RE01 1500KB, 256KB Group

R_FLASH Driver of CMSIS software package

R01AN4768EJ0103
Rev.1.03
Jun. 18, 2021

Introduction

This application note will describe an R_FLASH flash driver of the RE01 1500KB, 256KB Group CMSIS software package. In this application note, the R_FLASH flash driver will be referred to as a flash driver.

The flash driver has been developed to allow users to easily integrate reprogramming abilities of the flash into their applications using self-programming. The self-programming is the feature to reprogram the on-chip flash memory while running in single-chip mode. This application note focuses on using the flash driver and integrating it with your application program.

The source files provided by the flash driver comply with the IAR compiler and the GNU GCC compiler only.

Target Device

- RE01 1500KB Group
- RE01 256KB Group

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

Related Documents

- RE01 1500KB, 256KB Group Getting Started Guide to Development Using CMSIS Package (R01AN4660)

Contents

1. Overview

The flash driver is provided to customers to make the process of programming and erasing on-chip flash areas easier. The flash driver supports the code flash. The flash driver can be used to perform programming and erasing operations in the blocking or non-blocking background operation (BGO) mode. In the blocking mode, when a programming or erasing function is called, the function does not return until the operation is finished. In the BGO mode, the API functions return immediately after the operation is begun. When the code flash operation is on-going, the code flash area cannot be accessed by the user application. If an attempt is made to access the code flash area, the sequencer will transition into an error state. In the BGO mode, the user must poll for operation completion or provide a flash interrupt callback function (if flash interrupt support is available on the MCU).

1.1 Features

Below is a list of the features supported by the flash driver.

- Erasing and programming the code flash in the blocking mode or non-blocking BGO mode
- Area protection via access windows
- Start-up area protection; this function is used to safely rewrite block 0 to block 7 in the code flash.

2. API Information

Sample codes in this application note have been verified to operate under the following conditions.

2.1 Hardware Requirements

This driver requires that your MCU supports the following peripheral:

- Flash

2.2 Software Requirements

This driver is dependent upon the following packages:

- R-CORE
- R_SYSTEM driver

2.3 Limitations

- This API code is not re-entrant and protects against multiple concurrent function calls (not including RESET).

2.4 Supported Toolchains

This driver is tested with the following toolchains:

- IAR Embedded Workbench for ARM

  -1500KB :8.32.1.18631
  -256KB :8.40.2.22891

- GNU GCC ARM(arm-none-eabi)

  -1500KB : v5.4.1.20160919
  -256KB :v6.3.1.20170620

2.5 Header Files

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

Build-time configuration options can be selected or defined in "r_flash_cfg.h”.

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 Compilation Configuration

Configuring this driver is done through r_flash_cfg.h. Each configurable item is detailed in the table below.

Configuration options in r_flash_cfg.h		
Equate	**Default Value**	**Description**
FLASH_CFG_PARAM_CHECKING_ENABLE	1	Setting to 1 includes generation of a code for parameter checking. Setting to 0 omits the generation.
FLASH_CFG_CODE_FLASH_ENABLE	1 (Fixed)	Generate of a code to program the code flash area. When programming the code flash, the code must be executed from the RAM. See section 2.11 for details on how to set up the code and the linker to execute the code from the RAM. See section 2.13 for driver definition of the BGO mode.
FLASH_CFG_CODE_FLASH_BGO	0	Setting this to 0 forces the code flash API function to block until completed. Setting to 1 places the module in the BGO (background operations/interrupt) mode. In the BGO mode, the API function returns immediately after the code flash operation has been started. Notification of the operation completion is done via the callback function. When programing the code flash, a vector table and interrupt processing corresponding to the vector table need to be placed at a location other than the code flash in advance. See the usage notes in section 2.14.
FLASH_CFG_CODE_FLASH_RUN_FROM_ROM	0	Valid only when FLASH_CFG_CODE_FLASH_ENABLE is set to 1. ● RE01 1500KB Group: Set this to 0 when programming the code flash while executing a code in the RAM. Set this to 1 when programming the code flash while executing the code from another segment in the code flash (see section 2.12). ● RE01 256KB Group: The function of this configuration is not supported. Be sure to set this configuration to 0.

2.8 Code Size

The code size for the toolchain in Section 2.4 is based on the condition that no optimization is performed. Sizes of the ROM (code and constants) and the RAM (global data and codes executed on the RAM) are determined by the build-time configuration options set in the configuration header file of the driver. The code size shown in this item is for RE01 1500KB.

-RE01 1500KB

IAR Embedded Workbench for ARM 8.32.1.18631	
PARAM_CHECKING_ENABLE 1 > PARAM_CHECKING_ENABLE 0 **CODE_FLASH_BGO 1 > CODE_FLASH_BGO 0**	
Minimum Size	ROM: 3772 bytes
	RAM: 3820 bytes (48 bytes if FLASH_CFG_CODE_FLASH_RUN_FROM_ROM is 1)
Maximum Size	ROM: 4162 bytes
	RAM: 4210 bytes (48 bytes if FLASH_CFG_CODE_FLASH_RUN_FROM_ROM is 1)

IAR Embedded Workbench for ARM 8.32.1.18631	
PARAM_CHECKING_ENABLE 1 > PARAM_CHECKING_ENABLE 0 **CODE_FLASH_BGO 1 > CODE_FLASH_BGO 0**	
Minimum Size	ROM: 3772 bytes
	RAM: 3820 bytes (48 bytes if FLASH_CFG_CODE_FLASH_RUN_FROM_ROM is 1)
Maximum Size	ROM: 4162 bytes
	RAM: 4210 bytes (48 bytes if FLASH_CFG_CODE_FLASH_RUN_FROM_ROM is 1)

-RE01 256KB

GNU GCC ARM(arm-none-eabi) v5.4.1.20160919	
PARAM_CHECKING_ENABLE 1 > PARAM_CHECKING_ENABLE 0 **CODE_FLASH_BGO 1 > CODE_FLASH_BGO 0**	
Minimum Size	ROM: 5932 bytes
	RAM: 5930 bytes (46 bytes if FLASH_CFG_CODE_FLASH_RUN_FROM_ROM is 1)
Maximum Size	ROM: 6372 bytes
	RAM: 6370 bytes (46 bytes if FLASH_CFG_CODE_FLASH_RUN_FROM_ROM is 1)

GNU GCC ARM(arm-none-eabi) v5.4.1.20160919	
PARAM_CHECKING_ENABLE 1 > PARAM_CHECKING_ENABLE 0 **CODE_FLASH_BGO 1 > CODE_FLASH_BGO 0**	
Minimum Size	ROM: 5932 bytes
	RAM: 5930 bytes (46 bytes if FLASH_CFG_CODE_FLASH_RUN_FROM_ROM is 1)
Maximum Size	ROM: 6372 bytes
	RAM: 6370 bytes (46 bytes if FLASH_CFG_CODE_FLASH_RUN_FROM_ROM is 1)

2.9 API Data Structures

The API data structures are located in “r_flash_api.h” and discussed in Section 3.

2.10 Return Values

Different values API functions can return will be shown below. This return type is defined in “r_flash_api.h”.

```
/* Flash API error codes */
typedef enum_flash_err
{
FLASH_SUCCESS = 0,
FLASH_ERR_BUSY,         /* Flash module busy */
FLASH_ERR_ACCESSW,      /* Access window error */
FLASH_ERR_FAILURE,      /* Flash operation failure; programming error,
                           erasing error, etc. */
FLASH_ERR_CMD_LOCKED,   /* Peripheral in command locked state */
FLASH_ERR_LOCKBIT_SET, /* Programming/erasing error due to lock bit */ (not
                           supported)
FLASH_ERR_FREQUENCY,    /* Illegal frequency value attempted
                           (except for 1-32 MHz) */
FLASH_ERR_ALIGNED,      /* The address that was supplied is not
                           aligned correctly for code flash. */
FLASH_ERR_BOUNDARY,     /* Cannot write beyond the 1MB
                           boundary on some parts */
FLASH_ERR_OVERFLOW,     /* 'Address + number of bytes' for this
                           operation exceeds the end of this memory area. */
FLASH_ERR_BYTES,        /* Invalid number of bytes */
FLASH_ERR_ADDRESS,      /* Invalid address */
FLASH_ERR_BLOCKS,       /* The "number of blocks" parameter is invalid. */
FLASH_ERR_PARAM,        /* Illegal parameter */
FLASH_ERR_NULL_PTR,     /* Missing required parameter */
FLASH_ERR_UNSUPPORTED, /* Command is not supported. */
FLASH_ERR_SECURITY,     /* Programming/erasing error due to protection by
                           AWS.FSPR */
FLASH_ERR_TIMEOUT,      /* Timeout condition */
FLASH_ERR_ALREADY_OPEN /* Open() called twice without calling Close() */
} flash_err_t;
```

2.11 Programming Code Flash from RAM (RE01 1500KB Group only)

The API functions for programming the code flash will need to be copied into the RAM to keep it. The RAM will need to be initialized after a reset.

When the API functions are copied into the RAM, set FLASH_CFG_CODE_FLASH_RUN_FROM_ROM to 0 in “r_flash_cfg.h”. In addition, in order to enable programming the code flash, set FLASH_CFG_CODE_FLASH_ENABLE to 1 in “r_flash_cfg.h”.

Use the API functions of the R_SYSTEM driver to copy the API functions into the RAM. An example for copying the API functions into the RAM using the R_SYSTEM driver is shown below.

```
Int main ()
{
    /* Copy code from ROM to RAM */
    R_SYS_CodeCopy();

    R_SYS_Initialize();

    …
}
```

For details of the method for copying the API functions into the RAM, refer to relevant application notes.

2.12 Programming Code Flash from Code Flash

The code flash can be programmed while executing the code from the code flash. The code flash is divided into three areas. The code can be executed on one area and erase/write operations can be performed on the other areas.

To use this method, set FLASH_CFG_CODE_FLASH_ENABLE and FLASH_CFG_CODE_FLASH_RUN_FROM_ROM to 1 in “r_flash_cfg.h”.

Be sure not set up the linker as just described in section 2.11, and do guarantee that the region the code is running from is not the region the code is running on.

2.13 Operations in BGO Mode

In general, the background operation (BGO) mode refers to the non-blocking mode of the driver, which is the ability to execute instructions from the RAM while a code flash operation is running in the background.

When operating in the BGO mode, the API function calls are not blocked and return immediately. The user should not access the flash area being operated on until the operation has finished. If the area is accessed during the operation, the sequencer will go into an error state and the operation will fail.

The completion of the operation is indicated by the FRDYI interrupt. The completion of processing is checked in the FRDYI interrupt handler and the callback function is called. To register the callback function, call the R_FLASH_Control function with the FLASH_CMD_SET_BGO_CALLBACK command. The callback function receives an event indicating the completion status. The possible events (some MCU-specific) are located in "r_flash_api.h" and are defined as follows:

```
typedef enum
{
    FLASH_INT_EVENT_INITIALIZED,
    FLASH_INT_EVENT_ERASE_COMPLETE,
    FLASH_INT_EVENT_WRITE_COMPLETE,
    FLASH_INT_EVENT_BLANK,
    FLASH_INT_EVENT_NOT_BLANK,
    FLASH_INT_EVENT_TOGGLE_STARTUPAREA,
    FLASH_INT_EVENT_SET_ACCESSWINDOW,
    FLASH_INT_EVENT_ERR_DF_ACCESS,
    FLASH_INT_EVENT_ERR_CF_ACCESS,
    FLASH_INT_EVENT_ERR_SECURITY,
    FLASH_INT_EVENT_ERR_CMD_LOCKED,
    FLASH_INT_EVENT_ERR_LOCKBIT_SET,
    FLASH_INT_EVENT_ERR_FAILURE,
    FLASH_INT_EVENT_END_ENUM
} flash_interrupt_event_t;
```

When programming the code flash, the vector table and associated interrupt handlers must be allocated in an area other than the code flash in advance.

2.14 Registering Interrupts to NVIC

When allowing the FLASH driver to use the interrupt sources(FRDYI or FIFEER) for interrupts or polling, register the interrupts to the NVIC in r_system_cfg.h and then enable the interrupts in the Control function. For details, refer to

"Interrupt (NVIC) Settings" in the RE01 1500KB, 256KB Group Getting Started Guide to Development Using CMSIS Package（r01an4660）.

If the FRDYI interrupt or the FIFEER interrupt are not registered in NVIC, FLASH_ERR_PARAM will return when an interrupt enable command of the Control function is executed.

Table 2-1 shows the definition of NVIC registration for each intended use and Figure 2-1 shows the coding example for registering the interrupts to the NVIC.

Table 2-1 Definitions of NVIC Registration for Each Intended Use

Intended Use	NVIC Registration Definition	Remarks
When using FCU_FRDYI interrupt	**SYSTEM_CFG_EVENT_NUMBER_FCU_FRDYI**	
When using FCU_FIFERR interrupt	**SYSTEM_CFG_EVENT_NUMBER_FCU_FIFERR**	

```
 . . .
#define SYSTEM_CFG_EVENT_NUMBER_DMAC2_INT
          (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 2/6/10/14/18/22/26/30 only */
#define SYSTEM_CFG_EVENT_NUMBER_FCU_FRDYI
          (SYSTEM_IRQ_EVENT_NUMBER2)  /*!< Numbers 2/6/10/14/18/22/26/30 only
#define SYSTEM_CFG_EVENT_NUMBER_AGT0_AGTCMAI
        (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 2/6/10/14/18/22/26/30 only */
 . . .
```

Figure 2-1 Example of Interrupt Registration to NVIC (FCU_FRDYI used) in r_system_cfg.h

2.15 Usage Notes

2.15.1 Code Flash Operations in BGO Mode

When programming the code flash in the BGO/non-blocking mode, an external memory and the RAM can be accessed. Since the API function of the flash driver returns before the code flash programming is completed, a code that calls the API function is allocated in the RAM. In addition, it is necessary to check completion of running operations before other flash commands are issued. The commands include specifying the access window of the code flash, toggling the startup area select flag, erasing the code flash, and programing the code flash.

2.15.2 Code Flash Operations and General Interrupts

Code flash areas cannot be accessed while a flash operation is on-going for a particular memory area. This means that allocation of the vector table will need to be taken care of when allowing interrupts to occur during flash operations. The vector table is allocated in the code flash by default. If an interrupt occurs during the code flash operation, the code flash will be accessed to fetch the interrupt's starting address and an error will occur. To fix this situation, the user will need to allocate the vector table and any interrupt handlers that may occur outside of the code flash. The user will also need to change the-vector table offset register (VTOR).

By using the R_SYSTEM driver, the vector table and the interrupt handler can be relocated.

3. API Functions

3.1 Summary

The following functions are included in this driver:

Function	Description
R_FLASH_Open()	Initializes the flash driver.
R_FLASH_Close()	Closes the flash driver.
R_FLASH_Erase()	Erases the specified blocks in the code flash.
R_FLASH_Write_8Bytes()	Writes data to the code flash in a unit of 8 bytes.
R_FLASH_ Write_256Bytes()	Writes data to the code flash in a unit of 256 bytes.
R_FLASH_Control()	Configures settings for the status check, area protection, and swapping of start-up area protection.
R_FLASH_GetVersion()	Returns the current version of this driver.

3.2 R_FLASH_Open()

The function initializes the flash driver. This function must be called before calling any other API functions.

Format

```
flash_err_t R_FLASH_Open(void);
```

Parameters

None

Return Values

```
FLASH_SUCCESS:         /* Flash driver initialized successfully */
FLASH_ERR_BUSY:        /* Another flash operation in progress, try again later */
FLASH_ERR_ALREADY_OPEN:  /* Open() called twice without calling Close() */
```

Properties

Prototyped in “r_flash_api.h”

Description

This function initializes the flash driver. Note that this function must be called before any other API function is called.

Reentrant

No

Example

```
flash_err_t err;

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

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

Special Notes:

None

3.3 R_FLASH_Close()

The function closes the flash driver.

Format

```
flash_err_t R_FLASH_Close(void);
```

Parameters

None

Return Values

```
FLASH_SUCCESS:      /* Flash driver closed successfully */
FLASH_ERR_BUSY:     /* Another flash operation in progress, try again later */
```

Properties

Prototyped in “r_flash_api.h”

Description

This function closes the flash driver. It disables the flash interrupts (if enabled) and sets the driver to an uninitialized state.

Reentrant

No

Example

```
flash_err_t err;

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

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

Special Notes:

None

3.4 R_FLASH_Erase()

This function is used to erase the specified blocks in the code flash.

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 block to be erased. The enum “flash_block_address_t” is defined in “r_flash_re01_1500kb.h” ・“r_flash_re01_256kb.h”. The blocks are labeled in the same fashion as they are in the UMH of the MCU. For example, the block allocated at address 0x00007000 is block 7 in the UMH, and therefore “FLASH_CF_BLOCK_7” should be passed for this parameter.

num_blocks

Specifies the number of blocks to be erased.

Return Values

```
FLASH_SUCCESS:        /* Operation successful (if BGO mode is enabled, this
                         means the operation was started successfully.) */
FLASH_ERR_BLOCKS:     /* Invalid number of blocks specified */
FLASH_ERR_OVERFLOW:   /* Range to be erased exceeds code flash area */
FLASH_ERR_ADDRESS:    /* Invalid address specified */
FLASH_ERR_BUSY:       /* Another flash operation in progress, or the module is
                         not initialized */
FLASH_ERR_FAILURE:    /* Erasing failure. Sequencer has been reset, or callback
                         function not registered (if BGO/non-blocking mode is
                         enabled) */
```

Properties

Prototyped in “r_flash_api.h”

Description

This function erases contiguous blocks in the code flash. The block size (FLASH_CF_BLOCK_SIZE) is 4 Kbytes. With a block specified by the first parameter as the start address of blocks to be erased, blocks having larger block numbers (address increasing direction) are erased.

When the API is used in the BGO/non-blocking mode, the callback function is called when all the specified blocks have been erased.

Reentrant

No

Example

```
flash_err_t err;

/* Erase code flash blocks 2 to 6 */
err = R_FLASH_Erase((uint32 t)FLASH_DF_BLOCK_2, 5);

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

Special Notes:

In order to erase the blocks in the code flash, the area to be erased needs to be a rewritable area (set to be accessible with the access window).

3.5 R_FLASH_BlankCheck()

This function is not supported.

3.6 R_FLASH_Write_8Bytes()

This function is used to write data to the code flash in a unit of 8 bytes.

Format

```
flash_err_t R_FLASH_Write_8bytes(uint32_t   src_address,
                                 uint32_t   dest_address,
                                 uint32_t   num_bytes);
```

Parameters

src_address

This is a pointer to the buffer containing the data to write to the code flash.

dest_address

This is a pointer to the code flash area to write the data. The address specified must be divisible by the programming size (8 bytes). See *Description* below for important restrictions regarding this parameter.

num_bytes

This is the number of bytes contained in the buffer specified with *src_address*. This number must be a multiple of the programming size (8 bytes) of the code flash memory.

Return Values

```
FLASH_SUCCESS:          /* Operation successful (in BGO/non-blocking mode, this
                           means the operation was started successfully.) */
FLASH_ERR_FAILURE:      /* Programming failed. Possibly the destination address
                           was controlled with access window; or callback function
                           was not present (when BGO mode and flash interrupt
                           supported) */
FLASH_ERR_BUSY:         /* Another flash operation in progress or the module not
                           initialized */
FLASH_ERR_OVERFLOW:     /* Range to be programed exceeds code flash area */
FLASH_ERR_BYTES:        /* Number of bytes specified was not a multiple of the
                           programming size or exceeds the maximum range */
FLASH_ERR_ADDRESS:      /* Invalid address was input or address was not divisible
                           by the programming size */
FLASH_ERR_CMD_LOCKED    /* Operation disabled due to FCU in command-lock state */
```

Properties

Prototyped in “r_flash_api.h”

Description

This function is used to write data to the flash memory. Before writing to any flash area, the area must already be erased.

When performing the writing operation, the user must make sure to start the writing operation from an address divisible by the programming size (8 bytes) and make the number of bytes to write be a multiple of the programming size (8 bytes).

An area used to write data in the code flash must be a rewritable area (set to be accessible with the access window).

When the API is used in the BGO/non-blocking mode, the callback function is called when all writing operations are completed.

Reentrant

No

Example

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

/* Write data to internal memory */
err = R_FLASH_Write_8Bytes((uint32_t)write_buffer, dst_addr,
sizeof(write_buffer));

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

Special Notes:

None

3.7 R_FLASH_Write_256Bytes()

This function is used to write data into the code flash in a unit of 256 bytes.

Format

```
flash_err_t R_FLASH_Write_256bytes(uint32_t src_address,
                                   uint32_t dest_address,
                                   uint32_t num_bytes);
```

Parameters

src_address

This is a pointer to the buffer containing the data to write to the code flash.

dest_address

This is a pointer to the code flash area to write. The address specified must be divisible by the programming size (256 bytes). See *Description* below for important restrictions regarding this parameter.

num_bytes

This is the number of bytes contained in the buffer specified with *src_address*. This number must be a multiple of the programming size (256 bytes) of the code flash memory.

Return Values

```
FLASH_SUCCESS:          /* Operation successful (in BGO/non-blocking mode, this
                           means the operation was started successfully.) */
FLASH_ERR_FAILURE:      /* Programming failed. Possibly the destination address
                           was controlled with access window; or callback function
                           was not present (when BGO mode and flash interrupt
                           supported) */
FLASH_ERR_BUSY:         /* Another flash operation in progress or the module not
                           initialized */
FLASH_ERR_OVERFLOW:     /* Range to be programed exceeds code flash area */
FLASH_ERR_BYTES:        /* Number of bytes specified was not a multiple of the
                           programming size or exceeds the maximum range */
FLASH_ERR_ADDRESS:      /* Invalid address was input or address not divisible by
                           the programming size */
FLASH_ERR_CMD_LOCKED    /* Operation disabled due to FCU in command-lock state */
```

Properties

Prototyped in “r_flash_api.h”

Description

This function is used to write data into the flash memory. Before writing to any flash area, the area must already be erased.

When performing the writing operation, the user must make sure to start the writing operation from an address divisible by the programming size (256 bytes) and make the number of bytes to write be a multiple of the programming size (256 bytes).

An area used to write data in the code flash must be a rewritable area (set to be accessible with the access window).

When the API is used in the BGO/non-blocking mode, the callback function is called when all writing operations are completed.

Reentrant

No

Example

```
flash_err_t err;
uint8_t write_buffer[256] = "Hello World...";

/* Write data to internal memory */
err = R_FLASH_Write_256Bytes((uint32_t)write_buffer, dst_addr,
sizeof(write_buffer));

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

Special Notes:

None

3.8 R_FLASH_Control()

This function implements functions other than the functions for programing and erasing.

Format

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

Parameters

cmd

Command to execute.

**pcfg*

Configuration parameters requested by the specific command. This may be NULL if the command does not request it.

Return Values

```
FLASH_SUCCESS:          /* Operation successful (in BGO mode, this means the
                        operations was started successfully.) */
FLASH_ERR_FAILURE:      /* Callback function was not configured, or hardware
                       operation was failed */
FLASH_ERR_NULL_PTR:     /* parameter pcfg used in command for requesting for a
                         configuration structure was NULL*/
FLASH_ERR_BUSY:         /* Another flash operation in progress or API not
                         initialized */
FLASH_ERR_CMD_LOCKED:   /* Flash control circuit was in a command locked state
                        and was reset */
FLASH_ERR_ACCESSW:      /* Access window error: Incorrect area specified */
FLASH_ERR_PARAM:        /* Invalid command */
FLASH_ERR_FREQUENCY:    /* When executing FLASH_CMD_CONFIG_CLOCK command,
                        invalid frequency was passed */
FLASH_ERR_UNSUPPORTED:   /* Unsupported command*/
```

Properties

Prototyped in “r_flash_api.h”

Description

This function is an expanded function that implements functions of the sequencer other than the functions for programing and erasing. Depending on the command type, a different parameter type has to be passed.

Command	Parameter	Operation
FLASH_CMD_RESET	NULL	Resets the sequencer. This may or may not wait for the current flash operation to complete (operation dependent).
FLASH_CMD_STATUS_GET	NULL	Returns the status of the API (Busy or Idle). In the Idle state, FLASH_SUCCESS is returned.
FLASH_CMD_SET_BGO_CALLBACK	flash_interrupt_config_t *	Registers the callback function.
FLASH_CMD_ACCESSWINDOW_GET	flash_access_window_config_t *	Returns the access window boundaries for the code flash.
FLASH_CMD_ACCESSWINDOW_SET	flash_access_window_config_t *	Sets the access window boundaries for the code flash. In the BGO/non-blocking mode, after the access window is set, the FRDYI interrupt occurs and the callback function is called.**
FLASH_CMD_SWAPFLAG_GET	uint32_t *	Loads a current value of a start-up area setting monitor flag (BTFLG).
FLASH_CMD_SWAPFLAG_TOGGLE	NULL	Toggles the start-up program areas to swap the area with a function allocated in the RAM. After swapping the area, reset the MCU without returning to the code flash. In the BGO/non-blocking mode, after swapping the area, the FRDYI interrupt occurs and the callback function is called.**
FLASH_CMD_SWAPSTATE_GET	uint8_t *	Loads a current value of a start-up area selection bit (a value of SAS).

Command	Parameter	Operation
FLASH_CMD_SWAPSTATE_SET	uint8_t *	Sets a value of the start-up area selection bit (FSUACR.SAS) by defining it in r_flash_api.h. #define (value) FLASH_SAS_EXTRA (0) FLASH_SAS_DEFAULT (2) FLASH_SAS_ALTERNATE (3) FLASH_SAS_SWITCH_AREA (4) When FLASH_SAS_EXTRA, FLASH_SAS_DEFAULT, or FLASH_SAS_ALTERNATE is set, the value is directly set in FSUACR.SAS, and the area is swapped depending on the value. When FLASH_SAS_SWITCH_AREA is set, the area is immediately swapped using a function allocated in the RAM. The area after the reset is an area specified with FLASH_SAS_EXTRA.
FLASH_CMD_CONFIG_CLOCK	uint32_t *	Speed in Hz at which ICLK is running. Only needs to be called if a clock speed is changed at run time.

**When using these commands in BGO/non-blocking mode, it must wait for the command processing to complete before the next flash programming. Before the flash programming, wait for the callback function to be called, or check that the API is in the Idle state with the FLASH_CMD_STATUS_GET command of the R_FLASH_Control function.

Reentrant

No, except for the FLASH_CMD_RESET command which can be executed at any time.

Example 1: Polling in BGO mode

In the blocking mode, other operations cannot be performed while waiting for the flash operation to complete. The BGO mode is used when other operations must be performed while waiting for the flash operation to complete.

```
flash_err_t err;

/* erase all of code flash */
R_FLASH_Erase(FLASH_CF_BLOCK_0, FLASH_NUM_BLOCKS_CF);

/* wait for operation to complete */
while (R_FLASH_Control(FLASH_CMD_STATUS_GET, NULL) == FLASH_ERR_BUSY)
{
/* do critical system checks here */
}
```

Example 2: Setting BGO mode with interrupt support

The BGO/non-blocking mode is enabled when FLASH_CFG_CODE_FLASH_BGO equals 1. When programming the code flash, relocate the vector table in the RAM. Also, the callback function must be registered prior to programming/erasing.

```
void func(void)
{
    flash_err_t_err;
    flash_interrupt_config_t cb_function;

    /* Copy code from ROM to RAM */
    R_SYS_CodeCopy();

    /* Relocate vector table in RAM */
    R_SYS_Initialize();

    /* Initialize API */
    err = R_FLASH_Open();
    If (FLASH_SUCCESS != err)
    {
        //...(omission)
    }

    /* Set callback function and interrupt priority level */
    Cb_function.pcallback = u_cb_function;
    Cb_function.int prority = 1;

    err = R_FLASH_Control(FLASH_CMD_SET_BGO_CALLBACK, (void *)&cb function);
    If (FLASH_SUCCESS != err)
    {
        Printf("Control FLASH_CMD_SET_BGO_CALLBACK command failure.");
    }

    /* Perform operations on code flash */
    do rom operations();

}

 __attribute__ ((section(".ramfunc")))
void u_cb_function(void *event) /* Callback Function */
{
    flash_int_cb_args_t *ready event = event;

    /* ISR callback function process */
}

void do_rom_operations(void)
{
/* Write setting of code flash access window, toggle of start-up area flag,
erase, or code flash program process here */

... (omission)
}
```

Example 3: Getting range of current access window

```
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)
{
    printf("Control FLASH_CMD_ACCESSWINDOW_GET command failure.");
}
```

Example 4: Setting access window of code flash

The area protection is used to prevent unauthorized programming or erasure of code flash blocks. The following example makes only block 3 writable.

```
flash_err_t                  err;
flash_access_window_config_t access_info;

/* Allow rewrite to code flash block 3 */

access_info.start_addr = (uint32_t) FLASH_CF_BLOCK_3;
access_info.end_addr = (uint32_t) FLASH_CF_BLOCK_4;
err = R_FLASH_Control(FLASH_CMD_ACCESSWINDOW_SET, (void *)&access_info);
if (FLASH_SUCCESS != err)
{
    printf("Control FLASH_CMD_ACCESSWINDOW_SET command failure.");
}
```

In the following example, block 0 to block 2 are set to be rewritable.

```
flash_err_t err;
flash_access_write_window_config_t access_info;

/* Set block 0 to block 2 to be rewritable */

access_info.start_addr = (unit32 t) FLASH_CF_BLOCK_0;
access_info.end_addr = (unit32 t) FLASH_CF_BLOCK_3;
err = R_FLASH_Control(FLASH_CMD_ACCESSWINDOW_SET, (void *)&access info);
if (FLASH_SUCCESS != err)
{
    Printf("Control FLASH_CMD_ACCESSWINDOW_SET command failure.");
}
```

In the following example, block 380 to block 383 are set to be rewritable.

```
flash_err_t err;
flash_access_window_config_t access_info;

/* Set block 380 to block 383 to be rewritable */

access_info.start_addr = (uint32_t) FLASH_CF_BLOCK_380;
access_info.end_addr = (uint32_t) FLASH_CF_BLOCK_383 + FLASH_CF_BLOCK_SIZE;
err = R_FLASH_Control(FLASH_CMD_ACCESSWINDOW_SET, (void *)&access_info);
if (FLASH_SUCCESS != err)
{
    printf("Control FLASH_CMD_ACCESSWINDOW_SET command failure.");
}
```

Example 5: Getting value of active start-up area

The following example shows how to read the value of the start-up area setting monitor flag (FAWMON.BTFLG).

```
uint32_t    swap_flag;
flash_err_t err;

err = R_FLASH_Control(FLASH_CMD_SWAPFLAG_GET, (void *)&swap_flag);
if (FLASH_SUCCESS != err)
{
    printf("Control FLASH_CMD_SWAPFLAG_GET command failure.");
}
```

Example 6: Swapping active start-up area

The following example shows how to toggle the active start-up program area. Swap the area with the function placed in the RAM. After the area has been swapped, reset the MCU without returning to the code flash.

```
flash_err_t err;

/* Swap active area from default to alternate or vice versa */

err = R_FLASH_Control(FLASH_CMD_SWAPFLAG_TOGGLE, FLASH_NO_PTR);
if(FLASH_SUCCESS != err)
{
    printf("Control FLASH_CMD_SWAPFLAG_TOGGLE command failure.");
}
```

Example 7: Getting value of start-up area select bit

The following example shows how to read the current value in the start-up area select bit (FSUACR.SAS).

```
uint8_t    swap_area;
flash_err_t err;

err = R_FLASH_Control(FLASH_CMD_SWAPSTATE_GET, (void *)&swap_area);
if (FLASH_SUCCESS != err)
{
    printf("Control FLASH_CMD_SWAPSTATE_GET command failure.");
}
```

Example 8: Setting value of start-up area select bit

The following example shows how to set the value to the start-up area select bit (FSUACR.SAS). Swap the area with the function placed in the RAM. After a reset, the area will be the one specified with FLASH_SAS_EXTRA.

```
uint8_t    swap_area;
flash_err_t err;

swap_area = FLASH_SAS_SWITCH_AREA;
err = R_FLASH_Control(FLASH_CMD_SWAPSTATE_SET, (void *)&swap_area);
if (FLASH_SUCCESS != err)
{
    printf("Control FLASH_CMD_SWAPSTATE_SET command failure.");
}
```

Special Notes:

None

3.9 R_FLASH_GetVersion ()

This function returns the current version of this driver.

Format

```
uint32_t R_FLASH_GetVersion(void);
```

Parameters

None

Return Values

Version number

Properties

Prototyped in “r_flash_api.h”

Description

This function will return the version number of this driver. 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.

Reentrant

Yes

Example

```
uint32_t cur_version;

/* Get version of installed flash API */
cur_version = R_FLASH_GetVersion();

/* Check whether the version is new enough for this application's use */
if (MIN_VERSION > cur_version)
{
    /* WARNING: This flash API version is not new enough and does not support
       XXX feature that is supported by newer versions and is needed
       for this application. */
    ...
}
```

Special Notes:

None

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Oct. 11, 2019	—	First edition issued
1.01	Dec. 02, 2019	4	2.4 Supported Tool chains, updated
		6,7	2.8 Code size, updated
		10	2.14 Registering Interrupts to NVIC ,added
1.02	May. 25, 2020	6	2.8 Note that the code size is for RE01 1500KB.
1.03	Jun. 18, 2021	5	2.7 Compilation Configuration ● Change description of FLASH_CFG_CODE_FLASH_RUN_FROM_ROM by product group.
		9	2.11 Programming Code Flash from RAM (RE01 1500KB Group only) ● Note that only RE01 1500KB Group is supported.
		11	2.15.1 Code Flash Operations in BGO Mode ● Correct the description.
		15	3.4 R_FLASH_Erase() ● Correct the Description.
		23-24	3.8 R_FLASH_Control() ● Note the return value in the Operation of the FLASH_CMD_STATUS_GET command. ● Correct the register in the Operation of the FLASH_CMD_SWAPSTATE_SET command. (FISR -> FSUACR) ● Correct notes. ○ Note that it must wait for command processing to complete.

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/.