

# RX200 Series

R01AN0823EU0100

Rev.1.00

## Simple Flash API for RX200

December 12, 2011

### Introduction

A simple Application Program Interface (API) has been created to allow users of flash based RX200 Series devices to easily integrate reprogramming abilities into their applications using User Mode programming. User Mode programming is the term used to describe a Renesas MCU's ability to reprogram its own internal flash memory while running in its normal operational mode. This application note focuses on using that API and integrating it with your application program.

The API source files comply with the Renesas RX compiler only.

### Target Device

The following is a list of devices able to use this API:

- **RX210 Group**

### Contents

1. Overview .....	2
2. Configuring the API .....	3
3. Usage Notes.....	5
4. Boot Loader Implementations .....	10
5. API Functions.....	12
6. Example Code.....	21

## 1. Overview

The Simple Flash API 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 API in its simplest form can be used to perform blocking erase and program operations. The term ‘blocking’ means that when a program or erase function is called, the function does not return until the operation has finished. When a flash operation is on-going, that flash area cannot be accessed by the user. If an attempt to access the flash area is made, the flash control unit will transition into an error state. For this reason ‘blocking’ operations are preferred by some users to prevent the possibility of a flash error. But there are other cases where blocking operations are not desired. If the user is writing data to the data flash for example, the ROM can still be read. In this case many users would like for the data flash write or erase to occur in the background (non-blocking) while their application continues to run in ROM. RX200 MCUs support this feature and it is available in the Simple Flash API. The user can also perform non-blocking ROM operations as well, but application code will need to be outside of ROM.

### 1.1 Features

Below is a list of the features supported by the Simple Flash API.

- Blocking erasing and programming of User ROM
- Non-blocking, background operation, erasing and programming of User ROM
- Blocking erasing, programming, and blank checking of data flash
- Non-blocking, background operation, erasing, programming, and blank checking of data flash
- Callback functions for when flash operation has finished (only with non-blocking)
- ROM to ROM transfers
- Data flash to data flash transfers
- Lock bit protection
- Lock bit set/read

### 1.2 Example

An example project that goes through all of these features is included with this application note. *FlashAPI\_Sample\_RX200.c* is the file that contains the *main()* function and the demo code.

### 1.3 API Files

The API files needed for MCUs belonging to the RX200 Series are shown in the table below.

Group	API Files
RX210	r_Flash_API_RX200.h r_Flash_API_RX200.c r_Flash_API_RX200_UserConfig.h

## 2. Configuring the API

Before using the API, you must first configure the code. All general settings are done by modifying *#define* statements in the `r_Flash_API_RX200_UserConfig.h` file. Some of the parameters configure the code for which MCU you are using and others configure the behavior of the Flash API.

### 2.1 Specifying your system and peripheral clock speeds

The layout and block sizes of the flash and data flash memories differ depending on which RX200 device is being used. In order for the Flash API code to run correctly the user needs to define which device they are targeting. Since the current release of the RX200 Flash API only has support for the RX210 device, it is the only listed option in the file `r_Flash_API_RX200_UserConfig.h`.

The size of the ROM on the MCU should also be chosen. This is done using the `ROM_SIZE` *#define*. For example, to use a 512KB MCU, the user would use the following:

```
//#define ROM_SIZE          2097152      //2MB
//#define ROM_SIZE          1048576      //1MB
#define ROM_SIZE           524288        //512KB
//#define ROM_SIZE          262144       //256KB
```

The actual programming of the device is done using a dedicated sequencer called the Flash Control Unit (FCU). The FCU needs to know the frequency of the clock that is supplied to it so it can set its delays appropriately. The system clock (ICLK) also needs to be known for setting software timeout values. The timeout values enable the code to properly exit in the event that a command fails to complete. Please specify the speeds at which the FCU and system clocks will be running during the time of the flash operation. It is perfectly acceptable to run your clocks at a different speed when not performing an erase/write operation.

The operating speed of the clock supplied to the FCU is identified by modifying the `PCLK_FREQUENCY` parameter in the `r_Flash_API_RX200_UserConfig.h` file. This name was chosen because on RX610 and RX62x devices the peripheral clock was the clock supplied to the FCU, and the Flash API for the RX210 is a derivative of that version. The speed is set in increments of MHz. For example, if the FCU clock will be running at 50MHz at the time you call the Erase and Write flashing functions, then you would specify '50'.

```
#define PCLK_FREQUENCY 50
```

The operating speed of the system clock is identified by modifying the `ICLK_FREQUENCY` parameter in the `r_Flash_API_RX200_UserConfig.h` file. The speed is set in increments of MHz. For example, if your system clock will be running 100MHz at the time you call the Erase and Write flashing functions, then you would specify '100'.

```
#define ICLK_FREQUENCY 100
```

**NOTE:** When programming/erasing the flash memory the ICLK and PCLK frequencies should not be below 8MHz.

## 2.2 Flash API configuration parameters

This section deals with parameters that change the behavior of the Flash API. These settings are valid regardless of the settings from Section 2.1.

Configuration options found in <code>r_Flash_API_RX200_UserConfig.h</code>	
<b>ENABLE_ROM_PROGRAMMING</b>	If defined then ROM programming is enabled and code required for this operation is copied to RAM. If undefined then only data flash operations are available and all code will be located in ROM.
<b>FLASH_TO_FLASH</b>	If defined then ROM to ROM and data flash to data flash operations will be enabled. When enabled the Flash API will require a RAM buffer to hold the data to be programmed. The size of the RAM buffer will be maximum number of bytes between the programming size of the data flash and ROM.
<b>DATA_FLASH_BGO</b>	Enables non-blocking data flash operations. When enabled, data flash operations will occur in the background and API functions will return before the operation has finished. When disabled API functions will not return until the data flash operation has completed.
<b>ROM_BGO</b>	Enables non-blocking ROM operations. When enabled, ROM operations will occur in the background and API functions will return before the operation has finished. When disabled API functions will not return until the ROM operation has completed.
<b>FLASH_READY_IPL</b>	This is the interrupt priority level that will be used for the flash ready interrupt when BGO operations are enabled.
<b>IGNORE_LOCK_BITS</b>	If defined then lock bit protection will be ignored. If undefined then lock bit protection will be used and if a program/erase is attempted on a block with its lock bit set, the operation will fail.

## 2.3 What happened to DATA\_FLASH\_OPERATION\_PIPL AND ROM\_OPERATION\_PIPL?

In v2.0 of the Simple Flash API for RX there were two extra `#define`'s in the user configuration file that are not shown in the table above. These definitions were removed due to a bug that was found in the code. The way the definitions were meant to work was that when a flash operation was called, the API would set the MCU's IPL to a certain level. When the flash operation was finished, the API would set the IPL back to what it was before the flash operation was called. Using this method, the user could easily prevent certain interrupts from occurring during flash operation which could cause a ROM or data flash access violation. The problem occurred when trying to restore the MCU's IPL at the end of a flash operation. If the flash operation was done using BGO then it would finish inside of the flash ready ISR. The IPL could be changed inside of the ISR but since the IPL is restored from the stack when returning from an ISR, the change essentially had no effect. This means that after the flash operation was finished the MCU's IPL was not correctly restored. To fix this, the definitions were removed. This means the user must take extra care to make sure no interrupts occur during flash operations that may cause an access violation.

If the user would like to restore these features, two options are presented here. The first is to have code that alters the IPL value that is stored on the stack when an ISR is taken. This can be tricky since the location on the stack can change depending on how many stack variables are used and how many registers are saved. The other option is to make the flash ready interrupt the fast interrupt. This option is easier to code for and safer since the IPL will always be stored in the backup PSW register. The downside to this approach is that the user loses the ability to use the fast interrupt for another interrupt.

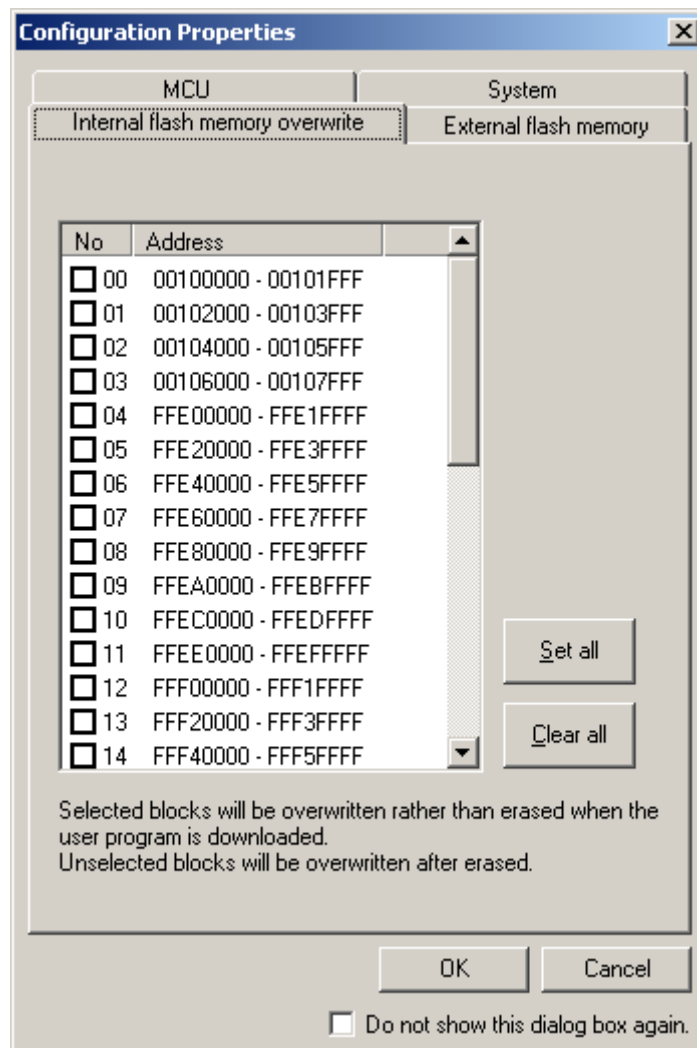
### 3. Usage Notes

#### 3.1 Debugging within HEW

Using the E1 and E20, you are allowed to debug while erasing and programming the on board flash memory and data flash memory. Care should be taken to make sure that the flash block holding the user program is not erased unless the user has some way of programming new code while executing in RAM.

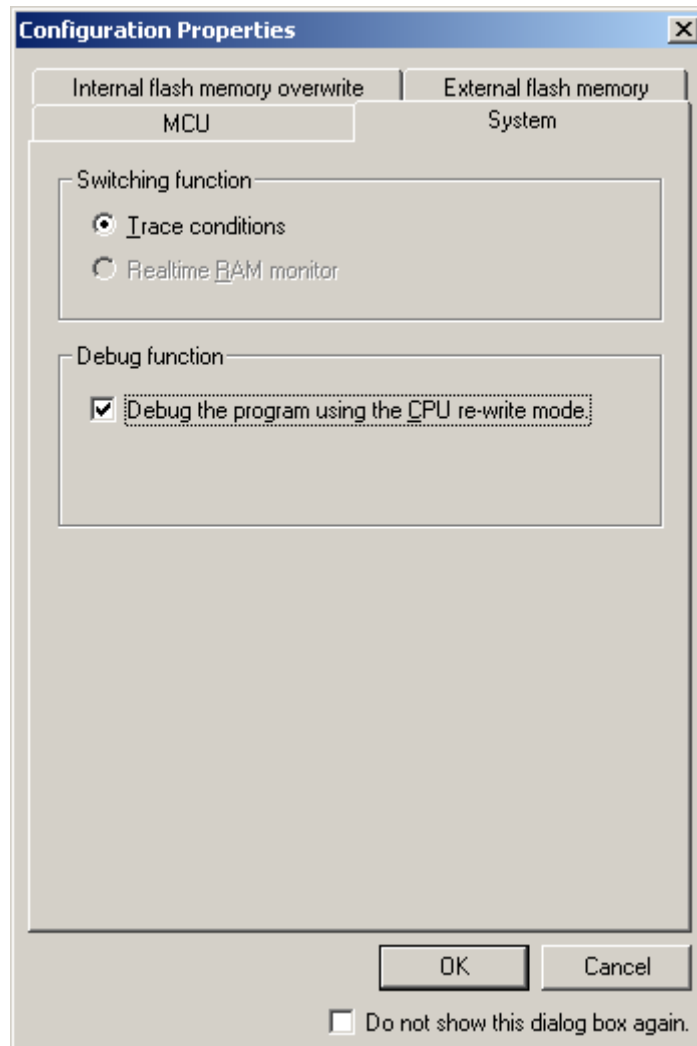
You cannot use the FDT programming software to view previously written data to the flash memory because an RX200 Series device will automatically erase all flash memory when it enters boot mode as a built-in security feature.

If you attempt to disconnect and then re-connect to your system with HEW, the entire flash memory will be erased upon re-connecting with the E1/E20 under default debugger settings. In order to preserve the flash values you will need to specify which flash blocks you want to be overwritten, rather than erased. This is done in the ‘Configuration Properties’ window underneath the ‘Internal flash memory overwrite’ tab. Place a check in the boxes next to the flash blocks you desire to be overwritten instead of being erased. A screenshot of the window is below.



### 3.2 Viewing Programmed/Erased Flash Memory in HEW

Use of the Memory window inside HEW to view the flash memory contents after an erase or write will not work under the default debugger settings. The reason for this is that HEW will cache the flash contents when the debug session starts and will not refresh the values after the program/erase command finishes. There is an option when connecting though that specifies you are using CPU rewrite code and therefore to refresh the flash memory values. This option is in the 'Configuration Properties' window that will come up when connecting to the E1/E20. Switch to the 'System' tab and check the box next to 'Debug the program using the CPU re-write mode'. A screenshot is shown below of the window. Now when using the memory window the current flash memory values will be displayed.

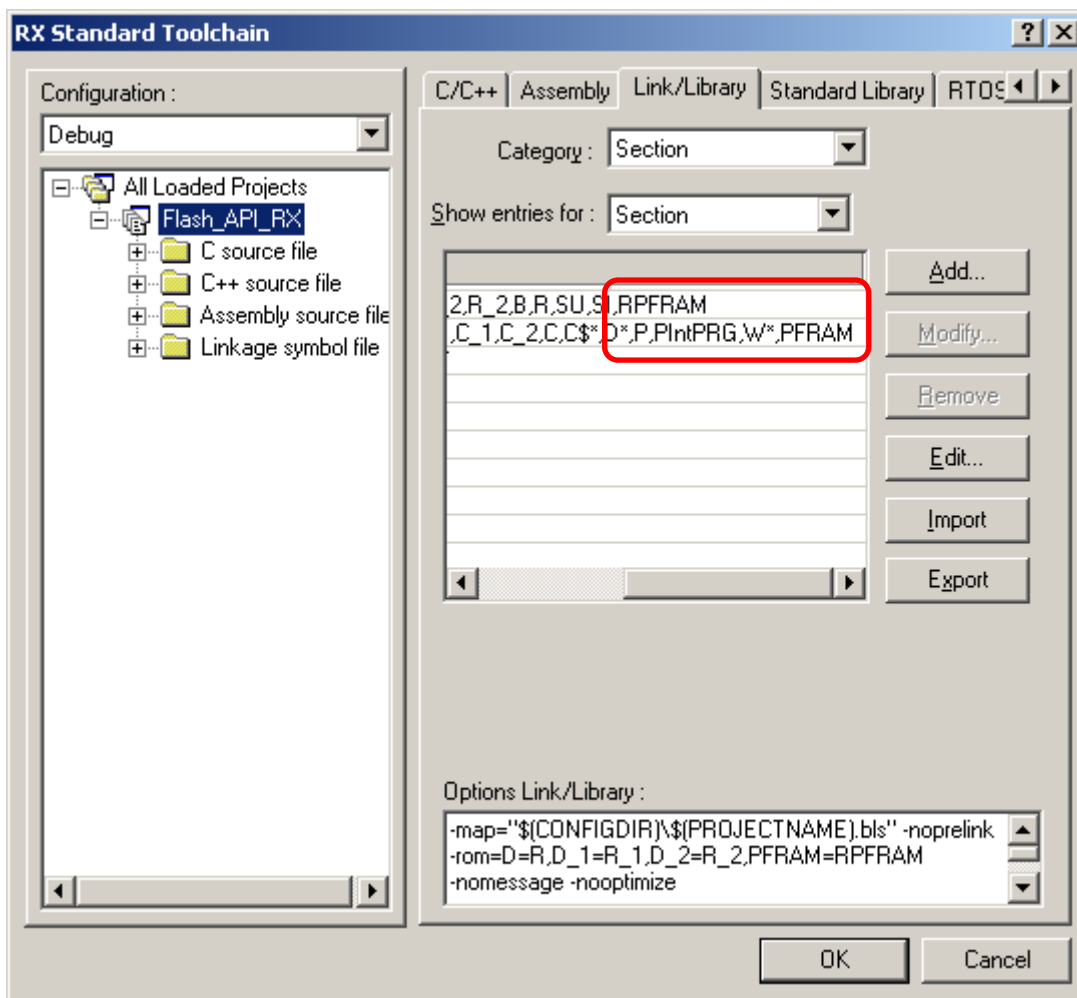


### 3.3 RX200 Series

The flash implementation for this group requires that sections in RAM and Flash be created to hold the API functions for reprogramming ROM. Also, the RAM section will need to be initialized after RESET. Note that this is only for ROM programming. If you are only programming the data flash area, you do not need these settings, but you should change the configuration setting 'ENABLE\_ROM\_PROGRAMMING' to undefined in the file 'Flash\_API\_RX200\_UserConfig.h'. Please follow steps 1-4 below if you are programming or erasing ROM:

**Step 1:** In HEW, add a new section titled 'RPFRAM' in a RAM area.

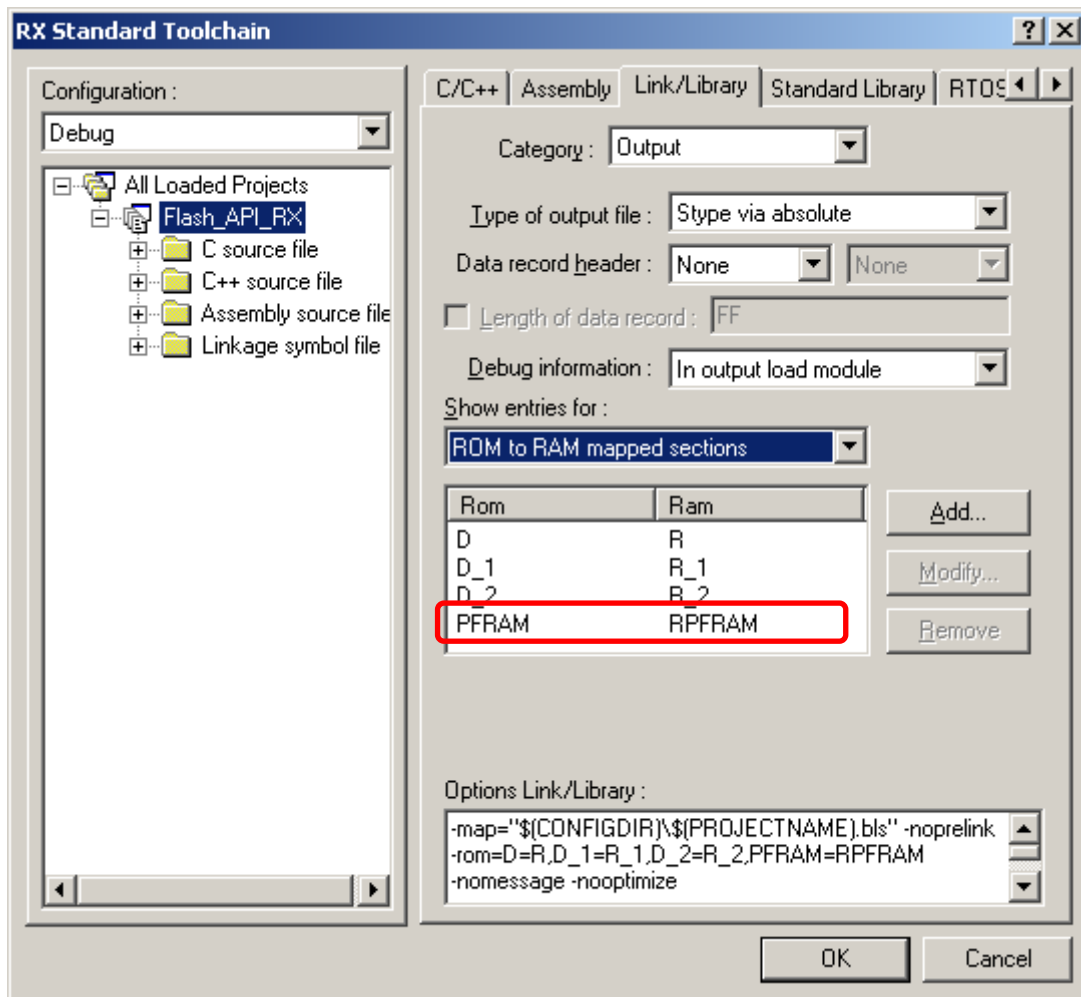
**Step 2:** In HEW, add a new section titled 'PFRAM' in a Flash area.



**Step 3:** In the start file 'dbsct.c', add the initialization of this code for the RAM section as seen below in **RED** (note: don't forget to add the comma on the previous line)

```
-- FILE [dbsct.c] --
#pragma section $DSEC
static const struct {
    _UBYTE *rom_s; /* Initial address on ROM of initialization data section */
    _UBYTE *rom_e; /* Final address on ROM of initialization data section */
    _UBYTE *ram_s; /* Initial address on RAM of initialization data section */
} DTBL[] = {
    { __sectop("D"), __secend("D"), __sectop("R") } ,
    { __sectop("PFRAM"), __secend("PFRAM"), __sectop("RPFRAM") }
};
```

**Step 4:** In HEW, add the linker option to map the ROM section (PFRAM) address to RAM section address (RPFRAM) as seen below.



### 3.4 ROM Area Boundaries

The RX200 Series currently does not have any MCUs that have more than one ROM Area, although there could be devices with multiple ROM areas in future. If you do try to write over a boundary, then the R\_FlashWrite() function will return an error code before performing any write operations stating that this has occurred. In order to write over a boundary, the user will have to take precautions to make sure and split the write up where the first write programs up to the boundary and then the second write starts at the boundary.

Which ROM Area is currently selected for programming and erasure is controlled by the FENTRY bits located in the FENTRYR register. The reason programming cannot go across the boundary is because only one of these bits can be set at a time. Which bit is set is automatically taken care of when the user calls the R\_FlashWrite() function.

### 3.5 Using Non-Blocking Background Operations

When background operations (BGO) for ROM or data flash are enabled, API function calls will not block and will return before the flash operation has finished. The user should take care in these instances that they do not try to access the flash area that is being operated on until the operation has finished. If the area is accessed during an operation then the FCU will go into an error state and the operation will fail.

The user will be alerted when a background flash operation has finished through a callback function. There are 3 callback functions that the Simple Flash API uses when an operation completes. The user should write these functions in their application code. For an example, look in the *FlashAPI\_Sample\_RX200.c* file that is included with this application note. The 3 callback functions are:

- **void FlashEraseDone(void)**
  - This function is called when a data flash or ROM erase has completed
- **void FlashWriteDone(void)**
  - This function is called when a data flash or ROM write has completed
- **void FlashBlankCheckDone(uint8\_t result)**
  - This function is called when a data flash blank check has completed. The 'result' parameter will be 'FLASH\_BLANK' in the event that the block was blank and 'FLASH\_NOT\_BLANK' in the event that the block was not blank.

There is also a callback function in the event that a flash error has occurred.

- **void FlashError(void)**

The Flash API will reset the FCU when an error is detected but this callback is included to alert the user that the flash operation did not complete successfully.

#### 3.5.1 Data Flash BGO Precautions

When using data flash BGO the User ROM, RAM, and external memory can still be accessed. This means that care should only be taken to make sure that the data flash is not accessed during a data flash operation. This includes interrupts that may access the data flash.

#### 3.5.2 ROM BGO Precautions

When using ROM BGO external memory and RAM can still be accessed. Since most users will put their code in ROM, extra care should be taken compared to performing BGO data flash operations. Since the API code will return before the ROM operation has finished the code that calls the API function will need to be outside of the User ROM. Another important issue to be aware of is the relocatable vector table. The vector table by default resides in the User ROM. If an interrupt occurs during the 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 service routines that may occur outside of ROM. The user will also need to change the variable vector table's pointer register (INTB). Examples of this are shown in the example workspace that comes with this application note.

### 3.6 Interrupts

ROM or data flash areas cannot be accessed while a flash operation is on-going for that particular memory area. This means that care will need to be taken when allowing interrupts to occur during flash operations. These precautions apply whether the user is using BGO operations or not.

## 4. Boot Loader Implementations

If you wish to create a boot loader that will have the ability to erase/program the entire memory space of the device, then you will need to either put all of your code in the User Boot flash area or move your entire boot loader application to RAM. You cannot leave your code in the User flash area since you cannot erase a block of flash that you are currently running from.

### 4.1 Moving an entire application to the User Boot area

Some RX200 Series devices have a separate flash area designated as the User Boot area. The MCU can be setup such that it boots into this area instead of using the user application reset vector. This flash area cannot be programmed or erased by a user program running on the MCU. These features make this flash area convenient for holding a boot loader application. Some steps and considerations for this implementation are below.

- Change the linker settings within HEW such that your application code and Interrupt Vector Table are placed in the User Boot area. If these settings are not changed then by default the code and IVT will be placed in the regular User flash area.
- In User Boot mode the reset vector is moved from 0xFFFFF0FC to 0xFF7FFF0C. Therefore make sure for your boot loader application you mark 0xFF7FFF0C as the reset vector. This can be done by doing the following:
  1. Setup a section at 0xFF7FFF0C in the linker settings. We'll call it 'BOOTVECT' for this example.
  2. Create a new C source file, add it your project, and add an array of function pointers. An example of this is shown below where the function we want run after reset is called 'BootLoader'.

```
#pragma section C BOOTVECT

void* const Boot_Vectors[] = {
    //0xFF7FFF0C is reset vector in User Boot Mode
    (void*) BootLoader,
}
```

### 4.2 Moving a bootloader application to RAM

If your RX200 Series device does not have a User Boot flash area, or if you do not wish to use the User Boot area for some other reason, you can move your bootloader application to RAM. Some steps and considerations for this implementation are detailed below.

- Allocate a section in RAM within HEW's linker settings where you want your program to execute from. Make sure when you name the section that the first letter is an 'F', which signifies a 'Fixed' area section. For example, if you want a section called MY\_APP\_RAM, you would name the section named 'FMY\_APP\_RAM' in the linker settings.
- Because your RAM executable code will need a ROM location to be loaded and stored, you must first allocate a section within HEW's linker settings. Make sure when you name the section that the first letter is a 'P' which is required by the toolchain to signify a 'Program' area. For example, if you want a section called MY\_APP, you would name the section named 'PMY\_APP' in the linker settings.
- The RX linker has a special option that will assign RAM memory addresses for functions (and data), but then physically place it in a ROM location. This is done with the assumption that the application program will copy the code or data from the ROM storage location to the RAM execution location before it is referenced. After your code is moved to RAM, all the absolute address references will match your RAM location. Also, whenever any other source module attempts to reference this code, it will be given its RAM address, not its ROM storage address. This is done by using the linker's ROM-to-RAM mapping option "-rom=xxxx=yyyy" where 'xxxx' would be the ROM section you have allocated and 'yyyy' would be the RAM section you have allocated. This can be configured in HEW following the directions shown in Step 4 of Section 3.3. In our example, it would look like:

```
-rom=PMY_APP=FMY_APP_RAM
```

- When it is time to execute your RAM based program, you must first copy the executable binary from its ROM storage location to its RAM executable location. Below is an example of how you could do that. You could also add your sections to the code in 'dbsct.c' file as shown in Step 3 of Section 3.3.

```
unsigned char *src;
unsigned char *dst;

src = (unsigned char *)(__sectop("PMY_APP"));
dst = (unsigned char *)(__sectop("FMY_APP_RAM"));
for( ; src < (unsigned char *)(__secend("PMY_APP")); src++, dst++)
{
    *dst = *src;
}
```

- When writing your code, you will also need to tell the linker which functions should be part of this special ROM section that will be relocated to RAM. To do that, place "#pragma section MY\_APP" before the function. Note that the 'P' before 'MY\_APP' has been removed. This is because the compiler will automatically insert a 'P' at the beginning of each section that is intended to hold executable code. Please note that once the '#pragma section' is used in a source file, all function and data declarations following it will be placed in that section as well until the end of the file unless another '#pragma section' is encountered. If this next '#pragma section' has no section name, then the default sections will be used. You can also specify a section name and it will be used for the code and data after it. For more information, please refer to the RX Toolchain Manual. Below is an example of its usage.

```
#pragma section MY_APP
void function1( void )
{
    {THIS FUNCTION WILL BE PLACED IN 'PMY_APP'}
}
void function2( void )
{
    {THIS FUNCTION WILL BE PLACED IN 'PMY_APP'}
}
```

- One final consideration is to make sure that all reference functions be part of that section that will be relocated to RAM. It will be no good moving and executing your code from RAM if your code still accidentally calls a function in ROM (that you might have already erased). In some cases, compiler optimization may have your code call a common standard library function to increase code efficiency because calling a single library function will use less code than implementing the functionality multiple times throughout the code. An example of this would be doing 32-bit multiply operations in your application code. This sometimes may be tricky to spot unless you are examining the compiler's generated output. Since these libraries will be located in their default ROM based locations (not your special ROM-to-RAM section), your special re-programming code may execute OK for erasing a few blocks, but then after erasing the block with the library function call in it, your application will terminally crash.

## 5. API Functions

### 5.1 R\_FlashErase

This function allows an entire flash block to be erased.

#### Format

```
uint8_t R_FlashErase(uint8_t block);
```

#### Parameters

*block*

Specifies the block to erase. This value is defined in the `r_Flash_API_RX200.h` file. The blocks are labeled in the same fashion as they are in the device's Hardware Manual. For example, on the RX210 the block located at address `0xFFFFC000` is called Block 0 in the hardware manual therefore "BLOCK\_DB0" should be passed for this parameter.

#### Return Values

Returns the outcome of the erase operation:

FLASH\_SUCCESS = Operation Successful (if BGO is enabled this means the operations was started successfully)

FLASH\_FAILURE = Operation Failed

FLASH\_BUSY = Another flash operation is in progress

#### Properties

Prototyped in file "r\_Flash\_API\_RX200.h"

Implemented in file "r\_Flash\_API\_RX200.c"

#### Description

Erases a single block of Flash memory. Starting with RX63x MCUs some RX MCUs now have much smaller erase blocks for the data flash. For example, the RX210 has 128 byte erase blocks. This means that for a 8KB data flash there are 64 blocks. Instead of having a definition for each block (e.g. BLOCK\_DB0, BLOCK\_DB1, ..., BLOCK\_DB63) data flash blocks were grouped into 2KB virtual blocks. Each virtual block therefore consists of 16 real data flash blocks. This was done to make it easier on users to delete larger regions of data flash as has been done in the past. Users still have the option of deleting with 128 byte granularity using the `R_FlashEraseRange()` function.

NOTE: Do not attempt to erase a flash block that you are currently executing from.

## 5.2 R\_FlashEraseRange

The function starts erasing data flash blocks at a given address and stops when the number of bytes to erase has been reached.

### Format

```
uint8_t R_FlashEraseRange(uint32_t start_addr, uint32_t bytes);
```

### Parameters

#### *start\_addr*

Specifies the address where the erase should begin. This must be on an erase boundary and the address must be in the data flash area.

#### *bytes*

Specifies the number of bytes to erase. This must be a multiple of the data flash erase size. For example, on the RX210 the data flash erase size is 128 bytes so 128, 256 etc... could be used for this parameter.

### Return Values

Returns the outcome of the erase operation:

FLASH\_SUCCESS = Operation Successful (if BGO is enabled this means the operation was started successfully)

FLASH\_FAILURE = Operation Failed

FLASH\_ERROR\_ALIGNED = Flash address was not on correct boundary

FLASH\_BUSY = Another flash operation is in progress

FLASH\_ERROR\_BYTES = Number of bytes did not match erase size

FLASH\_ERROR\_ADDRESS = Invalid address, this is only for data flash

### Properties

Prototyped in file "r\_Flash\_API\_RX200.h"

Implemented in file "r\_Flash\_API\_RX200.c"

### Description

Erases at least 1 data flash block. This function was first introduced for RX63x MCUs that had significantly smaller data flash erase sectors than previous RX600 MCUs. Instead of having the user deal with a large number of data flash block #defines, this function allows the user to send in an address and how many bytes they wish to erase.

NOTE: This function is only available for data flash blocks. Cannot be used on ROM blocks.

### 5.3 R\_FlashWrite

This function allows data to be written into flash.

#### Format

```
uint8_t R_FlashWrite( uint32_t  flash_addr,
                     uint32_t  buffer_addr,
                     uint16_t  bytes);
```

#### Parameters

##### *flash\_addr*

This is a pointer to the Flash or Data Flash area to write. The address must be on a programming line boundary. See *Description* below for important restrictions regarding this parameter.

##### *buffer\_addr*

This is a pointer to the buffer containing the data to write to Flash.

##### *bytes*

The number of bytes contained in the *buffer\_addr* buffer. This number must be a multiple of the programming size for memory area you are writing to. See *Description* below for important restrictions regarding this parameter.

#### Return Values

Returns the outcome of the write operation:

FLASH\_SUCCESS = Operation Successful (if BGO is enabled this means the operations was started successfully)

FLASH\_FAILURE = Operation Failed

FLASH\_ERROR\_ALIGNED = Flash address was not on a programming boundary

FLASH\_ERROR\_BYTES = Number of bytes provided was not a multiple of the programming size

FLASH\_ERROR\_ADDRESS = Invalid address

FLASH\_ERROR\_BOUNDARY = (ROM) Cannot write across ROM Area Boundaries

FLASH\_BUSY = Flash is busy with another operation

#### Properties

Prototyped in file "r\_Flash\_API\_RX200.h"

Implemented in file "r\_Flash\_API\_RX200.c"

#### Description

Writes data to flash memory.

When performing a write the user must make sure to start the write on a programming boundary and the number of bytes to write must be a multiple of the programming size. The boundaries and programming sizes differ depending on what MCU is being used and whether the ROM or data flash is being written to. Programming boundaries start at the beginning of the flash area and then each boundary is a multiple of the programming size. For example, if the programming line size is 256, then the flash address you pass must have bits B0-B7 all be '0'.

Some MCUs have ROM Area boundaries (different than programming boundaries previously discussed) that cannot be written over. If the user is writing over this location then they will need to make sure to split up the writes such that the first write will program up to the boundary, and the second write will start at the boundary. If the user tries to write over this boundary the function will return an error before doing any programming operations. The user can see the boundaries for their device by looking at the ROM\_AREA\_# definitions for their device in r\_Flash\_API\_RX200.h.

MCU	ROM Programming Line Size	Data Flash Programming Line Size
RX210 Group	2, 8 or 128 bytes	2 or 8 bytes

## 5.4 R\_FlashDataAreaAccess

This function allows Data Flash areas to be accessed or modified.

### Format

```
void R_FlashDataAreaAccess(uint16_t read_en_mask,  
                           uint16_t write_en_mask);
```

### Parameters

#### *read\_en\_mask*

This is a bitmapped value where bits are used to determine which Data blocks should be able to be read by the MCU. A '0' indicates the block cannot be accessed and a '1' indicates it can. Bits 0-3 represent Data Blocks 0-3 respectively.

#### *write\_en\_mask*

This is a bitmapped value where bits are used to determine which Data blocks should be able to be modified (Erase/Write) by the Flash Control Unit (FCU). A '0' indicates the block cannot be modified and a '1' indicates it can. Bits 0-3 represent Data Blocks 0-3 respectively.

### Return Values

None.

### Properties

Prototyped in file "r\_Flash\_API\_RX200.h"  
Implemented in file "r\_Flash\_API\_RX200.c"

### Description

After RESET, the Data Flash area is not readable by the MCU. It is also not enabled for reprogramming. This function is used to select what blocks you would like to be read or modifiable. You only have to set this function once at the beginning of your application.

## 5.5 R\_FlashDataAreaBlankCheck

This function is used to determine if an area in the Data Flash area is blank or not, since this cannot be determined by simply reading the memory location.

### Format

```
uint8_t R_FlashDataAreaBlankCheck(uint32_t address,
                                   uint8_t size);
```

### Parameters

#### *address*

The address of the area to blank check.

If the parameter 'size' is specified as 'BLANK\_CHECK\_8\_BYTE' (available on RX610 and RX62x devices), this should be set to an 8-byte address boundary.

If the parameter 'size' is specified as 'BLANK\_CHECK\_2\_BYTE' (available on RX63x devices), this should be set to a 2-byte address boundary.

If the parameter 'size' is specified as 'BLANK\_CHECK\_ENTIRE\_BLOCK' (available on all RX600 devices), this should be set to a defined Data Block Number ('BLOCK\_DB0', 'BLOCK\_DB1', 'BLOCK\_DB2' or 'BLOCK\_DB3') or an address in the data flash block. Either option will work.

#### *size*

This specifies if you are checking an 8-byte location, 2-byte location, or an entire 8KB block. You must set this to either 'BLANK\_CHECK\_2\_BYTE', 'BLANK\_CHECK\_8\_BYTE', or 'BLANK\_CHECK\_ENTIRE\_BLOCK'.

### Return Values

Returns the outcome of the write operation:

FLASH\_BLANK = (2 or 8 Byte check or non-BGO) Blank. (Entire Block & BGO) Blank check operation started.

FLASH\_NOT\_BLANK = Not Blank

FLASH\_FAILURE = Operation Failed

FLASH\_BUSY = Another flash operation is in progress

FLASH\_ERROR\_ADDRESS = Invalid address was input

FLASH\_ERROR\_BYTES = Incorrect 'size' was submitted

### Properties

Prototyped in file "r\_Flash\_API\_RX200.h"

Implemented in file "r\_Flash\_API\_RX200.c"

### Description

Before you can write to any flash area in an MCU, the area must already be blank. Since the memory locations in RX200 Series Data Flash areas are not represented by a defined 'blank' value of 0xFF like they are in the User Program area, an additional function is needed to test a section of flash to determine if it is blank.

RX200 Series devices have two methods for checking for blank areas; one checks a smaller area and the other a larger area. The number of bytes checked by the smaller method is same as the smallest programming size for the data flash (i.e. 2 bytes on RX210). The larger check performs the blank check on the entire Data Flash block at once. This function does not have to be called for each section prior to programming. It is simply here to assist in application programming.

## 5.6 R\_FlashProgramLockBit

Sets the lock bit for a flash block.

### Format

```
uint8_t R_FlashProgramLockBit(uint8_t block);
```

### Parameters

*block*

The ROM erasure block that will have its lock bit set.

### Return Values

Returns the outcome of the operation:

FLASH\_SUCCESS = Operation Successful, lock bit set

FLASH\_FAILURE = Operation Failed

FLASH\_BUSY = Another flash operation is in progress

### Properties

Prototyped in file "r\_Flash\_API\_RX200.h"

Implemented in file "r\_Flash\_API\_RX200.c"

### Description

Each block of ROM has a lock bit associated with it. If lock bit protection is enabled and the lock bit is set for a given block then that block cannot be programmed or erased. If an attempt to erase or program the block is made, the operation will be ignored. This function will set the lock bit for the selected flash block. Whether lock bit protection is enabled or not is controlled by the API function R\_FlashSetLockBitProtection().

## 5.7 R\_FlashReadLockBit

Reads the lock bit for a flash block.

### Format

```
uint8_t R_FlashReadLockBit(uint8_t block);
```

### Parameters

*block*

This ROM erasure block will have its lock bit read.

### Return Values

Returns the outcome of the operation:

FLASH\_LOCK\_BIT\_SET = Lock bit is set

FLASH\_LOCK\_BIT\_NOT\_SET = Lock bit is not set

FLASH\_FAILURE = Operation Failed

FLASH\_BUSY = Another flash operation is in progress

### Properties

Prototyped in file "r\_Flash\_API\_RX200.h"

Implemented in file "r\_Flash\_API\_RX200.c"

### Description

Each block of ROM has a lock bit associated with it. If lock bit protection is enabled and the lock bit is set for a given block then that block cannot be programmed or erased. If an attempt to erase or program the block is made, the operation will be ignored. This function will return whether a flash block has its lock bit set or not. Whether lock bit protection is enabled or not is controlled by the API function R\_FlashSetLockBitProtection().

## 5.8 R\_FlashSetLockBitProtection

Enables or disables lock bit protection.

### Format

```
uint8_t R_FlashSetLockBitProtection(uint8_t lock_bit);
```

### Parameters

*lock\_bit*

Boolean value that determines whether to enable or disable lock bit protection. If set to 'true' then lock bit protection will be enabled. If set to 'false' then lock bit protection will be disabled.

### Return Values

Returns the outcome of the operation:

FLASH\_SUCCESS = Operation was successful

FLASH\_BUSY = Flash is busy with another operation

### Properties

Prototyped in file "r\_Flash\_API\_RX200.h"

Implemented in file "r\_Flash\_API\_RX200.c"

### Description

Each block of ROM has a lock bit associated with it. If lock bit protection is enabled and the lock bit is set for a given block then that block cannot be programmed or erased. If an attempt to erase or program the block is made, the operation will be ignored. This function controls whether lock bit protection is enabled. If disabled then all flash blocks are eligible for programming and erasure regardless of whether their lock bit is set or not.

## 5.9 R\_FlashGetStatus

Returns the current state of the flash.

### Format

```
uint8_t R_FlashGetStatus(void);
```

### Parameters

None.

### Return Values

Returns the state of the flash:

FLASH\_SUCCESS = Flash is ready to use

FLASH\_BUSY = Flash is busy with another operation

### Properties

Prototyped in file "r\_Flash\_API\_RX200.h"

Implemented in file "r\_Flash\_API\_RX200.c"

### Description

This function will return the current state of the flash. If BGO operations are used then this function call can be used to poll for detecting when the last flash operation has finished.

## 6. Example Code

The following is an example of using the API in order to erase and program Block 2 on a RX210 Group MCU.

```
#include "r_Flash_API_RX200.h"

char write_buffer[512]; // Enough to hold two 256 byte lines
void main(void)
{
    unsigned char result;

    /* Erase Block 2 */
    result = R_FlashErase( BLOCK_2 );

    /* Write our data buffer into Flash block 2.
       Note that we are writing two 256 byte lines. */
    result = R_FlashWrite( (unsigned long)0xFFFF4000, // Where to write
                          (unsigned long)write_buffer, // Our Data to write
                          512); // How much to write

    while(1); /* END OF DEMO */
}
```

The following is an example of using the API in order to erase and program Data Block 0 on a RX210 MCU.

```
#include "r_Flash_API_RX200.h"

char write_buffer[8]; // Data to program
void main(void)
{
    unsigned char result;

    /* Enable Read and Write access for the all the Data Flash blocks */
    R_FlashDataAreaAccess( 0xF, 0xF);

    /* Checks if an 8-byte area is blank */
    result = R_FlashDataAreaBlankCheck( (unsigned long)0x100400,
                                        BLANK_CHECK_8_BYTE);

    /* Erase Data Block 0 */
    result = R_FlashErase(BLOCK_DB0);

    /* Write our data buffer into Flash Data Block 0. */
    result = R_FlashWrite((unsigned long) 0x100000, // Where to write
                          (unsigned long) write_buffer, // Our Data to write
                          8); // How much to write

    while(1); /* END OF DEMO */
}
```

Please refer to the FlashAPI\_Sample\_RX200.c source file that comes packaged with this application note for more advanced sample code.

## Website and Support

Renesas Electronics Website

<http://www.renesas.com/>

Inquiries

<http://www.renesas.com/inquiry>

All trademarks and registered trademarks are the property of their respective owners.

## Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Dec.12.11	—	First edition issued

## 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 manual, refer to the relevant sections of the manual. If the descriptions under General Precautions in the Handling of MPU/MCU Products and in the body of the manual differ from each other, the description in the body of the manual takes precedence.

### 1. Handling of Unused Pins

Handle unused pins in accord 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 one with a different type number, confirm that the change will not lead to problems.

- The characteristics of MPU/MCU in the same group but having different type numbers may differ because of the differences in internal memory capacity and layout pattern. When changing to products of different type numbers, implement a system-evaluation test for each of the products.

## Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
  2. Renesas Electronics does not assume any liability for infringement of 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. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
  3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
  4. 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 of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
  5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may 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.
  6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
  7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheet or data books, etc.  
"Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.  
"High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.  
"Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
  8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
  9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, 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, please evaluate the safety of the final products or system manufactured by you.
  10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
  11. This document may not be 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, or if you have any other inquiries.
- (Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.  
(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.



### 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.

2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.  
Tel: +1-408-588-6000, Fax: +1-408-588-6130

#### Renesas Electronics Canada Limited

1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada  
Tel: +1-905-898-5441, Fax: +1-905-898-3220

#### Renesas Electronics Europe Limited

Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K  
Tel: +44-1628-585-100, Fax: +44-1628-585-900

#### Renesas Electronics Europe GmbH

Arcadiastrasse 10, 40472 Düsseldorf, Germany  
Tel: +49-211-65030, Fax: +49-211-6503-1327

#### Renesas Electronics (China) Co., Ltd.

7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China  
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

#### Renesas Electronics (Shanghai) Co., Ltd.

Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China  
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

#### Renesas Electronics Hong Kong Limited

Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong  
Tel: +852-2886-9318, Fax: +852-2886-9022/9044

#### Renesas Electronics Taiwan Co., Ltd.

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

#### Renesas Electronics Singapore Pte. Ltd.

1 HarbourFront Avenue, #06-10, Keppel Bay Tower, Singapore 098632  
Tel: +65-6213-0200, Fax: +65-6278-8001

#### Renesas Electronics Malaysia Sdn.Bhd.

Unit 906, 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 Korea Co., Ltd.

11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea  
Tel: +82-2-558-3737, Fax: +82-2-558-5141