RX Family

Flash Module Using Firmware Integration Technology

R01AN2184EU0340
Rev. 3.40
Mar 8, 2018

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 RX600 and the RX200 Series of MCUs (R01AN0544EU).

The source files accompanying the Flash FIT module comply with the Renesas RX compiler only.

Target Device

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

- RX110, RX111, RX113 Groups
- RX130, RX130-512KB Groups
- RX210, RX21A Groups
- RX220 Group
- RX231, RX230 Groups
- RX23T, RX24T Groups
- RX24U Group
- RX610 Group
- RX621, RX62N, RX62T, RX62G Groups
- RX630, RX631, RX63N, RX63T Groups
- RX64M Group
- RX651, RX65N, RX65N-2M Groups
- RX66T Group
- RX71M Group

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

Related Documents

- Firmware Integration Technology User's Manual (R01AN1833EU)
- Board Support Package Firmware Integration Technology Module (R01AN1685EU)
- Adding Firmware Integration Technology Modules to Projects (R01AN1723EU)
- Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826EJ)

Contents

1. Overview 4
1.1 Features 4
1.2 Optional BSP 4

2. API Information 5
2.1 Hardware Requirements 5
2.2 Software Requirements 5
2.3 Limitations 5
2.4 Supported Toolchains 5
2.5 Header Files 5
2.6 Integer Types 5
2.7 Flash Types and Features 5
2.8 Configuration Overview 6
2.9 Code Size 8
2.10 API Data Types 9
2.11 Return Values 9
2.12 Adding the FIT Flash Module to Your Project 10
2.12.1 Adding source tree and project include paths 10
2.12.2 Setting driver and FIT BSP use options 10
2.12.3 Project generated files (no FIT BSP) 11
2.12.4 Migrating from version 1.x to version 2.x 11
2.12.5 Migrating from version 2.x to version 3.20 11
2.13 Programming Code Flash from RAM 12
2.14 Programming Code Flash from ROM 14
2.15 Operations in BGO Mode 14
2.16 Dual Bank Operation 16
2.17 Usage Notes 18
2.17.1 Data Flash Operations in BGO Mode 18
2.17.2 ROM Operations in BGO Mode 18
2.17.3 ROM Operations and General Interrupts 19
2.17.4 Emulator Debug Configuration 19

3. API Functions 20
3.1 Summary 20
3.2 R_FLASH_Open 21
3.3 R_FLASH_Close 22
3.4 R_FLASH_Erase 23
3.5 R_FLASH_BlankCheck 25
3.6 R_FLASH_Write 27
3.7 R_FLASH_Control 29
3.8 R_FLASH_GetVersion 36

4. Demo Projects 37
4.1 flash_demo_rskrx113 37
4.2 flash_demo_rskrx231 37
4.3 flash_demo_rskrx23T 37
4.4 flash_demo_rskrx130 38
4.5 flash_demo_rskrx24T 38
4.6 flash_demo_rskrx65N 38
4.7 flash_demo_rskrx24U 38
4.8 flash_demo_rx65n2mb_bank1_bootapp / _bank0_otherapp 39
4.9 flash_demo_rdkrx63n 39
4.10 flash_demo_rskrx64m 39
4.11 flash_demo_rskrx64m_romrun 39
4.12 flash_demo_rskrx66T 40
4.13 Adding a Demo to a Workspace 40

Website and Support 41

Revision Record 42

General Precautions in the Handling of MPU/MCU Products 44

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

1.2 Optional BSP

As of v2.00, this driver may be built with or without the BSP. When not using the BSP, flash dependent settings such as clock speed and memory sizes normally set in r_bsp_config.h are set in r_mcu_config.h instead

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) v3.91.

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 v2.07.00

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 API.

Build-time configuration options are selected or defined in the file "r_flash_rx_config.h”.

When building without the BSP, additional configuration options are selected or defined in the file "r_mcu_config.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 Flash Types and Features

The flash driver is divided into four 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 2
RX210, RX220, RX21A
RX610,

RX62G, RX62N, RX62T
RX630, RX631, RX63N, RX63T

FLASH TYPE 3
RX64M, RX66T, RX71M

FLASH TYPE 4
RX651*, RX65N**
**has no data flash, ***no data flash on parts with less than 1.5M 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. Some of these features/#defines include:

#define FLASH_HAS_ISR_CALLBACK_CMD

#define FLASH_NO_BLANK_CHECK

#define FLASH_HAS_CF_BLANK_CHECK

#define FLASH_ERASE_ASCENDING_BLOCK_NUMS

#define FLASH_ERASE_ASCENDING_ADDRESSES

#define FLASH_HAS_ROM_CACHE

#define FLASH_HAS_DIFF_CF_BLOCK_SIZES

#define FLASH_HAS_BOOT_SWAP

#define FLASH_HAS_APP_SWAP

#define FLASH_HAS_CF_ACCESS_WINDOW

#define FLASH_HAS_DF_ACCESS_WINDOW

#define FLASH_HAS_INDIVIDUAL_CF_BLOCK_LOCKS

#define FLASH_HAS_SEQUENTIAL_CF_BLOCKS_LOCK

#define FLASH_HAS_ERR_ISR

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.

Configuration options in r_flash_rx_config.h		
Equate	**Default Value**	**Description**
FLASH_CFG_USE_FIT_BSP	1	Setting to 1 builds driver with constants from r_bsp_config.h Setting to 0 build driver with constants from r_mcu_config.h
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 57.16) under certain restrictions. See section 2.13 for details on how to set up code and the linker to execute code from RAM. See section 2.15 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.
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 2.17 Usage Notes.
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 2.14).
FLASH_CFG_FLASH_READY_IPL	5	For FLASH_TYPE_2. This defines the interrupt priority level for that interrupt
FLASH_CFG_IGNORE_LOCK_BITS	1	For FLASH_TYPE_2. This applies only to ROM as Data Flash does not support lock bits. Each erasure block has a corresponding lock bit that can be used to protect that block from being programmed/erased after the lock bit is set. Setting this to 1 causes lock bits to be ignored and programs/erases to a block will not be limited. Setting this to 0 will cause lock bits to be used as the user configures through the Control command.

Table 1: Flash general configuration settings

Configuration options in r_mcu_config.h

Equate	Default Value	Description
MCU_CFG_ICLK_HZ	(FIT BSP default)	Set value to MCU ICLK speed (e.g. 80000000 for 80Mhz)
MCU_CFG_FCLK_HZ	(FIT BSP default)	Set value to MCU flash clock speed (e.g. 20000000 for 20Mhz)
MCU_CFG_PART_MEMORY_SIZE	(FIT BSP default)	Set value (0x0 – 0xF) to memory size designation found in part number. The possible values are also found just below this equate in the r_mcu_config.h file.

Table 2: Configuration settings when FIT BSP is not used

2.9 Code Size

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.

Flash Type 1 ROM and RAM usage	
ROM usage: **PARAM_CHECKING_ENABLE 1 > PARAM_CHECKING_ENABLE 0** **DATA_FLASH_BGO 1 > DATA_FLASH_BGO 0** **CODE_FLASH_ENABLE 1 > CODE_FLASH_ENABLE 0** **CODE_FLASH_BGO 1 > CODE_FLASH_BGO 0**	
Minimum Size	ROM: 2198 bytes (2098 if no DF)
	RAM: 84 bytes (84+1944=2028 if no DF)
Maximum Size	ROM: 3386 bytes (2567 if no DF)
	RAM: 84 + 2690 = 2774 bytes (84+2384=2468 if no DF)

Flash Type 2 ROM and RAM usage	
ROM usage: **PARAM_CHECKING_ENABLE 1 > PARAM_CHECKING_ENABLE 0** **DATA_FLASH_BGO 1 > DATA_FLASH_BGO 0** **CODE_FLASH_ENABLE 1 > CODE_FLASH_ENABLE 0** **CODE_FLASH_BGO 1 > CODE_FLASH_BGO 0** **IGNORE_LOCK_BITS 0 > IGNORE_LOCK_BITS 1**	
Minimum Size	ROM: 2179 bytes
	RAM: 32 bytes
Maximum Size	ROM: 2953 bytes

	RAM: 44 + 2626 = 2670 bytes

Flash Type 3 ROM and RAM usage	
ROM usage: **PARAM_CHECKING_ENABLE 1 > PARAM_CHECKING_ENABLE 0** **CODE/DATA_FLASH_BGO 1 > CODE/DATA_FLASH_BGO 0** **CODE_FLASH_ENABLE 1 > CODE_FLASH_ENABLE 0**	
Minimum Size	ROM: 1797 bytes
	RAM: 64 bytes
Maximum Size	ROM: 3262 bytes
	RAM: 64 + 2900 = 2964 bytes

Flash Type 4 ROM and RAM usage	
ROM usage: **PARAM_CHECKING_ENABLE 1 > PARAM_CHECKING_ENABLE 0** **CODE/DATA_FLASH_BGO 1 > CODE/DATA_FLASH_BGO 0**	
Minimum Size	ROM: 1967 bytes
	RAM: 64 + 1707 = 1771 bytes
Maximum Size	ROM: 2656 bytes
	RAM: 64 + 2368 = 2432 bytes

2.10 API Data Types

The API data structures are located in the file “r_flash_rx_if.h” and discussed 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 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, 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_ALIGNED,      /* Type2 - The address that was supplied was not
                           on aligned correctly for code flash or data flash */
FLASH_ERR_BOUNDARY,     /* Type2 - Writes cannot cross the 1MB boundary
```

```
                          on some parts */
FLASH_ERR_OVERFLOW,    /* Type2 - 'Address + number of bytes' for this
                          operation went past the end of this memory area. */
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

For detailed explanation of how to add a FIT Module to your project, see document R01AN1723EU “Adding FIT Modules to Projects”.

2.12.1 Adding source tree and project include paths

In general, a FIT Module may be added in 3 ways:

1. Using an e2studio FIT tool, such as File>New>Renesas FIT Module (prior to v5.3.0), Renesas Views->e2 solutions toolkit->FIT Configurator (v5.3.0 or later), or projects created using the Smart Configurator (v5.3.0 or later). This adds the module and project include paths.
2. Using e2studio File>Import>General>Archive File from the project context menu.
3. Unzipping the .zip file into the project directory directly from Windows.

When using methods 2or 3, the include paths must be manually added to the project. This is done in e2studio from the project context menu by selecting Properties>C/C++ Build>Settings and selecting Compiler>Source in the ToolSettings tab. The green “+” sign in the box to the right is used to pop a dialog box to add the include paths. In that box, click on the Workspace button and select the directories needed from the project tree structure displayed. The directories needed for this module are:

- ${workspace_loc:/${ProjName}/r_flash_rx
- ${workspace_loc:/${ProjName}/r_flash_rx/src
- ${workspace_loc:/${ProjName}/r_flash_rx/src/targets
- ${workspace_loc:/${ProjName}/r_flash_rx/src/flash_type_1
- ${workspace_loc:/${ProjName}/r_flash_rx/src/flash_type_2
- ${workspace_loc:/${ProjName}/r_flash_rx/src/flash_type_3
- ${workspace_loc:/${ProjName}/r_flash_rx/src/flash_type_4
- ${workspace_loc:/${ProjName}/r_config

2.12.2 Setting driver and FIT BSP use options

The flash-specific options are found and edited in \r_config\r_flash_rx_config.h.

A reference copy (not for editing) containing the default values for this file is stored in \r_flash_rx\ref\r_flash_rx_config_reference.h.

If you are building your application with the FIT BSP, nothing else is required.

If you are building your application without using the FIT BSP, the user must:

COPY \r_flash_rx\src\targets\<mcu>\r_mcu_config_reference.h

TO \r_config\r_mcu_config_reference.h, then

RENAME to \r_config\r_mcu_config.h

Next, set FLASH_CFG_USE_FIT_BSP to 0 in r_flash_rx_config.h. Additionally, change the clock speeds and memory size value in r_mcu_config.h if needed.

Any application file which calls an API function should include the interface file “r_flash_rx_if.h” (which in turn includes “r_flash_rx_config.h” and indirectly “r_mcu_config.h” if needed). This file contains the API function declarations and all structures and enumerations necessary to use the module.

2.12.3 Project generated files (no FIT BSP)

If you are using the project generator, and your application will use flash interrupts, you will need to comment out the generated ISR templates. Specifically, comment out:

src\vect.h:

```
#pragma interrupt (Excep_FCUIF_FRDYI(vect=23))
#pragma interrupt (Excep_FCUIF_FIFERR(vect=21))
```

src\interrupt_handlers.c

```
void Excep_FCUIF_FRDYI(void){ }
void Excep_FCUIF_FIFERR(void){ }
```

2.12.4 Migrating from version 1.x to version 2.x

To migrate from version 1.x to 2.x when using the FIT BSP, after installing the new source tree, a single file must be copied and renamed:

COPY \r_flash_rx\src\targets\<mcu>\r_mcu_config_reference.h

TO \r_config\r_mcu_config_reference.h, then

RENAME to \r_config\r_mcu_config.h

No other action is required.

2.12.5 Migrating from version 2.x to version 3.20

To migrate from version 2.x to 3.20 when using the FIT BSP, after installing the new source tree, the file “r_config\r_mcu_config.h” should be removed.

To migrate from version 2.x to 3.20 when *not* using the FIT BSP, after installing the new source tree, the “r_mcu_config_reference.h” file still needs to be copied and renamed:

COPY \r_flash_rx\src\targets\<mcu>\r_mcu_config_reference.h

TO \r_config\r_mcu_config_reference.h, then

RENAME to \r_config\r_mcu_config.h

This will overwrite the previous “r_mcu_config.h” file. No other action is required.

2.13 Programming Code Flash from RAM

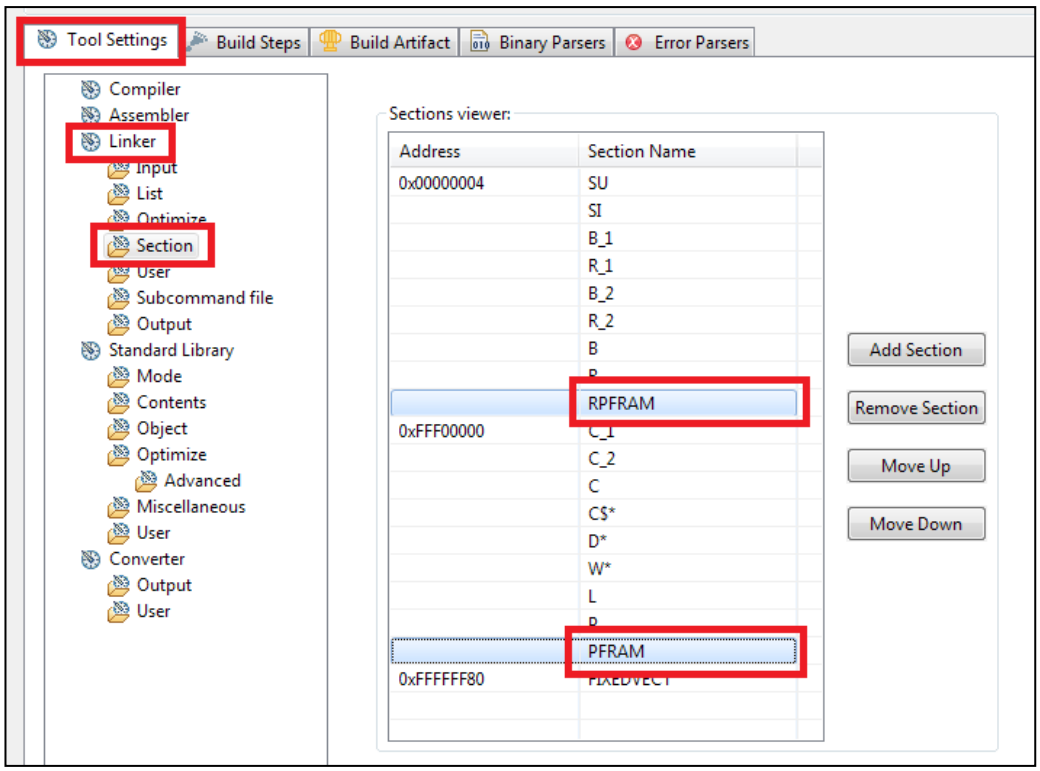
MCUs require that sections in RAM and ROM be created to hold the API functions for reprogramming ROM. This is required because the sequencer (with some exceptions in Type 3) cannot program or erase ROM while executing from ROM. The RAM section will need to be initialized after reset.

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. Please follow the steps below when programming or erasing ROM:

Example when configuring in e2studio:

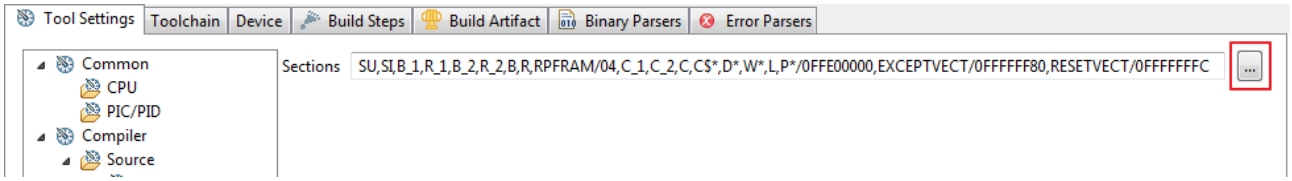
The process of setting up the linker sections and mapping ROM to RAM needs to be done in e2 studio as listed below.

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

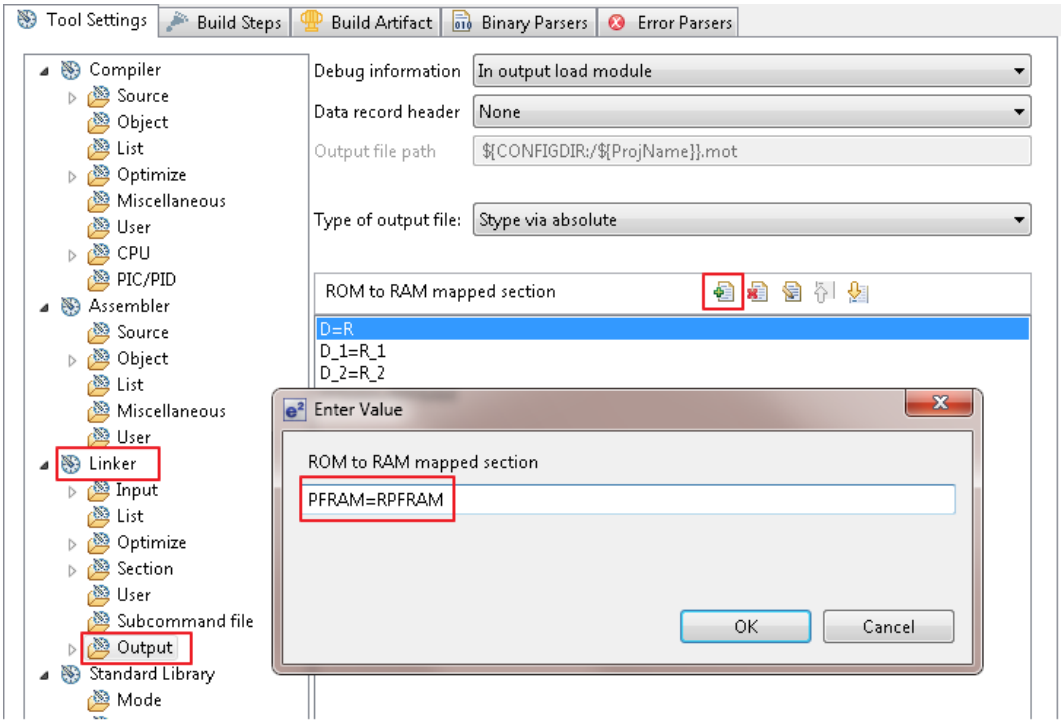


NOTE: Depending upon the e2studio version you are using, there will be a section called P or P*. If it is P*, then there is no need to specify a separate PFRAM section.

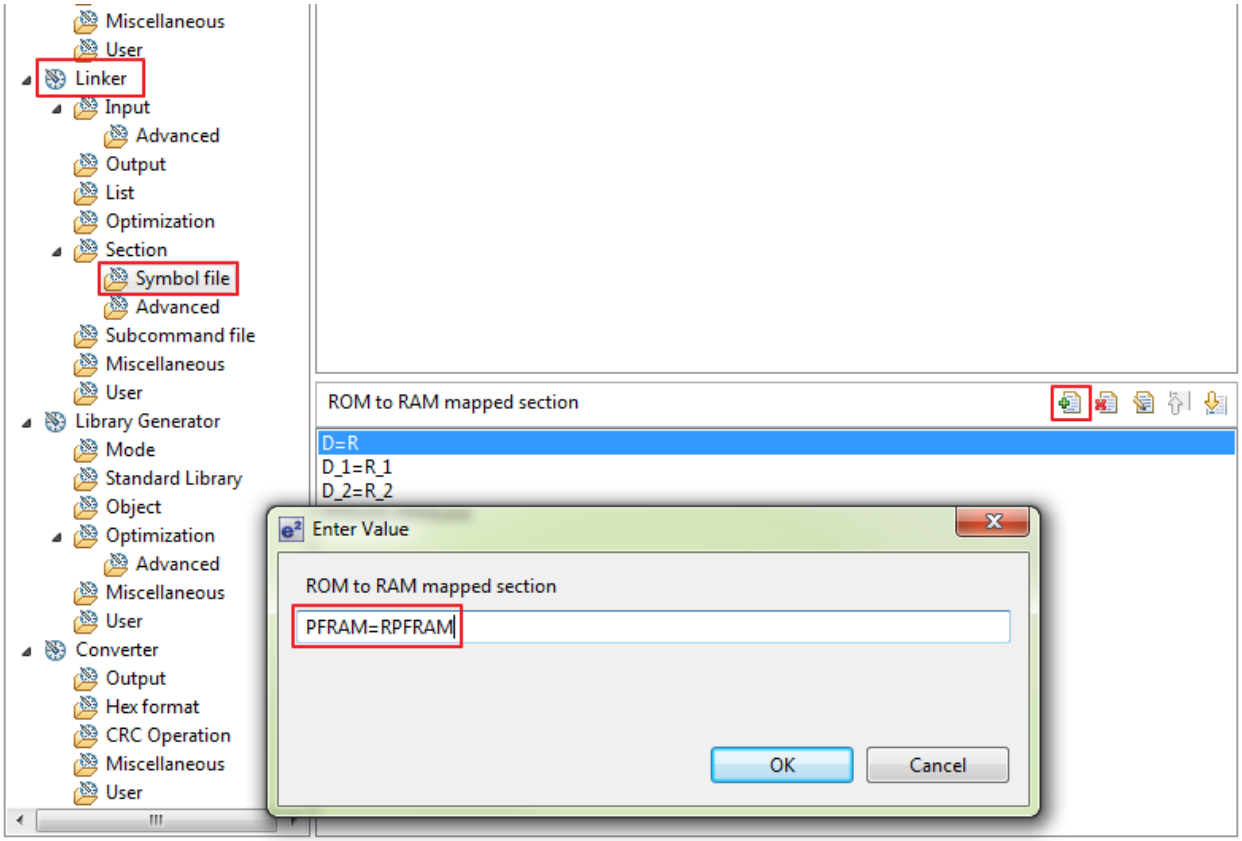
If you are using e2studio v6.0.0 or later, the table does not appear by default. To make the table appear, click the "…" button to the right of the "Sections" entry.



3. Add the linker option to map the ROM section (PFRAM) address to the RAM section address (RPFRAM) by adding 'PFRAM=RPFRAM' to the linker Output options as seen below. This is done using the Linker -> Output section of the Tool Settings in e2Studio prior to v6.0.0.



If you are using e2studio v6.0.0 or later, add 'PFRAM=RPFRAM' via Linker -> Section -> Symbol file.



4. The linker is now setup to correctly allocate the appropriate API code to RAM. The operation to copy code from ROM to RAM is done automatically upon calling the R_FLASH_Open() function. If this is not done before the API functions are called, then the MCU will jump to uninitialized RAM.
5. The interrupt callback functions and the code which operate on ROM should be enclosed within the FRAM section.

```
#pragma section FRAM

/* functions to operate on ROM (and interrupt callbacks) goes here */

#pragma
```

2.14 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 2.13, but do guarantee that the region the code is running from is not the region being operated on!

2.15 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 (non-Flash Type 2), 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;
```

For Flash Type 2 MCUs, instead of passing an event to a single callback function, the driver provides predefined callback functions for each event category (backwards compatible with old Simple Flash API):

- FlashEraseDone(void)
- FlashBlankCheckDone(result)
- FlashWriteDone(void)
- FlashError(void)

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 2.14).

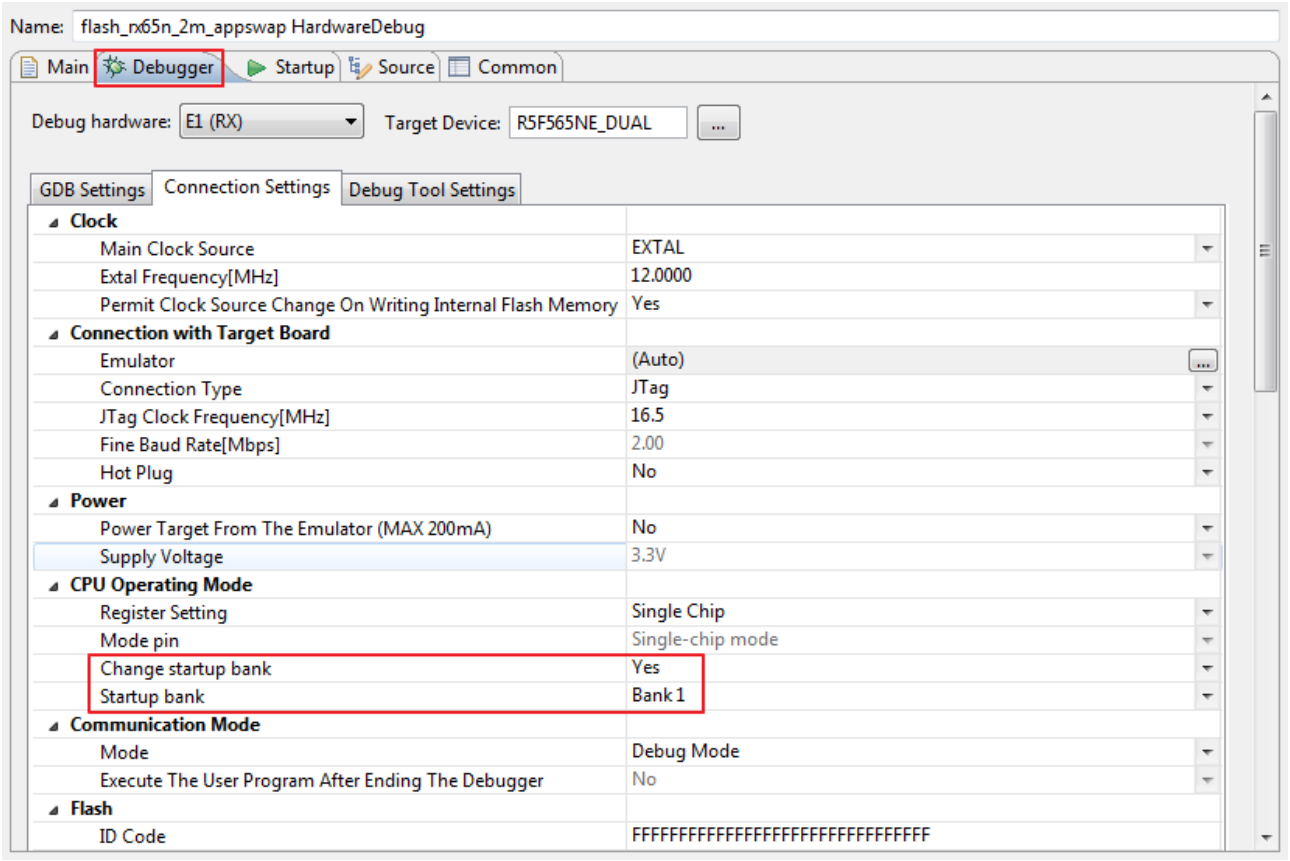
2.16 Dual Bank Operation

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:

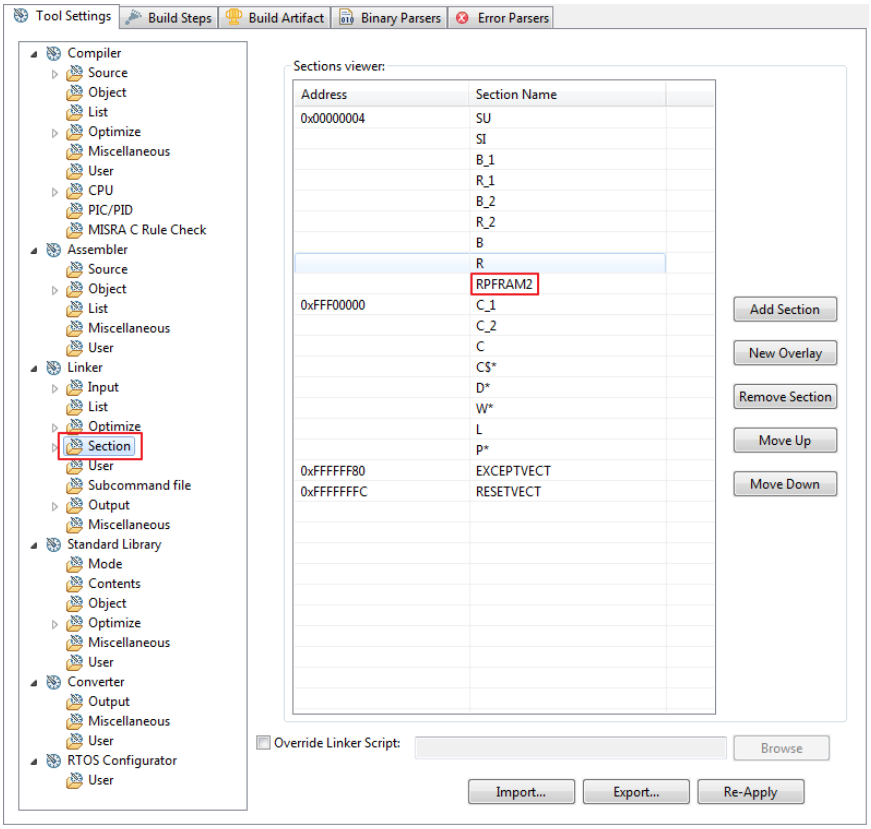
```
BSP_CFG_CODE_FLASH_BANK_MODE     0     // set to 0 for dual mode (not default)
BSP_CFG_CODE_FLASH_START_BANK    1     // different value for each application
```

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.

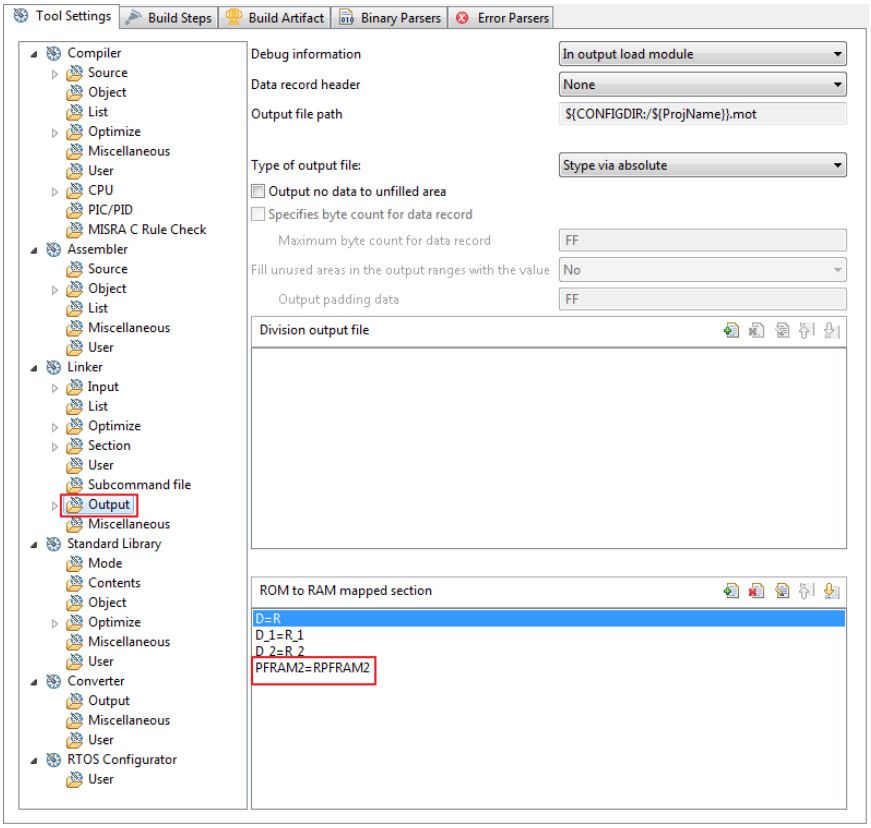


In the above example, the application built with BSP_CFG_CODE_FLASH_START_BANK set to 1 will be the application that is executed at reset.

As mentioned in Section 2.14, 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:

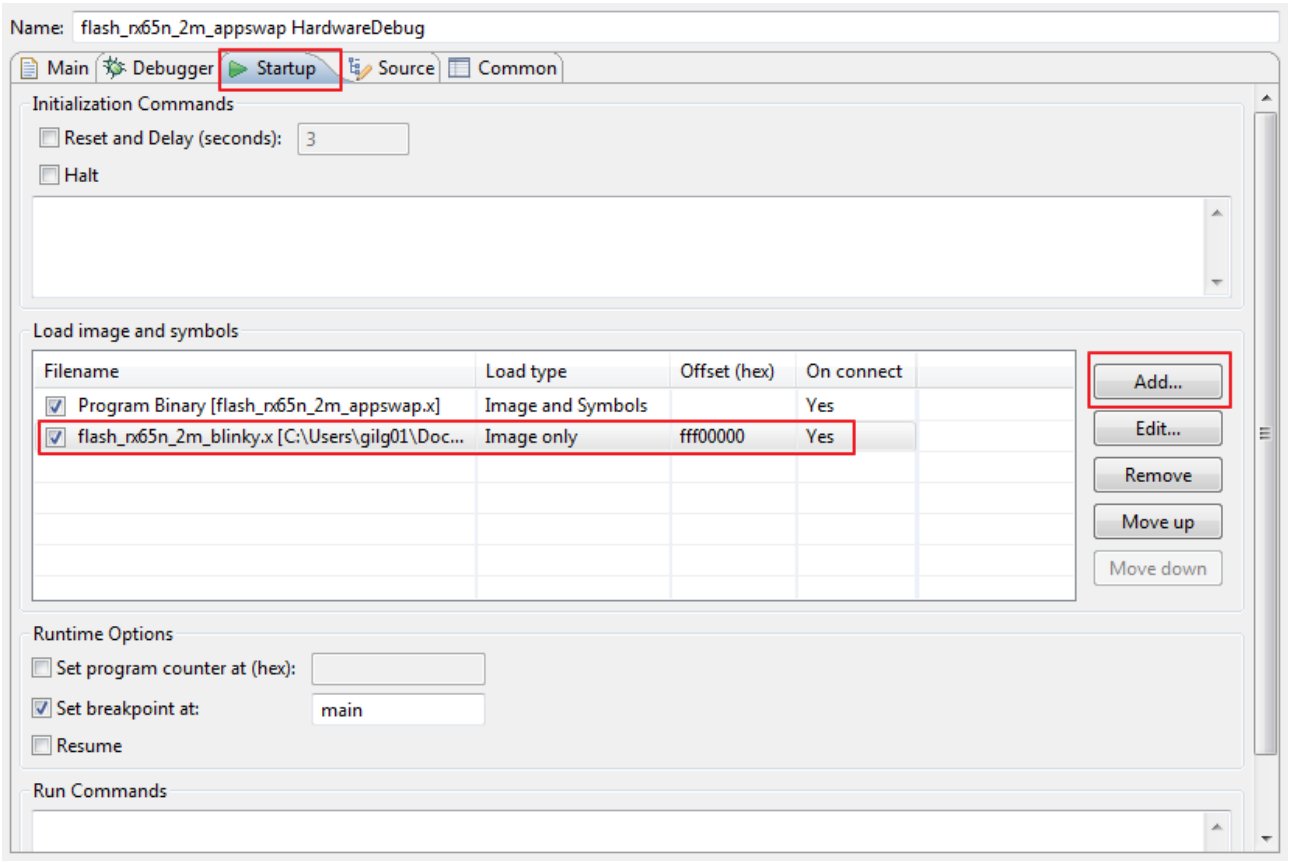


Note: For e2studio v6.0.0 or later, a "…" button appears at the top right which must be pressed to display the section table as shown in this earlier release.



Note: For e2studio 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.



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, e2studio 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, e2studio 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.

2.17 Usage Notes

2.17.1 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.

2.17.2 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 (R01AN2191EJ).

2.17.3 ROM Operations and General Interrupts

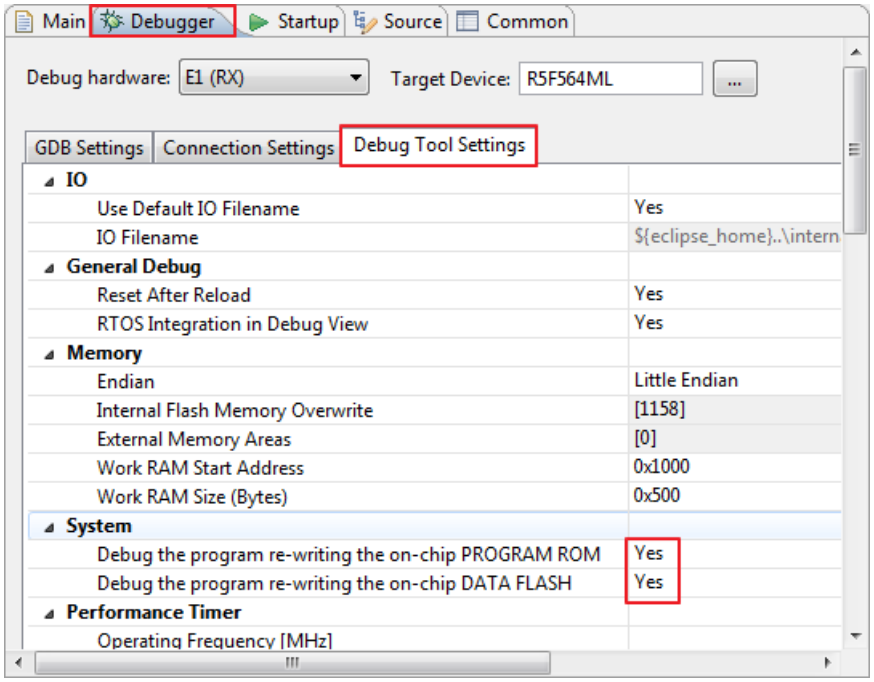
ROM or data flash areas cannot be accessed while a flash operation is on-going for that particular memory area. This means that the relocatable vector table will need to be taken care of when allowing interrupts to occur during flash operations.

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.

2.17.4 Emulator Debug Configuration

As a safety feature, the emulator prohibits modification of code and data flash by the application. To use this driver, writing must be enabled. This is accomplished by changing the Debug Tool Settings in the debug configuration as follows:



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

```
uint8_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.

For flash type 2, *num_bytes* must be either BLANK_CHECK_SMALLEST or BLANK_CHECK_ENTIRE_BLOCK. These values are used to be compatible with legacy Simple Flash API code. BLANK_CHECK_SMALLEST denotes that FLASH_DF_MIN_PGM_SIZE will be checked.

**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. */
        . . .
    }
}
```

Example: Flash Type 2

Second argument must be BLANK_CHECK_SMALLEST (checks FLASH_DF_MIN_PGM_SIZE bytes) or BLANK_CHECK_ENTIRE_BLOCK.

```
flash_err_t err;
flash_res_t result;

/* Blank check all of data flash block 0 */
err = R_FLASH_BlankCheck((uint32_t)FLASH_DF_BLOCK_0,
                         BLANKCHECK_ENTIRE_BLOCK, &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 1,2,3,4 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 1,2,3,4 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 1,2,4 FLASH_CMD_ACCESSWINDOW_SET	flash_access_window_config_t * (different structure for flash types)	Specifies the access window boundaries for ROM (types 1,4) or the data flash read/write block enable masks (type 2). 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).
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 2 FLASH_CMD_LOCKBIT_PROTECTION	flash_lockbit_enable_t *	Setting argument to “false” allows erasing/writing of blocks with lockbit set. Setting argument to “true” prohibits erasing/writing of blocks with lockbit set. NOTE: Erasing a block clears the lockbit.
Flash type 2 FLASH_CMD_LOCKBIT_PROGRAM	flash_program_lockbit_config_t *	Sets lockbit for block whose address is provided as argument.
Flash type 2, 3 FLASH_CMD_LOCKBIT_READ	2) flash_read_lockbit_config_t * 3) flash_lockbit_config_t *	Type 2: Loads argument with FLASH_LOCKBIT_SET or FLASH_LOCKBIT_NOT_SET for block address provided. Type 3: Loads argument with FLASH_RES_LOCKBIT_STATE_PROTECTED or FLASH_RES_LOCKBIT_STATE_NON_PROTECTED for block address provided. Type 3 uses callback function in BGO/non-blocking mode.
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 FLASH_CMD_ROM_CACHE_ENABLE	NULL	Enables caching of ROM (invalidates cache first).
RX24T, RX24U, RX65x FLASH_CMD_ROM_CACHE_DISABLE	NULL	Disables caching of ROM. Call before rewriting ROM.
RX24T, RX24U, RX65x FLASH_CMD_ROM_CACHE_STATUS	uint8_t *	Sets the value to 1 if caching is enabled; 0 if disabled.
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).

**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: Set access window data flash (flash type 2)

The area protection is used to prevent unauthorized reading, programming or erasure of data flash blocks. The following example allows only blocks 64-127 to be read/writeable on an RX63N.

```
flash_err_t                  err;
flash_access_window_config_t df_access;

/* Map of access window bits to data flash blocks
 *      RX62*   RX63*       RX21*
 * b0   0       0-63        0-15
 * b1   1       64-127      16-31
 * b2   2       128-191     32-47
 * b3   3       192-255     48-63
 * b4   4       256-319
 * b5   5       320-383
 * b6   6       384-447
 * b7   7       448-511
 * b8   8       512-575
 * b9   9       576-639
 * b10  10      640-703
 * b11  11      704-767
 * b12  12      768-831
 * b13  13      832-895
 * b14  14      896-959
 * b15  15      960-1023
 */
/* Allow reads and writes to Data Flash blocks 64-127 on an RX63N */

df_access.read_en_mask = 0x0002;
df_access.write_en_mask = 0x0002;
err = R_FLASH_Control(FLASH_CMD_ACCESSWINDOW_SET, (void *)&df_access);
if (FLASH_SUCCESS != err)
{
    printf("Control FLASH_CMD_ACCESSWINDOW_SET command failure.");
}
```

Example 6: 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 7: 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. After the area has been swapped, reset the MCU without returning to ROM.

```
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 8: 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 9: 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 10: 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);
```

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 API. */
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 API 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 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_rdkrx63n

This is a simple demo for the RDKRX63N demonstration 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

RDKRX63N

4.10 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 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

4.11 flash_demo_rskrx64m_romrun

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 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

4.12 flash_demo_rskrx66T

This is a simple demo for the RSKRX66T starter kit for e2studio v6.2.0. 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 ->Symbol file).

Setup and Execution

1. Compile and download the sample code.

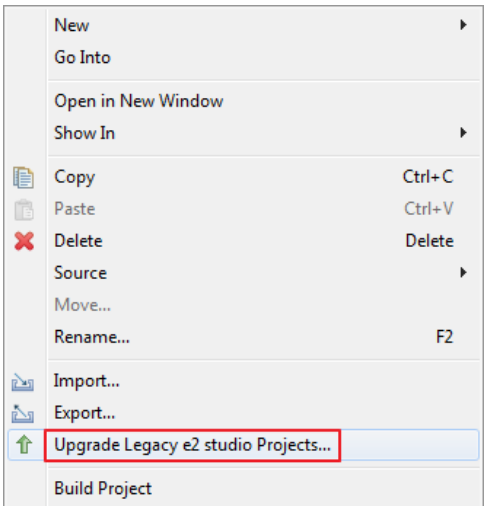
2. Click 'Reset Go' to start the software. If the program stops at main(), press F8 to resume.

Boards Supported

RSKRX66T

4.13 Adding a Demo to a Workspace

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 and Browse to the .zip file for the demo. If you are using e2studio 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 e2studio Projects".



Website and Support

Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/inquiry

Revision Record

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.

3.20	Aug.11.17	1, 5, 10-14, 16, 36	Added RX130-512KB support. Added e2studio 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.

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU 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. 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 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. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment 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 moment 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 moment 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 moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has 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. Moreover, 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.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of an MPU or MCU in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

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

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

SALES OFFICES

Renesas Electronics Corporation

http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
1001 Murphy Ranch Road, Milpitas, CA 95035, U.S.A.
Tel: +1-408-432-8888, Fax: +1-408-434-5351

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700, Fax: +44-1628-651-804

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709 Quantum Plaza, No.27 ZhichunLu, Haidian District, Beijing, 100191 P. R. China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, 200333 P. R. China
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL 2nd Stage, Indiranagar, Bangalore 560 038, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
17F, KAMCO Yangjae Tower, 262, Gangnam-daero, Gangnam-gu, Seoul, 06265 Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5338

Colophon 7.0