RX Family

Flash Module Using Firmware Integration Technology

Introduction

The Flash Module Using Firmware Integration Technology (FIT) has been developed to allow users of supported RX devices to easily integrate reprogramming abilities into their applications using self-programming. 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 FIT module and integrating it with your application program.

The Flash FIT module is different from the Simple Flash API that supports the earlier RX600 and the RX200 Series of MCUs (R01AN0544).

Target Device

The following is a list of devices that are currently supported by this API:

- RX110, RX111, RX113 Groups
- RX130 Group
- RX230, RX231 Groups
- RX23T, RX24T Groups
- RX24U Group
- RX64M Group
- RX65N, RX651 Groups
- RX66T Group
- RX71M Group
- RX72T Group

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

Target Compilers

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

Related Documents

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

Contents

1. Overview

The Flash FIT module is provided to customers to make the process of programming and erasing on-chip flash areas easier. Both ROM and data flash areas are supported. The module can be used to perform erase and program operations in blocking or non-blocking BGO mode. In blocking mode, when a program or erase function is called, the function does not return until the operation has finished. In Background Operations (BGO) mode, the API functions return immediately after the operation has begun. When a ROM operation is on-going, that ROM area cannot be accessed by the user. If an attempt is made to access the ROM area, the sequencer will transition into an error state. In BGO mode, whether operating on ROM or data flash, the user must poll for operation completion or provide a flash interrupt callback (if flash interrupt support is available on MCU).

1.1 Features

Below is a list of the features supported by the Flash FIT module.

- Erasing, programming, and blank checking for ROM and data flash in blocking mode or non-blocking BGO mode.
- Area protection via access windows or lockbits.
- Start-up program protection; this function is used to safely rewrite block 0 to block 7 in ROM

2. API Information

This Driver API follows the Renesas API naming standards.

2.1 Hardware Requirements

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

- Flash

2.2 Software Requirements

This driver is dependent upon the following FIT packages:

- Renesas Board Support Package (r_bsp) v5.00 or later

2.3 Limitations

- This code is not re-entrant and protects against multiple concurrent function calls (not including RESET).
- During ROM reprogramming, ROM cannot be accessed. When reprogramming ROM, make sure application code runs from RAM.

2.4 Supported Toolchains

This driver is tested and working with the following toolchains:

- Renesas RX Toolchain v3.01.00

2.5 Header Files

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

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

2.6 Integer Types

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

2.7 Flash Types and Features

The flash driver is divided into three separate types based upon the technology and sequencer used. The compiled flash driver size is based upon the flash type (see section 2.9).

FLASH TYPE 1
RX110*, RX111, RX113, RX130
RX230, RX231. RX23T*, RX24T, RX24U
**has no data flash*

FLASH TYPE 3
RX64M, RX66T, RX71M, RX72T

FLASH TYPE 4
RX651*, RX65N*
**no data flash on parts with less than or equal to 1M code flash*

Because of the different flash types, not all flash commands or features are available on all MCUs. The file r_flash_rx_if.h identifies which features are available on each MCU using #defines.

2.8 Configuration Overview

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

Table 1 Flash general configuration settings

Configuration options in r_flash_rx_config.h		
Equate	**Default Value**	**Description**
FLASH_CFG_PARAM_CHECKING_ENABLE	1	Setting to 1 includes parameter checking. Setting to 0 omits parameter checking.
FLASH_CFG_CODE_FLASH_ENABLE	0	If you are only using data flash, set this to 0. Setting to 1 includes code to program the ROM area. When programming ROM, code must be executed from RAM, except for FLASH_TYPE_3 (see HW Manual Table 63.18) and RX65N-2M (see HW Manual Table 59.15) under certain restrictions. See section 2.14 for details on how to set up code and the linker to execute code from RAM. See section 2.16 for driver definition of BGO mode.
FLASH_CFG_DATA_FLASH_BGO	0	Setting this to 0 forces data flash API function to block until completed. Setting to 1 places the module in BGO (background operations/interrupt) mode. In BGO mode, data flash operations return immediately after the operation has been started. Notification of the operation completion is done via the callback function. When FLASH_CFG_CODE_FLASH_ENABLE is set to 1, make the same setting as FLASH_CFG_CODE_FLASH_BGO.
FLASH_CFG_CODE_FLASH_BGO	0	Setting this to 0 forces ROM API function to block until completed. Setting to 1 places the module in BGO (background operations/interrupt) mode. In BGO mode, ROM operations return immediately after the operation has been started. Notification of the operation completion is done via the callback function. When reprogramming ROM, the relocatable vector table and corresponding interrupt routines must be relocated to an area other than ROM in advance. See sections 2.18 Usage Notes. When FLASH_CFG_CODE_FLASH_ENABLE is set to 1, make the same setting as FLASH_CFG_CODE_FLASH_BGO.

Configuration options in r_flash_rx_config.h		
Equate	**Default Value**	**Description**
FLASH_CFG_CODE_FLASH_RUN_FROM_ROM	0	For FLASH_TYPE_3, RX65N-2M. Valid only when FLASH_CFG_CODE_FLASH_ENABLE is set to 1. Set this to 0 when programming code flash while executing in RAM. Set this to 1 when programming code flash while executing from another segment in ROM (see section 2.15).

2.9 Code Size

The ROM size, RAM size, and the maximum stack size of this module are described in the following table. One device is listed at a time as the representative of each flash type.

The code size is based on optimization level 2 and optimization type for size for the RXC toolchain in Section 2.4. The ROM (code and constants) and RAM (global data) sizes are determined by the build-time configuration options set in the module configuration header file.

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

Module Revision:	r_flash_rx Rev.3.51
Compiler Version:	Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00 (The option of "-lang = c99" is added to the default settings of the integrated development environment.) GCC for Renesas RX 4.08.04.201801 (The option of "-std = gnu99" is added to the default settings of the integrated development environment.) IAR C/C++ Compiler for Renesas RX version 4.10.01 (The default settings of the integrated development environment.)
Configuration Options:	The setting of configuration options that are different is described in each table. Other configuration options are default settings.

Flash Type 1: ROM, RAM and Stack Code Sizes (Maximum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX130	ROM	860 bytes	833 bytes	1880 bytes	1724 bytes	−	−
	RAM	5391 bytes		10767 bytes		−	
	STACK	972 bytes		−		−	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_DATA_FLASH_BGO 1 FLASH_CFG_CODE_FLASH_ENABLE 1 FLASH_CFG_CODE_FLASH_BGO 1							

Flash Type 1: ROM, RAM and Stack Code Sizes (Minimum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX130	ROM	1823 bytes	1713 bytes	3790 bytes	3524 bytes	2655 bytes	2519 bytes
	RAM	73 bytes		76 bytes		55 bytes	
	STACK	964 bytes		−		1036 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_DATA_FLASH_BGO 0 FLASH_CFG_CODE_FLASH_ENABLE 0 FLASH_CFG_CODE_FLASH_BGO 0							

Flash Type 3: ROM, RAM and Stack Code Sizes (Maximum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX64M	ROM	557 bytes	544 bytes	1172 bytes	1140 bytes	−	−
	RAM	5993 bytes		12604 bytes		−	
	STACK	48 bytes		−		−	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE/DATA_FLASH_BGO 1 FLASH_CFG_CODE_FLASH_ENABLE 1							

Flash Type 3: ROM, RAM and Stack Code Sizes (Minimum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX64M	ROM	2147 bytes	2017 bytes	4564 bytes	4388 bytes	3639 bytes	3488 bytes
	RAM	65 bytes		332 bytes		48 bytes	
	STACK	32 bytes		−		72 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE/DATA_FLASH_BGO 0 FLASH_CFG_CODE_FLASH_ENABLE 0							

Flash Type 4: ROM, RAM and Stack Code Sizes (Maximum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX65N	ROM	487 bytes	474 bytes	972 bytes	948 bytes	－	－
	RAM	6109 bytes		12520 bytes		－	
	STACK	44 bytes		－		－	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE/DATA_FLASH_BGO 1							

Flash Type 4: ROM, RAM and Stack Code Sizes (Minimum Size)							
		Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
Device	Category	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX65N	ROM	1978 bytes	1848 bytes	4132 bytes	3964 bytes	3362 bytes	3211 bytes
	RAM	61 bytes		328 bytes		47 bytes	
	STACK	28 bytes		－		68 bytes	
Configuration options: FLASH_CFG_PARAM_CHECKING_ENABLE 0: Without parameter check, 1: With parameter check FLASH_CFG_CODE/DATA_FLASH_BGO 0							

2.10 API Data Types

This section describes the structures used as arguments to API functions. These structures are included in the file r_flash_rx_if.h along with the API function prototype declarations. API functions are explained in Section 3.

2.11 Return Values

This shows the different values API functions can return. This return type is defined in “r_flash_rx_if.h”.

```
/* Flash FIT 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, blank check error, etc. */

FLASH_ERR_CMD_LOCKED,   /* Type3 - Peripheral in command locked state */
FLASH_ERR_LOCKBIT_SET,  /* Type3 - Program/Erase error due to lock bit. */
FLASH_ERR_FREQUENCY,    /* Type3 - Illegal Frequency value attempted (4-60Mhz) */
FLASH_ERR_BYTES,        /* Invalid number of bytes passed */
FLASH_ERR_ADDRESS,      /* Invalid address */
FLASH_ERR_BLOCKS,       /* The "number of blocks" argument is invalid. */
FLASH_ERR_PARAM,        /* Illegal parameter */
FLASH_ERR_NULL_PTR,     /* Missing required argument */
FLASH_ERR_UNSUPPORTED,  /* Command not supported for this flash type */
FLASH_ERR_SECURITY,     /* Type4 - Pgm/Erase err due to part locked (FAW.FSPR)*/
FLASH_ERR_TIMEOUT,      /* Timeout condition */
FLASH_ERR_ALREADY_OPEN  /* Open() called twice without intermediate Close() */
} flash_err_t;
```

2.12 Adding the FIT Flash Module to Your Project

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

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

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

(3) Adding the FIT module to your project using the Smart Configurator in CS+
By using the Smart Configurator Standalone version in CS+, the FIT module is automatically added to your project. Refer to “Renesas e^2 studio Smart Configurator User Guide (R20AN0451)” for details.

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

2.13 Usage Combined with Existing User Projects

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

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

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

There are some settings and notes for using the BSP startup disable function. For details, refer to the application note "RX Family Board Support Package Module Using Firmware Integration Technology (R01AN1685)".

2.14 Programming Code Flash from RAM

MCUs require that sections in RAM and ROM be created to hold the API functions for reprogramming ROM. This is required because the sequencer (with some exceptions in Type 3) cannot program or erase ROM while executing from ROM. The RAM section will need to be initialized after reset.

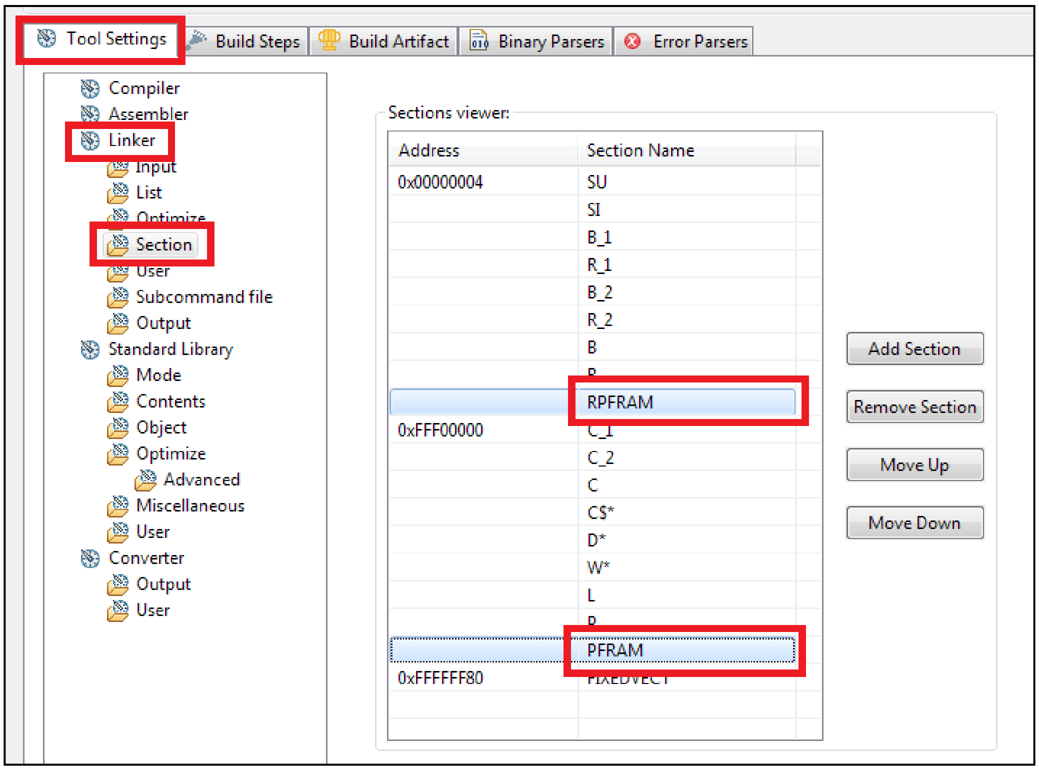
In order to enable ROM reprogramming, configure the FLASH_CFG_CODE_FLASH_ENABLE to 1 in the *r_flash_rx_config.h* file. Note that this is only for ROM programming.

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

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

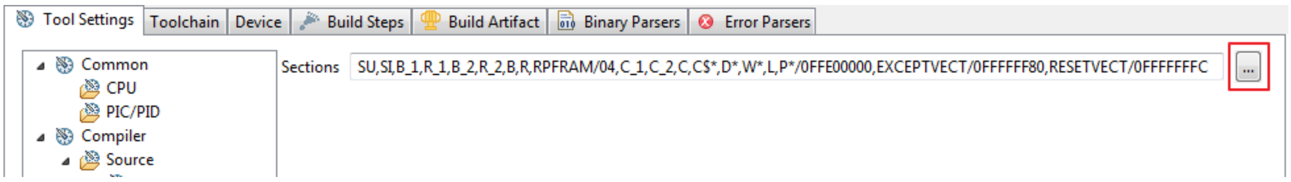
The process of setting up the linker sections and mapping from code flash to RAM need to be done in e^2 studio as listed below.

1. Add a new section titled '**RPFRAM'** in a RAM area.
2. Add a new section '**PFRAM'** in a code flash area.



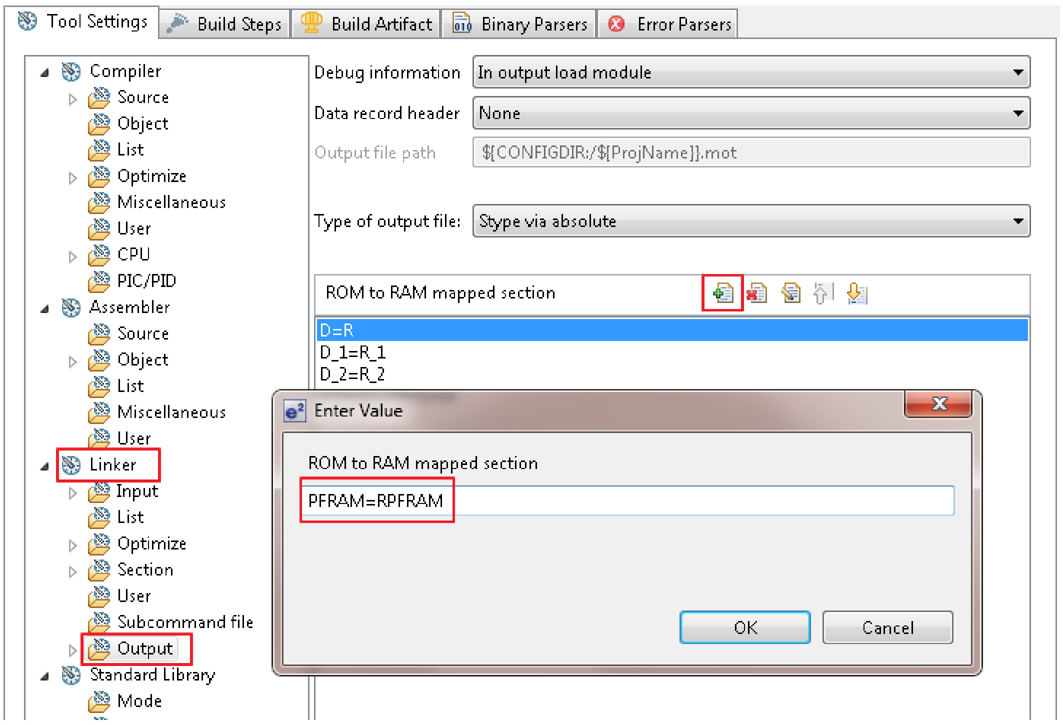
Address	Section Name
0x00000004	SU
	SI
	B_1
	R_1
	B_2
	R_2
	B
	R
	RPFRAM
0xFFF00000	C_1
	C_2
	C
	C$*
	D*
	W*
	L
	P
	PFRAM
0xFFFFFF80	FIXEDVECT

Note: Depending upon the e^{2}studio version you are using, there will be a section called P or P*. As mentioned above, if there is a P* section, it is not necessary to specify the PFRAM section separately. (If specified, a warning will occur.)
If you are using e^2 studio v6.0.0 or later, the section viewer does not appear by default. To display the section viewer, click the [...] button to the right of the [Sections] entry.



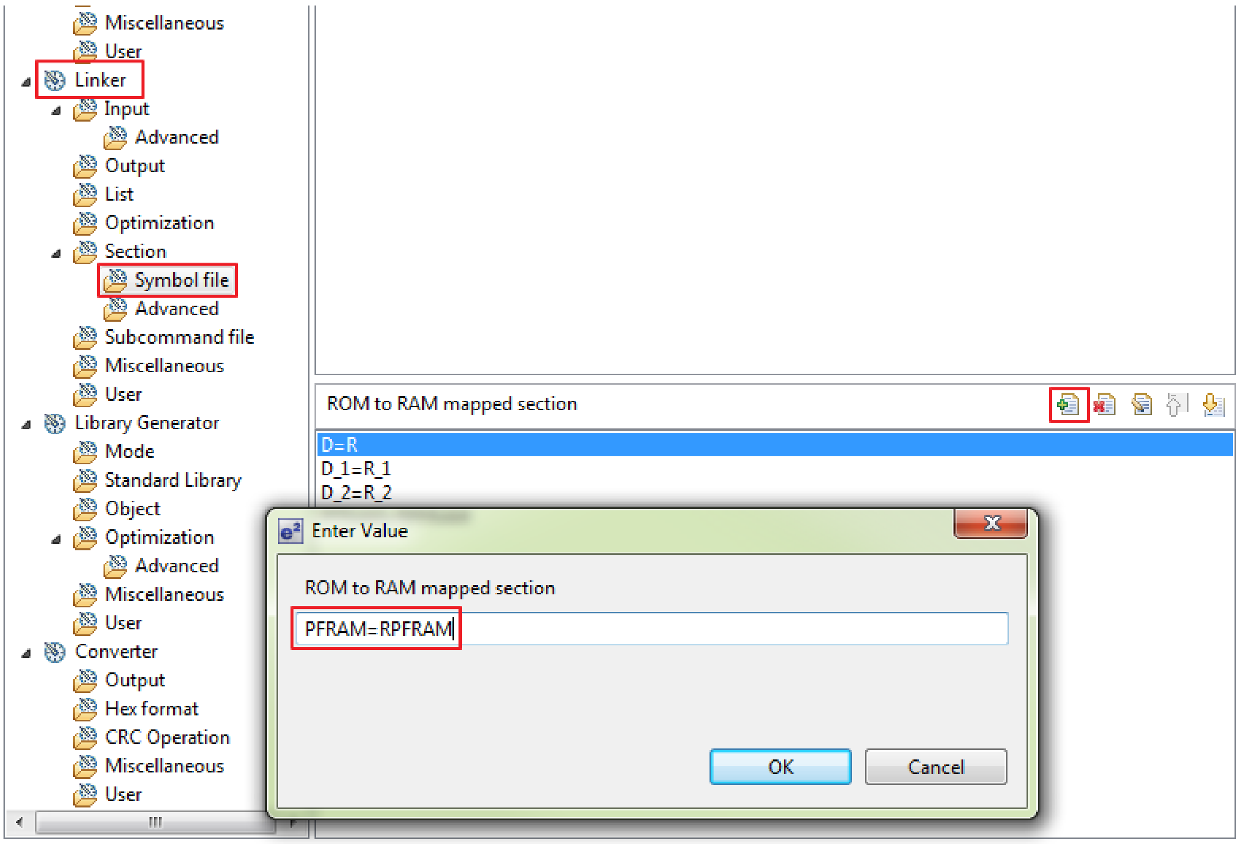
Sections SU,SI,B_1,R_1,B_2,R_2,B,R,RPFRAM/04,C_1,C_2,C,C$*,D*,W*,L,P*/0FFE00000,EXCEPTVECT/0FFFFFF80,RESETVECT/0FFFFFFFC

3. Add ‘PFRAM = RPFRAM’ to the linker output option (see below) and map the code flash section (PFRAM) address to the RAM section (RPFRAM) address. In e^2 studio earlier than v6.0.0, this is performed from the "Output" section of "Linker" in the "Tool Settings" tab.



Debug information: In output load module
Data record header: None
Output file path: ${CONFIGDIR:/${ProjName}}.mot
Type of output file: Stype via absolute
ROM to RAM mapped section
D=R
D_1=R_1
D_2=R_2
ROM to RAM mapped section
PFRAM=RPFRAM

If you are using e^2 studio v6.0.0 or later, add ‘PFRAM = RPFRAM’ via Linker > Section > Symbol file.



ROM to RAM mapped section
D=R
D_1=R_1
D_2=R_2
ROM to RAM mapped section
PFRAM=RPFRAM

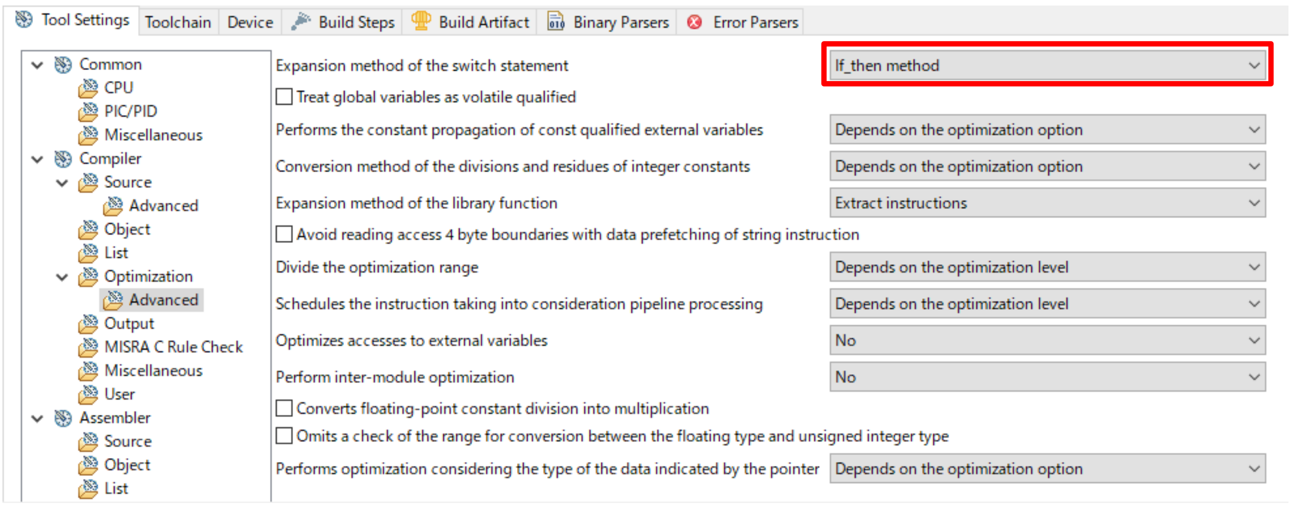
4. At this point the linker has been correctly set and the appropriate API codes have been allocated to RAM. Copying code from code flash to RAM is performed automatically when calling the R_FLASH_Open () function. If this is not completed before calling the API function, the MCU will jump to uninitialized RAM.
5. Codes that operate with the interrupt callback function and code flash need to be placed in the FRAM section.

```
#pragma section FRAM

/* Place functions (and interrupt callbacks) that operate on code flash here
*/

#pragma
```

6. When including the switch text code in the code copied from code flash to RAM, it will not be copied to RAM if the switch text code is added to the branch table. To ensure the code is securely placed in the FRAM section, carry out the following settings.



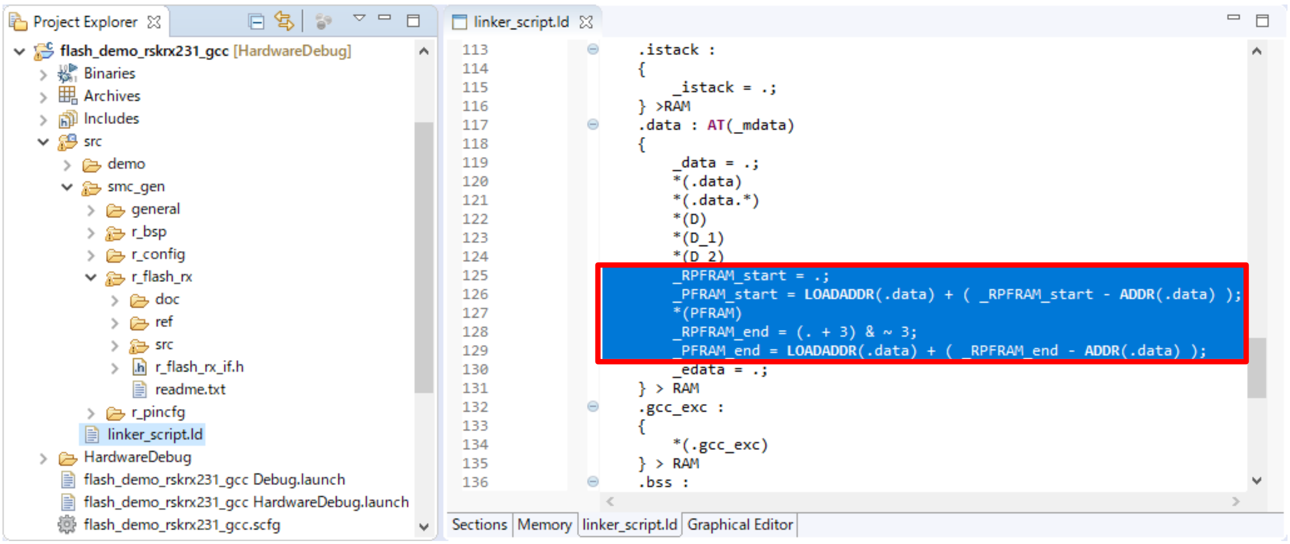
Expansion method of the switch statement	If_then method
☐ Treat global variables as volatile qualified	
Performs the constant propagation of const qualified external variables	Depends on the optimization option
Conversion method of the divisions and residues of integer constants	Depends on the optimization option
Expansion method of the library function	Extract instructions
☐ Avoid reading access 4 byte boundaries with data prefetching of string instruction	
Divide the optimization range	Depends on the optimization level
Schedules the instruction taking into consideration pipeline processing	Depends on the optimization level
Optimizes accesses to external variables	No
Perform inter-module optimization	No
☐ Converts floating-point constant division into multiplication	
☐ Omits a check of the range for conversion between the floating type and unsigned integer type	
Performs optimization considering the type of the data indicated by the pointer	Depends on the optimization option

2.14.2 Using GCC for Renesas RX

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

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

1. Add the following to the linker settings file (linker_script.ld).



```
.istack :
{
    _istack = .;
} >RAM
.data : AT(_mdata)
{
    _data = .;
    *(.data)
    *(.data.*)
    *(D)
    *(D_1)
    *(D_2)
    _RPFRAM_start = .;
    _PFRAM_start = LOADADDR(.data) + ( _RPFRAM_start - ADDR(.data) );
    *(PFRAM)
    _RPFRAM_end = (. + 3) & ~ 3;
    _PFRAM_end = LOADADDR(.data) + ( _RPFRAM_end - ADDR(.data) );
    _edata = .;
} > RAM
.gcc_exc :
{
    *(.gcc_exc)
} > RAM
.bss :
```

＊GCC for Renesas RX has confirmed only little endian operation.

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

Code flash rewriting is not supported.
Supports only data flash rewrite.

2.15 Programming Code Flash from ROM

For Flash Type 3 and RX65N-2M, with certain limitations ROM can be programmed while running from ROM. Basically ROM is broken into two regions. Code can run from one region and erase/write operations can be performed on the other. The size of these regions vary based upon the amount of ROM on the MCU. See Table 63.16 in the RX64M and Table 63.18 in the RX71M Hardware Manuals for boundary details. Only MCUs with large ROM areas support this feature. For the RX65N-2M group, the boundaries correspond to bank boundaries.

When this method is used, set FLASH_CFG_CODE_FLASH_ENABLE and FLASH_CFG_CODE_FLASH_RUN_FROM_ROM to 1 in the r_flash_rx_config.h file. FLASH_CFG_CODE_FLASH_BGO (functions do not block/wait for completion) may be set to 0 or 1, but must match the setting for data flash BGO.

Be sure not set up the linker as just described in section 2.14, but do guarantee that the region the code is running from is not the region being operated on!

2.16 Operations in BGO Mode

Historically, Background Operation (BGO) mode refers to the non-blocking mode of the driver- the ability to execute instructions from RAM while a code flash operation is running in the background. For Flash Type 3 and RX65N-2M devices, the hardware manual redefines BGO as the ability to program one region of code flash while executing from another region as discussed in section 2.14.

The #defines in r_flash_config.h use the historical definition of BGO as to whether or not to use interrupts or to perform blocking. For Flash Type 3 and RX65N-2M devices, the new “BGO” feature is indicated with the equate FLASH_CFG_CODE_FLASH_RUN_FROM_ROM.

When operating in BGO mode, API function calls do not block 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 an 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 is passed an event to indicate the completion status. The possible events (some MCU-specific) are located in “r_flash_rx_if.h” and are as follows:

```
typedef enum _flash_interrupt_event
{
    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_LOCKBIT_WRITTEN,
    FLASH_INT_EVENT_LOCKBIT_PROTECTED,
    FLASH_INT_EVENT_LOCKBIT_NON_PROTECTED,
    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_TOGGLE_BANK,
    FLASH_INT_EVENT_END_ENUM
} flash_interrupt_event_t;
```

When reprogramming ROM, the relocatable vector table and associated interrupts must be relocated to an area other than ROM in advance (exception- Flash Type 3 and RX65N-2M usage as explained in section 2.15).

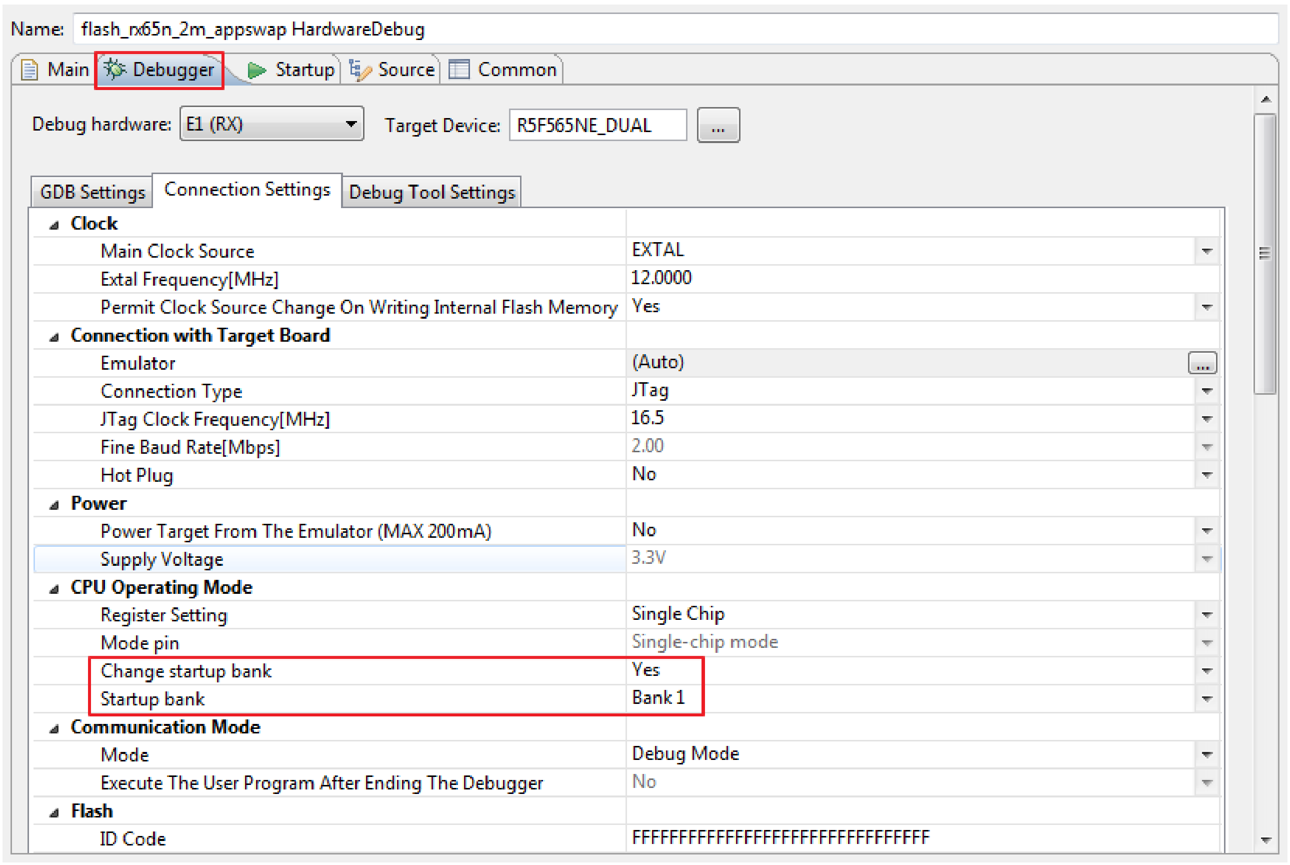
2.17 Dual Bank Operation

The RX65N-2M Group can operate in two modes- linear and dual bank. Linear mode is the standard mode where a single application runs out of code flash. Dual bank mode allows two applications to be loaded into code flash simultaneously. The application loaded into the upper half of code flash (the part which contains the fixed vector table) is the application that runs. Applications can be swapped at runtime using the command R_FLASH_Control(FLASH_CMD_BANK_TOGGLE). Note that the swap does not take effect until the next MCU reset.

When developing these applications, two constants in bsp_config.h must be modified:

- BSP_CFG_CODE_FLASH_BANK_MODE 0 // set to 0 for dual mode (not default)
- BSP_CFG_CODE_FLASH_START_BANK 1 // different value for each application

The mode constant should be set to 0 in both applications. The start bank should be set to 1 in one application and 0 in the other. Either bank can actually be the boot bank. The bank that is booted from is selected in the debug configuration.

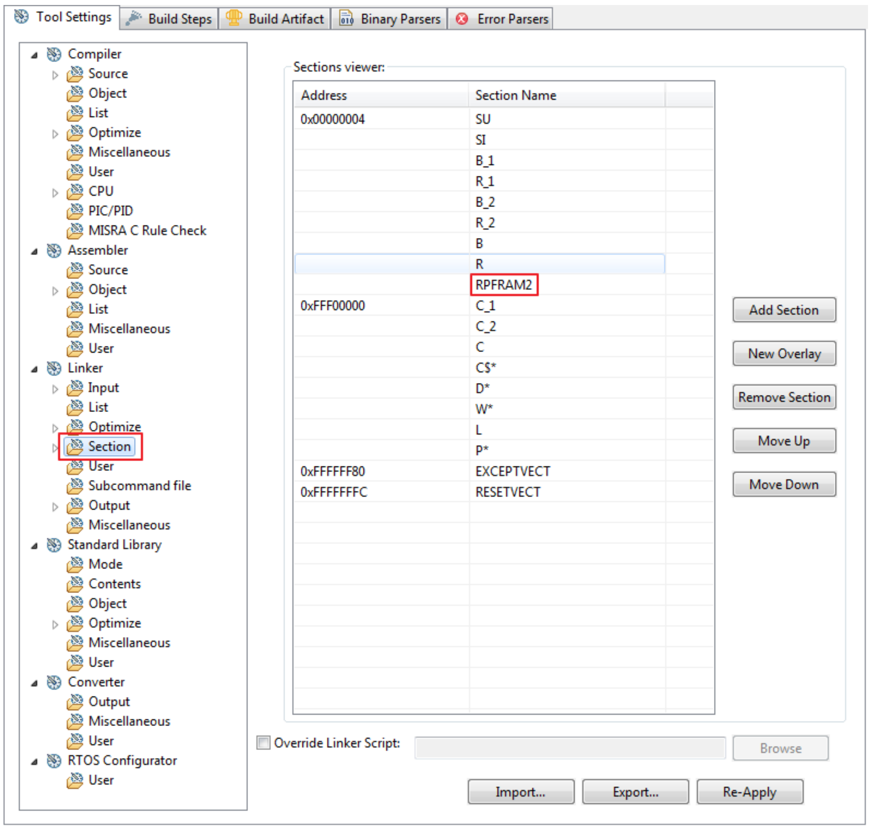

Name: flash_rx65n_2m_appswap HardwareDebug

Debug hardware: E1 (RX) Target Device: R5F565NE_DUAL

Clock	
Main Clock Source	EXTAL
Extal Frequency[MHz]	12.0000
Permit Clock Source Change On Writing Internal Flash Memory	Yes
Connection with Target Board	
Emulator	(Auto)
Connection Type	JTag
JTag Clock Frequency[MHz]	16.5
Fine Baud Rate[Mbps]	2.00
Hot Plug	No
Power	
Power Target From The Emulator (MAX 200mA)	No
Supply Voltage	3.3V
CPU Operating Mode	
Register Setting	Single Chip
Mode pin	Single-chip mode
Change startup bank	Yes
Startup bank	Bank 1
Communication Mode	
Mode	Debug Mode
Execute The User Program After Ending The Debugger	No
Flash	
ID Code	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

In the above example, the application built with BSP_CFG_CODE_FLASH_START_BANK set to 1 will be the application that is executed at reset.

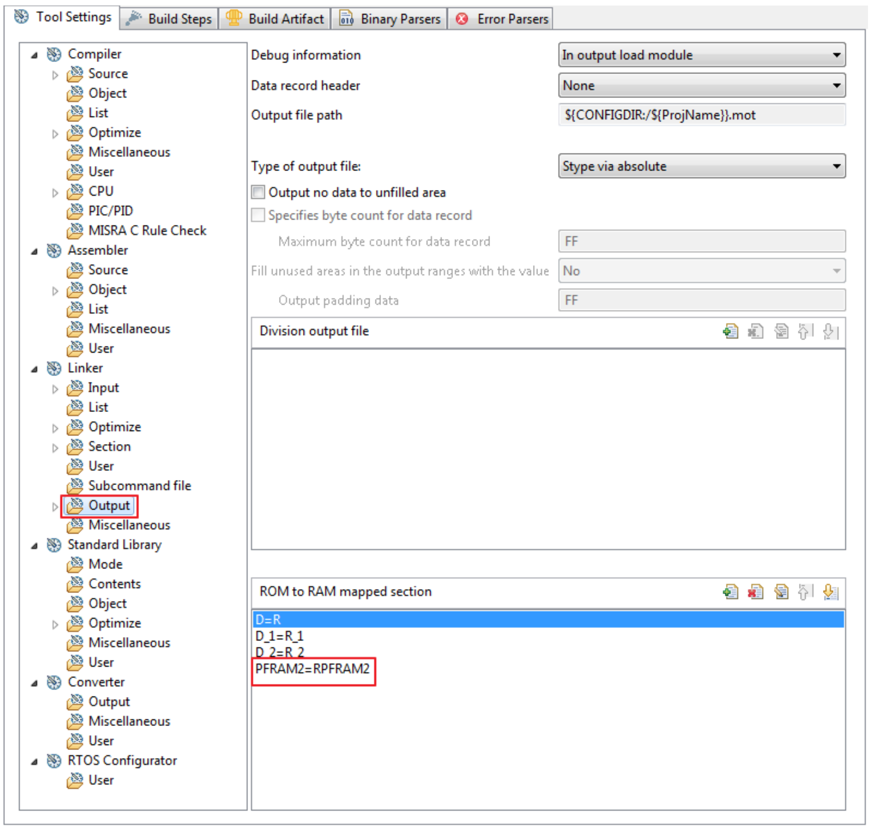
As mentioned in Section 2.15, the flash driver can erase and program code flash in the other bank without running from RAM (set FLASH_CFG_CODE_FLASH_RUN_FROM_ROM to 1 in r_flash_config.h). However, the code that handles swapping of the banks must execute from RAM. To accomplish this, add the section RPFRAM2 to the linker section table and linker mapped output as follows:



Sections viewer:

Address	Section Name
0x00000004	SU
	SI
	B_1
	R_1
	B_2
	R_2
	B
	R
	RPFRAM2
0xFFF00000	C_1
	C_2
	C
	C$*
	D*
	W*
	L
	P*
0xFFFFFF80	EXCEPTVECT
0xFFFFFFFC	RESETVECT

Note: For e^{2}studio v6.0.0 or later, a "…" button appears at the top right which must be pressed to display the section table as shown in this earlier release.



Debug information	In output load module
Data record header	None
Output file path	${CONFIGDIR:/${ProjName}}.mot
Type of output file:	Stype via absolute
☐ Output no data to unfilled area	
☐ Specifies byte count for data record	
Maximum byte count for data record	FF
Fill unused areas in the output ranges with the value	No
Output padding data	FF

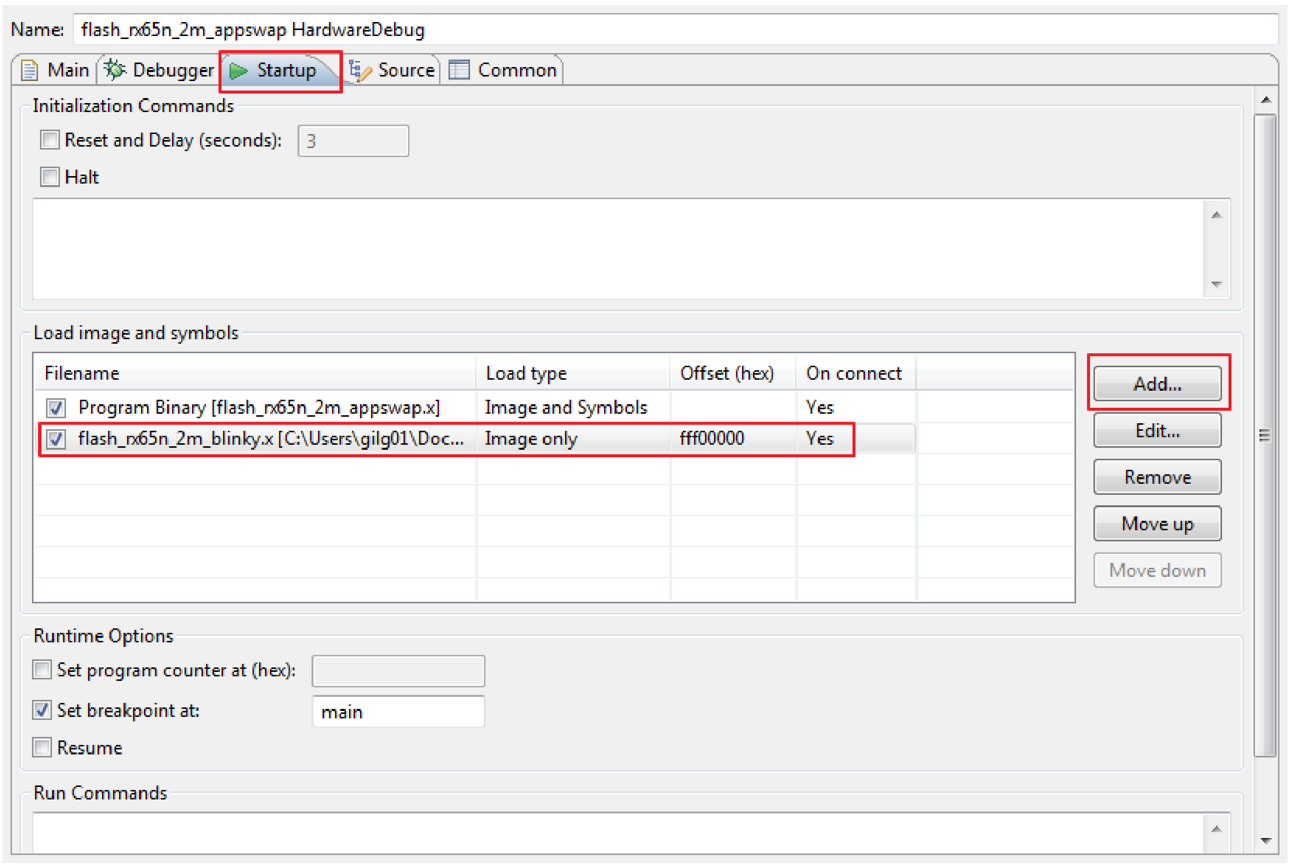
Division output file

ROM to RAM mapped section

```
D=R
D_1=R_1
D_2=R_2
PFRAM2=RPFRAM2
```

Note: For e^{2}studio v6.0.0 or later, "ROM to RAM mapped section" is in Linker->Section->Symbol file, not Linker->Output

To have the emulator also download the application built with BSP_CFG_CODE_FLASH_START_BANK set to 0, it must be added to the startup tab as shown below.



Name: flash_rx65n_2m_appswap HardwareDebug

Initialization Commands

☐ Reset and Delay (seconds): 3

☐ Halt

Load image and symbols

	Filename	Load type	Offset (hex)	On connect
✓	Program Binary [flash_rx65n_2m_appswap.x]	Image and Symbols		Yes
✓	flash_rx65n_2m_blinky.x [C:\Users\gilg01\Doc...	Image only	fff00000	Yes

Runtime Options

☐ Set program counter at (hex):

☑ Set breakpoint at: main

☐ Resume

Run Commands

Note that the offset for the second application should always be FFF00000 for a 2Mb part. This is two's compliment for -1M (not a starting address). This means that the application will be loaded into memory 1Mb lower than the values shown in the linker or map file. But after the banks are swapped, the addresses in memory will match those that are executing.

Currently, e^{2}studio can only maintain one debug symbol table at a time. Therefore, only one of the applications should have "Image and Symbols" selected for the load type. Whenever the debug session is paused, e^{2}studio will always use this symbol table for displaying source code and variables. Note that this may not even be correct for the application that was running. Because of this, it is recommended that the user uses an LED to indicate which application is running, and therefore know when the program is paused whether or not the source code displayed matches the executing code. This limitation should only be a minor inconvenience when you consider that the applications can be fully debugged independently first, and that the symbol table loaded can be easily changed when desired.

2.18 Usage Notes

2.18.1 Data Flash Operations in BGO Mode

When reprogramming data flash in BGO/non-blocking mode, ROM, RAM, and external memory can still be accessed. Care should be taken to make sure that the data flash is not accessed during data flash operations. This includes interrupts that may access the data flash.

2.18.2 ROM Operations in BGO Mode

When reprogramming ROM in BGO/non-blocking mode, external memory and RAM can still be accessed. Since the flash API functions will return before the ROM operation has finished, the code that calls the API function will need to be in RAM, and the code will need to check for completion before issuing another Flash command. Note that this includes setting the code flash access window, swapping boot blocks/toggling startup area flag, erasing code flash, writing code flash, as well as reading the Unique ID with another FIT Module (R01AN2191).

2.18.3 ROM Operations and General Interrupts

ROM or data flash areas cannot be accessed while a flash operation is on-going for that particular memory area. This means that the relocatable vector table will need to be taken care of when allowing interrupts to occur during flash operations.

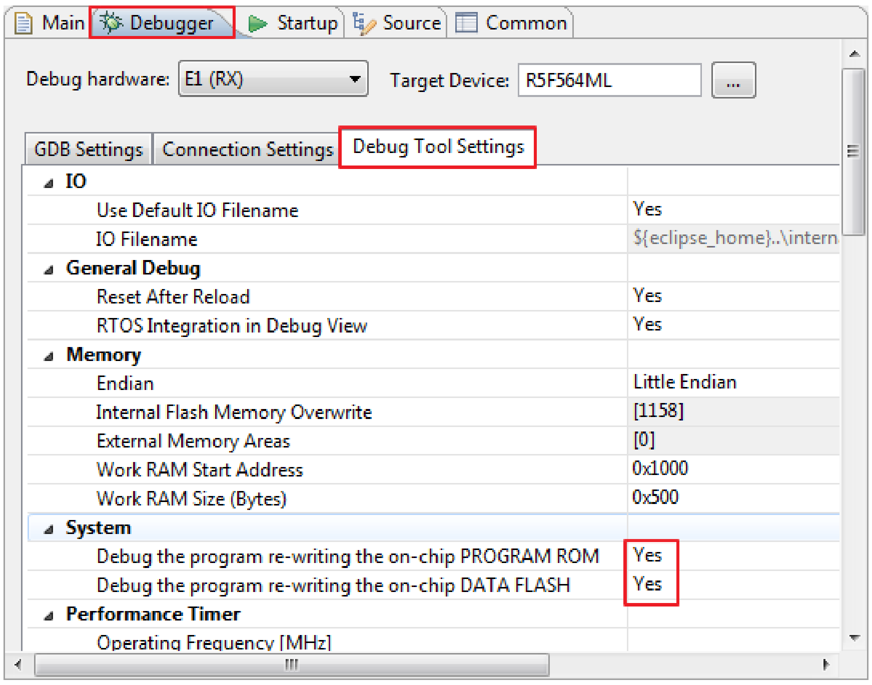
The vector table is placed in ROM by default. If an interrupt occurs during ROM operation, then ROM will be accessed to fetch the interrupt's starting address and an error will occur. To fix this situation the user will need to relocate the vector table and any interrupt handlers that may occur outside of ROM. The user will also need to change the interrupt table register (INTB).

The module does not include the function to relocate the vector table and the interrupt handler. Please consider an appropriate method to relocate them according to the user system.

2.18.4 Emulator Debug Configuration

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

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



Debug hardware: E1 (RX)   Target Device: R5F564ML

IO	
Use Default IO Filename	Yes
IO Filename	${eclipse_home}..\intern
General Debug	
Reset After Reload	Yes
RTOS Integration in Debug View	Yes
Memory	
Endian	Little Endian
Internal Flash Memory Overwrite	[1158]
External Memory Areas	[0]
Work RAM Start Address	0x1000
Work RAM Size (Bytes)	0x500
System	
Debug the program re-writing the on-chip PROGRAM ROM	Yes
Debug the program re-writing the on-chip DATA FLASH	Yes
Performance Timer	
Operating Frequency [MHz]	

3. API Functions

3.1 Summary

The following functions are included in this design:

Function	Description
R_FLASH_Open()	Initializes the Flash FIT module.
R_FLASH_Close()	Closes the Flash FIT module.
R_FLASH_Erase()	Erases the specified block of ROM or data flash.
R_FLASH_BlankCheck()	Checks if the specified data flash or ROM area is blank.
R_FLASH_Write()	Write data to ROM or data flash.
R_FLASH_Control()	Configures settings for the status check, area protection, and switching areas for start-up program protection.
R_FLASH_GetVersion()	Returns the current version of this FIT module.

3.2 R_FLASH_Open

The function initializes the Flash FIT module. 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 FIT module initialized successfully*
FLASH_ERR_BUSY:	*Another flash operation in progress, try again later*
FLASH_ERR_ALREADY_OPEN:	*Open() called twice without an intermediate Close()*

Properties

Prototyped in file “r_flash_rx_if.h”

Description

This function initializes the Flash FIT module, and if FLASH_CFG_CODE_FLASH_ENABLE is 1, copies the API functions necessary for ROM erasing/reprogramming into RAM (not including vector table). Note that this function must be called before any other API function.

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 FIT module.

Format

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

Parameters

None

Return Values

FLASH_SUCCESS:	*Flash FIT module closed successfully*
FLASH_ERR_BUSY:	*Another flash operation in progress, try again later*

Properties

Prototyped in file “r_flash_rx_if.h”

Description

This function closes the Flash FIT module. It disables the flash interrupts (if enabled) and sets the driver to an uninitialized state. This function is only required when the VEE (Virtual EEPROM) driver will be used. In that case, the flash driver must be closed (or never opened) prior to calling R_VEE_Open().

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 block in ROM or data 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 erase. The enum flash_block_address_t is defined in the corresponding MCU's r_flash_rx\src\targets\mcu\r_flash_mcu.h file. The blocks are labeled in the same fashion as they are in the device's Hardware Manual. For example, the block located at address 0xFFFFC000 is called Block 7 in the RX113 hardware manual, therefore "FLASH_CF_BLOCK_7" should be passed for this parameter. Similarly, to erase Data Flash Block 0 which is located at address 0x00100000, this argument should be FLASH_DF_BLOCK_0.

num_blocks

Specifies the number of blocks to be erased. For type 1 parts, *address* + *num_blocks* cannot cross a 256K boundary.

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_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/poling mode is enabled)*

Properties

Prototyped in file "r_flash_rx_if.h"

Description

Erases a contiguous number of ROM or data flash memory blocks.

The block size varies depending on MCU types. For example, on the RX111 both code and data flash block sizes are 1Kbytes. On the RX231 and RX23T the block size for ROM is 2 Kbytes and for data flash is 1Kbyte (no data flash on the RX23T). The equates FLASH_CF_BLOCK_SIZE for ROM and FLASH_DF_BLOCK_SIZE for data flash are provided for these values.

The enum *flash_block_address_t* is configured at compile time based on the memory configuration of the MCU device specified in the r_bsp module.

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

Reentrant

No.

Example

```
flash_err_t err;

/* Erase Data Flash blocks 0 and 1 */
err = R_FLASH_Erase(FLASH_DF_BLOCK_0, 2);

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

Special Notes:
In order to erase a ROM block, the area to be erased needs to be in a rewritable area. FLASH_TYPE_1 uses access windows to identify this. The other flash types use lock bits which must be off for erasing.

3.5 R_FLASH_BlankCheck

This function is used to determine if the specified area in either ROM or data flash is blank or not.

Format

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

Parameters

address

The address of the area to blank check. MCUs may support this feature on data flash, code flash, both, or neither.

num_bytes

For flash types 1, 3, and 4, this is the number of bytes to be checked. The number of bytes specified must be a multiple of FLASH_DF_MIN_PGM_SIZE for a data flash address or FLASH_CF_MIN_PGM_SIZE for a code flash address. These equates are defined in r_flash_rx\src\targets\<mcu>\r_flash_<mcu>.h and are MCU specific. For type 1 parts, *address* + *num_bytes* cannot cross a 256K boundary.

**blank_check_result*

Pointer that will be populated by the API with the results of the blank check operation in blocking (non-BGO) mode

Return Values

FLASH_SUCCESS:	*Operation successful (in BGO mode, this means the operation was started successfully)*
FLASH_ERR_FAILURE:	*Blank check Failed. Sequencer has been reset, or callback function not registered (if BGO mode is enabled with flash interrupt support)*
FLASH_ERR_BUSY:	*Another flash operation in progress or the module is not initialized*
FLASH_ERR_BYTES:	*num_bytes was either too large or not a multiple of the minimum programming size or exceed the maximum range*
FLASH_ERR_ADDRESS:	*Invalid address was input or address not divisible by the minimum programming size*

Properties

Prototyped in file “r_flash_rx_if.h”

Description

The result of the blank check operation is placed into blank_check_result when operating in blocking mode. This variable is of type flash_res_t which is defined in r_flash_rx_if.h. If the API is used in BGO/non-blocking mode, after the blank check is complete, the result of the blank check is passed as the argument of the callback function.

Reentrant

No.

Example: Flash Types 1, 3, and 4
Second argument is number of bytes to check (must be multiple of FLASH_DF_MIN_PGM_SIZE).

```
flash_err_t err;
flash_res_t result;

/* Blank check first 64 bytes of data flash block 0 */
err = R_FLASH_BlankCheck((uint32_t)FLASH_DF_BLOCK_0, 64, &result);
if (err != FLASH_SUCCESS)
{
    /* handle error */
}
else
{
    /* Check result. */
    if (FLASH_RES_NOT_BLANK == result)
    {
        /* Block is not blank. */
        . . .
    }
    else if (FLASH_RES_BLANK == ret)
    {
        /* Block is blank. */
        . . .
    }
}
```

Special Notes:
None

3.6 R_FLASH_Write

This function is used to write data to ROM or data flash.

Format

```
flash_err_t R_FLASH_Write(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 Flash.

dest_address

This is a pointer to the ROM or data flash area to write. The address specified must be divisible by the minimum programming size. See *Description* below for important restrictions regarding this parameter.

num_bytes

The number of bytes contained in the buffer specified with *src_address*. This number must be a multiple of the minimum programming size for memory area you are writing to.

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 under access window or lockbit control; or callback function not present(BGO mode with flash interrupt support)*
FLASH_ERR_BUSY:	*Another flash operation in progress or the module not initialized*
FLASH_ERR_BYTES:	*Number of bytes provided was not a multiple of the minimum programming size or exceed the maximum range*
FLASH_ERR_ADDRESS:	*Invalid address was input or address not divisible by the minimum programming size.*

Properties

Prototyped in file “r_flash_rx_if.h”

Description

Writes data to flash memory. Before writing to any flash area, the area must already be erased.

When performing a write the user must make sure to start the write on an address divisible by the minimum programming size and make the number of bytes to write be a multiple of the minimum programming size. The minimum programming size differs depending on what MCU package is being used and whether the ROM or data flash is being written to.

An area to write data to ROM must be rewritable area (access window or lockbit allowed).

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

Reentrant

No.

Example

```
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));

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

Special Notes:

FLASH_DF_MIN_PGM_SIZE defines the minimum data flash program size.

FLASH_CF_MIN_PGM_SIZE defines the minimum ROM (code flash) program size.

3.7 R_FLASH_Control

This function implements all non-core functionality of the sequencer.

Format

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

Parameters

cmd

Command to execute.

**pcfg*

Configuration parameters required by the specific command. This maybe NULL if the command does not require it.

Return Values

```
FLASH_SUCCESS:          Operation successful (in BGO mode,
                          this means the operations was started successfully)
FLASH_ERR_BYTES:        Number of blocks exceeds max range
FLASH_ERR_ADDRESS:      Address is an invalid Code/Data Flash block start
                          address
FLASH_ERR_NULL_PTR:     pcfg was NULL for a command that expects a configuration
                          structure
FLASH_ERR_BUSY:         Another flash operation in progress or API not
                          initialized
FLASH_ERR_LOCKED:       The 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
```

Properties

Prototyped in file “r_flash_rx_if.h”

Description

This function is an expansion function that implements non-core functionality of the sequencer. Depending on the command type a different argument type has to be passed.

Command	Argument	Operation
Flash type 1,3,4 FLASH_CMD_RESET	NULL	Resets the flash sequencer. This may or may not wait for the current flash operation to complete (operation dependent).
Flash type 1,3,4 FLASH_CMD_STATUS_GET	NULL	Returns the status of the API (Busy or Idle).
Flash type 1,3,4 FLASH_CMD_SET_BGO_CALLBACK	flash_interrupt_config_t *	Registers the callback function.
Flash type 1,4 FLASH_CMD_ACCESSWINDOW_GET	flash_access_window_config_t *	Returns the access window boundaries for ROM.
Flash type 1,4 FLASH_CMD_ACCESSWINDOW_SET	flash_access_window_config_t *	Specifies the access window boundaries for ROM (types 1,4). Types 1 & 4 use callback function in BGO/non-blocking mode.**
Flash type 1,4 FLASH_CMD_SWAPFLAG_GET	uint32_t *	Loads the flag indicating the designated boot block startup area (SASMF type 1, BTFLG type 4).

Command	Argument	Operation
Flash type 1,4 FLASH_CMD_SWAPFLAG_TOGGLE	NULL	Toggles the flag indicating the designated boot block start-up area. Boot block swap takes effect at next reset. Uses callback function in BGO/non-blocking mode.**
Flash type 1,4 FLASH_CMD_SWAPSTATE_GET	uint8_t *	Loads the flags (FLASH_SAS_xxx values) indicating temporary boot block startup area.
Flash type 1,4 FLASH_CMD_SWAPSTATE_SET	uint8_t *	Sets the flags (FLASH_SAS_xxx values) indicating the temporary boot block startup area. The value of SWAPFLAG still indicates boot block area used at next reset.
Flash type 3 FLASH_CMD_LOCKBIT_READ	flash_lockbit_config_t *	Loads argument with FLASH_RES_LOCKBIT_STATE_PROTECTED or FLASH_RES_LOCKBIT_STATE_NON_PROTECTED for block address provided.
Flash type 3 FLASH_CMD_LOCKBIT_WRITE	flash_lockbit_config_t *	Sets the lockbit for the number of blocks specified starting with the block address provided. Uses callback function in BGO/non-blocking mode.
Flash type 3 FLASH_CMD_LOCKBIT_ENABLE	NULL	Prohibits erasing/writing of blocks with lockbit set.
Flash type 3 FLASH_CMD_LOCKBIT_DISABLE	NULL	Allows erasing/writing of blocks with lockbit set. NOTE: Erasing a block clears the lockbit.
Flash type 3,4 FLASH_CMD_CONFIG_CLOCK	uint32_t *	Speed in Hz that FCLK is running at. Only needs to be called if clock speed changes at run time.
RX24T, RX24U, RX65x, RX66T, RX72T FLASH_CMD_ROM_CACHE_ENABLE	NULL	Enables caching of ROM (invalidates cache first).
RX24T, RX24U, RX65x, RX66T, RX72T FLASH_CMD_ROM_CACHE_DISABLE	NULL	Disables caching of ROM. Call before rewriting ROM.
RX24T, RX24U, RX65x, RX66T, RX72T FLASH_CMD_ROM_CACHE_STATUS	uint8_t *	Sets the value to 1 if caching is enabled; 0 if disabled.
RX66T, RX72T FLASH_CMD_SET_NON_CACHED_RANGE0	flash_non_cached_t *	Sets ROM range to limit/disable caching.
RX66T, RX72T FLASH_CMD_SET_NON_CACHED_RANGE1	flash_non_cached_t *	Sets another ROM range to limit/disable caching.

Command	Argument	Operation
RX66T, RX72T FLASH_CMD_GET_NON_CACHED_RANGE0	flash_non_cached_t *	Retrieves non-cached settings for RANGE0.
RX66T, RX72T FLASH_CMD_GET_NON_CACHED_RANGE1	flash_non_cached_t *	Retrieves non-cached settings for RANGE1.
RX65N-2M FLASH_CMD_BANK_TOGGLE	NULL	Swaps startup bank. Becomes effective at next reset. Uses callback function in BGO/non-blocking mode.**
RX65N-2M FLASH_CMD_BANK_GET	flash_bank_t *	Loads the current BANKSEL value (bank and address effective at next reset).

Note: ** These commands will block until completed even when in BGO (interrupt) mode. This is necessary while flash reconfigures itself. The callback function will still be called upon completion in BGO mode.

Reentrant

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

Example 1: Polling in BGO mode
To spin in a loop while waiting for a flash operation to complete and doing nothing else is functionally the same as operating in normal blocking mode. BGO mode is used when other processing must be performed while waiting for a flash operation to complete.

```
    flash_err_t err;

    /* erase all of data flash */
    R_FLASH_Erase(FLASH_DF_BLOCK_0, FLASH_NUM_BLOCKS_DF);

    /* 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 up BGO mode with interrupt support on flash types 1, 3 and 4.
BGO/non-blocking mode is enabled when FLASH_CFG_DATA_FLASH_BGO equals 1 or FLASH_CFG_CODE_FLASH_BGO equals 1. When reprogramming ROM, relocate the relocatable vector table to RAM. Also, the callback function must be registered prior to write/erase/blank check calls.

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

    /* Relocate the Relocatable Vector Table in RAM */

    /* It is also possible to set the address of the flash ready interrupt
       function directly to ram_vect_table[23]. Please consider the method
       according to the user's system.*/
    pvect_table = (uint32_t *)__sectop("C$VECT");
    ram_vect_table[23] = pvect_table[23]; /* FRDYI Interrupt function Copy */
    set_intb((void *)ram_vect_table);

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

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

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

    ... (omission)
}
```

```
#pragma section FRAM

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

    /* Perform ISR callback functionality here */
}

void do_rom_operations(void)
{
    /* Set cf access window, toggle startup area flag/swap boot blocks,
       erase, blank check, or write ROM here */

    ... (omission)
}

#pragma section
```

Example 3: Get 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: Set access window code flash (flash types 1 and 4)

The area protection is used to prevent unauthorized programming or erasure of ROM blocks. The following example makes only block 3 writeable (and everything else is not writeable).

```
flash_err_t                   err;
flash_access_window_config_t access_info;

/* Allow write to Code Flash block 3 */

access_info.start_addr = (uint32_t) FLASH_CF_BLOCK_3;
access_info.end_addr = (uint32_t) FLASH_CF_BLOCK_2;
err = R_FLASH_Control(FLASH_CMD_ACCESSWINDOW_SET, (void *)&access_info);
if (FLASH_SUCCESS != err)
{
    printf("Control FLASH_CMD_ACCESSWINDOW_SET command failure.");
}
```

Example 5: Get value of active startup area

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

```
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: Swap active startup area
The following example shows how to toggle the active start-up program area. Swap the area with the function placed in RAM.

```
flash_err_t err;

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

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

Example 7: Get value of startup area select bit
The example below shows how to read the current value in the start-up area select bit (FISR.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: Set value of startup area select bit
The example below shows how to set the value to the start-up area select bit (FISR.SAS) for the start-up program area. Swap the area with the function placed in 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.");
}
```

Example 9: Using ROM cache
The example below shows cache command usage when rewriting ROM.

```
uint8_t    status;

/* Enable caching towards beginning of application */
R_FLASH_Control(FLASH_CMD_ROM_CACHE_ENABLE, NULL);

/* Put main code here; optionally verify that flash is enabled */
R_FLASH_Control(FLASH_CMD_ROM_CACHE_STATUS, &status);
if (status != 1)
{
    // should never happen
}

/* Prepare to rewrite ROM */
R_FLASH_Control(FLASH_CMD_ROM_CACHE_DISABLE, NULL);

/* Erase, write, and verify new ROM code here */

/* Re-enable caching */
R_FLASH_Control(FLASH_CMD_ROM_CACHE_ENABLE, NULL);
```

Example 10: Limit ROM caching over a specific range.
The example below shows non-cached range command usage. It is legal to have non-cached ranges overlap.

```
flash_non_cached_t range;
uint8_t            status;

/* Do not cache fast-instruction fetching or operand access by the CPU
 * for the first 1K of ROM in flash block 10. ROM caching will be temporarily
 * disabled by the Control() command if was already enabled.
 */
range.start_addr = (uint32_t)FLASH_CF_BLOCK_10;
range.size = FLASH_NON_CACHED_1_KBYTE;
range.type_mask = FLASH_NON_CACHED_MASK_IF | FLASH_NON_CACHED_MASK_OA;

R_FLASH_Control(FLASH_CMD_SET_NON_CACHED_RANGE0, &range);

/* Enable caching */
R_FLASH_Control(FLASH_CMD_ROM_CACHE_ENABLE, NULL);


/* Retrieve non-cached settings for RANGE0 */
R_FLASH_Control(FLASH_CMD_GET_NON_CACHED_RANGE0, &range);
```

Special Notes:
None

3.8 R_FLASH_GetVersion

Returns the current version of the Flash FIT module.

Format

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

Parameters

None.

Return Values

Version of the Flash FIT module.

Properties

Prototyped in file “r_flash_rx_if.h”

Description

This function will return the version of the currently installed Flash API. 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 FIT. */
cur_version = R_FLASH_GetVersion();

/* Check to make sure version is new enough for this application’s use. */
if (MIN_VERSION > cur_version)
{
    /* This Flash FIT version is not new enough and does not have XXX feature
       that is needed by this application. Alert user. */
    ...
}
```

Special Notes:

This function is specified to be an inline function.

4. Demo Projects

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

4.1 flash_demo_rskrx113

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

Setup and Execution

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

Boards Supported

RSKRX113

4.2 flash_demo_rskrx231

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

Setup and Execution

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

Boards Supported

RSKRX231

4.3 flash_demo_rskrx23t

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

Setup and Execution

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

Boards Supported

RSKRX23T

4.4 flash_demo_rskrx130

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

Setup and Execution

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

Boards Supported

RSKRX130

4.5 flash_demo_rskrx24t

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

Setup and Execution

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

Boards Supported

RSKRX24T

4.6 flash_demo_rskrx65n

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

Setup and Execution

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

Boards Supported

RSKRX65N-1

4.7 flash_demo_rskrx24u

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

Setup and Execution

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

Boards Supported

RSKRX24U

4.8 flash_demo_rx65n2mb_bank1_bootapp / _bank0_otherapp

This is a simple demo for the dual bank operation of the RX65N-2MB demo board. The demo uses the r_flash_rx API with blocking functionality to read the BANKSEL register and swap banks/applications at next reset. The bank 1 application flashes LED1 when it is running. The bank 0 application flashes LED0 when it is running.

Setup and Execution

1. Build flash_demo_rx65n2mb_bank1_bootapp, and build flash_demo_rx65n2mb_bank0_otherapp.
2. Download (HardwareDebug) flash_demo_rx65n2mb_bank1_bootapp (its debug configuration also downloads the other app).
3. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.
4. Notice LED1 is flashing. Press the reset switch on the board. Notice LED0 is flashing (banks have swapped and the other application is now running). Continue this reset process if desired.

Boards Supported

RSKRX65N-2MB

4.9 flash_demo_rskrx64m

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

Setup and Execution

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

Boards Supported

RSKRX64M

4.10 flash_demo_rskrx64m_runrom

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

Setup and Execution

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

Boards Supported

RSKRX64M

4.11 flash_demo_rskrx66t

This is a simple demo for the RSKRX66T starter kit for e^{2}studio v6.2.0. The demo uses the r_flash_rx API with blocking functionality to erase and write data flash and code flash. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Symbol file).

Setup and Execution

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

Boards Supported

RSKRX66T

4.12 flash_demo_rskrx72t

This is a simple demo for the RSKRX72T starter kit for e^{2}studio v7.0.0. The demo uses the r_flash_rx API with blocking functionality to erase and write data flash and code flash. Each write function is verified with a read-back of data. Note the “pragma section FRAM” for writing to code flash and the corresponding section definitions in the linker (see project Properties->C/C++ Build ->Settings ->Tool Settings (tab) ->Linker ->Section and ->Symbol file).

Setup and Execution

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

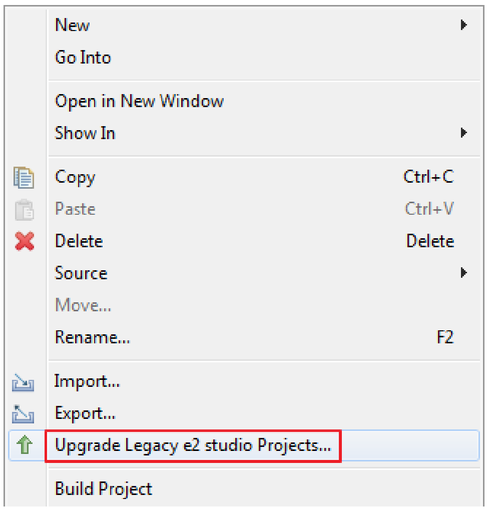
Boards Supported

RSKRX72T

4.13 Adding a Demo to a Workspace

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

If you are using e^{2}studio v6.0.0 or later, you may need to update the project after importing it in order for the project to build properly. This is done by right-clicking on the project folder and selecting “Upgrade Legacy e^{2}studio Projects”.



4.14 Downloading Demo Projects

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

5. Reference Documents

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

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

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

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	July.24.14	—	First edition issued
1.10	Nov.13.14	1, 4 7	Added RX113 support. Updated “ROM to RAM” image.
1.11	Dec.11.14	—	Added RX64M to xml support file.
1.20	Dec.22.14	1, 4	Added RX71M support.
1.30	Aug.28.15	All 5, 10	Updated template. Added RX231 support Added flash type 3 code flash run-from-rom info. Fixed RX64M/71M erase boundary issue.
1.40	Sep.03.15	1, 4	Added RX23T support Fixed Big Endian bug in R_DF_Write_Operation() for Flash Type 1. Fixed FLASH_xF_BLOCK_INVALID values for Flash Type 3.
1.50	Nov.11.15	1, 4	Added RX130 support
1.51	Nov.11.15	—	Repackaged demo with BSP v3.10
1.60	Nov.17.15	1, 5 22, 25	Added RX24T support Added ROM cache support Fixed incorrect FLASH_CF_BLOCK_INVALID for RX210/21A/62N/630/63N/63T in code (Flash Type 2).
1.61	May.20.16	10, 11	Added erase/write/blankcheck BGO support for RX64M/71M Fixed lockbit enable/disable commands.
1.62	May.25.16	—	Added lockbit write/read BGO support for RX64M/71M
1.63	Jun.13.16	—	Fixed bug where large flash writes returned success when actually failed (improper timeout handling) on RX64M/71M
1.64	Aug.11.16	—	Fixed RX64M/71M bug where R_FLASH_Control (FLASH_CMD_STATUS_GET, NULL) always returned BUSY. Added #if to exclude ISR code when not in BGO mode.
1.70	Aug.11.16	1, 4-6, 8 —	Added RX651/RX65N support (Flash Type 4) Fixed bug in Flash Type 2 that caused erroneous blankcheck results.
2.00	Aug.17.16	1, 3, 4, 6-9	Added RX230 and RX24T support (Flash Type 1) Added configuration option for operation without FIT BSP. Inserted document sections 2.12.2 thru 2.12.4. Modified values for FLASH_CF_LOWEST_VALID_BLOCK and FLASH_CF_BLOCK_INVALID for Flash TYPE 1.
2.10	Dec.20.16	1, 5-7, 11, 13, 17, 19, 21, 23-26, 31-32	Added RX24U and RX24T-512 support (Flash Type 1) Fixed several minor bugs in all flash types and added more parameter checking. See History in r_flash_rx_if.h for complete list of changes.
3.00	Dec.21.16	8, 9	Merged code common to types 1, 3, and 4 and restructured high level code for cleaner operation. Modified ROM/RAM size tables.
3.10	Feb.17.17	5-7, 13-17, 26-28, 35	Added RX65N-2M support. Added sections 2.16 and 2.17.4. Added commands FLASH_CMD_BANK_xxx. Fixed potential “BUSY” return from Flash Type 1 API calls (potential bug with very slow flash). Added clearing of ECC flag during initialization of Flash Type 3.

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

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

Rev.	Date	Description	
		Page	Summary
4.00	Apr.19.19	Program	Changed as a result of supporting the GCC/IAR compiler. Changed as a result of deletion of FLASH_CFG_USE_FIT_BSP. Changed as a result of deletion of FLASH_CFG_FLASH_READY_IPL. Changed as a result of deletion of FLASH_CFG_IGNORE_LOCK_BITS. Deleted flash type 2 device from target device. Deleted FLASH_ERR_ALIGNED. Deleted FLASH_ERR_BOUNDARY. Deleted FLASH_ERR_OVERFLOW. Added the process to output error when BSP is earlier than Rev.5.00.

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/.