RX600 Series

Virtual EEPROM for RX

R01AN0724EU0150
Rev.1.50
January 3, 2012

Introduction

Many users wish to use the data flash on their MCUs as they would an EEPROM. The problem with this is that many data flashes on the RX600 series do not have 1-byte write or 1-byte erase capabilities. Even if their RX MCU did offer this granularity there would also be the issue of wear leveling. To help solve these issues the Virtual EEPROM project (VEE for short) was created. The VEE project offers these features:

- Users can write any amount of data regardless of the data flash's true programming size
- Wear leveling is used to increase data flash longevity
- Easy to use API interface to safely read and write
- Uses background operation feature of MCUs so data flash operations do not block the user application
- Automatically recovers from resets or power downs that occur during programs and erases
- Adapts to flashes with different program sizes, erase sizes, block sizes, and number of data flash blocks
- Highly configurable to adjust to unique needs of each user

Target Device

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

- **RX621, RX62N Group**
- **RX62T Group**
- **RX630 Group**
- **RX631, RX63N Group**

Contents

1. Overview 2

2. API Information 5

3. API Functions 13

4. Demo Workspace 19

1. Overview

The Virtual EEPROM (VEE) project is a software layer that sits on top of the Renesas provided Flash API as shown in Figure 1.

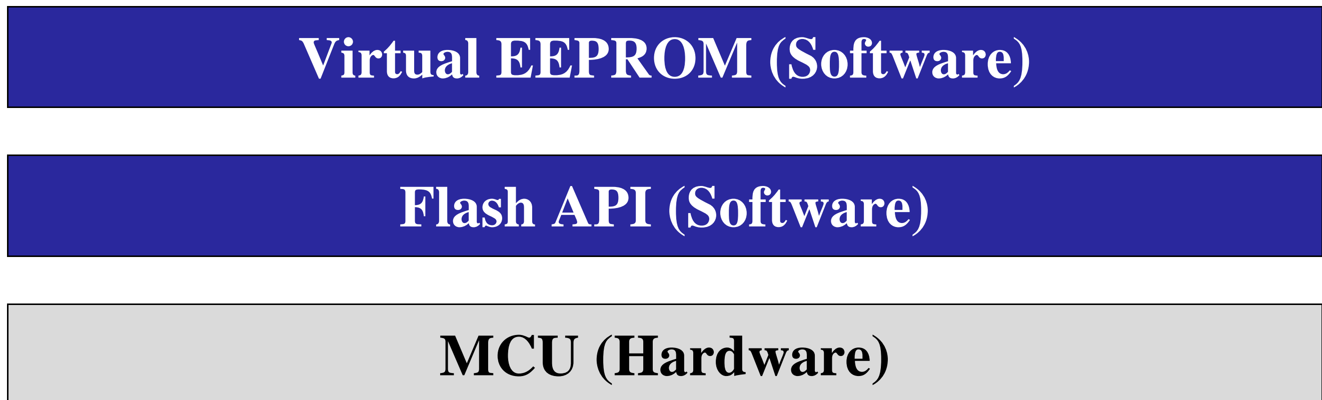


Figure 1 : Project Layers

1.1 Use of Background Operations

The Virtual EEPROM requires that the user's MCU have hardware support for background operations (BGO) on the data flash and that the Flash API has software support. Background operation means that flash operations do not block. In a blocking system (one that does not have support for BGO) when a flash operation is started, control is not given back to the user application until the operation has completed. With a system that supports BGO, control is given back to the user application immediately after the operation has been successfully started. The user will be alerted by a callback function, or can poll to see if the operation has completed. Since the VEE project uses the BGO capabilities of the MCU this means less time is taken away from the user application.

1.2 Records

When data is written to the VEE it is done using VEE Records. Each record has some information that must be filled in by the user as well as a pointer to the data to be stored. Users can store as much data as they wish with each record. When the user writes a record to the VEE it stores the data and the record information together. Each record is identified by a unique ID. If the user writes a record with the same ID as a record that was previously written then it will be written as a new record and the older record will no longer be valid. The reason this is done is for wear leveling purposes. The user does have the option of configuring the VEE code to ignore duplicate writes. An example of how records are stored is shown in Section 1.4.

1.3 Data Management

With the VEE project the MCU's data flash area is split up into VEE Sectors. These do not correspond to real sectors on the MCU. The VEE project requires at least one VEE Sector. Each VEE Sector is made up of at least two VEE Blocks. Each VEE Block can be made up of one or more flash blocks on the MCU. One VEE Block has the latest stored data for that VEE Sector at any given time.

The VEE Blocks ping-pong data back and forth as one becomes full. When a write occurs and there is no room left in the current VEE Block a defrag occurs in which the latest data is transferred to the next VEE Block in the VEE Sector. Once the data has been transferred the old VEE Block can be erased so that it will be ready when the new VEE Block becomes full. This moving of data is used for wear leveling purposes.

Having different VEE Sectors allows the user to separate data. One reason for doing this would be to separate frequently written data from infrequently written data. An example is if a user had one large block of data that was written once a day and a small block of data that was written every minute. The small block will fill up the current block quickly and will force a defrag very often which means the large block of data that has not been changed will need to be transferred even though the data is the same as before. The large block can also cause defrags to happen more often since it can take up a significant portion of the VEE Block's available space.

The user writes VEE Records, described in Section 1.2, to the VEE. After the record has been written to the virtual EEPROM the user can retrieve the data using the unique ID. If the user wants to store a new version of the record they can send in the same ID as before. The record does not have to be the same size as before, the API takes care of this.

1.4 Example of VEE in Action

This section explains how data is managed in a 1 sector, 2 block VEE setup. As explained earlier, VEE Records are stored in VEE Blocks. At least 2 VEE Blocks are needed to make a VEE Sector. At any point in time within the VEE project there is a maximum of one valid record per unique ID. As records with the same ID are written, the newest record becomes the valid record and the previously written records are ignored. Figure 2 shows what the VEE might look like during use. Notice that each record is represented twice in VEE Block 0. Only the record that is lower (records are stored downward in this example) is valid.

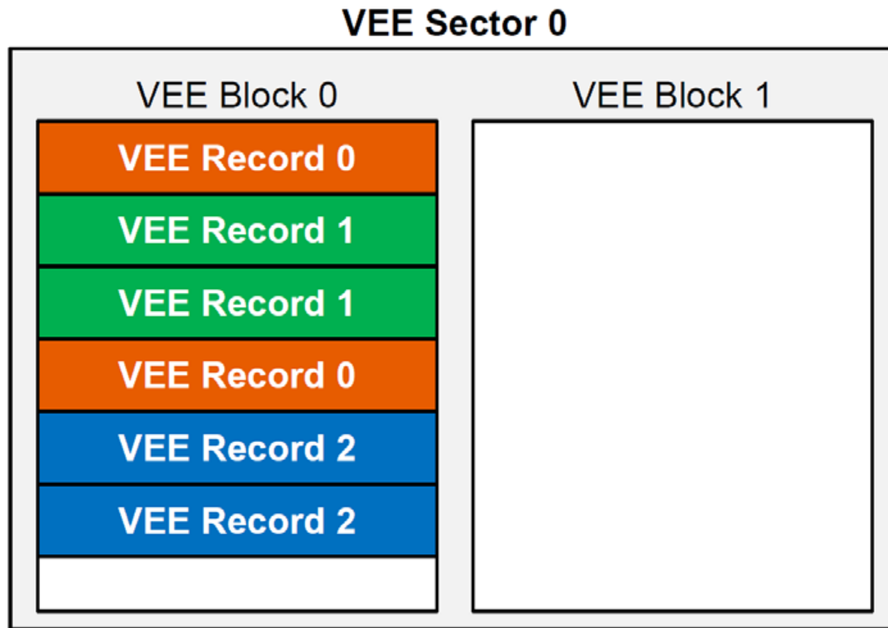


Figure 2 : Filling up Block 0

When a VEE write is issued to a VEE Block that does not have a enough room for the record, a defrag is needed. A defrag will move all valid records to another VEE Block. This is shown in Figure 3 where the write of VEE Record 1 will not fit into VEE Block 0. This forces a defrag where all valid records will be moved to VEE Block 1.

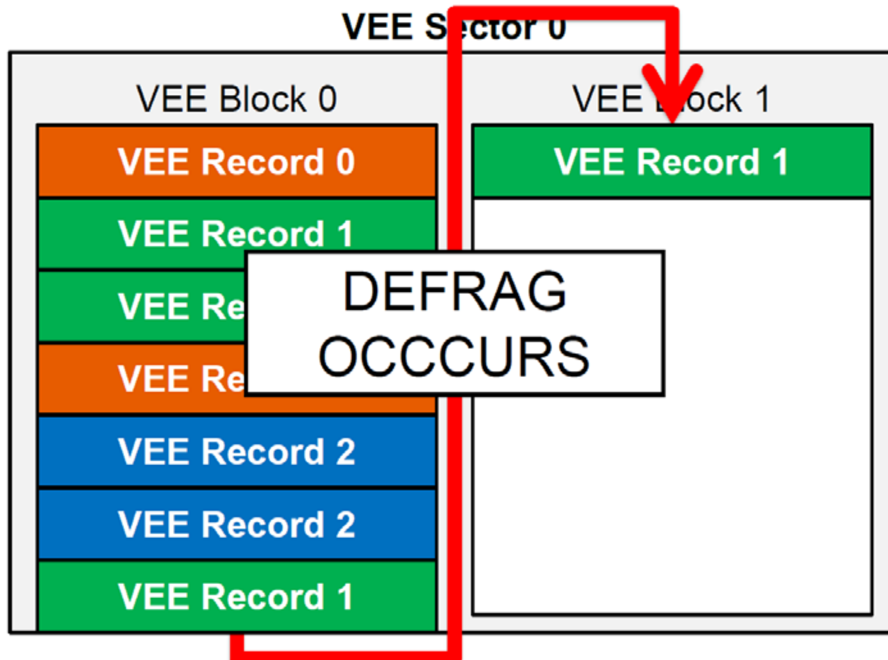


Figure 3 : Defrag moves data to Block 1

Notice that only the valid versions of each record are copied. The older records are ignored and later erased.

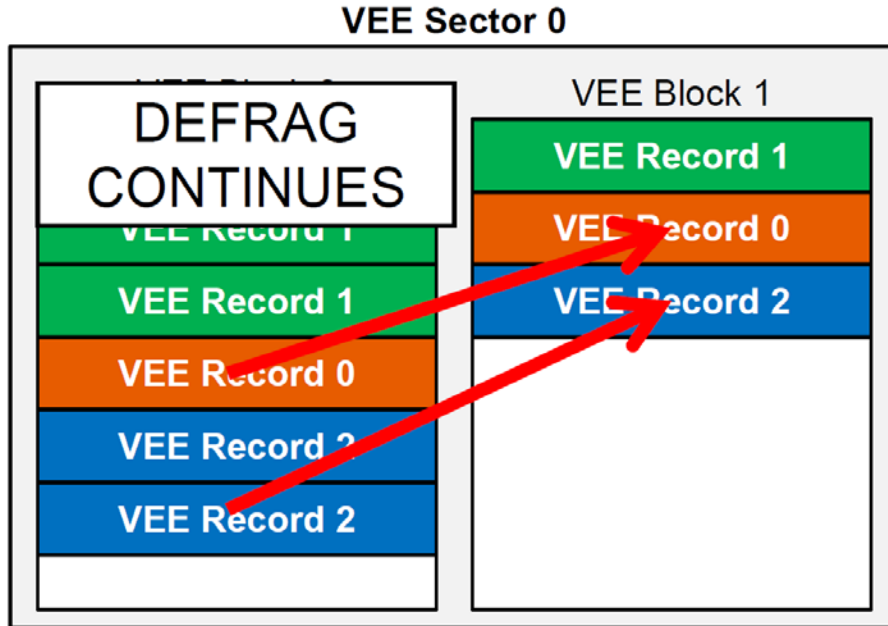


Figure 4 : Defrag only copies valid records

After the defrag has completed the previous VEE Block will be erased. Figure 5 shows the state of the VEE after the defrag and erase have completed. All of the valid records have been moved to VEE Block 1 and VEE Block 0 has been erased. Now that VEE Block 0 has been erased, it will be ready when a defrag is needed for VEE Block 1.

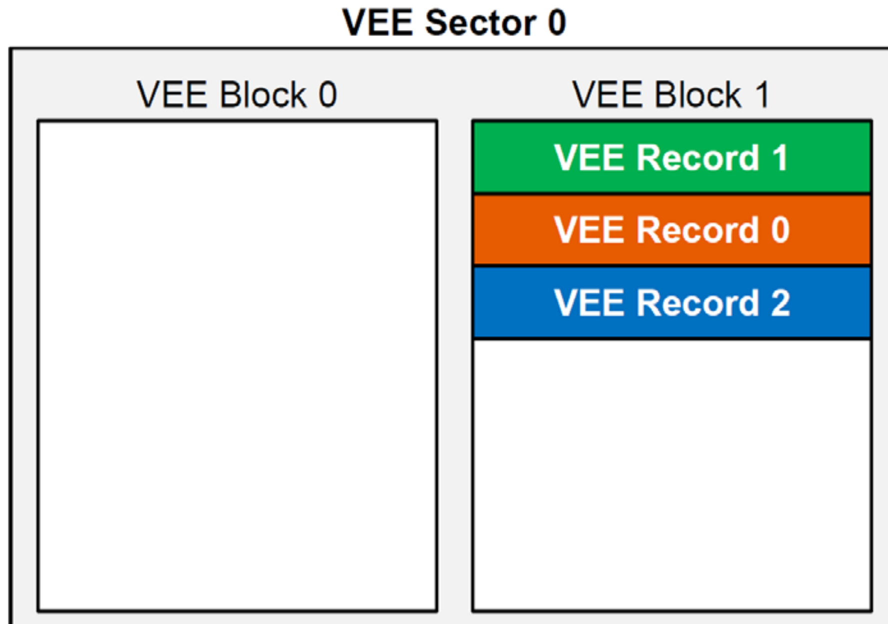


Figure 5 : Delete Block 0 after defrag

2. API Information

The VEE API follows the Renesas API naming standards.

2.1 Hardware Requirements

This middleware requires your MCU support the following features:

- Flash with background operation (BGO) feature

2.2 Hardware Resource Requirements

This section details the hardware peripherals that this middleware requires. Unless explicitly stated, these resources must be reserved for the middleware and the user cannot use them.

2.2.1 Data Flash

For safe operations the user should reserve the entire data flash on their MCU for VEE operations. If this rule is not followed then user will be responsible for making sure other data flash operations do not interfere with VEE operations.

2.3 Software Requirements

This middleware is dependent upon the following packages:

- Simple Flash API for RX600 (R01AN0544EU)

2.4 Header Files

All API calls are accessed by including a single file, *r_vee.h*, which is supplied with the VEE project code. This header file in turn references *r_vee_config.h* which has the user's VEE configuration information.

2.5 Integer Types

This project uses ANSI C 99 "Exact width integer types" in order to make the code clearer and more portable. These types are defined in *stdint.h*.

2.6 Configuration Overview

This section covers the definitions located in the *r_vee_config.h* file and discusses how each one changes the behavior of the VEE project.

Configuration options found in r_vee_config.h	
VEE_NUM_SECTORS	This defines the number of VEE Sectors that will be used. This definition will choose a sector configuration found in the *r_vee_config_*port*.h* file. For example, if a RX62N is being used with 2 VEE Sectors then there must be a 2 sector VEE configuration defined in *r_vee_config_rx62x.h*.
VEE_MAX_RECORD_ID	The number of unique records to be used in the system. It is not recommended to put more entries than you will use since there is a cache entry for each possible unique ID. If you attempt to issue a VEE operation for a VEE Record with an ID greater than or equal to this value then the API will return an error.
VEE_IGNORE_DUPLICATE_WRITES	This option allows the user to select to ignore record writes where the record being written is the same (data matches) as the record already stored in the VEE. This will potentially save space in VEE and lead to less defrags but takes extra time to check for existing matches when performing a write.

VEE_CACHE_FILL_ALL	This configuration value lets you choose whether you want to fill the cache all at once, or one record at a time. If this definition is uncommented then when a read is performed and a record is not found in the cache (e.g. first read after reset) then all the records will be searched for and the entire cache will be filled in at once. If this is commented out then only the record which was requested will be searched for. This comes down to do you want to spread out the time required for searching or do you want to get it all of out of the way at the very beginning so that subsequent reads will not require searching?
VEE_USE_DEFAULT_CHECK_FUNCTIONS	Lets the user choose whether to use the default R_VEE_GenerateCheck() and vee_check_record() functions or whether they want to write their own. For example, if the user wanted to use a checksum instead of the default static flag check then they could comment out this define and write their own functions. If the user writes their own function they must still use the same function name and arguments.
VEE_CALLBACK_FUNCTION	There are 2 parts to this configuration item. The first part is whether the definition VEE_CALLBACK_FUNCTION is defined or not. If it is not defined then callbacks will not be used in the VEE project and the user must poll to see if an operation has finished. If it is defined then the definition should be the name of the callback function the user is going to write. For example, if the user defined VEE_CALLBACK_FUNCTION to be 'MyCallback' then the user would write the MyCallback() function in their application and this function will be called when a VEE operation completes.

Table 1 : Description of Configuration Definitions

2.6.1 Using the r_bsp Package

Starting with v1.50, the VEE middleware uses the r_bsp package. The r_bsp package includes startup code and MCU information for different RX boards. The VEE code gets the information it needs about the MCU being used from the files in the r_bsp package. Users are encouraged to add their own boards to the r_bsp package. By having a clear foundation for middleware to be built on top of this should enable RX middleware to be more easily integrated.

2.7 VEE Sector Configuration

This section covers VEE Sector configuration using the *r_vee_config_*port*.h* (e.g. for RX62N this filename is *r_vee_config_rx62x.h*) file. The definitions and data structures presented in this section configure the VEE Sectors, VEE Blocks, and VEE Records used in the VEE project. The data structures are defined in *r_vee_config_*port*.h* but no space is allocated until they are declared in *r_vee.c*.

2.7.1 Number of VEE Sectors

The number of VEE Sectors to be used is defined by the VEE_NUM_SECTORS #define in *r_vee_config.h*. See Section 2.6 for more information.

2.7.2 Assigning VEE Records to VEE Sectors

VEE Records are assigned to VEE Sectors at compile time via the g_vee_RecordLocations[] array. There is one entry in this array per unique ID in the VEE project. How many unique IDs there are in the VEE project is controlled by the VEE_MAX_RECORD_ID definition discussed in Section 2.6. The value for each entry in the array is which VEE Sector the record will be located in. Below is an example configuration where there are 2 VEE Sectors and we want the first 4 records to be located in VEE Sector 0 and the last 4 records to be located in VEE Sector 1.

```
const uint8_t g_vee_RecordLocations[VEE_MAX_RECORD_ID] = {
    0,  /* Record 0 will be in sector 0 */
    0,  /* Record 1 will be in sector 0 */
    0,  /* Record 2 will be in sector 0 */
    0,  /* Record 3 will be in sector 0 */
    1,  /* Record 4 will be in sector 1 */
    1,  /* Record 5 will be in sector 1 */
    1,  /* Record 6 will be in sector 1 */
    1,  /* Record 7 will be in sector 1 */
};
```

2.7.3 Allocating VEE Blocks

After defining how many VEE Sectors will be used the user must decide where the VEE Blocks inside of the sectors will be allocated. This is done using two separate arrays. The names presented below are the defaults used.

The first array is named g_vee_sect#_block_addresses[] where the '#' is replaced with the sector number. Each entry of the array defines the starting address of a VEE Block in this particular sector. There will be one of these arrays defined for each VEE Sector.

The second array is named g_vee_sect#_df_blocks[][2] where the '#' is once again replaced with the sector number. This is 2D array so each entry is another array. The internal array is an array that holds the first and last data flash blocks on the MCU that make up this VEE Block. There will be one of these arrays defined for each VEE Sector.

An example is shown below of a system with the following setup:

- 2 VEE Sectors
- 2 VEE Blocks per VEE Sector
- 4 MCU data flash blocks per VEE Block
- VEE Sector 0 will be allocated lower in memory than VEE Sector 1

```
/* Sector 0 */
const uint32_t g_vee_sect0_block_addresses[] =
{
    0x100000, /* Start address of VEE Block 0 */
    0x102000  /* Start address of VEE Block 1 */
};
const uint16_t g_vee_sect0_df_blocks[][2] =
{
    {BLOCK_DB0, BLOCK_DB3}, /* Start & end DF blocks making up VEE Block 0 */
    {BLOCK_DB4, BLOCK_DB7}  /* Start & end DF blocks making up VEE Block 1 */
};
/* Sector 1 */
const uint32_t g_vee_sect1_block_addresses[] =
{
    0x104000, /* Start address of VEE Block 0 */
    0x106000  /* Start address of VEE Block 1 */
};
const uint16_t g_vee_sect1_df_blocks[][2] =
{
    {BLOCK_DB8,  BLOCK_DB11}, /* Start & end DF blocks making up VEE Block 0 */
    {BLOCK_DB12, BLOCK_DB15}  /* Start & end DF blocks making up VEE Block 1 */
};
```

The data flash block #defines (e.g. BLOCK_DB0) are defined by the Simple Flash API for RX600 package.

2.7.4 Data Structure that Holds VEE Project Data Configuration

The g_vee_Sectors array is the data structure that is used in the VEE project code for obtaining information about the current systems VEE data configuration. Each entry defines a VEE Sector and holds the following information:

- The ID of the sector
- How many VEE Blocks make up this sector
- The size (in bytes) of this sector
- The starting MCU addresses for each VEE Block in this sector
  - Discussed in Section 2.7.3.
- The number of MCU data flash blocks per VEE Block
- The start and end MCU data flash blocks for each VEE Block
  - Discussed in Section 2.7.3.

An example is shown below of a VEE project with 3 different sized VEE Sectors.

```
const vee_sector_t g_vee_Sectors[ VEE_NUM_SECTORS ] =
{
    /* Sector 0 */
    {
        /* ID is 0 */
        0,
        /* There are 2 VEE Blocks in this sector */
        2,
        /* Size of each VEE Block */
        8192,
        /* Starting addresses for each VEE Block */
        (const uint32_t *)g_vee_sect0_block_addresses,
        /* Number of data flash blocks per VEE Block
           (End Block # - Start Block # + 1) */
        4,
        /* Start & end DF blocks making up VEE Blocks */
        g_vee_sect0_df_blocks
    }
    ,
    /* Sector 1 */
    {
        /* ID is 1 */
        1,
        /* There are 2 VEE Blocks in this sector */
        2,
        /* Size of each VEE Block */
        6144,
        /* Starting addresses for each VEE Block */
        (const uint32_t *)g_vee_sect1_block_addresses,
        /* Number of data flash blocks per VEE Block
           (End Block # - Start Block # + 1) */
        3,
        /* Start & end DF blocks making up VEE Blocks */
        g_vee_sect1_df_blocks
    }
    ,
```

EXAMPLE CONTINUED ON NEXT PAGE

```
    /* Sector 2 */
    {
        /* ID is 2 */
        2,
        /* There are 2 VEE Blocks in this sector */
        2,
        /* Size of each VEE Block */
        2048,
        /* Starting addresses for each VEE Block */
        (const uint32_t *)g_vee_sect2_block_addresses,
        /* Number of data flash blocks per VEE Block
           (End Block # - Start Block # + 1) */
        1,
        /* Start & end DF blocks making up VEE Blocks */
        g_vee_sect2_df_blocks
    }

    /* To add more sectors copy the one above and change the values */
};
```

2.8 API Data Structures

2.8.1 VEE Record

When reading and writing to the VEE using the provided API functions the user sends in data using a VEE Record data structure. This structure is shown below.

```
/* VEE Record Structure */
typedef struct
{
    /* Unique record identifier, cannot be 0xFF! */
    vee_var_data_t    ID;
    /* Number of bytes of data for this record */
    vee_var_data_t    size;
    /* Valid or error checking field */
    vee_var_data_t    check;
    /* Which VEE Block this record is located in, user does not set this */
    vee_var_data_t    block;
    /* Pointer to record data */
    uint8_t     far  * pData;
} vee_record_t;
```

2.9 Return Values

The possible return values from the VEE API functions are listed in the following typedef from *r_vee_types.h*.

```
/* Return values for functions */
typedef enum
{
    VEE_SUCCESS,
    VEE_FAILURE,
    VEE_BUSY,
    VEE_NO_ROOM,
    VEE_NOT_FOUND,
    VEE_ERROR_FOUND
} vee_return_values_t;
```

2.10 VEE States

The possible states that the VEE project can be in are listed below and can be found in *r_vee_types.h*. One of these states will be returned by the R_VEE_GetState() function.

```
/* Defines the possible states of the VEE */
typedef enum
{
    VEE_READY,
    VEE_READING,
    VEE_WRITING,
    VEE_ERASING,
    VEE_DEFRAG,
    VEE_ERASE_AND_DEFRAG,
    VEE_WRITE_AND_DEFRAG,
    VEE_ERASE_AND_WRITE
} vee_states_t;
```

2.11 MCU Specific Typedefs

There are two typedefs used in the VEE project that change depending on the data flash characteristics of the MCU being used. These two typedefs are *vee_var_min_t* and *vee_var_data_t* and will be found in the MCU specific header file for your project (e.g. *r_vee_rx62x.h* for RX62x devices).

The size of *vee_var_min_t* should be set to the minimum program size of the data flash. For example, the RX62N group has 8-byte minimum writes on its data flash so uint64_t (8-byte integer) would be used. On the RX63N, which has 2-byte writes, *vee_var_data_t* would be set to uint16_t (2-byte integer).

The *vee_var_data_t* typedef is used in the VEE Record structure as can be seen in Section 2.8. The size of *vee_var_data_t* has to be at least the size of *vee_var_min_t* and can be larger. The more bytes used for *vee_var_data_t,* also means the more bytes of overhead per VEE Record. This is discussed in more detail in Section 2.12. Therefore, the smallest value for *vee_var_data_t* that can be used in your system should be used. The user would set the size of *vee_var_data_t* to be larger than the minimum write size of the data flash in the event that they needed the extra range. For example, on the R8C/38C the *vee_var_data_t* typedef could be set to uint8_t (1-byte integer) but this would limit the size of the data for each VEE Record to 255 bytes (0xFF).

Below is an example setup for the R8C/3x Group.

```
/* Set size of vee_var_data_t to the minimum write size of MCU's data flash
   or larger. This is the size of the variables in a record structure. */
typedef uint16_t vee_var_data_t;
/* Set size of vee_var_min_t to the minimum write size of MCU's data flash */
typedef uint8_t vee_var_min_t;
```

2.12 Overhead Associated with VEE Records

The overhead of each VEE Record can be seen by looking at the data structure used in Section 2.8. Other than the data being stored there are 4 members of type *vee_var_data_t*. This means that the overhead of each VEE Record will be '4 * sizeof(*vee_var_data_t*)'. For example, using the default settings shown in Section 2.11 for the R8C/3x, there will be 8-bytes of overhead since *vee_var_data_t* is set to be 2-bytes. On the RX62N, *vee_var_data_t* is set to 8-bytes by default, meaning that the overhead will be 32-bytes per record.

2.13 Reading Data from VEE

To read a record out of the VEE the user calls the R_VEE_Read(vee_record_t * VEE_TEMP) API function. If found the VEE will set the pData pointer to the address of the data in the data flash. The user can now use this pointer to read the data. A very important thing to remember is that after performing a read the R_VEE_ReleaseState() function must be called before any other VEE API functions can proceed other than another read command. This is enforced for safety reasons. The reason this is required is that when using BGO data flash operations there is no way to guarantee the data can be safely read unless the user has exclusive access to the data flash. Below is an example of how a problem could occur if these precautions were not in place.

1. User calls R_VEE_Read() for VEE Record 0 and gets address of data in data flash.
2. User reads data for VEE Record 0.
3. User calls R_VEE_Write() and writes VEE Record 1 to the VEE.
4. The R_VEE_Write() function returns successfully and the write is going on in the background using BGO.
5. The user reads data again from VEE Record 0, the address of which was obtained earlier.
6. A data flash access violation error occurs because a read was attempted to the data flash before the write of VEE Record 1 finished.

This same scenario could occur during a VEE erase or defrag. There are safety precautions in place to help prevent this from occurring. Even with these built-in precautions the user must still be aware of what they are doing. For example, the VEE project has no way of preventing a user from using a VEE Record data pointer that was obtained earlier during a current data flash operation. In order for users to protect themselves from this scenario they must make sure to always read, or re-read, data using the R_VEE_Read() command after any of the following commands:

- R_VEE_Write()
- R_VEE_Erase()
- R_VEE_Defrag()

For example, even if VEE Record 0 was read earlier, the R_VEE_Read(ID=0) command should be used before reading the data again after the R_VEE_Write(ID=2) command was issued. Below is an example of how a user should read and use data from the VEE and how API functions will try to enforce safe use.

Operation	Result
R_VEE_Read(ID=1)	Successful
R_VEE_Write(ID=2)	**Not successful**, VEE_BUSY returned
R_VEE_Read(ID=2)	Successful
Use data from VEE Records 1 & 2 using previously obtained data pointers	Successful
R_VEE_Erase(…)	**Not Successful**, VEE_BUSY returned
R_VEE_ReleaseState()	Successful
R_VEE_Erase(…)	Successful
R_VEE_Write(ID=3)	Successful
R_VEE_Read(ID=2)	Successful
Use data from VEE Record 2 using previously obtained data pointer	Successful

2.14 Error Recovery

When the R_VEE_Write() or R_VEE_Defrag() API functions are used the API will check the current VEE Sector for errors. Errors occur when a reset or power down occurs during a VEE operation. For example, if a reset occurs during a VEE write operation then the record's data will stop abruptly in the data flash. The VEE does protect users from reading records that were not completely programmed by using the 'check' member of the VEE Record structure. The 'check' member is written last and checked when a VEE read occurs.

The error recovery mechanisms built into the VEE project will try to recover as much data as possible. There is one case where the latest written record can be lost. This scenario is described below.

1. VEE Record 0 is written using R_VEE_Write().
2. There was not enough room for VEE Record 0 so a defrag occurs.
3. Defrag starts off by writing VEE Record 0 to new VEE Block.
4. After VEE Record 0 is written a reset occurs before the defrag can finish.
5. User tries to read VEE Record 0 using R_VEE_Read().

At this point there are two options. The first option is that the VEE will return the last written record for VEE Record 0 that was in the previous VEE Block that was being defragged before the reset occurred. The other option is to find the newer record that is located in the new VEE Block. The first option is chosen for simplicity and quickness. The R_VEE_Read() function does not check VEE Sectors for corruption which means that records can be returned in the quickest time possible. This especially critical to customers that need data as quick as possible on power-up. When the next write occurs to the VEE Sector in question it will detect the error and will erase the new block and then start the defrag operation again.

2.15 Adding Middleware to Your Project

Follow the steps below to add the VEE code to your project. These steps assume that the Flash API has already been added to your project.

1. Copy the 'r_vee' directory (packaged with this application note) to your project directory.
2. Add the file 'r_vee.c' to your project.
3. Add the C Source file for your MCU port to your project.
    a. e.g. for RX62x this file is named 'r_vee_rx62x.c'
4. Add an include path to the 'r_vee' directory.
5. Add an include path to the 'r_vee\src' directory.
6. Configure the middleware using *r_vee_config.h*.

2.16 Detecting Flash Errors

In the event that a data flash error occurs the user will be alerted using the FlashError() callback function. This is the same callback function that is used by the Flash API. Since it is a callback function the user should write the function in their own application. The prototype for the function is shown below.

```
void FlashError(void);
```

3. API Functions

3.1 R_VEE_Read

Attempts to find a record in the VEE.

Format

```
uint8_t R_VEE_Read(vee_record_t * vee_temp);
```

Parameters

vee_temp

Pointer to structure with record information to look for.

Return Values

VEE_SUCCESS:	*Successful, structure members set accordingly*
VEE_NOT_FOUND:	*Record not found*
VEE_BUSY:	*Other VEE operation in progress, try again later*

Properties

Prototyped in file “r_vee.h”

Description

This function is called to try and retrieve a record from the VEE. The user sends in a VEE Record structure with the ID filled in for the record they wish to find. The VEE will first search the VEE Cache. If the record is not found in the cache then it will be searched for in the data flash. If the record is found in the data flash then its location will be stored in the cache for future reads.

Reentrant

Yes, but only when a VEE write, defrag, or erase is not on-going. When one of these operations is on-going, the function will return VEE_BUSY.

Example

```
vee_record_t example_record;

/* We want to find VEE Record 1 */
example_record.ID = 1;

/* Search VEE for record */
if (VEE_SUCCESS == R_VEE_Read(&example_record))
{
    /* Send data */
    for (loop = 0; loop < example_record.size; loop++)
    {
        TransmitByte(example_record.pData[loop]);
        ...
    }
}
```

Special Notes:

This function will not cause recovery methods to be started in the event that the VEE is corrupt. For example, if a reset or power down occurs during a VEE write, erase, or defrag then the VEE system will be left in a corrupt state. If a VEE read is issued when the MCU powers back on then the read will be issued on the corrupt system. This function will ignore the corrupt system and will attempt to read the latest valid VEE Record with the supplied ID. The reason this function operates this way is for users to be able to read their data as quick as possible after a reset. If recovery methods were started with this function then the user’s application would have to stall waiting for the VEE system to be fixed.

3.2 R_VEE_Write

Writes a record to the VEE.

Format

```
uint8_t R_VEE_Write(vee_record_t * vee_temp);
```

Parameters

vee_temp

Pointer to structure with record to write

Return Values

VEE_SUCCESS:	*Successful, write in progress*
VEE_BUSY:	*Other VEE operation in progress, try again later*
VEE_NO_ROOM:	*No room, need to call R_VEE_Erase()*
VEE_FAILURE:	*Data flash operation failed*

Properties

Prototyped in file “r_vee.h”

Description

This function is used to write a VEE Record to the data flash. When sending in the VEE Record the user should fill in the following members:

- ID
- size
- check
- pData

If the function returns VEE_SUCCESS then the record has not yet been written. It is in the process of being written. If the user chose to use the VEE callback function then it will trigger when the write has finished. Otherwise, the user can use the R_VEE_GetState() function to poll the VEE. When a record is written it is automatically entered into the VEE Cache for faster retrieval in the future.

Reentrant

No, but is protected by lock to prevent errors from concurrent function calls

Example

```
vee_record_t example_record;

/* Fill in data for VEE Record 1 */
example_record.ID = 1;
example_record.size = sizeof(record_data);
example_record.pData = &record_data[0];

/* Generate 'check' field */
R_VEE_GenerateCheck(&example_record);

/* Write record */
if (VEE_SUCCESS == R_VEE_Write(&example_record))
{
    ...
}
```

Special Notes:

This function will check the current VEE Sector for errors and will attempt to fix any that are detected. If recovery operations are needed then the API will return VEE_BUSY and the user will need to issue the write again later.

3.3 R_VEE_Defrag

Defrags a sector of the VEE.

Format

```
uint8_t R_VEE_Defrag(uint8_t sector);
```

Parameters

sector

ID of which VEE Sector to defrag

Return Values

VEE_SUCCESS:	*Successful, defrag in progress*
VEE_BUSY:	*Other VEE operation in progress, try again later*
VEE_NOT_FOUND:	*No ACTIVE block found to defrag*

Properties

Prototyped in file “r_vee.h”

Description

This function is used to defrag a sector. Defrags will be done automatically when R_VEE_Write() is called and there is no more room in the active VEE Block. The user might want to use this function to force a defrag during idle time so that a defrag will have less chance of happening during a more busy ‘writing’ phase.

If the function returns VEE_SUCCESS then the defrag has not yet finished. It is in the process of being defragged. If the user chose to use the VEE callback function then it will trigger when the defrag has finished. Otherwise, the user can use the R_VEE_GetState() function to poll the VEE.

Reentrant

No, but is protected by lock to prevent errors from concurrent function calls

Example

```
uint8_t sector;

for (sector = 0; sector < VEE_NUM_SECTORS; sector++)
{
    /* Defrag sector */
    ret = R_VEE_Defrag(sector);

    /* Check result */
    if (VEE_SUCCESS == ret)
    {
        ...
    }

    /* Wait for defrag to finish */
    ...
}
```

Special Notes:

This function will check the current VEE Sector for errors and will attempt to fix any that are detected. If recovery operations are needed then the API will return VEE_BUSY and the user will need to issue the write again later.

3.4 R_VEE_Erase

Erases a sector of the VEE.

Format

```
uint8_t R_VEE_Erase(uint8_t sector);
```

Parameters

sector

ID of which VEE Sector to erase

Return Values

VEE_SUCCESS:	*Successful, erase in progress*
VEE_BUSY:	*Other VEE operation in progress, try again later*
VEE_FAILURE:	*Data flash operation failed*

Properties

Prototyped in file “r_vee.h”

Description

This function is used to erase the data in a VEE Sector. If no active VEE Block is found in the given VEE Sector then VEE_SUCCESS will be returned. If an active VEE Block is found then that block will be erased. This function uses the VEE Block’s flags to determine whether it is empty or not. It does not check every memory location in a VEE Block.

If the function returns VEE_SUCCESS then the erase has not yet finished. It is in the process of being erased. If the user chose to use the VEE callback function then it will trigger when the erase has finished. Otherwise, the user can use the R_VEE_GetState() function to poll the VEE.

Reentrant

No, but is protected by lock to prevent errors from concurrent function calls

Example

```
uint8_t sector;

/* Erase all data from VEE */
for (sector = 0; sector < VEE_NUM_SECTORS; sector++)
{
    /* Erase sector */
    ret = R_VEE_Erase(sector);

    /* Check result */
    if (VEE_SUCCESS == ret)
    {
        ...
    }

    /* Wait for erase to finish */
    ...
}

/* VEE is empty */
```

Special Notes:

The VEE project has no way of invalidating VEE Record structures the user may have in their project when the VEE Sector that the records are stored in is erased. To prevent any errors from this the user should take care to not read data using a data pointer to a VEE Sector that has been erased. Instead the user should always use the R_VEE_Read() function after an erase to make sure they have valid data.

3.5 R_VEE_GetState

Returns the current state of the VEE.

Format

```
vee_states_t R_VEE_GetState(void);
```

Parameters

none

Return Values

State of VEE. Refer to Section 2.10 for information on possible VEE states.

Properties

Prototyped in file “r_vee.h”

Description

This function returns the current state of the VEE. This function can be used to poll the VEE to detect when a VEE operation has finished.

Reentrant

Yes.

Example

```
uint8_t sector;

/* Erase all data from VEE */
for (sector = 0; sector < VEE_NUM_SECTORS; sector++)
{
    /* Erase sector */
    ret = R_VEE_Erase(sector);

    /* Check result */
    if (VEE_SUCCESS == ret)
    {
        ...
    }

    while (VEE_READY != R_VEE_GetState())
    {
        /* Wait for erase to finish */
    }
}

/* VEE is empty */
```

3.6 R_VEE_ReleaseState

After a successful read this function sets the VEE State to VEE_READY so that VEE operations can continue.

Format

```
uint8_t R_VEE_ReleaseState(void);
```

Parameters

none

Return Values

VEE_SUCCESS: *Successful, state released*
VEE_FAILURE: *Can only release state when state is VEE_READING*

Properties

Prototyped in file “r_vee.h”

Description

This function attempts to release the state of the VEE so that other VEE operations can occur. This function is necessary to call after the user has read a record from the VEE using the R_VEE_Read() function. For more information on why this is required please refer to Section 2.13. This function should only be called after a successful read. If the user uses this function after a VEE write, defrag, or erase then the function will return VEE_FAILURE.

Reentrant

Yes.

Example

```
vee_record_t example_record;
uint8_t ret;

/* We want to find VEE Record 1 */
example_record.ID = 1;

/* Search VEE for record */
if (VEE_SUCCESS == R_VEE_Read(&example_record))
{
    /* Read data and use it */
    ...
}

/* Release state so other VEE operations can occur */
ret = R_VEE_ReleaseState();
```

4. Demo Workspace

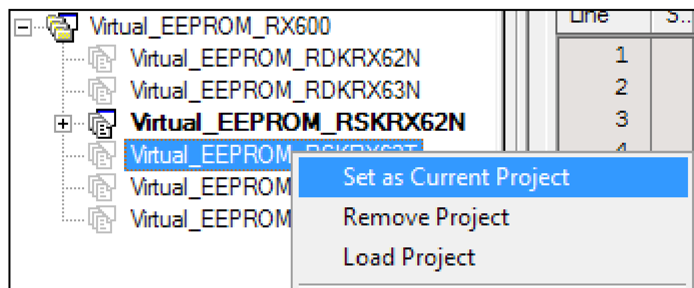
4.1 Configuring the Workspace for your Board and MCU

The HEW workspace that comes packaged with this application note has a project for each supported Renesas development board. This version of the VEE package includes projects for the following boards:

- RSKRX62N
- RSKRX62T
- RSKRX63N
- RSKRX630
- RDKRX62N
- RDKRX63N

The only code that changes between these projects is the board support code that is used along with the demo and VEE code. To choose a project follow these steps:

1. Open the HEW workspace
2. Right-click on the project you wish to load in the navigation pane (by default on left) and click 'Set as Current Project'.



3. You can now build and execute the demo.

4.2 Configuring the r_bsp Package

The VEE API code and demo workspace use the r_bsp package for startup code, board support code, and for getting MCU information. The r_bsp package is easily configured through the *platform.h* header file which is located in the r_bsp folder. To configure the r_bsp package, open up *platform.h* and uncomment the definition for the board you are using. For example, to run the demo on a RDKRX62N board, the user would uncomment the 'PLATFORM_BOARD_RDKRX62N' macro and make sure all other board macros are commented out.

```
/*********************************************
DEFINE YOUR SYSTEM
**********************************************
//#define PLATFORM_BOARD_RSKRX610      (1)

//#define PLATFORM_BOARD_RSKRX62N      (1)
//#define PLATFORM_BOARD_RSKRX62T      (1)
#define PLATFORM_BOARD_RDKRX62N      (1)

//#define PLATFORM_BOARD_RSKRX630      (1)
//#define PLATFORM_BOARD_RSKRX63N      (1)
//#define PLATFORM_BOARD_RDKRX63N      (1)
```

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	Jul.15.11	—	First edition issued
1.50	Jan.03.12	—	Added support for RX63x Group. Revised document to reflect different location of VEE Sector configuration definitions. Other minor changes due to code being updated to be compliant with latest coding standard.

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 part number, confirm that the change will not lead to problems.
   — The characteristics of MPU/MCU in the same group but having different part numbers may differ because of the differences in internal memory capacity and layout pattern. When changing to products of different part 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 sheets 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

© 2011 Renesas Electronics Corporation. All rights reserved.

Colophon 1.1