprogrammed.

This parameter must specify a multiple of the minimum programming size of the target flash memory region.

num_bytes

Specifies the number of bytes in flash memory to be written.

This parameter must specify a multiple of the minimum programming size of the target flash memory region.

Return Values

FLASH_SUCCESS	*/* Successful completion of programming. In non-blocking mode, this indicates that programming has started. */*
FLASH_ERR_FAILURE	*/* Programming failed due to flash sequencer error. In non-blocking mode, the callback function is not registered. */*
FLASH_ERR_BUSY	*/* A different flash memory process is being executed, or the module is not initialized. */*
FLASH_ERR_BYTES	*/* Number of bytes provided was not a multiple of the minimum programming size or exceeds the maximum range. */*
FLASH_ERR_ADDRESS	*/* Specified address is invalid. */*

Properties

Prototyped in file “r_flash_rx_if.h”.

Description

Flash memory regions must be erased before being reprogrammed.

The address specified by the second argument and the number of bytes specified by the third argument of this API function must be in multiples of the minimum programming size. The minimum programming size varies depending on the MCU and flash memory as described in Table 3.4.

Table 3.4 Minimum Programming Sizes by MCU Group

MCU Group	Code Flash Memory*1	Data Flash Memory*2
RX110	4 bytes	—*3
RX111	4 bytes	1 byte
RX113	4 bytes	1 byte
RX130	4 bytes	1 byte
RX13T	4 bytes	1 byte
RX140	8 bytes	1 byte
RX230, RX231	8 bytes	1 byte
RX23E-A, RX23E-B	8 bytes	1 byte
RX23T	8 bytes	—*3
RX23W	8 bytes	1 byte
RX24T	8 bytes	1 byte
RX24U	8 bytes	1 byte
RX260, RX261	8 bytes	1 byte
RX26T	128 bytes	4 bytes
RX64M	256 bytes	4 bytes
RX65N, RX651	128 bytes	4 bytes*4
RX660	256 bytes	4 bytes
RX66N	128 bytes	4 bytes
RX66T	256 bytes	4 bytes
RX671	128 bytes	4 bytes
RX71M	256 bytes	4 bytes
RX72M	128 bytes	4 bytes
RX72N	128 bytes	4 bytes
RX72T	256 bytes	4 bytes

*1 Defined as FLASH_CF_MIN_PGM_SIZE in the specific MCU definitions file (“r_flash_rx\src\targets\<mcu>\r_flash_<mcu>.h”).

*2 Defined as FLASH_DF_MIN_PGM_SIZE in the specific MCU definitions file (“r_flash_rx\src\targets\<mcu>\r_flash_<mcu>.h”).

*3 Does not contain any data flash memory.

*4 Products with no more than 1 Mbyte of code flash memory do not have data flash memory.

When this API function is used in non-blocking mode, the callback function is called when all write operations are complete.

Reentrant

- Not allowed

Example

The second argument specifies the addresses in flash memory to be reprogrammed.

The third argument specifies the number of bytes to be written in flash memory.

Both of these arguments must be expressed in multiples of the minimum programming size.

```
flash_err_t err;
uint8_t write_buffer[16] = “Hello World...”;

/* Write data to internal memory.*/
err = R_FLASH_Write((uint32_t)write_buffer, dst_addr, sizeof(write_buffer));

if (FLASH_SUCCESS != err)
{
    . . .
}
```

Special Notes:

None

3.6 R_FLASH_Control()

This API function perform processing other than programming, erasing, and blank check.

Format

```
flash_err_t R_FLASH_Control(
        flash_cmd_t     cmd,
        void            *pcfg
)
```

Parameters

cmd

Specifies the command to execute.

**pcfg*

Specifies the required arguments depending on the command specified by argument 1. Set this to NULL if no arguments are required for the particular command.

Return Values

FLASH_SUCCESS	*/* Successful completion. In non-blocking mode, this indicates that processing has started successfully. */*
FLASH_ERR_ADDRESS	*/* Specified address is invalid. */*
FLASH_ERR_NULL_PTR	*/* NULL was specified even though the second argument was required. */*
FLASH_ERR_BUSY	*/* A different flash module process is being executed, or the module is not initialized. */*
FLASH_ERR_CMD_LOCKED	*/* Flash sequencer is in command lock state. */* */* The forced stop command was issued, and the return processing was performed. */*
FLASH_ERR_ACCESSW	*/* An access window error occurred. Incorrect region specified. */*
FLASH_ERR_PARAM	*/* Invalid parameter was passed. */*

Properties

Prototyped in file “r_flash_rx_if.h”.

Description

This API function performs processing according to the command specified as an argument. Table 3.5 describes the supported commands by flash type.

Table 3.5 Supported Commands by Flash Type

Type of Command	Command	Flash Type			
		1	3	4	5
Common among all flash types					
Retrieve flash module API function running status	FLASH_CMD_STATUS_GET	✔	✔	✔	✔
Register callback function	FLASH_CMD_SET_BGO_CALLBACK	✔	✔	✔	✔
Flash sequencer reset	FLASH_CMD_RESET	✔	✔	✔	✔
Flash sequencer usage frequency notification					
Notify flash sequencer usage frequency	FLASH_CMD_CONFIG_CLOCK	—	✔	✔	✔
Access window					
Retrieve access window configuration	FLASH_CMD_ACCESSWINDOW_GET	✔	—	✔	✔
Configure access window	FLASH_CMD_ACCESSWINDOW_SET	*1		*1	*1
Startup program protection					
Retrieve startup region setting	FLASH_CMD_SWAPFLAG_GET	✔	—	✔	✔
Swap startup region	FLASH_CMD_SWAPFLAG_TOGGLE	*2			
Retrieve startup region selection bit setting	FLASH_CMD_SWAPSTATE_GET				
Set startup region selection bit	FLASH_CMD_SWAPSTATE_SET				
Lockbit					
Retrieve lockbit configuration	FLASH_CMD_LOCKBIT_READ	—	✔	—	—
Set lockbit	FLASH_CMD_LOCKBIT_WRITE		*3		
Enable lockbit configuration	FLASH_CMD_LOCKBIT_ENABLE				
Disable lockbit configuration	FLASH_CMD_LOCKBIT_DISABLE				
ROM cache					
Enable ROM cache configuration	FLASH_CMD_ROM_CACHE_ENABLE	✔	✔	✔	—
Disable ROM cache configuration	FLASH_CMD_ROM_CACHE_DISABLE				
Retrieve ROM cache configuration	FLASH_CMD_ROM_CACHE_STATUS	*4	*5		
Disable cache					
Set non-cached RANGE0	FLASH_CMD_SET_NON_CACHED_RANGE0	—	✔	✔	—
Set non-cached RANGE1	FLASH_CMD_SET_NON_CACHED_RANGE1				
Retrieve configuration of non-cached RANGE0	FLASH_CMD_GET_NON_CACHED_RANGE0		*5	*6	
Retrieve configuration of non-cached RANGE1	FLASH_CMD_GET_NON_CACHED_RANGE1				
Dual bank					
Swap banks	FLASH_CMD_BANK_TOGGLE	—	—	✔	✔
Retrieve bank configuration	FLASH_CMD_BANK_GET			*7	*8

*1 Access window can only be used on code flash memory.
*2 Only supported on products with at least 32 Kbytes of code flash memory.
*3 Lockbit can only be used on code flash memory.
*4 Supported by RX24T and RX24U only.
*5 Supported by RX66T and RX72T only.
*6 Supported by RX66N, RX671, RX72M, and RX72N only.
*7 Only supported on products with at least 1 Mbytes of code flash memory.
*8 Only supported on products with at least 512 Kbytes of code flash memory.

Table 3.6 through Table 3.9 describe details of supported commands organized by flash type.

Table 3.6 Details of Commands Supported by Flash Type 1

Command	Contents
FLASH_CMD_STATUS_GET (Set the argument value to NULL.) *Refer to Example 3 for usage examples.	Retrieves the running state of the flash sequencer for flash memory. This command can be used even while flash memory processing is running. FLASH_SUCCESS: Flash sequencer is not running. FLASH_ERR_BUSY: Flash sequencer is running.
FLASH_CMD_SET_BGO_CALLBACK (Argument: flash_interrupt_config_t *) *Refer to Example 1 and Example 2 for usage examples.	Registers the callback function. This command requires operation in non-blocking mode.
FLASH_CMD_RESET (Set the argument value to NULL.)	Resets the flash sequencer. This command can be used even while flash memory processing is running.
FLASH_CMD_ACCESSWINDOW_GET (Argument: flash_access_window_config_t *) *Refer to Example 4 for usage examples.	Retrieves the start and end addresses of the blocks defining the region to which the access window is applied in code flash memory.
FLASH_CMD_ACCESSWINDOW_SET (Argument: flash_access_window_config_t *) *Refer to Example 5 for usage examples.	Specifies the start and end addresses of the blocks defining the region to which the access window is applied in code flash memory. The start address must be a smaller number than the end address in access window configurations. Programming and erase processes cannot be performed on blocks outside the range specified with the start and end addresses. Multiple ranges defined by start and end addresses cannot be specified. Specify the same start and end addresses to delete an access window configuration. When using in non-blocking mode, FRDYI interrupt occurs after setting the access window, and then callback function is called.
FLASH_CMD_SWAPFLAG_GET (Argument: uint32_t *) *Refer to Example 6 for usage examples.	Retrieves the startup region setting. 0: Startup from the alternate region 1: Startup from the default region
FLASH_CMD_SWAPFLAG_TOGGLE (Set the argument value to NULL.) *Refer to Example 7 for usage examples.	Swaps the startup region. The swapped startup region takes effect after the next reset. When using in non-blocking mode, FRDYI interrupt occurs after the startup region is swapped, and then the callback function is called. Make sure that the FLASH_CFG_CODE_FLASH_ENABLE configuration option is set to “1” when using this command.

Command	Contents
FLASH_CMD_SWAPSTATE_GET (Argument: uint8_t *) *Refer to Example 8 for usage examples.	Retrieves the value of the startup region selection bit (FISR.SAS). FLASH_SAS_EXTRA: The startup region selection bit follows the startup region configuration. FLASH_SAS_DEFAULT: Sets the startup region selection bit to the default region. FLASH_SAS_ALTERNATE: Sets the startup region selection bit to the alternate region.
FLASH_CMD_SWAPSTATE_SET (Argument: uint8_t *) *Refer to Example 9 for usage examples.	Sets the value of the startup region selection bit (FISR.SAS). The set startup region takes effect immediately. The default value after a reset is FLASH_SAS_EXTRA. FLASH_SAS_EXTRA: Follows the configuration of the startup region in extra area. FLASH_SAS_DEFAULT: Temporarily changes the startup region to the default region. FLASH_SAS_ALTERNATE: Temporarily changes the startup region to the alternate region. FLASH_SAS_SWITCH_AREA: Swaps the startup region.
FLASH_CMD_ROM_CACHE_ENABLE (Set the argument value to NULL.) *Refer to Example 10 for usage examples.	Enables the code flash memory cache.
FLASH_CMD_ROM_CACHE_DISABLE (Set the argument value to NULL.) *Refer to Example 10 for usage examples.	Disables the code flash memory cache. Call before reprogramming code flash memory.
FLASH_CMD_ROM_CACHE_STATUS (Argument: uint8_t *) *Refer to Example 10 for usage examples.	Retrieves the status of code flash memory cache. 0: Code flash memory cache is disabled 1: Code flash memory cache is enabled

Table 3.7 Details of Commands Supported by Flash Type 3

Command	Contents
FLASH_CMD_STATUS_GET (Set the argument value to NULL.) *Refer to Example 3 for usage examples.	Retrieves the running state of the flash sequencer for flash memory. This command can be used even while flash memory processing is running. FLASH_SUCCESS: Flash sequencer is not running. FLASH_ERR_BUSY: Flash sequencer is running.
FLASH_CMD_SET_BGO_CALLBACK (Argument: flash_interrupt_config_t *) *Refer to Example 1 and Example 2 for usage examples.	Registers the callback function. This command requires operation in non-blocking mode.
FLASH_CMD_RESET (Set the argument value to NULL.)	Resets the flash sequencer. This command can be used even while flash memory processing is running.
FLASH_CMD_LOCKBIT_READ (Argument: flash_lockbit_config_t *) *Refer to Example 12 for usage examples.	Retrieves the status of the lockbit configuration for the specified block in code flash memory. When using in non-blocking mode, FRDYI interrupt occurs after the status of the lockbit configuration is retrieved, and then the callback function is called. *1 FLASH_RES_LOCKBIT_STATE_PROTECTED: Protected FLASH_RES_LOCKBIT_STATE_NON_PROTECTED: Not protected
FLASH_CMD_LOCKBIT_WRITE (Argument: flash_lockbit_config_t *) *Refer to Example 12 for usage examples.	Sets the starting block address and the number of blocks defining the region to which the lockbit is applied in code flash memory. For the lockbit configuration, multiple regions specified by the starting block address and the number of blocks can be set. When using in non-blocking mode, FRDYI interrupt occurs after setting the lockbit, and then the callback function is called. *1
FLASH_CMD_LOCKBIT_ENABLE (Set the argument value to NULL.) *Refer to Example 12 for usage examples.	Prohibits the program and erase processes from being performed on the blocks in code flash memory set as the lockbit region.
FLASH_CMD_LOCKBIT_DISABLE (Set the argument value to NULL.) *Refer to Example 12 for usage examples.	Allows the program and erase processes from being performed on the blocks in code flash memory set as the lockbit region. The blocks with lockbit set can be erased after using this command. Note that erasing the blocks with lockbit set also clears the lockbit configuration for the erased blocks.
FLASH_CMD_ROM_CACHE_ENABLE (Set the argument value to NULL.) *Refer to Example 10 for usage examples.	Enables the code flash memory cache.
FLASH_CMD_ROM_CACHE_DISABLE (Set the argument value to NULL.) *Refer to Example 10 for usage examples.	Disables the code flash memory cache. Call before reprogramming code flash memory.
FLASH_CMD_ROM_CACHE_STATUS (Argument: uint8_t *) *Refer to Example 10 for usage examples.	Retrieves the status of code flash memory cache. 0: Code flash memory cache is disabled 1: Code flash memory cache is enabled

Command	Contents
FLASH_CMD_SET_NON_CACHED_RANGE0 (Argument: flash_non_cached_t *) *Refer to Example 11 for usage examples.	Sets the area specified in code flash memory as non-cacheable RANGE0. Caching will be disabled for the specified area. Note that running this command while cache is enabled causes the cache to be temporarily disabled.
FLASH_CMD_SET_NON_CACHED_RANGE1 (Argument: flash_non_cached_t *) *Refer to Example 11 for usage examples.	Sets the area specified in code flash memory as non-cacheable RANGE1. Caching will be disabled for the specified area. Note that running this command while cache is enabled causes the cache to be temporarily disabled.
FLASH_CMD_GET_NON_CACHED_RANGE0 (Argument: flash_non_cached_t *) *Refer to Example 11 for usage examples.	Retrieves the configuration of non-cacheable RANGE0.
FLASH_CMD_GET_NON_CACHED_RANGE1 (Argument: flash_non_cached_t *) *Refer to Example 11 for usage examples.	Retrieves the configuration of non-cacheable RANGE1.
FLASH_CMD_CONFIG_CLOCK (Argument: uint32_t *)	Provides notification of the frequency used by the flash sequencer. This command is used to change the flash clock (FLCK) speed from the frequency as set by BSP while a program is running. This command is not needed if not changing the flash clock (FCLK).

*1 Blocks until completion even when operating in non-blocking mode.

Table 3.8 Details of Commands Supported by Flash Type 4

Command	Contents
FLASH_CMD_STATUS_GET (Set the argument value to NULL.) *Refer to Example 3 for usage examples.	Retrieves the running state of the flash sequencer for flash memory. This command can be used even while flash memory processing is running. FLASH_SUCCESS: Flash sequencer is not running. FLASH_ERR_BUSY: Flash sequencer is running.
FLASH_CMD_SET_BGO_CALLBACK (Argument: flash_interrupt_config_t *) *Refer to Example 1 and Example 2 for usage examples.	Registers the callback function. This command requires operation in non-blocking mode.
FLASH_CMD_RESET (Set the argument value to NULL.)	Resets the flash sequencer. This command can be used even while flash memory processing is running.
FLASH_CMD_ACCESSWINDOW_GET (Argument: flash_access_window_config_t *) *Refer to Example 4 for usage examples.	Retrieves the start and end addresses of the blocks defining the region to which the access window is applied in code flash memory.
FLASH_CMD_ACCESSWINDOW_SET (Argument: flash_access_window_config_t *) *Refer to Example 5 for usage examples.	Specifies the start and end addresses of the blocks defining the region to which the access window is applied in code flash memory. The start address must be a smaller number than the end address in access window configurations. Programming and erase processes cannot be performed on blocks outside the range specified with the start and end addresses. Multiple ranges defined by start and end addresses cannot be specified. Specify the same start and end addresses to delete an access window configuration. When using in non-blocking mode, FRDYI interrupt occurs after setting the access window, and then callback function is called. *1
FLASH_CMD_SWAPFLAG_GET (Argument: uint32_t *) *Refer to Example 6 for usage examples.	Retrieves the startup region setting. 0: Swaps the configuration of startup regions 0 and 1. 1: Keeps the configuration of startup regions 0 and 1 to the defaults.
FLASH_CMD_SWAPFLAG_TOGGLE (Set the argument value to NULL.) *Refer to Example 7 for usage examples.	Swaps the startup region. The swapped startup region takes effect after the next reset. When using in non-blocking mode, FRDYI interrupt occurs after the startup region is swapped, and then the callback function is called. *1
FLASH_CMD_SWAPSTATE_GET (Argument: uint8_t *) *Refer to Example 8 for usage examples.	Retrieves the value of the startup region selection bit (FSUACR.SAS) when operating in linear mode. This command cannot be used in dual mode. FLASH_SAS_SWAPFLG: The startup region selection bit follows the startup region configuration. FLASH_SAS_DEFAULT: Sets the startup region selection bit to startup region 0. FLASH_SAS_ALTERNATE: Sets the startup region selection bit to startup region 1.

Command	Contents
FLASH_CMD_SWAPSTATE_SET (Argument: uint8_t *) *Refer to Example 9 for usage examples.	Sets the value of the startup region selection bit (FSUACR.SAS) when operating in linear mode. The set startup region takes effect immediately. The default value after a reset is FLASH_SAS_SWAPFLG. This command cannot be used in dual mode. FLASH_SAS_SWAPFLG: Follows the configuration of the startup region in option settings memory. FLASH_SAS_DEFAULT: Temporarily changes the startup region to startup region 0. FLASH_SAS_ALTERNATE: Temporarily changes the startup region to startup region 1. FLASH_SAS_SWITCH_AREA: Swaps the startup region.
FLASH_CMD_ROM_CACHE_ENABLE (Set the argument value to NULL.) *Refer to Example 10 for usage examples.	Enables the code flash memory cache.
FLASH_CMD_ROM_CACHE_DISABLE (Set the argument value to NULL.) *Refer to Example 10 for usage examples.	Disables the code flash memory cache. Call before reprogramming code flash memory.
FLASH_CMD_ROM_CACHE_STATUS (Argument: uint8_t *) *Refer to Example 10 for usage examples.	Retrieves the status of code flash memory cache. 0: Code flash memory cache is disabled 1: Code flash memory cache is enabled
FLASH_CMD_SET_NON_CACHED_RANGE0 (Argument: flash_non_cached_t *) *Refer to Example 11 for usage examples.	Sets the area specified in code flash memory as non-cacheable RANGE0. Caching will be disabled for the specified area. Note that running this command while cache is enabled causes the cache to be temporarily disabled.
FLASH_CMD_SET_NON_CACHED_RANGE1 (Argument: flash_non_cached_t *) *Refer to Example 11 for usage examples.	Sets the area specified in code flash memory as non-cacheable RANGE1. Caching will be disabled for the specified area. Note that running this command while cache is enabled causes the cache to be temporarily disabled.
FLASH_CMD_GET_NON_CACHED_RANGE0 (Argument: flash_non_cached_t *) *Refer to Example 11 for usage examples.	Retrieves the configuration of non-cacheable RANGE0.
FLASH_CMD_GET_NON_CACHED_RANGE1 (Argument: flash_non_cached_t *) *Refer to Example 11 for usage examples.	Retrieves the configuration of non-cacheable RANGE1.
FLASH_CMD_BANK_TOGGLE[*2] (Set the argument value to NULL.) *Refer to Example 13 for usage examples.	This command cannot be used in linear mode. Swaps the startup bank when operating in dual mode. The swap of the startup bank takes effect after the next reset. When using in non-blocking mode, FRDYI interrupt occurs after setting the bank selection register (BANKSEL), and then the callback function is called. [*1]

Command	Contents
FLASH_CMD_BANK_GET[*2] (Argument: flash_bank_t *) *Refer to Example 13 for usage examples.	This command cannot be used in linear mode. Retrieves the current startup bank setting from the bank selection register (BANKSEL) when operating in dual mode. FLASH_BANK0: 1 FLASH_BANK1: 0
FLASH_CMD_CONFIG_CLOCK (Argument: uint32_t *)	Provides notification of the frequency used by the flash sequencer. Also sets the read speed for data flash memory. [*2] This command is used to change the flash clock (FLCK) speed from the frequency as set by BSP while a program is running. This command is not needed if not changing the flash clock (FCLK).

[*1] Blocks until completion even when operating in non-blocking mode.

[*2] Only supported on products with at least 1.5 Mbyte of code flash memory.

Table 3.9 Details of Commands Supported by Flash Type 5

Command	Contents
FLASH_CMD_STATUS_GET (Set the argument value to NULL.) *Refer to Example 3 for usage examples.	Retrieves the running state of the flash sequencer for flash memory. This command can be used even while flash memory processing is running. FLASH_SUCCESS: Flash sequencer is not running. FLASH_ERR_BUSY: Flash sequencer is running.
FLASH_CMD_SET_BGO_CALLBACK (Argument: flash_interrupt_config_t *) *Refer to Example 1 and Example 2 for usage examples.	Registers the callback function. This command requires operation in non-blocking mode.
FLASH_CMD_RESET (Set the argument value to NULL.)	Resets the flash sequencer. This command can be used even while flash memory processing is running.
FLASH_CMD_ACCESSWINDOW_GET (Argument: flash_access_window_config_t *) *Refer to Example 4 for usage examples.	Retrieves the start and end addresses of the blocks defining the region to which the access window is applied in code flash memory.
FLASH_CMD_ACCESSWINDOW_SET (Argument: flash_access_window_config_t *) *Refer to Example 5 for usage examples.	Specifies the start and end addresses of the blocks defining the region to which the access window is applied in code flash memory. The start address must be a smaller number than the end address in access window configurations. Programming and erase processes cannot be performed on blocks outside the range specified with the start and end addresses. Multiple ranges defined by start and end addresses cannot be specified. Specify the same start and end addresses to delete an access window configuration. When using in non-blocking mode, FRDYI interrupt occurs after setting the access window, and then callback function is called. *1
FLASH_CMD_SWAPFLAG_GET (Argument: uint32_t *) *Refer to Example 6 for usage examples.	Retrieves the startup region setting. 0: Swaps the configuration of startup regions 0 and 1. 1: Keeps the configuration of startup regions 0 and 1 to the defaults.
FLASH_CMD_SWAPFLAG_TOGGLE (Set the argument value to NULL.) *Refer to Example 7 for usage examples.	Swaps the startup region. The swapped startup region takes effect after the next reset. When using in non-blocking mode, FRDYI interrupt occurs after the startup region is swapped, and then the callback function is called. *1
FLASH_CMD_SWAPSTATE_GET (Argument: uint8_t *) *Refer to Example 8 for usage examples.	Retrieves the value of the startup region selection bit (FSUACR.SAS) when operating in linear mode. This command cannot be used in dual mode. FLASH_SAS_SWAPFLG: The startup region selection bit follows the startup region configuration. FLASH_SAS_DEFAULT: Sets the startup region selection bit to startup region 0. FLASH_SAS_ALTERNATE: Sets the startup region selection bit to startup region 1.

Command	Contents
FLASH_CMD_SWAPSTATE_SET (Argument: uint8_t *) *Refer to Example 9 for usage examples.	Sets the value of the startup region selection bit (FSUACR.SAS) when operating in linear mode. The set startup region takes effect immediately. The default value after a reset is FLASH_SAS_SWAPFLG. This command cannot be used in dual mode. FLASH_SAS_SWAPFLG: Follows the configuration of the startup region in option settings memory. FLASH_SAS_DEFAULT: Temporarily changes the startup region to startup region 0. FLASH_SAS_ALTERNATE: Temporarily changes the startup region to startup region 1. FLASH_SAS_SWITCH_AREA: Swaps the startup region.
FLASH_CMD_BANK_TOGGLE*2 (Set the argument value to NULL.) *Refer to Example 13 for usage examples.	This command cannot be used in linear mode. Swaps the startup bank when operating in dual mode. The swap of the startup bank takes effect after the next reset. When using in non-blocking mode, FRDYI interrupt occurs after setting the bank selection register (BANKSEL), and then the callback function is called. *1
FLASH_CMD_BANK_GET*2 (Argument: flash_bank_t *) *Refer to Example 13 for usage examples.	This command cannot be used in linear mode. Retrieves the current startup bank setting from the bank selection register (BANKSEL) when operating in dual mode. FLASH_BANK0: 1 FLASH_BANK1: 0
FLASH_CMD_CONFIG_CLOCK (Argument: uint32_t *)	Provides notification of the frequency used by the flash sequencer. This command is used to change the flash clock (FLCK) speed from the frequency as set by BSP while a program is running. This command is not needed if not changing the flash clock (FCLK).

*1 Blocks until completion even when operating in non-blocking mode.

*2 Only supported on products with at least 512 Kbyte of code flash memory.

Example 1: Writing to code flash memory in non-blocking mode

To use flash module API functions in non-blocking mode, set both configuration options FLASH_CFG_DATA_FLASH_BGO and FLASH_CFG_CODE_FLASH_BGO to “1”.

To program code flash memory by running code from RAM, set the configuration option FLASH_CFG_CODE_FLASH_ENABLE to “1”. Also vector tables of possible interrupts must be relocated to RAM.

The registered callback function can be used by running R_FLASH_Open (), using R_FLASH_Control () to register the callback function, and then running a flash module API function (R_FLASH_Write (), R_FLASH_Erase (), or R_FLASH_BlankCheck ()).

```
/* Region in RAM storing vector tables */
static uint32_t ram_vect_table[256];

void func(void)
{
    flash_err_t err;
    flash_interrupt_config_t cb_func_info;
    uint32_t *pvect_table;

    /* Relocate interrupt vector tables in RAM */
    /* Directly set the FRDYI interrupt function address into
ram_vect_table[23]. */
    /* Please consider the method according to the user's system. */
    pvect_table = (uint32_t *)__sectop("C$VECT");
    ram_vect_table[23] = pvect_table[23]; /* FRDYI Interrupt function copy */
    set_intb((void *)ram_vect_table);

    /* Initialize the API. */
    err = R_FLASH_Open();
    /* Check for errors. */
    if (FLASH_SUCCESS != err)
    {
        /* Handle error */
    }

    /* Set callback function and interrupt priority */
    cb_func_info.pcallback = u_cb_function;
    cb_func_info.int_priority = 1;
    err = R_FLASH_Control(FLASH_CMD_SET_BGO_CALLBACK,(void *)&cb_func_info);
    if (FLASH_SUCCESS != err)
    {
        /* Handle error */
    }

    /* Perform operations on code flash memory */
    do_rom_operations();

    ... (omission)
}
```

```
#pragma section FRAM
void u_cb_function(void *event) /* Callback function */
{
    flash_int_cb_args_t *ready_event = event;

    /* Perform ISR callback functionality here */
    ... (omission)
}

void do_rom_operations(void)
{
    /* Set code flash memory access window, toggle startup area flag */
    /* Swap boot blocks, erase, blank check, or programming processing here */
    ... (omission)
}
#pragma section
```

Example 2: Writing to data flash memory in non-blocking mode

To use flash module API functions in non-blocking mode, set both configuration options FLASH_CFG_DATA_FLASH_BGO and FLASH_CFG_CODE_FLASH_BGO to “1”.

To program data flash memory, the code for reprogramming to flash memory can be ran in code flash memory.

The registered callback function can be used by running R_FLASH_Open (), using R_FLASH_Control () to register the callback function, and then running a flash module API function (R_FLASH_Write (), R_FLASH_Erase (), or R_FLASH_BlankCheck ()).

```
void func(void)
{
    flash_err_t err;
    flash_interrupt_config_t cb_func_info;

    /* Initialize the API. */
    err = R_FLASH_Open();
    /* Check for errors. */
    if (FLASH_SUCCESS != err)
    {
        /* Handle error */
    }

    /* Set callback function and interrupt priority */
    cb_func_info.pcallback = u_cb_function;
    cb_func_info.int_priority = 1;
    err = R_FLASH_Control(FLASH_CMD_SET_BGO_CALLBACK,(void *)&cb_func_info);
    if (FLASH_SUCCESS != err)
    {
        /* Handle error */
    }

    /* Set erase, blank check, or programming processing of data flash memory
here */
    ... (omission)
}

void u_cb_function(void *event) /* Callback function */
{
    flash_int_cb_args_t *ready_event = event;

    /* Perform ISR callback functionality here */
    ... (omission)
}
```

Example 3: Checking running status of flash module API functions
The following example shows the use of R_FLASH_Erase() in non-blocking mode.

```
    flash_err_t err;

    /* Erase all of data flash */
    err = R_FLASH_Erase(FLASH_DF_BLOCK_0, FLASH_NUM_BLOCKS_DF);
    if (FLASH_SUCCESS != err)
    {
        /* Handle error */
    }

    /* Check flash module API function running status */
    while (FLASH_ERR_BUSY == R_FLASH_Control(FLASH_CMD_STATUS_GET, NULL))
    {
        /* Execute any process */
    }
```

Example 4: Retrieving the access window configuration area for code flash memory

```
    flash_err_t err;
    flash_access_window_config_t access_info;

    err = R_FLASH_Control(FLASH_CMD_ACCESSWINDOW_GET, (void *)&access_info);
    if (FLASH_SUCCESS != err)
    {
            /* Handle error */
    }
```

Example 5: Configuring the access window area for code flash memory
Access window-based region protection is used to prevent configured areas in the code flash memory from being accidentally programmed or erased.

```
    flash_err_t err;
    flash_access_window_config_t access_info;

    /* Allow programming and erasing of block 3 in code flash memory. */
    access_info.start_addr = (uint32_t) FLASH_CF_BLOCK_3;
    access_info.end_addr = (uint32_t) FLASH_CF_BLOCK_2;
    err = R_FLASH_Control(FLASH_CMD_ACCESSWINDOW_SET, (void *)&access_info);
    if (FLASH_SUCCESS != err)
    {
      /* Handle error */
    }

    /* Allow programming and erasing of block 2, block 1,and block 0 in code
flash memory. */
    /* Use FLASH_CF_BLOCK_END to specify end address if block 0 is included in
setting range. */
    access_info.start_addr = (uint32_t) FLASH_CF_BLOCK_2;
    access_info.end_addr = (uint32_t) FLASH_CF_BLOCK_END;
    err = R_FLASH_Control(FLASH_CMD_ACCESSWINDOW_SET, (void *)&access_info);
    if (FLASH_SUCCESS != err)
    {
      /* Handle error */
    }
```

Example 6: Retrieving the startup region setting

```
flash_err_t err;
uint32_t      swap_flag;

err = R_FLASH_Control(FLASH_CMD_SWAPFLAG_GET, (void *)&swap_flag);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}
```

Example 7: Swapping the startup region setting

The following example shows how to toggle the active start-up program area.

```
flash_err_t err;

/* Swap the active area from Default to Alternate or vice versa. */

err = R_FLASH_Control(FLASH_CMD_SWAPFLAG_TOGGLE, FIT_NO_PTR);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}
```

Example 8: Retrieving the value of the startup region selection bit

```
flash_err_t err;
uint8_t       swap_area;

err = R_FLASH_Control(FLASH_CMD_SWAPSTATE_GET, (void *)&swap_area);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}
```

Example 9: Setting the value of the startup region selection bit

The following example shows how to set the startup region selection bit. The region specified by the startup region selection bit will be used after a reset.

```
flash_err_t err;
uint8_t       swap_area;

swap_area = FLASH_SAS_SWITCH_AREA;
err = R_FLASH_Control(FLASH_CMD_SWAPSTATE_SET, (void *)&swap_area);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}
```

Example 10: Enabling/disabling caching of code flash memory

The following example shows a process of enabling code flash memory caching, then disabling caching to perform erase or programming processes, and then re-enabling caching.

```
flash_err_t err;
uint8_t status;

/* Enable caching */
err = R_FLASH_Control(FLASH_CMD_ROM_CACHE_ENABLE, NULL);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}

/* Confirm that caching is enabled */
err = R_FLASH_Control(FLASH_CMD_ROM_CACHE_STATUS, &status);
if ((FLASH_SUCCESS != err) || (1 != status))
{
    /* Handle error */
}

... (omission)

/* Disable caching in preparation for programming */
err = R_FLASH_Control(FLASH_CMD_ROM_CACHE_DISABLE, NULL);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}

/* Erase, program, and verify new code here */

/* Re-enable caching */
err = R_FLASH_Control(FLASH_CMD_ROM_CACHE_ENABLE, NULL);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}
```

Example 11: Disabling caching in a specific area of code flash memory

The following shows how to disable caching of a specific area of code flash memory. Up to two areas of disabled caching can be configured, and these areas can overlap.

```
flash_err_t err;
flash_non_cached_t range;

/* Do not cache fast-instruction fetching or operand access by the CPU */
/* for the first 1 Kbyte of code flash in FLASH_CF_BLOCK_10. */
range.start_addr = (uint32_t)FLASH_CF_BLOCK_10;
range.size = FLASH_NON_CACHED_1_KBYTE;
range.type_mask = FLASH_NON_CACHED_MASK_IF | FLASH_NON_CACHED_MASK_OA;

err = R_FLASH_Control(FLASH_CMD_SET_NON_CACHED_RANGE0, &range);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}

/* Enable caching */
/* This command is eliminated if caching is already enabled. */
err = R_FLASH_Control(FLASH_CMD_ROM_CACHE_ENABLE, NULL);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}

/* Retrieve non-cached settings for RANGE0 */
err = R_FLASH_Control(FLASH_CMD_GET_NON_CACHED_RANGE0, &range);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}
```

Example 12: Configuring lockbit-based protection on code flash memory

The following example shows a process of setting lockbit on specific blocks in code flash memory, retrieving lockbit information, disabling lockbit-based protection, and then enabling protection.

```
flash_err_t err;
flash_lockbit_config_t lockbit_info;

/* Set lockbit on block 3 in code flash memory */
lockbit_info.block_start_address = FLASH_CF_BLOCK_3;
lockbit_info.num_blocks = 1;
err = R_FLASH_Control(FLASH_CMD_LOCKBIT_WRITE, (void *)&lockbit_info);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}

/* Retrieve lockbit information on block 3 in code flash memory */
err = R_FLASH_Control(FLASH_CMD_LOCKBIT_READ, (void *)&lockbit_info);
if ((FLASH_SUCCESS != err) ||
     (lockbit_info.result != FLASH_RES_LOCKBIT_STATE_PROTECTED))
{
    /* Handle error */
}


/* Disable lockbit-based protection, */
/* which enables erasing or programming of the block with lockbit set. */
err = R_FLASH_Control(FLASH_CMD_LOCKBIT_DISABLE, NULL);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}

/* Erasing or programming of the block with lockbit set is now enabled */

/* Enables lockbit-based protection, */
/* which disables erasing or programming of the block with lockbit set. */
err = R_FLASH_Control(FLASH_CMD_LOCKBIT_ENABLE, NULL);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}

/* Erasing or programming of the block with lockbit set is now disabled */
```

Example 13: Swapping startup banks
Performs swapping startup banks. The swap of the startup bank takes effect after the next reset.

The startup bank that will take effect after the next reset can be retrieved by the second argument by specifying the FLASH_CMD_BANK_GET command in the first argument of R_FLASH_Control().

If the value of the second argument is FLASH_BANK0, bank 0 will be the startup bank after the next reset. If the value is FLASH_BANK1, bank 1 will be the startup bank after the next reset.

```
flash_err_t err;
flash_bank_t bank_info;

/* Swap the bank selected as the startup bank */
err = R_FLASH_Control(FLASH_CMD_BANK_TOGGLE, NULL);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}

/* Retrieve the bank selected as the startup bank */
err = R_FLASH_Control(FLASH_CMD_BANK_GET, (void *)&bank_info);
if (FLASH_SUCCESS != err)
{
    /* Handle error */
}

/* The swap of the startup bank takes effect after the next reset */
```

Special Notes:
None

3.7 R_FLASH_GetVersion()

This API function retrieves the version number of the flash module.

Format

```
uint32_t R_FLASH_GetVersion(void)
```

Parameters

None

Return Values

Version Number

Properties

Prototyped in file “r_flash_rx_if.h”.

Description

This API function returns the version number of the flash module. The version number is encoded where the top 2 bytes are the major version number and the bottom 2 bytes are the minor version number. For example, Version 4.25 would be returned as 0x00040019.

Example

```
uint32_t cur_version;

/* Retrieve the version of the installed flash modules */
cur_version = R_FLASH_GetVersion();

/* Version determination processing */
if (0x00040019 > cur_version)
{
    /* Version-specific processing */
}
```

Special Notes:

None

4. Demo Projects

Demo projects are complete stand-alone programs. They include function main() that utilizes the module and its dependent modules (e.g. r_bsp). The standard naming convention for the demo project is <module>_demo_<board> where <module> is the peripheral acronym (e.g. s12ad, cmt, sci) and the <board> is the standard RSK (e.g. rskrx113). For example, s12ad FIT module demo project for RSKRX113 will be named as s12ad_demo_rskrx113. Similarly the exported .zip file will be <module>_demo_<board>.zip. For the same example, the zipped export/import file will be named as s12ad_demo_rskrx113.zip

Note that demo projects do not support a compiler other than Renesas Electronics C/C++ Compiler Package for RX Family.

4.1 flash_demo_rskrx113

This is a simple demo for the RSKRX113 starter kit. The demo uses blocking mode to execute flash erasing, blank check, and programming. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Output).

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX113

Evaluation Environment

Version used: BSP Rev. 5.30, FLASH FIT Rev. 4.30

4.2 flash_demo_rskrx231

This is a simple demo for the RSKRX231 starter kit. The demo uses blocking mode to execute flash erasing, blank check, and programming. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Output).

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX231

Evaluation Environment

Version used: BSP Rev. 5.30, FLASH FIT Rev. 4.30

4.3 flash_demo_rskrx23t

This is a simple demo for the RSKRX23T starter kit. The demo uses blocking mode to execute flash erasing, blank check, and programming. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Output).

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX23T

Evaluation Environment

Version used: BSP Rev. 5.30, FLASH FIT Rev. 4.30

4.4 flash_demo_rskrx130

This is a simple demo for the RSKRX130 starter kit. The demo uses blocking mode to execute flash erasing, blank check, and programming. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Output).

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX130

Evaluation Environment

Version used: BSP Rev. 5.30, FLASH FIT Rev. 4.30

4.5 flash_demo_rskrx24t

This is a simple demo for the RSKRX24Tstarter kit. The demo uses blocking mode to execute flash erasing, blank check, and programming. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Output).

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX24T

Evaluation Environment

Version used: BSP Rev. 5.30, FLASH FIT Rev. 4.30

4.6 flash_demo_rskrx65n

This is a simple demo for the RSKRX65N starter kit. The demo uses blocking mode to execute flash erasing, blank check, and programming. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Output).

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX65N-1

Evaluation Environment

Version used: BSP Rev. 5.30, FLASH FIT Rev. 4.30

4.7 flash_demo_rskrx24u

This is a simple demo for the RSKRX24U starter kit. The demo uses blocking mode to execute flash erasing and programming. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Output).

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX24U

Evaluation Environment

Version used: BSP Rev. 5.30, FLASH FIT Rev. 4.30

4.8 flash_demo_rskrx64m

This is a simple demo for the RSKRX64M starter kit. The demo uses blocking mode to execute flash erasing and programming. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Output).

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX64M

Evaluation Environment

Version used: BSP Rev. 5.30, FLASH FIT Rev. 4.30

4.9 flash_demo_rskrx64m_runrom

This is a simple demo for the RSKRX64M starter kit. What sets this apart from other demos is that this makes use of the RX64M feature which allows an application to run from one region of code flash while erasing/writing to another. (Most other MCUs require code that could execute during a code flash erase/write to be located in RAM.) The demo uses blocking mode to execute flash erasing and programming. Each write function is verified with a read-back of data. Notice that the typical Linker set up for supporting code flash erase/write (RAM locating) is not necessary in this demo, and that FLASH_CFG_CODE_FLASH_RUN_FROM_ROM is set to 1 in “r_flash_rx_config.h”.

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX64M

Evaluation Environment

Version used: BSP Rev. 5.30, FLASH FIT Rev. 4.30

4.10 flash_demo_rskrx66t

This is a simple demo for the RSKRX66T starter kit. The demo uses blocking mode to execute flash erasing and programming. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Symbol file).

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX66T

Evaluation Environment

Version used: BSP Rev. 5.30, FLASH FIT Rev. 4.30

4.11 flash_demo_rskrx72t

This is a simple demo for the RSKRX72T starter kit. The demo uses blocking mode to execute flash erasing and programming. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Symbol file).

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX72T

Evaluation Environment

Version used: BSP Rev. 5.30, FLASH FIT Rev. 4.30

4.12 flash_demo_mckrx26t

This is a simple demo for the MCKRX26T flexible motor control kit. The demo uses blocking mode to execute flash erasing and programming. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Symbol file).

Setup and Execution

1. Compile and download the sample code.
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

MCKRX26T

Evaluation Environment

Version used: BSP Rev. 7.30, FLASH FIT Rev. 5.00

4.13 Adding a Demo to a Workspace

Demo projects are found in the FITDemos subdirectory of the distribution file for this application note. To add a demo project to a workspace, select *File >> Import >> General >> Existing Projects into Workspace*, then click “Next”. From the Import Projects dialog, choose the “Select archive file” radio button. “Browse” to the FITDemos subdirectory, select the desired demo zip file, then click “Finish”.

4.14 Downloading Demo Projects

Demo projects are not included in the RX Driver Package. When using the demo project, the FIT module needs to be downloaded. To download the FIT module, right click on this application note and select “Sample Code (download)” from the context menu in the *Smart Brower >> Application Notes* tab.

5. Appendices

5.1 Confirmed Operation Environment

This section describes confirmed operation environment for this module.

Table 5.1 Confirmed Operation Environment (Rev. 4.00)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 7.3.0 IAR Embedded Workbench for Renesas RX 4.12.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 4.08.04.201902 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.12.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.4.00
Board used	Renesas Starter Kit for RX113 (product No.: R0K505113xxxxxx) Renesas Starter Kit for RX130 (product No.: RTK5005130xxxxxxxx) Renesas Starter Kit for RX231 (product No.: R0K505231xxxxxx) Renesas Starter Kit for RX23T (product No.: RTK500523Txxxxxxxx) Renesas Starter Kit for RX24T (product No.: RTK500524Txxxxxxxx) Renesas Starter Kit for RX24U (product No.: RTK500524Uxxxxxxxx) Renesas Starter Kit+ for RX64M (product No.: R0K50564Mxxxxxx) Renesas Starter Kit for RX66T (product No.: RTK50566Txxxxxxxxx) Renesas Starter Kit for RX72T (product No.: RTK5572Txxxxxxxxxx) Renesas Starter Kit+ for RX65N (product No.: RTK500565Nxxxxxxxx) Renesas Starter Kit+ for RX65N-2MB (product No.: RTK50565Nxxxxxxxxx)

Table 5.2 Confirmed Operation Environment (Rev. 4.10)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 7.3.0
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
Endian	Big endian/little endian
Revision of the module	Rev.4.10
Board used	Renesas Solution Starter Kit for RX23W (product No.: RTK5523Wxxxxxxxxx)

Table 5.3 Confirmed Operation Environment (Rev. 4.20)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 7.3.0 IAR Embedded Workbench for Renesas RX 4.12.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 4.08.04.201902 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.12.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.4.20
Board used	Renesas Starter Kit+ for RX72M (product No.: RTK5572Mxxxxxxxxxx)

Table 5.4 Confirmed Operation Environment (Rev. 4.30)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 7.4.0 IAR Embedded Workbench for Renesas RX 4.12.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 4.08.04.201902 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.12.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.4.30
Board used	RX13T CPU Card (product No.: RTK0EMXA10xxxxxxxx)

Table 5.5 Confirmed Operation Environment (Rev. 4.40)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 7.5.0 IAR Embedded Workbench for Renesas RX 4.12.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 4.08.04.201902 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.12.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.4.40
Board used	Renesas Solution Starter Kit for RX23E-A (product No.: RTK0ESXB10xxxxxxxx)

Table 5.6 Confirmed Operation Environment (Rev. 4.50)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 7.5.0 IAR Embedded Workbench for Renesas RX 4.12.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 4.08.04.201902 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.12.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.4.50
Board used	Renesas Starter Kit+ for RX72N (product No.: RTK5572Nxxxxxxxxxx)

Table 5.7 Confirmed Operation Environment (Rev. 4.70)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio 2021-01 IAR Embedded Workbench for Renesas RX 4.14.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.02.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202002 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.14.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.4.70
Board used	Renesas Starter Kit+ for RX671 (product No.: RTK55671xxxxxxxxxx)

Table 5.8 Confirmed Operation Environment (Rev. 4.80)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio 2021-07 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.03.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202102 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.4.80
Board used	Target Board for RX140 (product No.: RTK5RX140xxxxxxxxx)

Table 5.9 Confirmed Operation Environment (Rev. 4.90)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio 2022-04 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.04.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202104 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.4.90
Board used	Renesas Starter Kit for RX660 (product No.: RTK55660xxxxxxxxxx)

Table 5.10 Confirmed Operation Environment (Rev. 5.00)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio 2022-10 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.05.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202204 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.5.00
Board used	Renesas Flexible Motor Control Kit for RX26T MCU Group (product No.: RTK0EMXE70xxxxxxxx)

Table 5.11 Confirmed Operation Environment (Rev. 5.10)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio 2022-10 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.05.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202204 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.5.10
Board used	Renesas Solution Starter Kit for RX23E-B (product No.: RTK0ES1001xxxxxxxx)

Table 5.12 Confirmed Operation Environment (Rev. 5.20)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio 2024-07 IAR Embedded Workbench for Renesas RX 5.10.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.06.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202405 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 5.10.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.5.20
Board used	EK Board for RX261 (product No.: RTK5RX261xxxxxxxxx)

5.2 Troubleshooting

(1) Q: I have added this module to the project and built it. Then I got the error: Could not open source file “platform.h”.

A: The FIT module may not be added to the project properly. Check if the method for adding FIT modules is correct with the following documents:

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

When using this module, the board support package FIT module (BSP module) must also be added to the project. Refer to the application note “Board Support Package Module Using Firmware Integration Technology (R01AN1685)”.

(2) Q: I have added this module to the project and built it. Then I got the error:

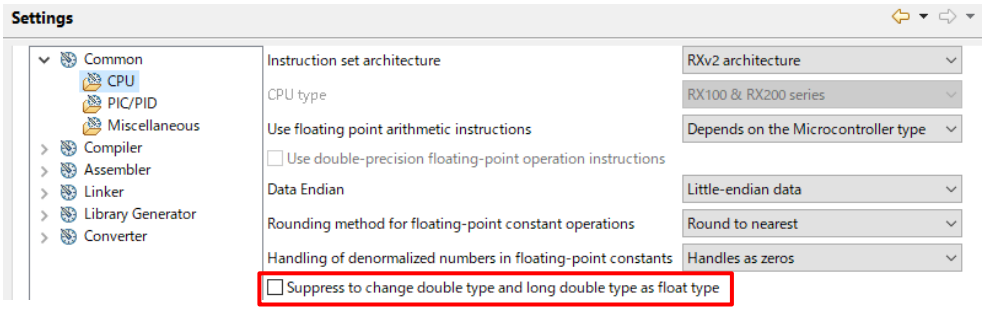
“No data flash on this MCU. Set FLASH_CFG_CODE_FLASH_ENABLE to 1 in r_flash_rx_config.h.”

A: The setting values in the “r_flash_rx_config.h” file could be incorrect. Review the “r_flash_rx_config.h” file and correct any incorrect values. Refer to “2.7 Configuration Overview” for details.

(3) Q: I have added this module to the project, changed the compiler option and built it. Then a ROM access violation is detected.

A: To use this module to run codes from RAM to reprogram the code flash memory, all codes used need to be loaded to the RAM.
Depending on the compiler option setting, the loaded destination may be ROM or RAM.
If the compiler option needs to be changed, confirm by outputting to a list file the fact that the codes may not be loaded to the ROM as a result of the change of the compiler option.
The following shows an example of a ROM access violation due to a change in the compiler option.

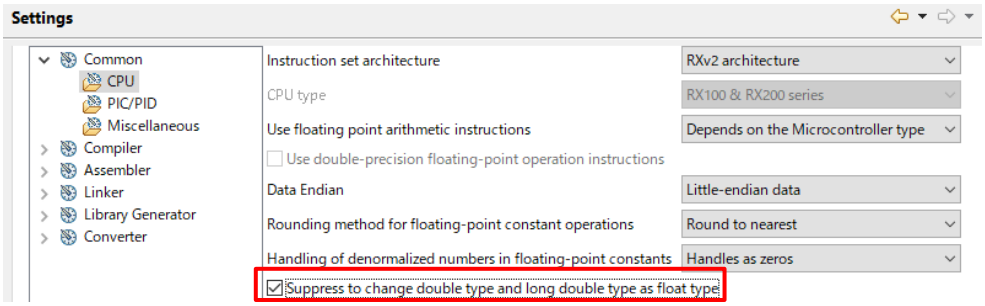
A-1: Default compiler option settings



Output result of the list file of the default compiler option settings

```
000003AC FC472E                 ITOF R2, R14
000003AF FD723E005CC149         FMUL #49C15C00H, R14
000003B6                L127:^  ; if_break_bb33
000003B6 92C5                   MOV.W R5, 14H[R4]
000003B8 FCA7E1                 FTOU R14, R1
000003BB A1C1                   MOV.L R1, 18H[R4]
000003BD 6601                   MOV.L #00000000H, R1
```

A-2: Compiler option change



Output result of the list file after the change in the compiler option

```
000003C1 EF21                   MOV.L R2, R1
000003C3 05rrrrrr           A   BSR __COM_CONV32ud
000003C7 6603                   MOV.L #00000000H, R3
000003C9 FB42802B3841           MOV.L #41382B80H, R4
000003CF 05rrrrrr           A   BSR __COM_MULd
000003D3 754740                 MOV.L #00000040H, R7
000003D6                L127:^  ; if_break_bb33
000003D6 05rrrrrr           A   BSR __COM_CONVd32u
000003DA A1E1                   MOV.L R1, 18H[R6]
000003DC 6601                   MOV.L #00000000H, R1
000003DE 92E7                   MOV.W R7, 14H[R6]
```

A-1 shows a list file of the default compiler option settings, and A-2 shows a list file after the change in the compiler option.

The difference between the A-1 and A-2 compiler option results in the difference between list file output results.

The red frame parts shown in the list file of A-2 indicate that they have been replaced with runtime library functions.

These runtime library functions are positioned in the “P” section by default and are not loaded to RAM.

For that reason, a ROM access violation will occur during program execution.

(4) Q: It is necessary to register a callback function when using non-blocking mode?

A: It is necessary to register a callback function. If no callback function is registered, FLASH_ERR_FAILURE will result when R_FLASH_Erase(), R_FLASH_BlankCheck(), or R_FLASH_Write() is run.

(5) Q: Return does not occur from R_FLASH_Erase() or R_FLASH_Write().
An undefined interrupt (excep_undefined_inst_isr()) is generated.

A: It is possible that another peripheral interrupt was generated and an interrupt handler allocated to an access-prohibited area in the code flash memory was run while R_FLASH_Erase() or R_FLASH_Write() were running. To prevent this, it is necessary to either disable interrupts while reprogramming the code flash memory or reallocate interrupt vector tables and interrupt handlers to the RAM for interrupts that may occur while the code flash memory is being reprogrammed. Refer to Example 1 in section 3.6, R_FLASH_Control(), for a usage example.

5.3 Compiler-Dependent Settings

This module of Rev. 4.00 or later supports multiple compilers. To use this module, different settings are required for each compiler as shown below.

5.3.1 Using Renesas Electronics C/C++ Compiler Package for RX Family

This section describes how to use Renesas Electronics C/C++ Compiler Package for RX Family as the compiler.

The process of setting up the linker sections and mapping from code flash to RAM need to be done in e^2 studio.

When Rev.5.00 or later of this module is used in combination with “e^2 studio 2022-07 or later”, the following settings are automatically performed.

- Sections 1 and 2 of "5.3.1.1 Programming Code Flash from RAM"
- Sections 1 and 2 of "5.3.1.2 Programming Code Flash Using the Dual Bank Function"

When Rev.5.00 or later of this module is used in combination with "e^2 studio 2022-04", a warning will occur, but the above settings will be performed automatically.

When Rev.5.00 or later of this module is used in combination with "e^2 studio 2022-01 or earlier", settings are not automatically performed.

5.3.1.1 Programming Code Flash from RAM

This section describes addition of sections, mapping from code flash to RAM, and placement of programs that operate during code flash re-writing.

1. Add a ‘RPFRAM’ section in a RAM area.

- e^2 studio 2022-04 or earlier

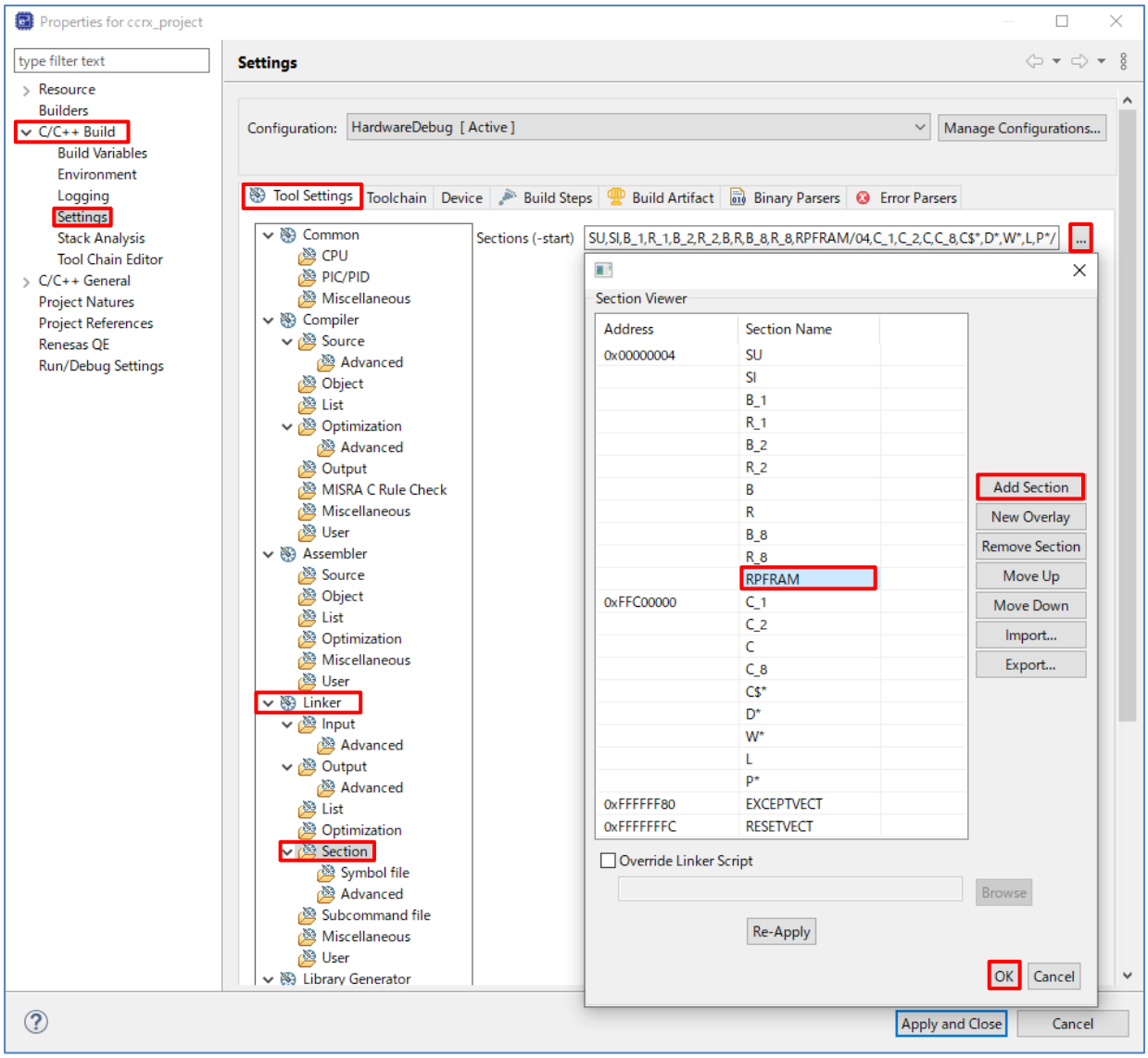
(1) In Project Explorer, click the project you want to debug.

(2) Click File > Properties to open the Properties window.

(3) On the Properties window, click C/C++ Build > Settings.

(4) Select the “Tool Settings” tab, click Linker > Section, and click the […] button to display the Section Viewer window.

(5) On the Section Viewer window, click the [Add Section] button to add a ‘RPFRAM’ section in a RAM area, and then click the [OK] button.



- e^2 studio 2022-07 or later

(1) In Project Explorer, click the project you want to debug.

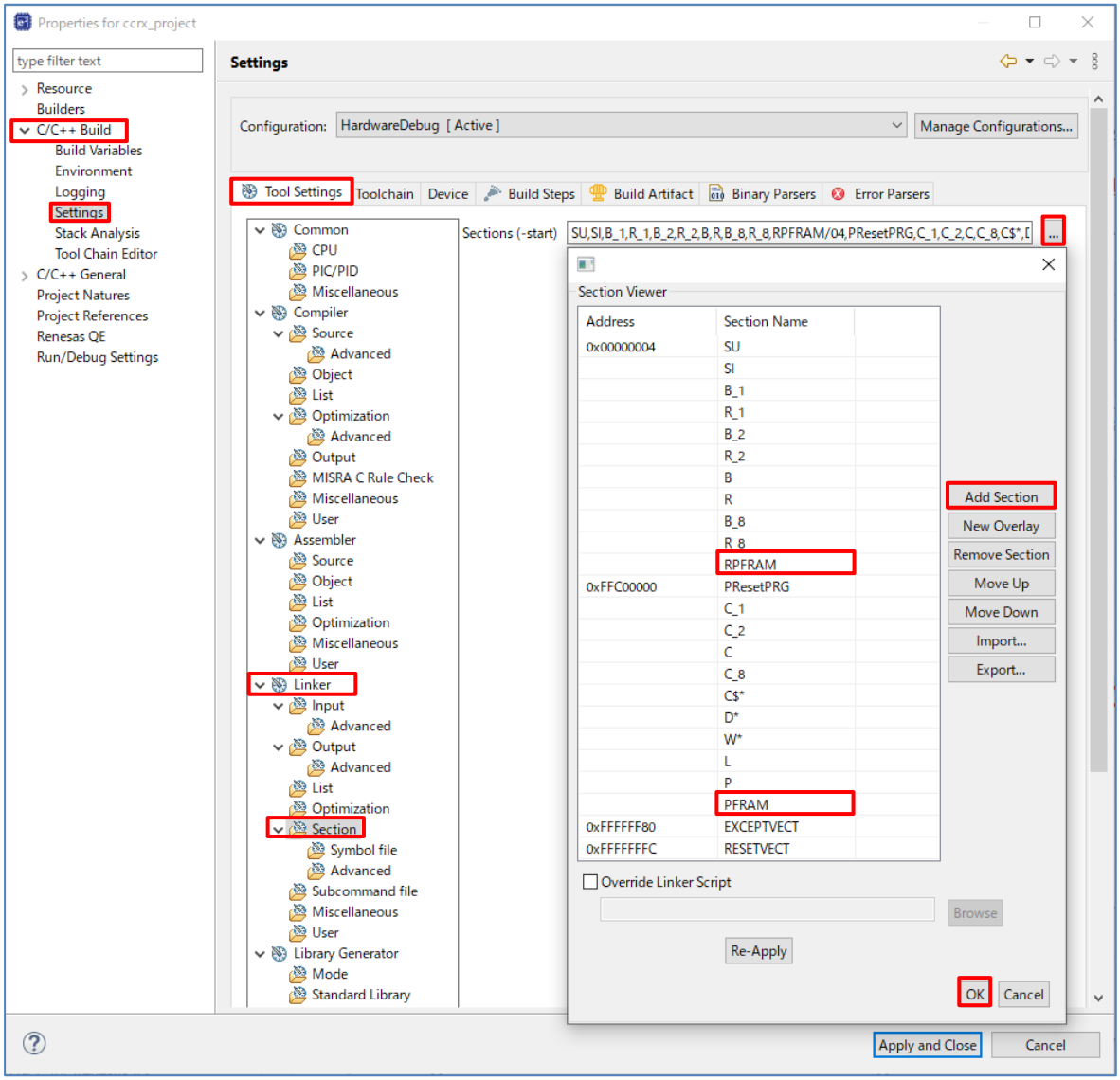
(2) Click File > Properties to open the Properties window.

(3) On the Properties window, click C/C++ Build > Settings.

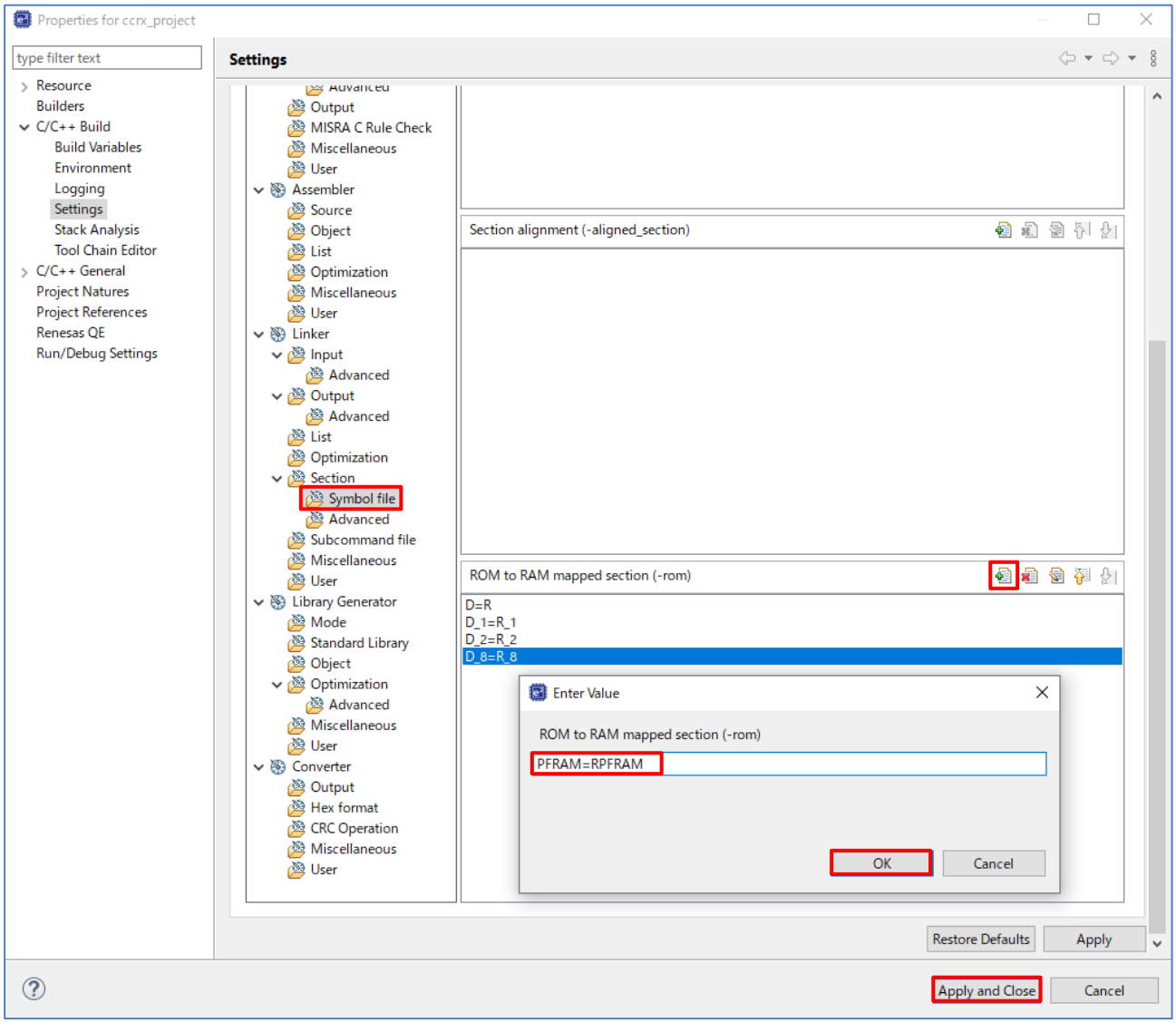
(4) Select the “Tool Settings” tab, click Linker > Section, and click the […] button to display the Section Viewer window.

(5) On the Section Viewer window, click the [Add Section] button to add a ‘RPFRAM’ section in a RAM area.

(6) On the Section Viewer window, click the [Add Section] button to add a ‘PFRAM’ section in a ROM area, and then click the [OK] button.



2. Map the code flash section (PFRAM) address to the RAM section (RPFRAM) address.
   (1) After clicking “Symbol file”, click the “Add” icon of the section to be mapped from ROM to RAM.
   (2) On the Enter Value window, enter ‘PFRAM=RPFRAM’ and then click the [OK] button.
   (3) Click the [Apply and Close] button.



3. Programs that operate during code flash re-writing such as interrupt callback function, etc. need to be placed in the FRAM section.

```
#pragma section FRAM
/* Function that operates during code flash re-writing */
void func(void){…}

/* Callback function that operates during code flash re-writing */
void cb_func(void){…}
#pragma section
```

5.3.1.2 Programming Code Flash Using the Dual Bank Function

This section describes addition of sections, mapping from code flash to RAM.

1. Add a ‘RPFRAM2’ section in a RAM area.

- e2 studio 2022-04 or earlier

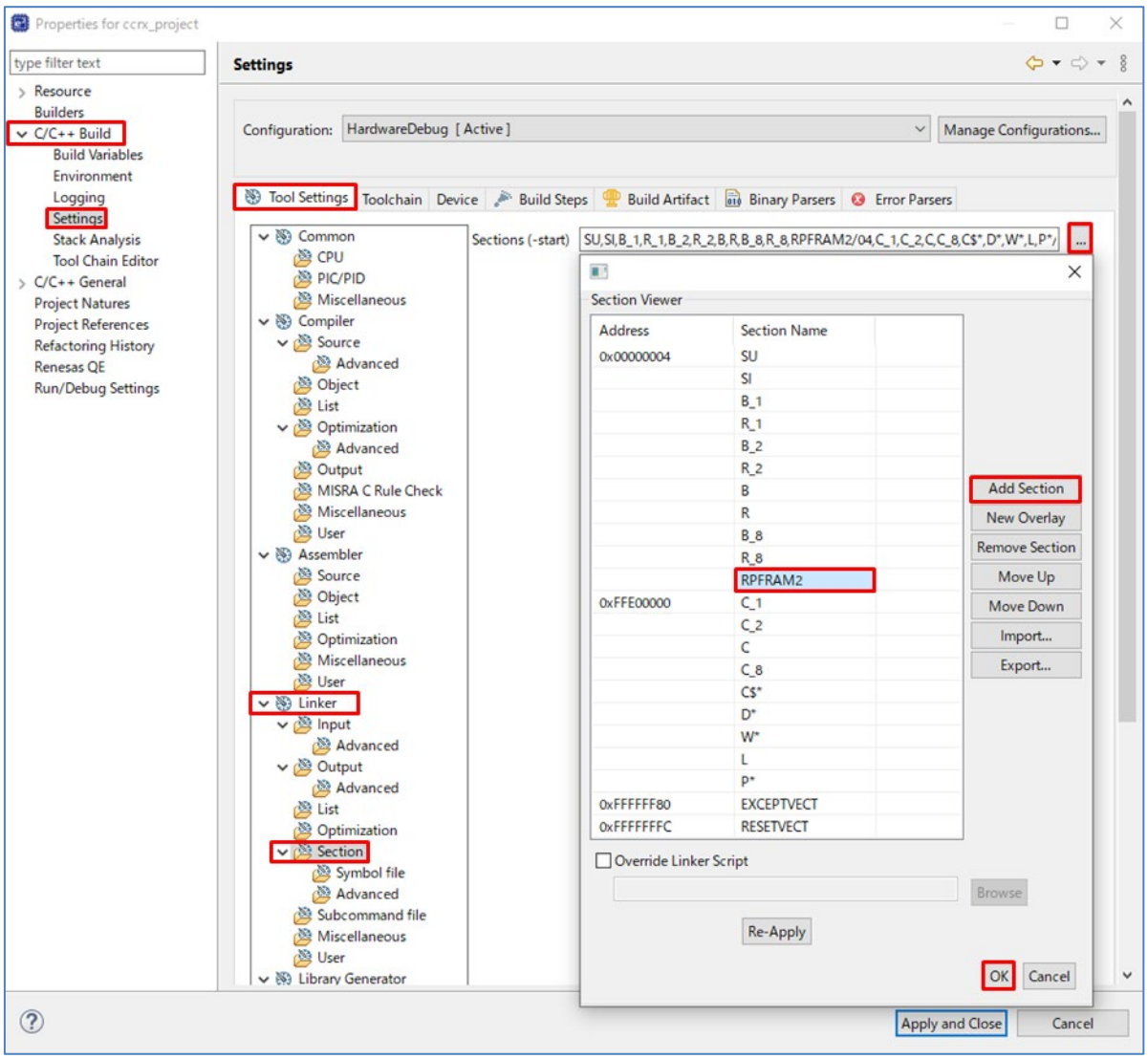
(1) In Project Explorer, click the project you want to debug.

(2) Click File > Properties to open the Properties window.

(3) On the Properties window, click C/C++ Build > Settings.

(4) Select the “Tool Settings” tab, click Linker > Section, and click the […] button to display the Section Viewer window.

(5) On the Section Viewer window, click the [Add Section] button to add a ‘RPFRAM2’ section in a RAM area, and then click the [OK] button.



- e2 studio 2022-07 or later

(1) In Project Explorer, click the project you want to debug.

(2) Click File > Properties to open the Properties window.

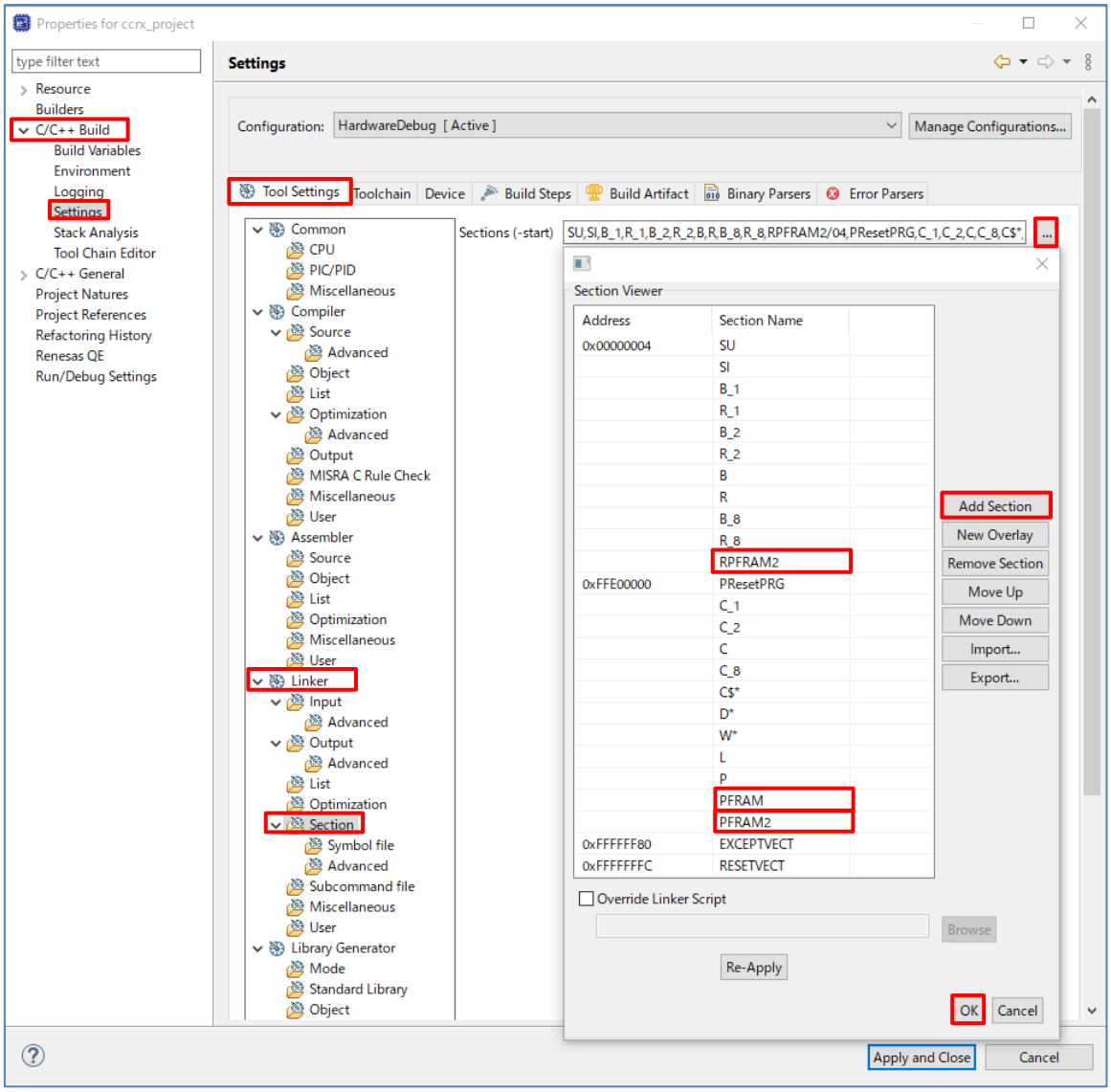
(3) On the Properties window, click C/C++ Build > Settings.

(4) Select the “Tool Settings” tab, click Linker > Section, and click the […] button to display the Section Viewer window.

(5) On the Section Viewer window, click the [Add Section] button to add a ‘RPFRAM2’ section in a RAM area.

(6) On the Section Viewer window, click the [Add Section] button to add a ‘PFRAM’ section in a ROM area.

(7) On the Section Viewer window, click the [Add Section] button to add a ‘PFRAM2’ section in a ROM area, and then click the [OK] button.

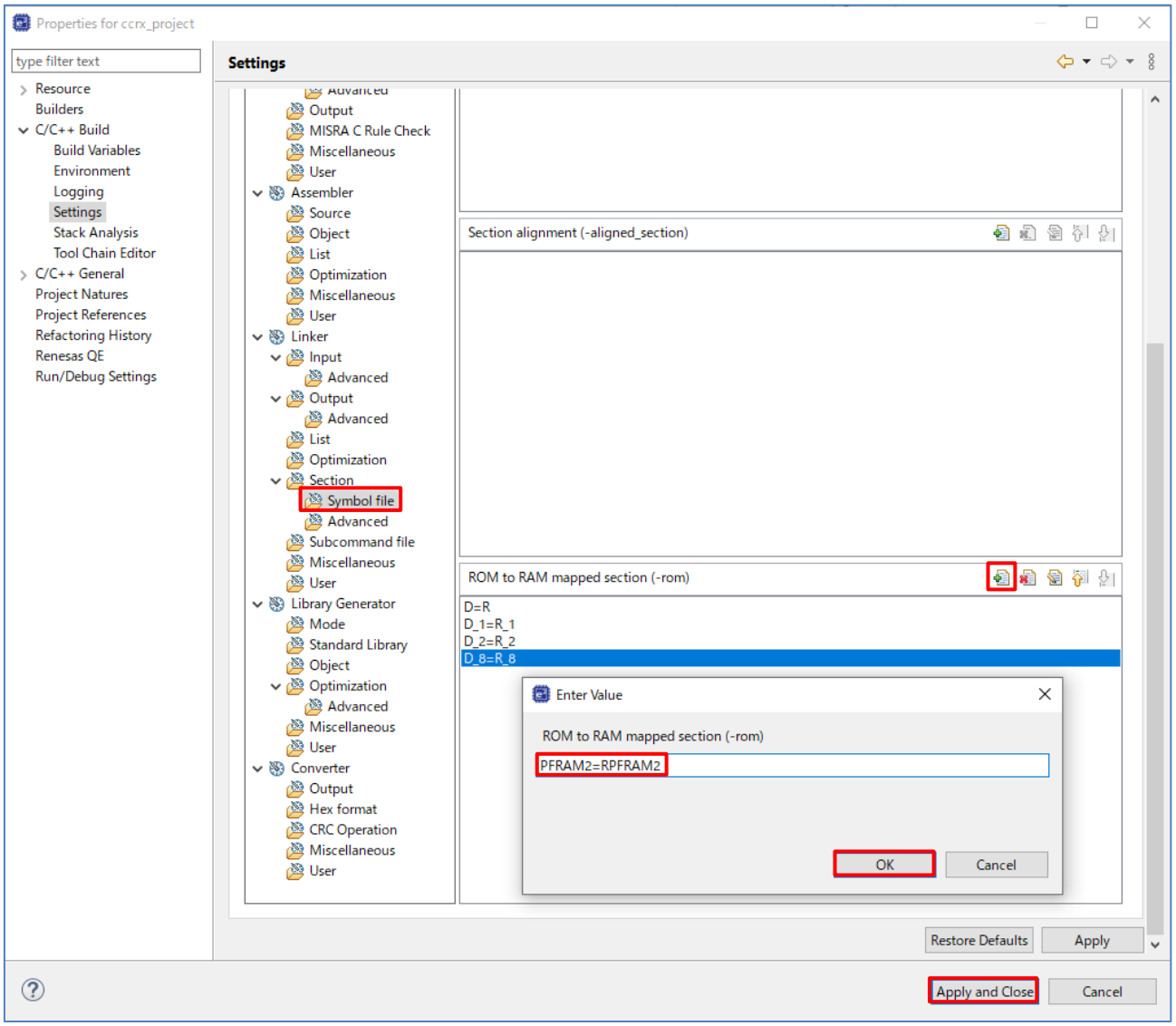


2. Map the code flash section (PFRAM2) address to the RAM section (RPFRAM2) address.

(1) After clicking “Symbol file”, click the “Add” icon of the section to be mapped from ROM to RAM.

(2) On the Enter Value window, enter ‘PFRAM2=RPFRAM2’ and then click the [OK] button.

(3) Click the [Apply and Close] button.



5.3.2 Using GCC for Renesas RX

This section describes how to use GCC for Renesas RX as the compiler.

For the linker setting, it is necessary to edit the linker settings file generated by e^2 studio.

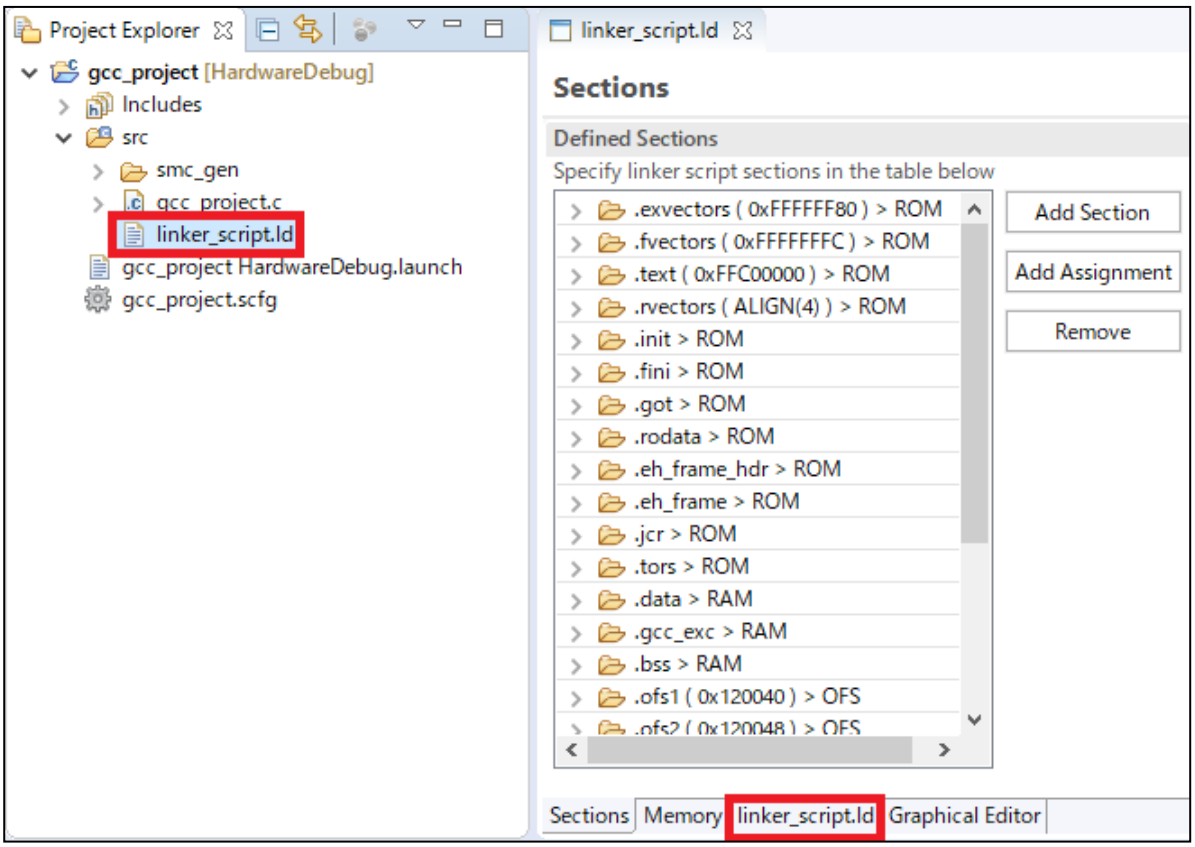
5.3.2.1 Programming Code Flash from RAM

This section describes addition of linker settings and placement of programs that operate during code flash re-writing.

1. Add a setting in the linker settings file (linker_script.ld).

(1) From Project Explorer, right-click the linker settings file (linker_script.ld), and select “Open”.

(2) On the linker_script.id window, click the “linker_script_id” tab.



(3) Add the following (a) to (c) in the linker settings file (linker_script.ld).

(a)
```
. += _edata - _data;
```

(b)
```
.pfram ALIGN(4):
{
     _PFRAM_start = .;
     . += _RPFRAM_end - _RPFRAM_start;
     _PFRAM_end = .;
} > ROM
```

(c)
```
.rpfram ALIGN(4): AT(_PFRAM_start)
{
     _RPFRAM_start = .;
     *(PFRAM)
     . = ALIGN(4);
     _RPFRAM_end = .;
} > RAM
```

linker_script.ld

```
77      .tors :
78      {
79          __CTOR_LIST__ = .;
80          . = ALIGN(2);
81          ___ctors = .;
82          *(.ctors)
83          ___ctors_end = .;
84          __CTOR_END__ = .;
85          __DTOR_LIST__ = .;
86          ___dtors = .;
87          *(.dtors)
88          ___dtors_end = .;
89          __DTOR_END__ = .;
90          . = ALIGN(2);
91          _mdata = .;
92          . += _edata - _data;
93      } > ROM
94      .pfram ALIGN(4):
95      {
96          _PFRAM_start = .;
97          . += _RPFRAM_end - _RPFRAM_start;
98          _PFRAM_end = .;
99      } > ROM
100     .data : AT(_mdata)
101     {
102         _data = .;
103         *(.data)
104         *(.data.*)
105         *(D)
106         *(D_1)
107         *(D_2)
108         _edata = .;
109     } > RAM
110     .rpfram ALIGN(4): AT(_PFRAM_start)
111     {
112         _RPFRAM_start = .;
113         *(PFRAM)
114         . = ALIGN(4);
115         _RPFRAM_end = .;
116     } > RAM
117     .gcc_exc :
118     {
119         *(.gcc_exc)
120     } > RAM
```

Sections | Memory | linker_script.ld | Graphical Editor

2. Programs that operate during code flash re-writing such as interrupt callback function, etc. need to be placed in a FRAM section by specifying the FRAM section for each function.

```
__attribute__((section("PFRAM")))
/* Function that operates during code flash re-writing */
void func(void){…}

__attribute__((section("PFRAM")))
/* Callback function that operates during code flash re-writing */
void cb_func(void){…}
```

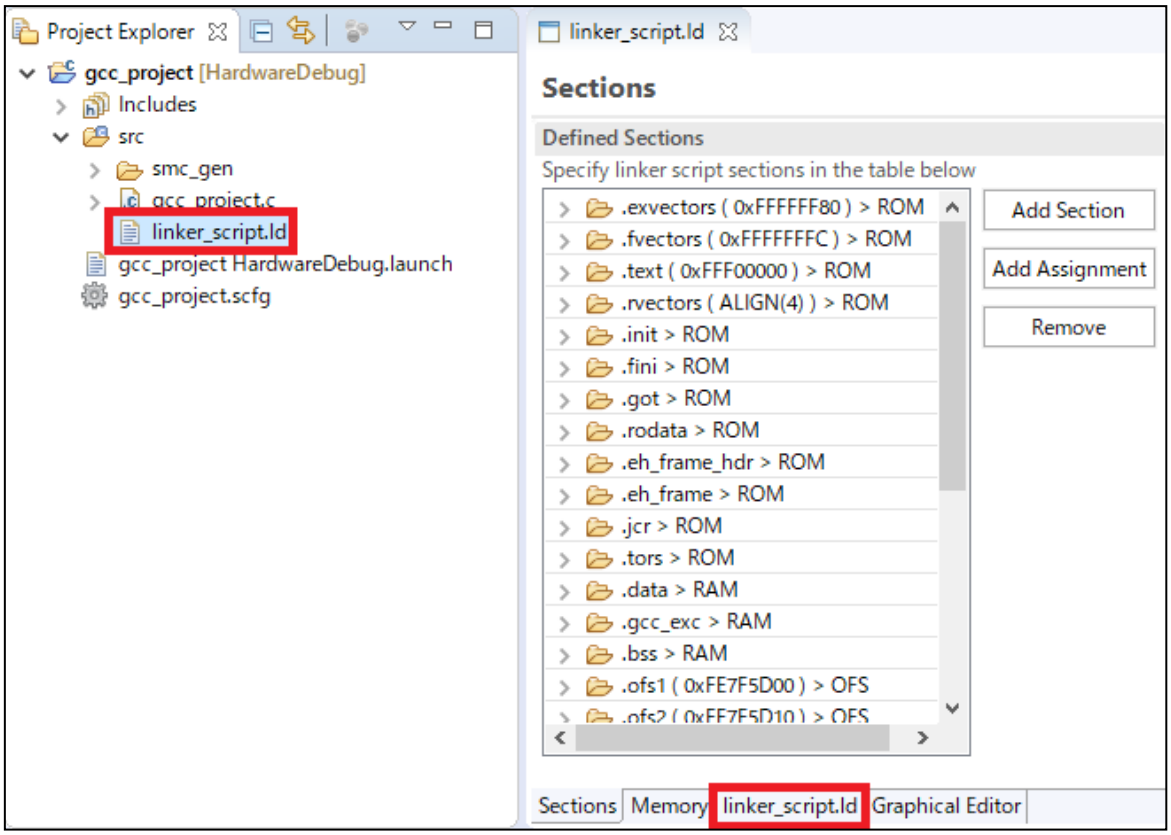
5.3.2.2 Programming Code Flash Using the Dual Bank Function

This section describes addition of linker settings and debugging with the dual bank function.

1. Add a setting in the linker settings file (linker_script.ld).

(1) From Project Explorer, right-click the linker settings file (linker_script.ld), and select “Open”.

(2) On the linker_script.id window, click the “linker_script_id” tab.

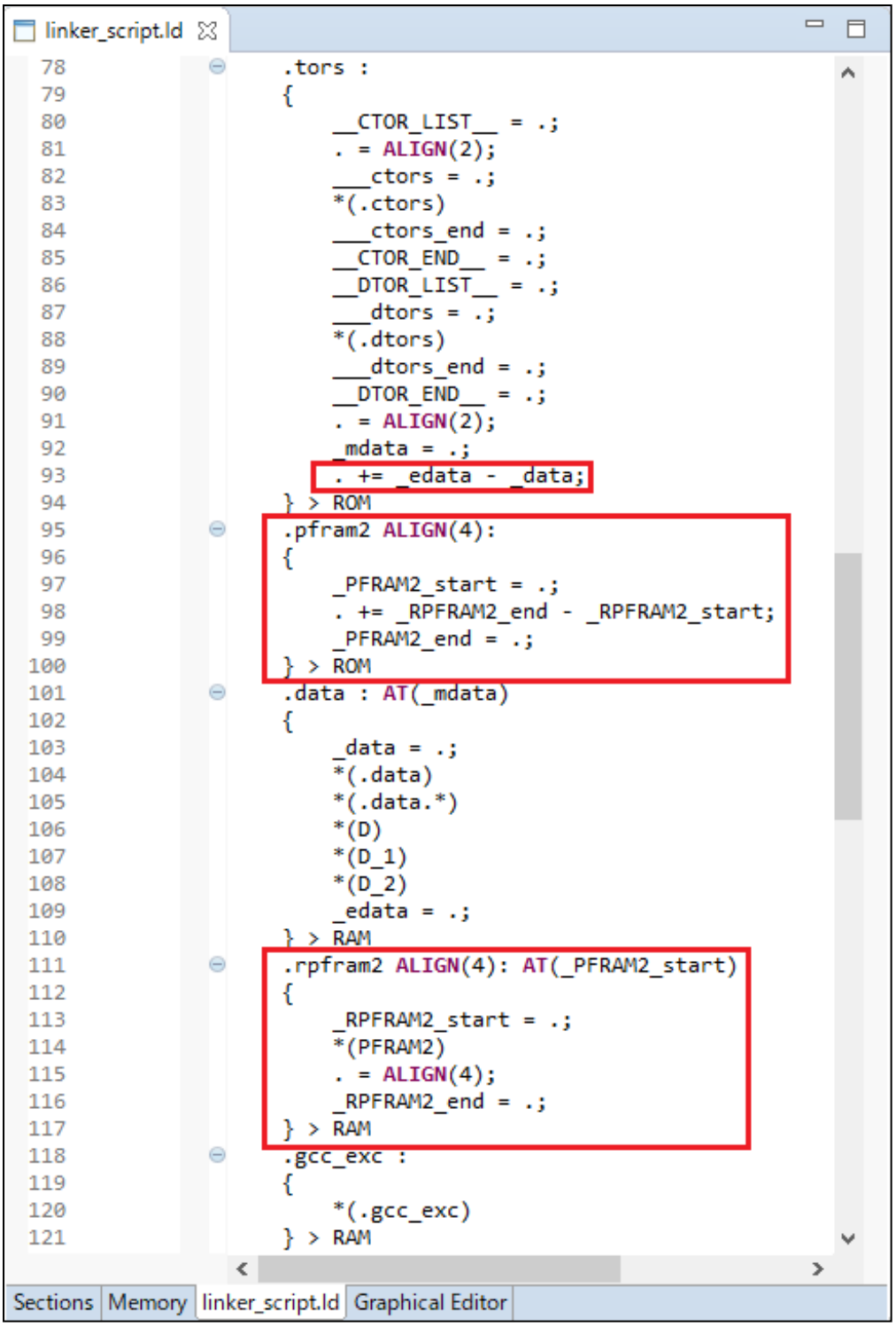


(3) Add the following (a) to (c) in the linker settings file (linker_script.ld).

(a)
```
. += _edata - _data;
```

(b)
```
.pfram2 ALIGN(4):
{
     _PFRAM2_start = .;
     . += _RPFRAM2_end - _RPFRAM2_start;
     _PFRAM2_end = .;
} > ROM
```

(c)
```
.rpfram2 ALIGN(4): AT(_PFRAM2_start)
{
     _RPFRAM2_start = .;
     *(PFRAM2)
     . = ALIGN(4);
     _RPFRAM2_end = .;
} > RAM
```


linker_script.ld
78 .tors :
79 {
80 __CTOR_LIST__ = .;
81 . = ALIGN(2);
82 ___ctors = .;
83 *(.ctors)
84 ___ctors_end = .;
85 __CTOR_END__ = .;
86 __DTOR_LIST__ = .;
87 ___dtors = .;
88 *(.dtors)
89 ___dtors_end = .;
90 __DTOR_END__ = .;
91 . = ALIGN(2);
92 _mdata = .;
93 . += _edata - _data;
94 } > ROM
95 .pfram2 ALIGN(4):
96 {
97 _PFRAM2_start = .;
98 . += _RPFRAM2_end - _RPFRAM2_start;
99 _PFRAM2_end = .;
100 } > ROM
101 .data : AT(_mdata)
102 {
103 _data = .;
104 *(.data)
105 *(.data.*)
106 *(D)
107 *(D_1)
108 *(D_2)
109 _edata = .;
110 } > RAM
111 .rpfram2 ALIGN(4): AT(_PFRAM2_start)
112 {
113 _RPFRAM2_start = .;
114 *(PFRAM2)
115 . = ALIGN(4);
116 _RPFRAM2_end = .;
117 } > RAM
118 .gcc_exc :
119 {
120 *(.gcc_exc)
121 } > RAM
Sections
Memory
linker_script.ld
Graphical Editor

5.3.3 Using IAR C/C++ Compiler for Renesas RX

This section describes how to use IAR C/C++ Compiler for Renesas RX as the compiler.

- Using the Smart Configurator Standalone version

  The Smart Configurator Standalone version is used to generate and use a project for IAR to which this module or BSP is added. Details of Smart Configurator Standalone version are described in the application note “RX Smart Configurator User Guide: IAREW (R20AN0535)”.

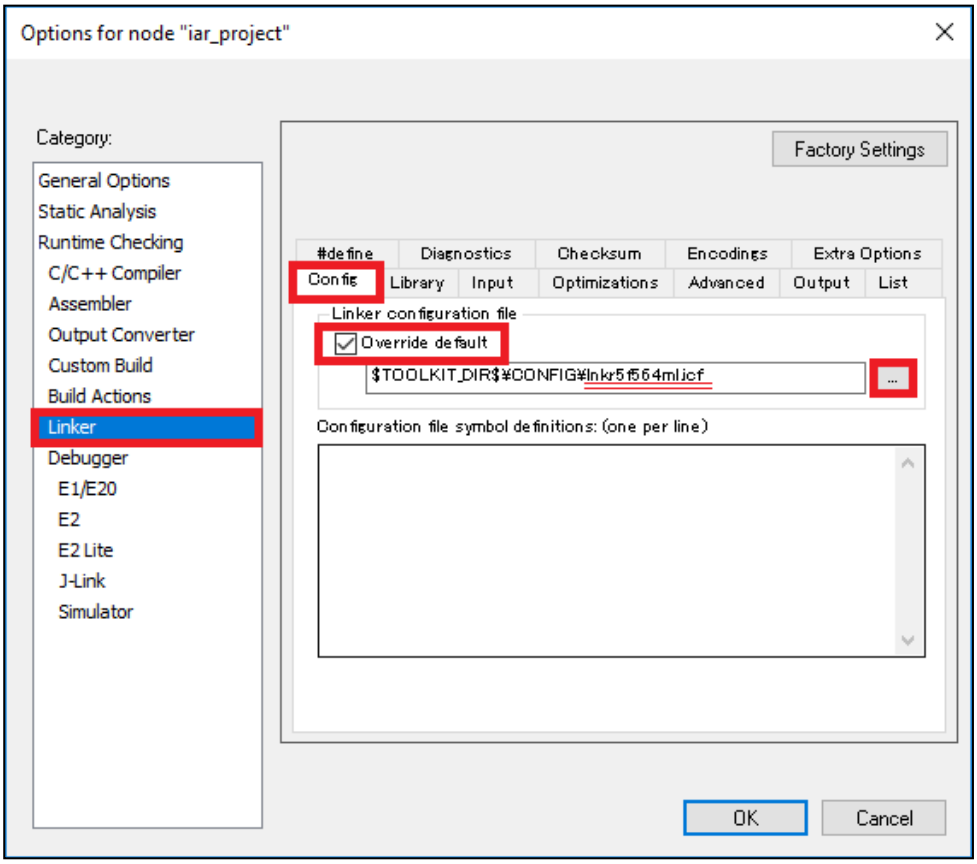
- Using the FIT Module Importer of IAR Embedded Workbench
  FIT Module Importer of IAR Embedded Workbench is used to generate and use a project for IAR to which this module or BSP is added. For details of FIT Module Importer, refer to the latest information on the IAR website.

To use this module for a project for IAR, the following settings are required.

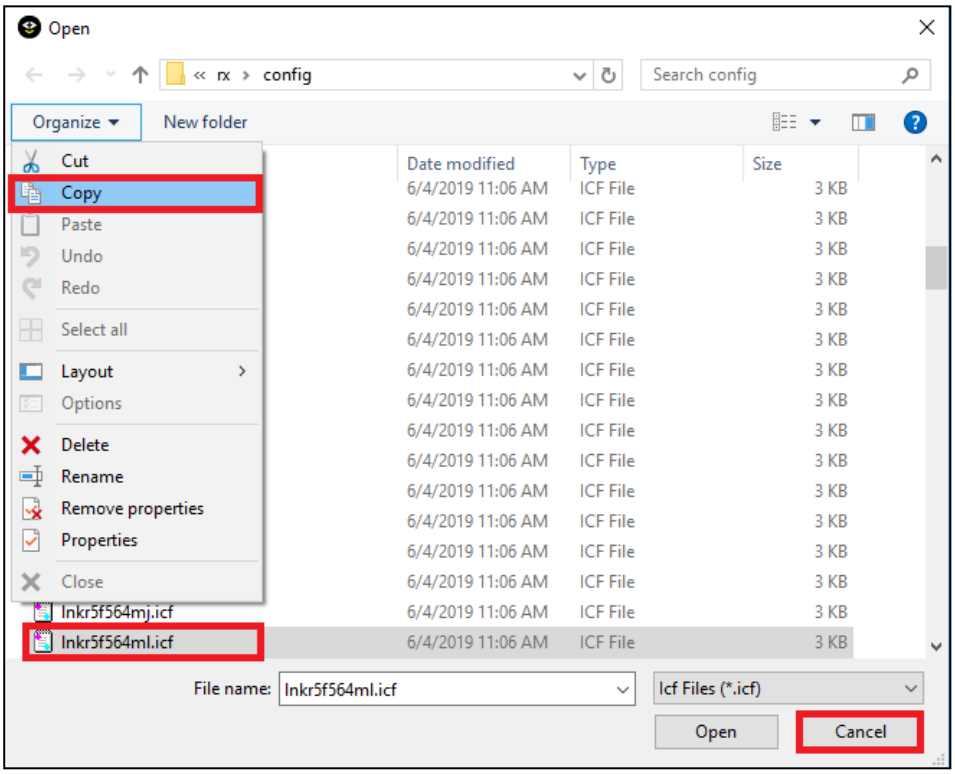
5.3.3.1 Programming Code Flash from RAM

This section describes addition of linker settings and placement of programs that operate during code flash re-writing.

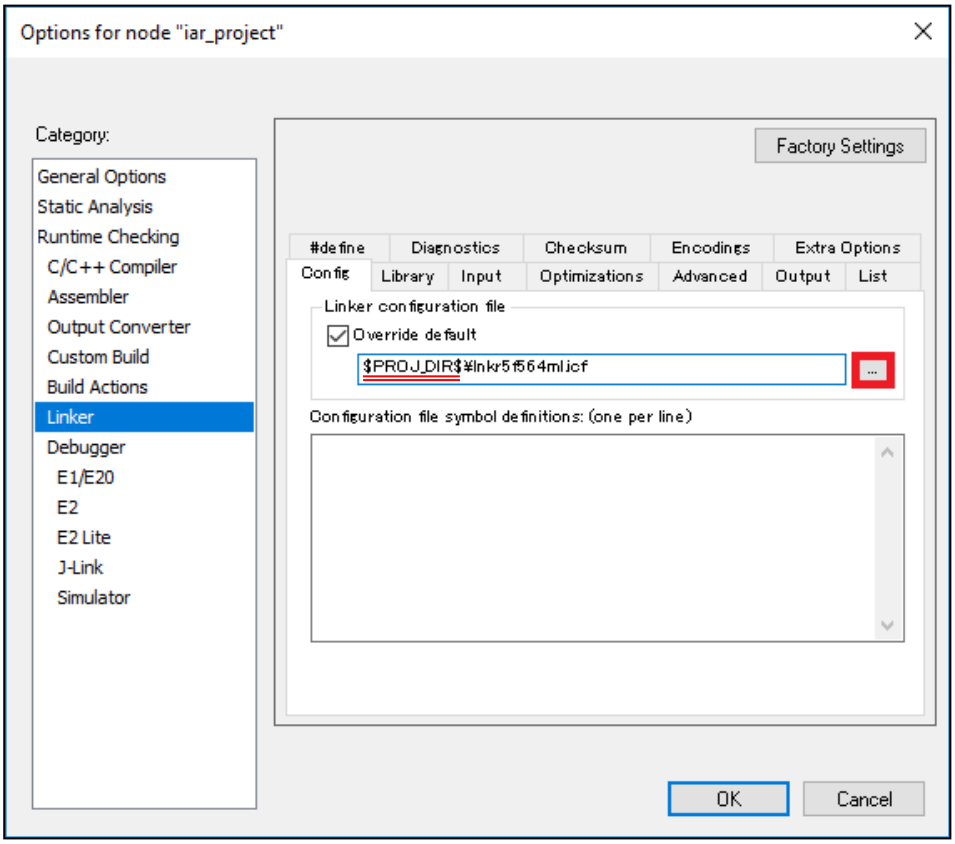
1. Open the Options window of the project for IAR, select “Linker” under “Category:”, and select the “Config” tab. Then, after confirming that the “Override default” check box has been selected, click the […] button.



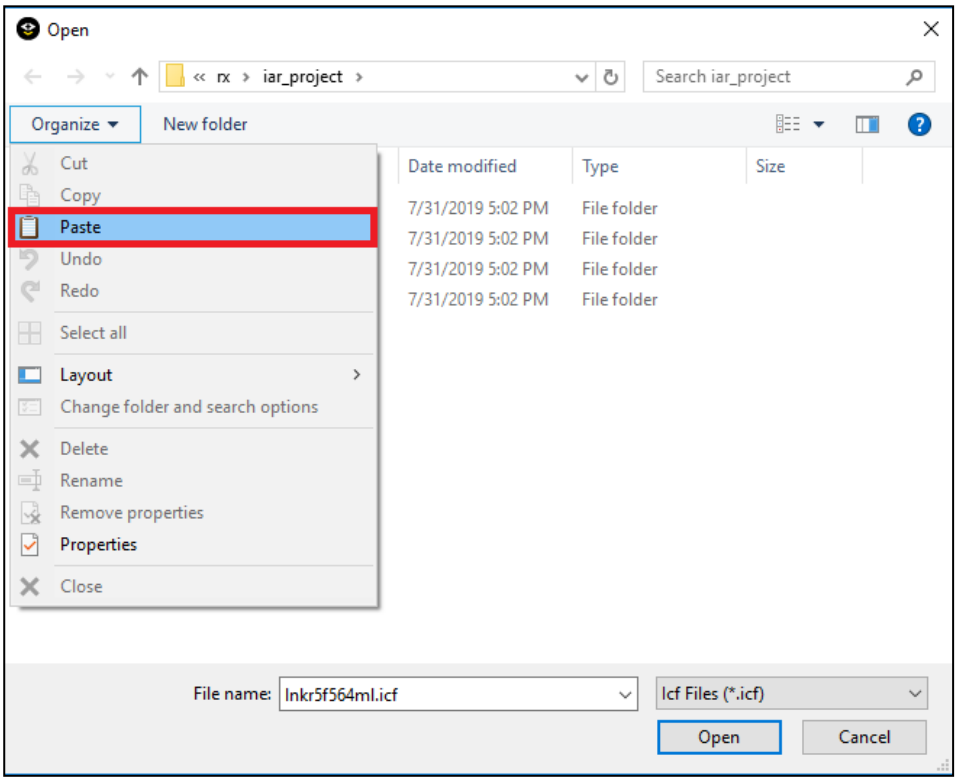
2. On the Open window, copy the .icf file of the target device (the double underlined part of the text box of the linker settings file in the Options window of step 1.), and click the [Cancel] button.



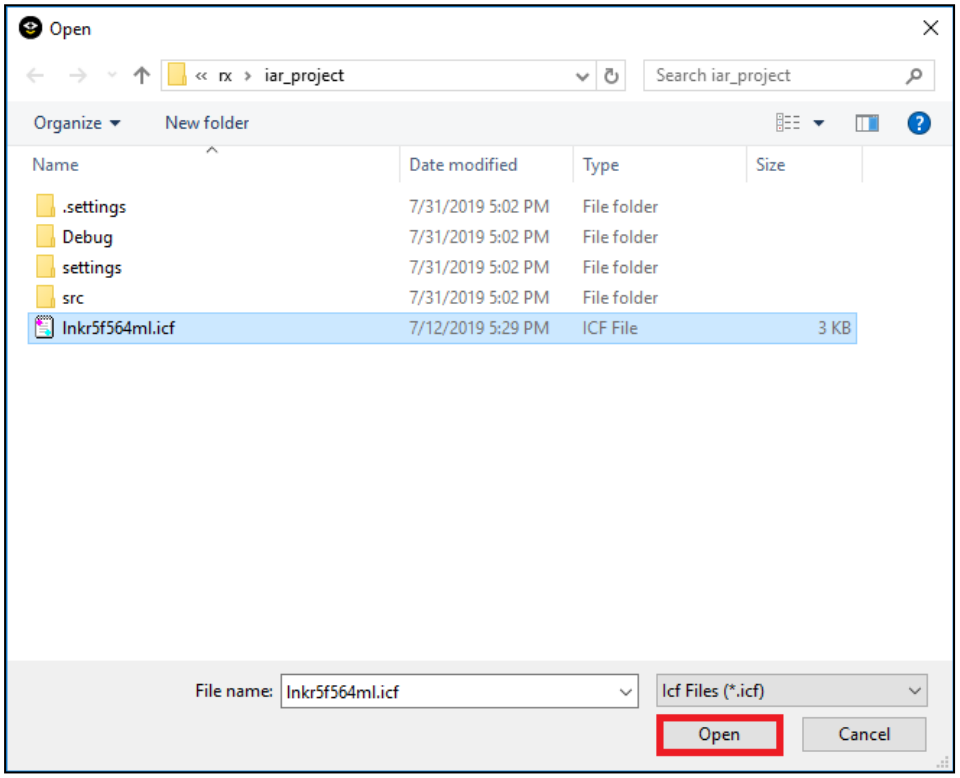
3. Re-write the path to the linker settings file on the Options window to the desired location. (In this example, "$PROJ_DIR$" is used as the path variable to place the file directly under the project folder.) After re-writing, click the […] button again.



4. On the Open window, paste the .icf file of the target device copied in 2 above. (In this example, it is pasted directly under the project folder)



Click the [Open] button.



Now, the default linker settings file was copied and it is ready to edit the copied linker settings file.

5. Copy the following (a) to (d) to add to the replaced linker settings file.

(a)
```
initialize manually { rw section .textrw, section PFRAM };
```

(b)
```
define block PFRAM      with alignment = 4 { section PFRAM };
define block PFRAM_init with alignment = 4 { section PFRAM_init };
```

(c)
```
"ROM32":place in ROM_region32        { ro,
                                       block PFRAM_init };
```

(d)
```
"RAM32":place in RAM_region32        { rw,
                                       ro section D,
                                       ro section D_1,
                                       ro section D_2,
                                       block PFRAM,
                                       block HEAP };
```

lnkr5f564ml.icf

```
//-------------------------------------------------------------------------
// Linker configuration file template for the Renesas RX microcontroller R5F564ML
//-------------------------------------------------------------------------
// Compatibility check
define exported symbol __link_file_version_4 = 1;

define memory mem with size = 4G;


define region RAM_region16 = mem:[from 0x00000004 to 0x00007FFF];
define region RAM_region24 = mem:[from 0x00000004 to 0x0007FFFF];
define region RAM_region32 = mem:[from 0x00000004 to 0x0007FFFF];


define region ROM_region16 = mem:[from 0xFFFF8000 to 0xFFFFFFFF];
define region ROM_region24 = mem:[from 0xFFC00000 to 0xFFFFFFFF];
define region ROM_region32 = mem:[from 0xFFC00000 to 0xFFFFFFFF];

define region DATA_FLASH   = mem:[from 0x00100000 to 0x0010FFFF];

initialize manually { rw section .textrw, section PFRAM };
initialize by copy { rw, ro section D, ro section D_1, ro section D_2 };
initialize by copy with packing = none { section __DLIB_PERTHREAD };
do not initialize  { section .*.noinit };

define block HEAP     with alignment = 4, size = _HEAP_SIZE { };
define block USTACK   with alignment = 4, size = _USTACK_SIZE { };
define block ISTACK   with alignment = 4, size = _ISTACK_SIZE { };

define block PFRAM      with alignment = 4 { section PFRAM };
define block PFRAM_init with alignment = 4 { section PFRAM_init };

define block STACKS with fixed order { block USTACK,
                                       block ISTACK };

place at address mem:0x00120040      { ro section .option_mem };

place at address mem:0xFFFFFFFC      { ro section .resetvect };
place at address mem:0xFFFFFF80      { ro section .exceptvect };

"ROM16":place in ROM_region16        { ro section .code16*,
                                       ro section .data16* };
"RAM16":place in RAM_region16        { rw section .data16*,
                                       rw section __DLIB_PERTHREAD };
"ROM24":place in ROM_region24        { ro section .code24*,
                                       ro section .data24* };
"RAM24":place in RAM_region24        { rw section .data24* };
"ROM32":place in ROM_region32        { ro,
                                       block PFRAM_init };
"RAM32":place in RAM_region32        { rw,
                                       ro section D,
                                       ro section D_1,
                                       ro section D_2,
                                       block PFRAM,
                                       block HEAP };
"STACKS":place at end of RAM_region32 { block STACKS };
```

6. Programs that operate during code flash re-writing such as interrupt callback function, etc. need to be placed in a FRAM section by specifying the FRAM section for each function.

```
#pragma location=”PFRAM”
/* Function that operates during code flash re-writing */
void func(void){…}

#pragma location=”PFRAM”
/* Callback function that operates during code flash re-writing
void cb_func(void){…}
```

5.3.3.2 Programming Code Flash Using the Dual Bank Function

This section describes addition of linker settings.

After performing items 1. to 4. in section 5.3.3.1, perform the following settings.

1. Copy the following (a) to (e) to change and add to the replaced linker settings file.

(a) Changes to the first address of bank 0 of dual mode.

```
define region ROM_region24 = mem:[from 0xFFF00000 to 0xFFFFFFFF];
define region ROM_region32 = mem:[from 0xFFF00000 to 0xFFFFFFFF];
```

(b)
```
initialize manually { rw section .textrw, section PFRAM2 };
```

(c)
```
define block PFRAM2      with alignment = 4 { section PFRAM2 };
define block PFRAM2_init with alignment = 4 { section PFRAM2_init };
```

(d)
```
"ROM32":place in ROM_region32        { ro,
                                       block PFRAM2_init };
```

(e)
```
"RAM32":place in RAM_region32        { rw,
                                       ro section D,
                                       ro section D_1,
                                       ro section D_2,
                                       block PFRAM2,
                                       block HEAP };
```

lnkr5f565ne_dual.icf

```
//-------------------------------------------------------------------------------
// Linker configuration file template for the Renesas RX microcontroller R5F565NE_DUAL
//-------------------------------------------------------------------------------
// Compatibility check
define exported symbol __link_file_version_4 = 1;

define memory mem with size = 4G;

define region RAM_region1  = mem:[from 0x00000004 to 0x0003FFFF];
define region RAM_region2  = mem:[from 0x00800000 to 0x0085FFFF];

define region RAM_region16 = mem:[from 0x00000004 to 0x00007FFF];
define region RAM_region24 = RAM_region1 | RAM_region2;
define region RAM_region32 = RAM_region1 | RAM_region2;

define region STANDBY_RAM  = mem:[from 0x000A4000 to 0x000A5FFF];

define region ROM_region16 = mem:[from 0xFFFF8000 to 0xFFFFFFFF];
define region ROM_region24 = mem:[from 0xFFF00000 to 0xFFFFFFFF];
define region ROM_region32 = mem:[from 0xFFF00000 to 0xFFFFFFFF];

define region DATA_FLASH   = mem:[from 0x00100000 to 0x00107FFF];

initialize manually { rw section .textrw, section PFRAM2 };
initialize by copy { rw, ro section D, ro section D_1, ro section D_2 };
initialize by copy with packing = none { section __DLIB_PERTHREAD };
do not initialize  { section .*.noinit };

define block HEAP     with alignment = 4, size = _HEAP_SIZE { };
define block USTACK   with alignment = 4, size = _USTACK_SIZE { };
define block ISTACK   with alignment = 4, size = _ISTACK_SIZE { };

define block PFRAM2      with alignment = 4 { section PFRAM2 };
define block PFRAM2_init with alignment = 4 { section PFRAM2_init };

define block STACKS with fixed order { block USTACK,
                                       block ISTACK };


place at address mem:0xFE7F5D00      { ro section .option_mem };
place at address mem:0xFFFFFFFC      { ro section .resetvect };
place at address mem:0xFFFFFF80      { ro section .exceptvect };

"ROM16":place in ROM_region16        { ro section .code16*,
                                       ro section .data16* };
"RAM16":place in RAM_region16        { rw section .data16*,
                                       rw section __DLIB_PERTHREAD };
"ROM24":place in ROM_region24        { ro section .code24*,
                                       ro section .data24* };
"RAM24":place in RAM_region24        { rw section .data24* };
"ROM32":place in ROM_region32        { ro,
                                       block PFRAM2_init };
"RAM32":place in RAM_region32        { rw,
                                       ro section D,
                                       ro section D_1,
                                       ro section D_2,
                                       block PFRAM2,
                                       block HEAP };

"STACKS":place at end of RAM_region1 { block STACKS };
```

6. Reference Documents

User's Manual: Hardware
The latest versions can be downloaded from the Renesas Electronics website.

Technical Update/Technical News
The latest information can be downloaded from the Renesas Electronics website.

User's Manual: Development Tools
RX Family C/C++ Compiler CC-RX User's Manual (R20UT3248)
The latest version can be downloaded from the Renesas Electronics website.

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	July.24.14	—	First edition issued
1.10	Nov.13.14	1, 4 7	Added RX113 support. Updated “ROM to RAM” image.
1.11	Dec.11.14	—	Added RX64M to xml support file.
1.20	Dec.22.14	1, 4	Added RX71M support.
1.30	Aug.28.15	All 5, 10	Updated template. Added RX231 support Added flash type 3 code flash run-from-rom info. Fixed RX64M/71M erase boundary issue.
1.40	Sep.03.15	1, 4	Added RX23T support Fixed Big Endian bug in R_DF_Write_Operation() for Flash Type 1. Fixed FLASH_xF_BLOCK_INVALID values for Flash Type 3.
1.50	Nov.11.15	1, 4	Added RX130 support
1.51	Nov.11.15	—	Repackaged demo with BSP v3.10
1.60	Nov.17.15	1, 5 22, 25	Added RX24T support Added ROM cache support Fixed incorrect FLASH_CF_BLOCK_INVALID for RX210/21A/62N/630/63N/63T in code (Flash Type 2).
1.61	May.20.16	10, 11	Added erase/write/blankcheck BGO support for RX64M/71M Fixed lockbit enable/disable commands.
1.62	May.25.16	—	Added lockbit write/read BGO support for RX64M/71M
1.63	Jun.13.16	—	Fixed bug where large flash writes returned success when actually failed (improper timeout handling) on RX64M/71M
1.64	Aug.11.16	—	Fixed RX64M/71M bug where R_FLASH_Control (FLASH_CMD_STATUS_GET, NULL) always returned BUSY. Added #if to exclude ISR code when not in BGO mode.
1.70	Aug.11.16	1, 4-6, 8 —	Added RX651/RX65N support (Flash Type 4) Fixed bug in Flash Type 2 that caused erroneous blankcheck results.
2.00	Aug.17.16	1, 3, 4, 6-9	Added RX230 and RX24T support (Flash Type 1) Added configuration option for operation without FIT BSP. Inserted document sections 2.12.2 thru 2.12.4. Modified values for FLASH_CF_LOWEST_VALID_BLOCK and FLASH_CF_BLOCK_INVALID for Flash TYPE 1.
2.10	Dec.20.16	1, 5-7, 11, 13, 17, 19, 21, 23-26, 31-32	Added RX24U and RX24T-512 support (Flash Type 1) Fixed several minor bugs in all flash types and added more parameter checking. See History in r_flash_rx_if.h for complete list of changes.
3.00	Dec.21.16	8, 9	Merged code common to types 1, 3, and 4 and restructured high level code for cleaner operation. Modified ROM/RAM size tables.

Rev.	Date	Description	
		Page	Summary
3.10	Feb.17.17	5-7, 13-17, 26-28, 35	Added RX65N-2M support. Added sections 2.16 and 2.17.4. Added commands FLASH_CMD_BANK_xxx. Fixed potential “BUSY” return from Flash Type 1 API calls (potential bug with very slow flash). Added clearing of ECC flag during initialization of Flash Type 3.
3.20	Aug.11.17	1, 5, 10-14, 16, 36	Added RX130-512KB support. Added e^{2}studio v6.0.0 differences. Modified driver so mcu_config.h only necessary when not using BSP. Fixed bug in RX65N-2M dual mode operation where sometimes when running in bank 0, performing a bank swap caused application execution to fail.
3.30	Nov.1.17	10, 20 19, 21 32 25	Added FLASH_ERR_ALREADY_OPEN. Added R_FLASH_Close(). Added Flash Type 2 set access window example Added Flash Type 2 blankcheck example.
3.40	Mar.8.18	1, 5, 6 14 14-15 39-40	Added support for RX66T. Added support for new 256K and 384K RX111 and RX24T variants. Updated table numbers in Section 2.14. Added interrupt event enumeration in Section 2.15 Added demos for RDKRX63N, RSKRX66T, and two for RSKRX64M.
3.41	Nov.8.18	6, 31, 36	Added NON_CACHED Control() commands. Added document number of the application note accompanying the sample program of the FIT module to xml file.
3.42	Feb.12.19	38-41	Modified typos in sections 4.1 to 4.12.
3.50	Feb.26.19	1, 5, 6, 31 41	Added support for RX72T. Added demo for RX72T. Fixed write failure bug in RX210 768K and 1M variants.

Rev.	Date	Description	
		Page	Summary
4.00	Apr.19.19	—	Added support for GCC/IAR compiler.
		1, 6	Deleted the following flash type 2 devices from the target device. RX210, RX21A, RX220, RX610, RX621, RX62N, RX62T, RX62G, RX630, RX631, RX63N, RX63T
		1	Deleted the following documents from Related Documents Adding Firmware Integration Technology Modules to e^2 studio Adding Firmware Integration Technology Modules to CS+ Projects Renesas e^2 studio Smart Configurator User Guide
		6	Deleted FLASH_CFG_USE_FIT_BSP. Deleted FLASH_CFG_FLASH_READY_IPL. Deleted FLASH_CFG_IGNORE_LOCK_BITS. Added the explanation of FLASH_CFG_DATA_FLASH_BGO. Added the explanation of FLASH_CFG_CODE_FLASH_BGO.
		7-10	Updated “2.9 Code Size” section.
		11	Deleted the following return values, which are no longer necessary, from “2.11 Return Values” section. FLASH_ERR_ALIGNED FLASH_ERR_BOUNDARY FLASH_ERR_OVERFLOW
		12	Updated “2.12 Adding the FIT FLASH Module to Your Project” section. Added “2.13 Usage Combined with Existing User Projects” section.
		13	Revised and updated as follows the structure of “2.14 Programming Code Flash from RAM” section. “2.14.1 Using Renesas Electronics C/C++ Compiler Package for RX Family”, “2.14.2 Using GCC for Renesas RX”, “2.14.3 Using IAR C/C++ Compiler for Renesas RX”
		22	Added “2.18.4 Emulator Debug Configuration” section.

Rev.	Date	Description	
		Page	Summary
4.00	Apr.19.19	Program	Changed as a result of supporting the GCC/IAR compiler. Changed as a result of deletion of FLASH_CFG_USE_FIT_BSP. Changed as a result of deletion of FLASH_CFG_FLASH_READY_IPL. Changed as a result of deletion of FLASH_CFG_IGNORE_LOCK_BITS. Deleted flash type 2 device from target device. Deleted FLASH_ERR_ALIGNED. Deleted FLASH_ERR_BOUNDARY. Deleted FLASH_ERR_OVERFLOW. Added the process to output error when BSP is earlier than Rev.5.00.
4.10	Jun.07.19	1, 5 7-11 17-18 48 49-50	Added support for RX23W. Updated “2.9 Code Size” section. Updated “2.14.2 Using GCC for Renesas” section. Added “5. Appendices” section. Added “5.1 Confirmed Operation Environment” section. Added “5.2 Troubleshooting” section.
		Program	Added support for RX23W. Modified FEARL and FSARL register settings. Updated the demo project environment.
4.20	Jul.19.19	1, 5 47 50	Added support for RX72M. Added “4.13 flash_demo_rskrx72m_bank0_bootapp / _bank1_otherapp” section. Updated “5.1 Confirmed Operation Environment” section.
		Program	Added support for RX72M. Added the RX72M demo project. Updated the demo project environment. Deleted the warning. Deleted definitions and include, which are no longer used. Granted the volatile declaration to global variables. Modified the section related to dual mode and linear mode. Modified part of Flash Type 4 timeout processing.

Rev.	Date	Description	
		Page	Summary
4.30	Sep.09.19	1, 6 5 7-11 14 15 42 45-62	Added support for RX13T. Added “2.5 Interrupt Vectors” section. Updated “2.10 Code Size” section. Modified the following descriptions and moved to “5.3 Compiler-Dependent Settings”. “2.14.1 Using Renesas Electronics C/C++ Compiler Package for RX Family” “2.14.2 Using GCC for Renesas RX” “2.14.3 Using IAR C/C++ Compiler for Renesas RX” Modified “2.18 Dual Bank Operation” section and moved the content that depends on the compiler to “5.3 Compiler-Dependent Settings”. Updated “5.1 Confirmed Operation Environment” section. Added “5.3 Compiler-Dependent Settings” section.
		Program	Added support for RX13T. Modified part of the flash type 1 error processing. Modified the copy method of R_FlashCodeCopy() when using IAR. Modified the implementation of r_flash_control() to the if_then method.
4.40	Sep.27.19	1, 5, 6 23 42	Added support for RX23E-A. Added FLASH_ERR_NULL_PTR to Return Values in “3.5 R_FLASH_BlankCheck()” section. Updated “5.1 Confirmed Operation Environment” section.
		Program	Added support for RX23E-A. Added the NULL check of the 3rd argument of r_flash_blankcheck().
4.50	Nov.18.19	1, 5, 6 5 15 16 21-37 29-32 45	Added support for RX66N and RX72N. Added limitations to “2.3 Limitations” section. Added “2.13 Blocking Mode and Non-blocking Mode” section. Deleted “2.17 Operations in BGO Mode” section. Deleted description of Reentrant from “3.2 R_FLASH_Open()”, “3.3 R_FLASH_Close()”, “3.4 R_FLASH_Erase()”, “3.5 R_FLASH_BlankCheck()”, “3.6 R_FLASH_Write()”, “3.7 R_FLASH_Control()”, and “3.8 R_FLASH_GetVersion()” sections. Modified the content of Description in “3.7 R_FLASH_Control()” section. Updated “5.1 Confirmed Operation Environment” section.
		Program	Added support for RX66N and RX72N. Supported Doxygen. Modified enabling and disabling IEN to use R_BSP_InterruptRequestEnable() and R_BSP_InterruptRequestDisable().

Rev.	Date	Description	
		Page	Summary
4.60	Jun.24.20	5-9 11 17-23 25-27 29 30 32-36 37-68 77 90	Modified the structure and content of “1. Overview” section. Modified the content of “2.7 Configuration Overview”. Modified the structure and content of section “2.9 Parameters”. Added “2.11 Callback Function” section. Modified the content of “2.13 Blocking Mode and Non-blocking Mode” section. Added “2.14 Region Protection via Access Windows and Lockbits” section. Modified the structure and content of “2.16 Reprogramming Flash Memory” section. Modified the structure and content of “3. API Functions” section. Updated “5.2 Troubleshooting” section. Updated “5.3.3 Using IAR C/C++ Compiler for Renesas RX” section.
		Program	Added processing to determine if R_FLASH_Open() has run. Modified the access window processing, including block 0. Modified the processing so that the enabling/disabling of IEN is performed in the flash module. Modified minor content, such as the deletion of unnecessary definitions.
4.70	Oct.23.20	1, 5, 6, 34, 42, 45, 48, 51 76	Added support for RX671. Updated “5.1 Confirmed Operation Environment” section.
		Program	Added support for RX671.
4.80	Apr.23.21	1, 5, 43, 46, 49 12-17 25 38 78	Added support for RX140. Updated “2.8 Code Size” section. Added the following return values to “2.10 Return Values” section. FLASH_ERR_HOCO Added “Return Values” to “3.1 R_Flash_Open()” section. Updated “5.1 Confirmed Operation Environment” section.
		Program	Added support for RX140.
4.81	Dec.10.21	8 34 46 81	Modified content of “1.3.1 Flash Memory Access Restrictions” section. Modified content of “2.16.1 Reprogramming Code Flash Memory by Running Code from RAM” section. Modified content of “Table 3.3 MCU Groups Supporting Blank Check.” Updated “5.2 Troubleshooting” section.
		Program	Added countermeasures as described in Tool News (R20TS0765 and R20TS0772).

Rev.	Date	Description	
		Page	Summary
4.90	May.13.22	1, 5, 43, 46, 49,	Added support for RX660.
		72	Deleted “4.8 flash_demo_rskrx65n2mb_bank0_bootapp / _bank1_otherapp” section.
		74	Deleted “4.13 flash_demo_rskrx72m_bank0_bootapp / _bank1_otherapp” section.
		79	Updated “5.1 Confirmed Operation Environment” section.
		88	Modified content of “5.3.1.2 Programming Code Flash Using the Dual Bank Function” section.
		92	Modified content of “5.3.2.2 Programming Code Flash Using the Dual Bank Function” section.
		Program	Added support for RX660. Added countermeasures as described in Tool News (R20TS0818). Added countermeasures as described in Technical Update (TN-RX*-A0261A). Deleted “flash_demo_rskrx65n2mb_bank0_bootapp / _bank1_otherapp”. Deleted “flash_demo_rskrx72m_bank0_bootapp / _bank1_otherapp”.
4.91	Dec.23.22	84, 85	Modified content of “5.3.1.1 Programming Code Flash from RAM” section.
		87, 88	Modified content of “5.3.1.2 Programming Code Flash Using the Dual Bank Function” section.

Rev.	Date	Description	
		Page	Summary
5.00	Apr.21.23	1	Added support for RX26T.
		5	Added "Flash Type 5/RX26T" to "Table 1.1 Supported MCU Groups by Flash Type" in "1.1.1 Flash Types Overview" section.
		6	Added "Flash Type 5" to "Table 1.2 Supported Features by Flash Type" in "1.1.2 Supported Features" section.
		10	Added "Flash Type 5" to "Table 2.1 Interrupt Vectors Used in this Module" in "2.4 Interrupt Vector" section.
		12-18	Updated "2.8 Code Size" section.
		30	Added "Table 2.5 Flash Type 5 Callback Function Arguments" in "2.11 Callback Function" section.
		33	Added "Flash Type 5" in "2.14.1 Access Window-based Region Protection" section.
		37	Added "Flash Type 5/RX26T" in "2.16.2 Reprogramming Code Flash Memory by Running Code from Code Flash Memory" section.
		38	Added "Flash Type 5" and modfied content in "2.16.3 Reprogramming Code Flash Memory by Utilizing the Dual Bank Function" section.
		39	Added "2.16.3.1 Reprogramming the other Bank different from the Startup Bank by Running Code from the Startup Bank" section.
		40	Added "2.16.3.2 Reprogramming the Startup Bank and the other Bank by Running Code from RAM" section.
		42	Updated "Table 3.1 Code Allocations in Relation to Configuration of Configuration Options" in "3.1 R_FLASH_Open()".
		46	Added "RX26T" to "Table 3.2 Block Sizes by MCU Group" in “3.3 R_FLASH_Erase()” section.
		49	Added "RX26T" to "Table 6.1 MCU Groups Supporting Blank Check" in "3.4 R_FLASH_BlankCheck()" section.
		52	Added "RX26T" to "Table 6.2 Minimum Programming Sizes by MCU Group" in "3.5 R_FLASH_Write()" section.
		55	Added "Flash Type 5" to "Table 3.5 Supported Commands by Flash Type" in "3.6 R_FLASH_Control()" section.
		63-64	Added "Table 3.9 Details of Commands Supported by Flash Type 5" in "3.6 R_FLASH_Control()" section.
		84	Added "Table 5.10 Confirmed Operation Environment (Rev. 5.00)" in "5.1 Confirmed Operation Environment" section.
		88	Updated "5.3.1 Using Renesas Electronics C/C++ Compiler Package for RX Family" section.
		Program	Added support for RX26T. Added countermeasures as described in Tool News (R20TS0872).
5.10	May.19.23	1, 5, 46, 49, 52	Added support for RX23E-B.
		85	Updated “5.1 Confirmed Operation Environment” section.
		Program	Added support for RX23E-B.
5.11	Oct.01.23	Program	Added countermeasures as described in Tool News (R20TS0963).
5.12	Jan.17.24	Program	Modified the dual mode definition of RX671 code flash memory 1 Mbyte.

Rev.	Date	Description	
		Page	**Summary**
5.20	Jul.30.24	1, 5, 46, 49, 52 85	Added support for RX260 and RX261. Updated “5.1 Confirmed Operation Environment” section.
		Program	Added support for RX260 and RX261. Added countermeasures as described in Technical Update (TN-RX*-A0274A). Removed the DFLWAITR register settings. Modified FISR register settings. When using the GCC or IAR compiler in non-blocking mode, ROM access occurs during P/E mode, so the following functions have been modified. flash_InterruptRequestDisable() flash_InterruptRequestEnable()
5.21	Nov.15.24	32	Added "2.13 for, while and do while statements"
		Program	Added WAIT_LOOP comments.

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

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

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

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

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

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

(Rev.5.0-1 October 2020)

Corporate Headquarters

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

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/.