RISC-V MCU

Renesas Flash Driver

Introduction

This application note describes a flash programming software module which is based on Software Integration System (SIS) technology.
This module has been developed to allow users of supported devices to easily implement flash memory self-programming[*1].
This application note describes how-to use this module and integrate it within an application program.

[*1] Self-programming is a method of reprogramming flash memory by the user applications.

Target Devices

R9A02G021

Target Compilers

- LLVM for RISC-V

For details on the tested environment please refer to section "4.1 Confirmed Operation Environment".

Related Documents

- Board Support Package Using Software Integration System (R01AN7177)

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	R9A02G021

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
Program	Programs the specified region.	✔
Erase	Erases the specified region.	✔
Blank check	Checks that a specified region is not programmed.	✔
Access window	Sets only specified regions as reprogrammable and protects the other regions.	✔[*1]
Startup program protection	Swaps the region containing the startup program after a reset to protect the startup region.	✔
Flash sequencer reset	Resets the flash sequencer.	✔

[*1] Access window can only be used on 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.

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
Data flash memory	Data flash memory	Code flash memory RAM

[*1] Excluding data flash memory, the reprogramming code and interrupt vector tables should be allocated in regions with read access. i.e. RAM.

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

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 section 4.3.1.1 for a usage example.

1.3.2 Clock limitation when reprogramming the flash memory

Do not modify the clock settings between the execution of the R_FLASH_Open function call and the completion of the R_FLASH_Close function call.

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 BSP module.

- Board Support Package (r_bsp) v1.00 or later.

2.3 Supported Toolchains

This module has been confirmed to work with the toolchain listed in 4.1 Confirmed Operation Environment.

2.4 Interrupt Vector

When the FLASH_CFG_DATA_FLASH_MODE or FLASH_CFG_CODE_FLASH_MODE configuration option (see section 2.7) is set to NON_BLOCKING(“1”), enable the interrupts shown in Table 2.1 below. When using in non-blocking mode, set the interrupt vector to be used. Refer to “interrupt Settings” in “RISC-V MCU Smart Configurator User’s Guide: e^2 studio (R20AN0730)” for details.

Table 2.1 Interrupt Vectors Used in this Module

Flash Type	Interrupt Vector
1	FCU_FRDYI interrupt (vector no.:21,29,37,45)

2.5 Header Files

All API calls and their supporting interface definitions are in “r_flash_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_config.h” file.

2.6 Integer Types

This project uses ANSI C99 “Exact width integer types” 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_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_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_MODE *Default value is “0”.	Specifies the processing method for data flash memory. A value of BLOCKING (“0”) processes data flash memory in blocking mode. A value of NON_BLOCKING (“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_MODE. Refer to section 2.13 for details on blocking mode and non-blocking mode.
FLASH_CFG_CODE_FLASH_MODE *Default value is “0”.	Specifies the processing method for code flash memory. A value of BLOCKING (“0”) processes code flash memory in blocking mode. A value of NON_BLOCKING (“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_MODE. Refer to section 2.13 for details on blocking mode and non-blocking mode.

2.8 Code Size

The ROM size, RAM size, and the maximum stack size of this module are described in the following table.

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 Rev.1.00
Compiler Version: LLVM for RISC-V V17.0.2.202401

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	
		LLVM for RISC-V	
Device	Category	With Parameter Checking	Without Parameter Checking
R9A02G021	ROM	6318 bytes	5654 bytes
	RAM	6120 bytes	5574 bytes
	STACK	148 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_MODE (NON_BLOCKING) FLASH_CFG_CODE_FLASH_MODE (NON_BLOCKING)			

Flash Type 1: ROM, RAM and Stack Code Sizes (Minimum Size)			
		Memory Used	
		LLVM for RISC-V	
Device	Category	With Parameter Checking	Without Parameter Checking
R9A02G021	ROM	2544 bytes	2250bytes
	RAM	40 bytes	
	STACK	72 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_MODE (BLOCKING) FLASH_CFG_CODE_FLASH_MODE (BLOCKING)			

2.9 Parameters

This section defines the structure and enumeration used for API function arguments.

2.9.1 Definitions

Structures and enumerations used as module arguments are defined in “r_flash_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_ERR_FAILURE,            // Error during program or erase process
    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 driver API
    FLASH_CMD_SET_BLOCKING_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_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_BLANK,                        // R_FLASH_BlankCheck result - blank
    FLASH_RES_NOT_BLANK                     // R_FLASH_BlankCheck result - not blank
} flash_res_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;
```

```
/* Selected of flash memory to be processed */
typedef enum _flash_type
{
    FLASH_TYPE_CODE_FLASH = 0,          // Specify Code Flash
    FLASH_TYPE_DATA_FLASH,              // Specify Data Flash
    FLASH_TYPE_INVALID                  // Abnormally specified flash memory.
} flash_type_t;
```

2.9.2 Definitions of Flash Memory Functionality and Capacity

The defined macros and enumerative arguments to be used as API parameters depend on the flash memory functionality and capacity. The provided definitions for R9A02G021 MCUs are listed below.

File name: r_flash\src\targets\r9a02g021\r_flash_r9a02g021.h

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

#define MCU_ROM_SIZE_BYTES            (131072)
#define MCU_RAM_SIZE_BYTES            (16384)

#define FLASH_NUM_BLOCKS_DF           (4)
#define FLASH_DF_MIN_PGM_SIZE         (1)
#define FLASH_CF_MIN_PGM_SIZE         (8)

#define FLASH_DF_BLOCK_SIZE           (1024)
#define FLASH_CF_BLOCK_SIZE           (2048)
#define FLASH_DF_FULL_SIZE            (FLASH_NUM_BLOCKS_DF*FLASH_DF_BLOCK_SIZE)

#define FLASH_NUM_BLOCKS_CF           (MCU_ROM_SIZE_BYTES / FLASH_CF_BLOCK_SIZE)

- omitted -

typedef enum _flash_block_address
{
    FLASH_CF_BLOCK_0       = 0x00000000,    /*   2KB: 0x00000000 - 0x000007FF */
    FLASH_CF_BLOCK_1       = 0x00000800,    /*   2KB: 0x00000800 - 0x00000FFF */

- omitted -

    FLASH_CF_BLOCK_63      = 0x0001F800,    /*   2KB: 0x0001F800 - 0x0001FFFF */
    FLASH_CF_BLOCK_END     = 0x0001FFFF,    /*   End of Code Flash Area       */
    FLASH_CF_BLOCK_INVALID = FLASH_CF_BLOCK_END,

    FLASH_DF_BLOCK_0       = 0x40100000,    /*   1KB: 0x40100000 - 0x401003FF */
    FLASH_DF_BLOCK_1       = 0x40100400,    /*   1KB: 0x40100400 - 0x401007FF */
    FLASH_DF_BLOCK_2       = 0x40100800,    /*   1KB: 0x40100800 - 0x40100BFF */
    FLASH_DF_BLOCK_3       = 0x40100C00,    /*   1KB: 0x40100C00 - 0x40100FFF */
    FLASH_DF_BLOCK_INVALID = FLASH_DF_BLOCK_0 + FLASH_DF_FULL_SIZE
} flash_block_address_t;

- omitted -
```

Use these definitions as the arguments for the module's 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_if.h".

```
/* Flash Driverreturn 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_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_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 FCU_FRDYI 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 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 FCU_FRDYI interrupt processing and indicates completion of the erase process.
FLASH_INT_EVENT_WRITE_COMPLETE	Called by the FCU_FRDYI interrupt processing and indicates completion of the program process.
FLASH_INT_EVENT_BLANK	Called by the FCU_FRDYI interrupt processing and indicates that the blank check resulted in a blank state.
FLASH_INT_EVENT_NOT_BLANK	Called by the FCU_FRDYI interrupt processing and indicates that the blank check resulted in a non-blank state.
FLASH_INT_EVENT_TOGGLE_STARTUPAREA	Called by the FCU_FRDYI interrupt processing and indicates completion of swapping the startup region.
FLASH_INT_EVENT_SET_ACCESSWINDOW	Called by the FCU_FRDYI interrupt processing and indicates completion of configuring the access window.
FLASH_INT_EVENT_ERR_FAILURE	Called by the FCU_FRDYI interrupt processing and indicates an error occurred during the program or erase process.

2.12 Adding the Software Integration System (SIS) 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) below.

(1) Adding the Flash driver to your project using Smart Configurator in e^2 studio
By using the Smart Configurator in e^2 studio, the Flash driver is automatically added to your project. Refer to "RISC-V MCU Smart Configurator User's Guide: e^2 studio (R20AN0730)" for details.

2.13 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.

2.13.1 Using in Blocking Mode

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

- FLASH_CFG_DATA_FLASH_MODE: BLOCKING
- FLASH_CFG_CODE_FLASH_MODE: BLOCKING

2.13.2 Using in Non-blocking Mode

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

- FLASH_CFG_DATA_FLASH_MODE: NON_BLOCKING
- FLASH_CFG_CODE_FLASH_MODE: NON_BLOCKING

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_BLOCKING_CALLBACK command for the argument of R_FLASH_Control(). (Refer to section 3.6 for details.)
Table 2.3 describes the API functions that send notification of processing results via the callback function.

Table 2.3 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()	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

A FCU_FRDYI 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.

When reprogramming data flash in non-blocking mode, the interrupt handler routine in the BSP must be assigned to the selected interrupt vector via the IELSRn register. The following files shall be edited.

\smc_gen\general\r_cg_inthandler.c

The following is an example of assignment to vector number 21. (IRQ number 2, vector offset 0x54 in the interrupt vector table). Add the exception handler function call between the ‘Start user code’/’End user code’ auto-generated comment lines, as shown below.

```
/*
 * INT_IELSR2 (0x54)
 */
void INT_IELSR2(void)
{
    /* Start user code for INT_IELSR2. Do not edit comment generated here */
    void Excep_FCU_FRDYI(void);
    Excep_FCU_FRDYI();
    /* End user code. Do not edit comment generated here */
}
```

2.14 Region Protection via Access Windows

A regions of MCU flash memory can be protected by using the access window, to prevent unintentional erasure or overwrite. API functions in this module support the following features.

2.14.1 Access Window-based Region Protection

Regions can be protected by using access window function in Flash Type 1 products.

The access window configuration is defined by specifying the start and end addresses of the flash blocks (region) to be protected.

The region defined by the start and end addresses can be re-programmed by the application. The application shall take care of defining the proper access window for areas of flash which shhall be write protected.

All regions are by default reprogrammable since the access window is not configured, until the registers are programmed with non-default values.

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

2.15 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 SIS modules to an existing user project without creating a new project.

BSP and this module (as applicable, other peripheral SIS 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 SISmodules 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 “Board Support Package Using Software Integration System (R01AN7177)” for details.

2.16 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 code or data flash memory.

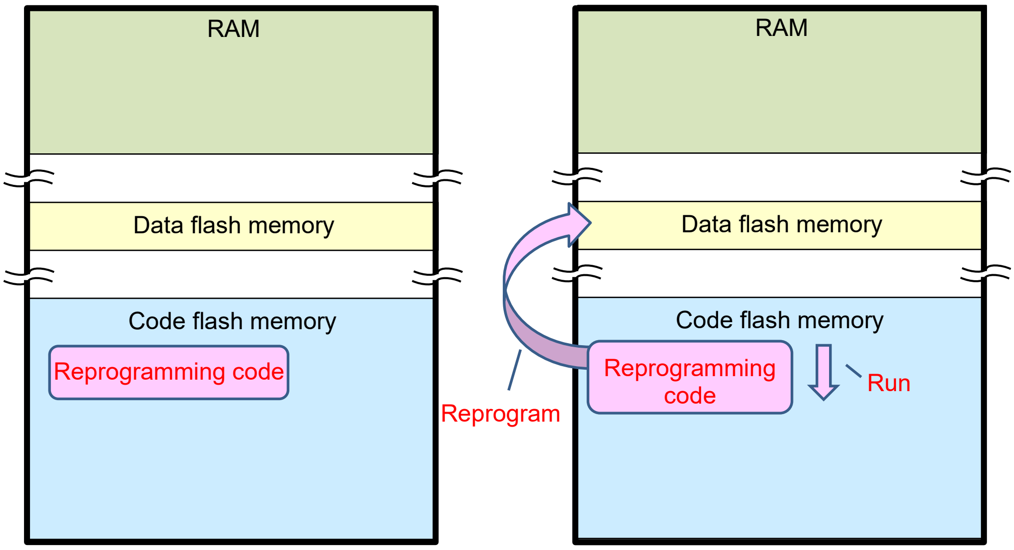


Figure 2.1 Location of Code Required to Perform Flash Memory Reprogramming and Reprogramming Process

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

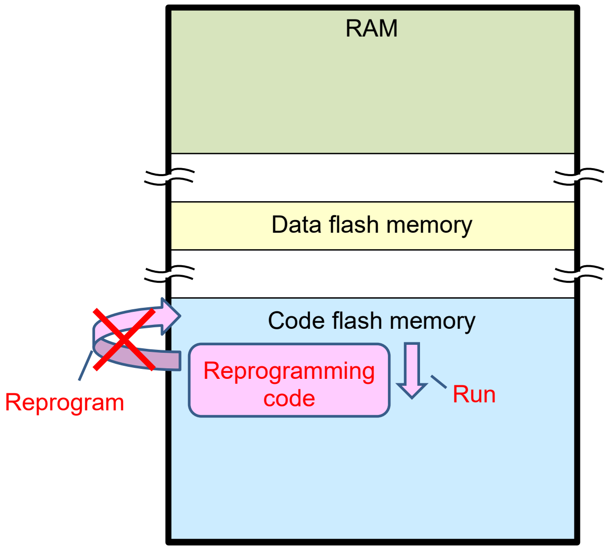


Figure 2.2 Reprogramming of Region Containing Code Required to Perform Flash Memory Reprogramming

Section 2.16.1 describe the available methods of reprogramming code flash memory.

2.16.1 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

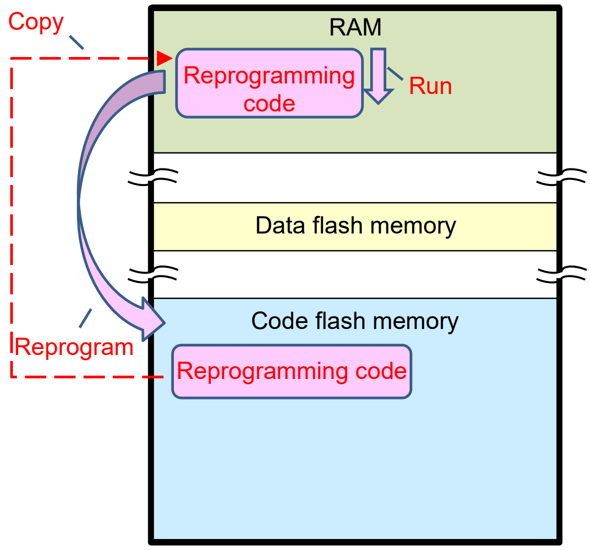


Figure 2.3 Reprogramming Code Flash Memory by Running Code on the RAM

Configure the configuration options of this module as follows.

- FLASH_CFG_CODE_FLASH_ENABLE: 1

*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 RAM for interrupts that may occur while the code flash memory is being reprogrammed. For details, refer to section 4.3.1.1.

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 Systemclock (ICLK) is invalid. */*
FLASH_ERR_HOCO	*/* The HOCO is not running. */*

Properties

Prototyped in file “r_flash_ 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 Data flash memory is allocated in code flash memory.
FLASH_CFG_CODE_FLASH_MODE	Don’t care	
FLASH_CFG_DATA_FLASH_MODE	0 or 1	
FLASH_CFG_CODE_FLASH_ENABLE	1	Code that processes flash memory is copied into RAM.
FLASH_CFG_CODE_FLASH_MODE	0	
FLASH_CFG_DATA_FLASH_MODE	0 or 1	
FLASH_CFG_CODE_FLASH_ENABLE	1	Code that processes flash memory is copied into RAM. The functionality to reallocate interrupt vector tables or interrupt processing is included[*1]
FLASH_CFG_CODE_FLASH_MODE	1	
FLASH_CFG_DATA_FLASH_MODE	0 or 1	

[*1] When FLASH_CFG_CODE_FLASH_MODE is set to NON_BLOCKING(“1”),the functionality of reallocating interrupt vector tables or interrupt processing is enabled. Refer to section 4.3.1.1 for details.

Reentrant

- Not allowed

Example

```
flash_err_t err;

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

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

Special Notes:

Do not modify the clock settings between the execution of the R_FLASH_Open function call and the completion of the R_FLASH_Close function call.

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_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\src\targets\<mcu>\r_flash_<mcu>.h".

num_blocks

Specifies the number of blocks to be erased.

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_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*1	Data Flash Memory*2
R9A02G021	2 Kbyte	1 Kbyte

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

*2 Defined as FLASH_DF_BLOCK_SIZE in the specific MCU definitions file ("r_flash\src\targets\<mcu>\r_flash_<mcu>.h").

When this API function is used in non-blocking mode, FCU_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.

```
flash_err_t err;

/* Erases code flash memory blocks in order from smaller to larger block numbers starting
from block 4. */
/* The following code causes blocks 4 and 5 in code flash memory to be erased. */
err = R_FLASH_Erase(FLASH_CF_BLOCK_4, 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.

**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_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
R9A02G021	●	●

●: Supported, —: Unsupported

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 */
        . . .
    }
}

/* Run the blank check on the first 64 bytes in block 8 of code flash memory. */
err = R_FLASH_BlankCheck((uint32_t)FLASH_CF_BLOCK_8, 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_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]
R9A02G021	8 bytes	1 byte

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

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

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_ACCESSW	*/* An access window error occurred. Incorrect region specified. */*
FLASH_ERR_PARAM	*/* Invalid parameter was passed. */*

Properties

Prototyped in file “r_flash_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
Common among all flash types		
Retrieve flash module API function running status	FLASH_CMD_STATUS_GET	✔
Register callback function	FLASH_CMD_SET_BLOCKING_CALLBACK	✔
Flash sequencer reset	FLASH_CMD_RESET	✔
Access window		
Retrieve access window configuration	FLASH_CMD_ACCESSWINDOW_GET	✔[*1]
Configure access window	FLASH_CMD_ACCESSWINDOW_SET	
Startup program protection		
Retrieve startup region setting	FLASH_CMD_SWAPFLAG_GET	✔[*2]
Swap startup region	FLASH_CMD_SWAPFLAG_TOGGLE	
Retrieve startup region selection bit setting	FLASH_CMD_SWAPSTATE_GET	
Set startup region selection bit	FLASH_CMD_SWAPSTATE_SET	

[*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.

Table 3.6 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_BLOCKING_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, FCU_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, FCU_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.

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_MODE and FLASH_CFG_CODE_FLASH_MODE to NON_BLOCKING (“1”).

To program code flash memory by running code from RAM, set the configuration option FLASH_CFG_CODE_FLASH_ENABLE to NON_BLOCKING (“1”).

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_BLOCKING_CALLBACK,(void
*)&cb_func_info);
    if (FLASH_SUCCESS != err)
    {
        /* Handle error */
    }

    /* Perform operations on RAM */
    do_rom_operations();

    ... (omission)
}
```

```
__attribute__((section("PFRAM")))
void u_cb_function(void *event) /* Callback function */
{
    flash_int_cb_args_t *ready_event = event;

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

__attribute__((section("PFRAM")))
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)
}
```

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_MODE to NON_BLOCKING (“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_BLOCKING_CALLBACK,(void
*)&cb_func_info);
    if (FLASH_SUCCESS != err)
    {
        /* Handle error */
    }

    /* Set data flash memory erase, blank check, or programming processing 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 2 in code flash memory. */
    access_info.start_addr = (uint32_t) FLASH_CF_BLOCK_2;
    access_info.end_addr = (uint32_t) FLASH_CF_BLOCK_3;
    err = R_FLASH_Control(FLASH_CMD_ACCESSWINDOW_SET, (void *)&access_info);
    if (FLASH_SUCCESS != err)
    {
      /* Handle error */
    }

    /* Allow programming and erasing of block 61 to 63 in code flash memory. */
    /* Use FLASH_CF_BLOCK_END to specify end address if block 63 is included in
setting range. */
    access_info.start_addr = (uint32_t) FLASH_CF_BLOCK_61;
    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, FLASH_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 */
    }
```

Special Notes:

None

4. Appendices

4.1 Confirmed Operation Environment

This section describes confirmed operation environment for this module.

Table 4.1 Confirmed Operation Environment (Rev. 1.00)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 2024-01
C compiler	LLVM for RISC-V V.17.0.2.202401 Compiler option: The following option is added to the default settings of the integrated development environment.
Endian	little endian
Revision of the module	Rev.1.00
Board used	FPB-R9A02G021 Board (product No.: RTK9FPG021S00001BJ)

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

- Using e[2] studio:
  Application note “RISC-V MCU Smart Configurator User's Guide: e[2] studio (R20AN0730)”

  When using this module, the board support package (BSP module) must also be added to the project. Refer to the application note “Board Support Package Module Using Software Integration System (R01AN7177)”.

(2) 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.

(3) Q: Return does not occur from R_FLASH_Erase() or R_FLASH_Write().

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. Interrupt vector tables and interrupt processing are relocated to RAM in r_flash.c. Please add such module as a reference implementation and adapt as needed.

4.3 Compiler-Dependent Settings

The compiler dependent settings necessary to use this software module are described in this section. The specific settings for the LLVM compiler are shown in section 4.3.1 below.

4.3.1 Using LLVM for RISC-V

This section describes how to use LLVM for RISC-V as the compiler.

For the linker setting, it is necessary to edit the linker settings file generated by e² studio.

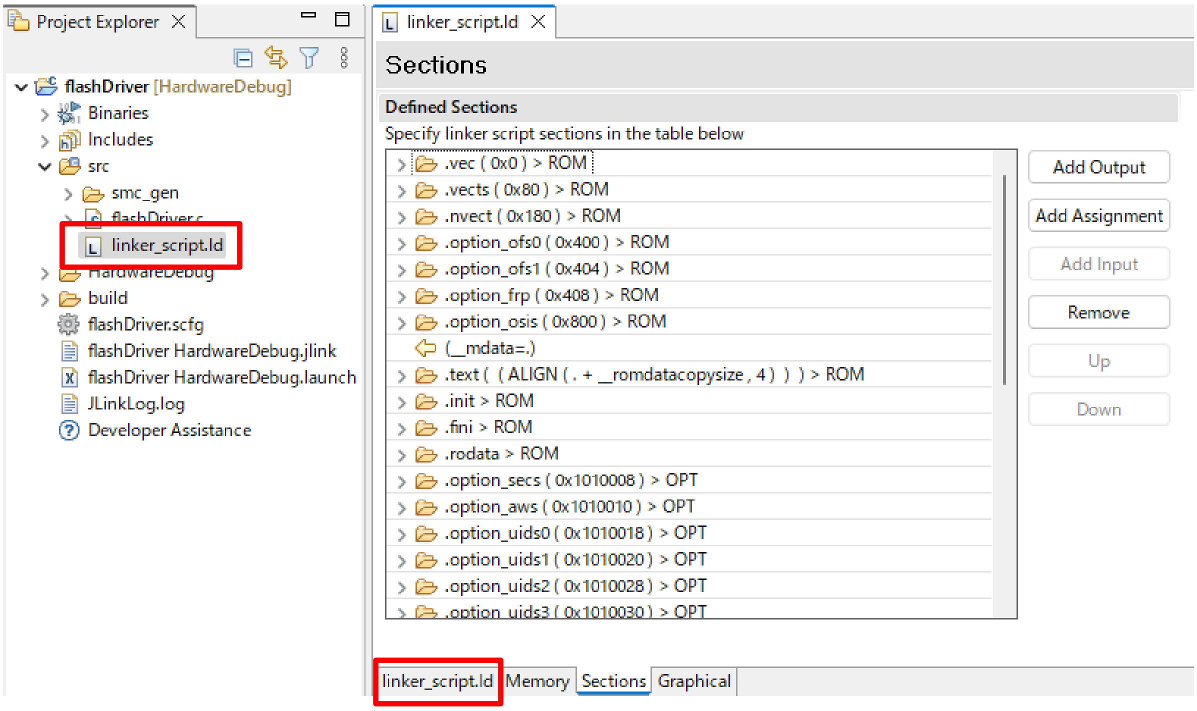
4.3.1.1 Programming Code Flash from RAM

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.



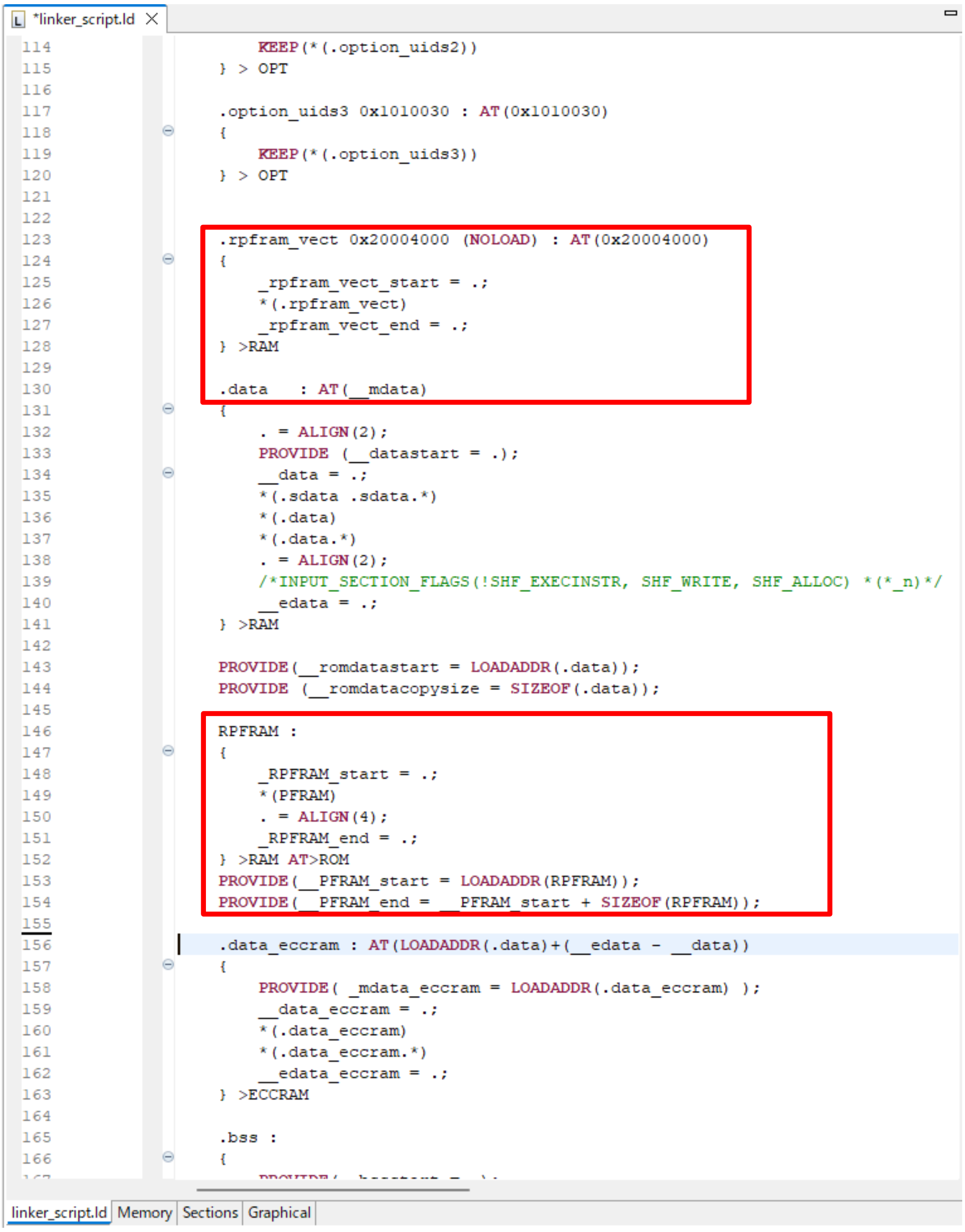
(3) Add in the linker settings file (linker_script.ld).

```
RPFRAM :
{
      _RPFRAM_start = .;
      *(PFRAM)
      . = ALIGN(4);
      _RPFRAM_end = .;
} >RAM AT>ROM
PROVIDE(__PFRAM_start = LOADADDR(RPFRAM));
PROVIDE(__PFRAM_end = __PFRAM_start + SIZEOF(RPFRAM));
```

(4) When rewriting code flash in non-blocking mode, an interrupt vector table must be relocated in RAM. Add the following settings to the linker configuration file(linker_script.ld).

```
.rpfram_vect 0x20004000 (NOLOAD) : AT(0x20004000)
{
      _rpfram_vect_start = .;
      *(.rpfram_vect)
      _rpfram_vect_end = .;
} > RAM

.data    : AT(__mdata)
```



2. Programs that operate during code flash reprogramming 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. Website and Support

Visit the following URLs to learn about key elements of the RISC-V MCU family, download components and related documentation, and get support:

RISC-V MCU Product Information	www.renesas.com/risc-v
RISC-V MCU Product Support Forum	https://community.renesas.com/risc-v/forum
RISC-V MCU Videos	www.renesas.com/risc-v/videos
Renesas Support	www.renesas.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Mar.23.24	—	Initial release

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.

Corporate Headquarters

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

Contact information

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