R8C, M16C, M32C

Virtual EEPROM

1. Abstract

Since the internal flash of a Renesas R8C/M16C/M32C microcontroller can be erased and programmed in-circuit, an application has been created in order to eliminate the need for an external EEPROM in a target system. This *Virtual EEPROM* implementation as it is called uses the highly erasable / programmable *Data flash* area that is available on all newer R8C/M16C/M32C MCUs. Specifically, many of these MCUs allow for a minimum 10,000 erase/write cycles of the Data flash blocks.

The “Virtual EEPROM” is a firmware implementation created by Renesas Technology to make use of the high erase/write cycle Data flash block technology. The Virtual EEPROM API developed and described here does not utilize any additional flash hardware in order to implement the Virtual EEPROM feature.

2. Theory of Virtual EEPROM

In many applications, the Virtual EEPROM implementation may remove the need for an external EEPROM that holds static values such as calibration data, periodically updated values such as time stamp data or system operation values such as user settings.

2.1 Understanding flash Specifications

To understand the Virtual EEPROM implementation, you must first understand the M16C’s internal flash requirements. First, the only way to erase data in a flash block is to erase the entire block at once. Second, flash data can only be written in Word (16-bit) units for M16C/M32C or Byte (8-bit) units for R8C. Once a Word (or Byte) is written to a specific address, that address cannot be written to again until the entire block is erased. Overwriting an already programmed location before it is erased is not recommended and can reduce the life of the flash block.

Another point to consider is that erasure of an entire flash block takes considerably longer than the time to program a Word(s) or Byte(s). Therefore, it is best to fill the entire block before performing an erase cycle in order to minimize disruption to normal system operations and care should be taken in designing in appropriate timing for erase cycles.

Also, a failure during a flash operation is most likely to occur when trying to erase the flash block. This is because over repeated use of the flash (erasing and re-programming), the voltage threshold of the individual flash cells themselves become harder to bring back to a default (or blank) state. Therefore, it is advantageous to program as much of the flash block as possible before erasing it in order to maximizes the total life of the flash.

And finally, the individual cells of a flash block have a finite number of erase/write cycles that can be performed on them. In addition to the user flash area which allows 100 or 1,000 erase/write cycle (depending on version), R8C, M16C, and M32C MCUs all have versions available with flash blocks that can be rated up to 10,000 e/w cycles. (these parts have #D7? Suffix). These new flash blocks are referred to as *Data flash* and will be used as the storage location for the Virtual EEPROM implementation.

2.2 Basic Theory of Virtual EEPROM

This section will discuss how to maximize the storage capability and efficiency of the Data flash blocks.

First, the most important requirement for Virtual EERPOM is that there are two identical sized flash blocks to be used. This is so as one becomes full, the necessary contents can be copied into the other flash block (which should be kept blank). After the contents have been successfully copied, that full block can then be erased. This eliminates the need for a temporary RAM buffer if we had to write the contents back into the same flash block. Also more importantly, this provides a fail safe for unexpected power failures while switching over to the other flash block because the original data is never destroyed (erased) until the copying is completed successfully. If a power failure does occur, your last known values are still in the original flash block to be used after system operation is restored.

Another important characteristic is to only update the data that has changed. Take for example if you had a 2k byte block of flash and you only needed to hold a total of 128 bytes at a time. Also, you periodically change only 10 bytes of that total 128 bytes. Given the fact that the flash memory has a finite number of erase cycles, it would make sense that you could get more life out of the flash if you only wrote the 10 new byte to flash that need to be changed instead of the entire 128 bytes every time. Also If for example the first 32 bytes where calibration data that never needed to be changed, there would be no reason to re-write that data back to flash if you were only changing another portion of that 128 bytes of data.

This concept of breaking your data up into smaller sections based on how frequently it will need to be changed is another aspect of using Virtual EEPROM. Some may find it easier to work with non-volatile data that has been grouped by the API as opposed to just storing to individual EEPROM addresses.

The source code for the Virtual EEPROM implementation makes use of a simple Flash API that takes care of the basic chip level flash programming operations. The actual erasing and writing of the flash is done by a separate software module called the Flash API. There exists flash API modules to support R8C, M16C or M32C devices that contain Flash Data Blocks suitable for this Virtual EERPOM implementation. Figure 1 shows the relationship between the different layers of hardware and firmware.

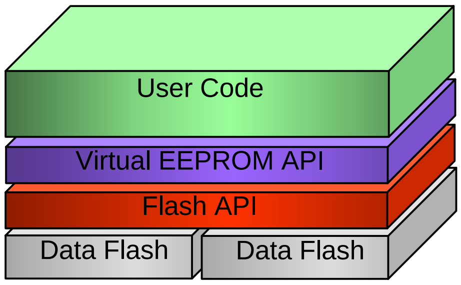


Figure 1 – Software Layers

3. Virtual EEPROM API

The Virtual EEPROM method of saving data to flash can be implemented in many ways. This application note will describe a very versatile method that Renesas has developed and is available to its customers.

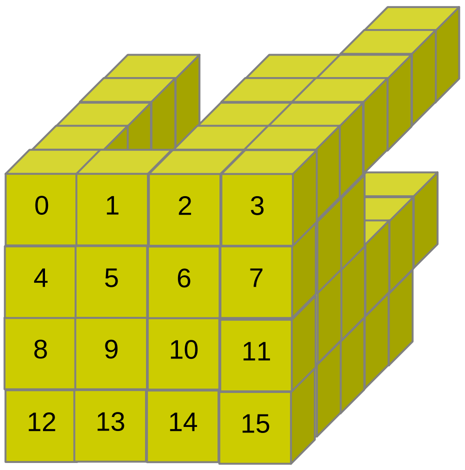
There are different implementations of the Virtual EEPROM concept because not all users have the same requirements when dealing with non-volatile data. Some may value the number of times they can rewrite a flash block and be willing to group data together (in an array) in order to save overhead associated with each EEPROM entry (also saved to that flash area). Other users may decide they would rather have the ability to store and retrieve data on a byte by byte basis like they did when they were using an EEPROM. That type of implementation would require much more overhead.

The following descriptions of the Virtual EERPOM implementations will help you decide which implementation would best fit your application.

3.1 Virtual EEPROM Overview

The best way to think about the Renesas Virtual EEPROM implementation is like a file cabinet with a set number of drawers. Each drawer can contain as much or as little data as you want, and its size is not dependant on any other drawer in the cabinet.

Furthermore, you may change the size of data you are storing in that drawer dynamically in your application and the amount of flash needed to hold your data will be sized according as well.



Think of the Virtual EEPROM as Cabinet where each drawer has an individual integer label

The following are some important aspects of the implementation to keep in mind.

- The maximum number of records (drawers) you may have at one time is set at compile time. This value may be anything from 1 to 244.
- The record (drawer) label you use to reference a particular record will be an integer value from 0 to MAX_RECORD_LABEL.
- A record label area is not allocated until the first time it is used. For example, if you define the maximum number of record labels to be 4, yet over the life of your application you only use 2 of them, no flash area is ever allocated for those 2 unused records.
- The size of an individual record is set during run time and may change.
- The maximum size of a record is 256 WORDS (or 512 bytes).
- The size of the data being stored in a drawer is always specified in WORD (2 byte) values. So if you only need to store 9 bytes, you would set the drawer size to 5 WORDS and simply ignore the last byte at the end.

3.2 Configuring the Header file Options

Configuring the API requires modifying the #define values found in the file “virtee_variable.h”. These values are as follows:

3.2.1 API_BLOCK_HI

This specifies the address flash block to be used for the virtual EEPROM storage area. This value must be assigned to match the valid values define in the flashAPI.h file. The “HI” portion of the name no longer has any mean as it did in previous versions.

Example:

```
#define API_BLOCK_HI    BLOCK_A
```

3.2.2 ADDR_BLOCK_HI

This specifies the address value of the beginning of the API_BLOCK_HI flash block to be used for the virtual EEPROM storage area. The address value must be expressed in hex with a leading 0 and an ‘h’ at the end.

Example:

```
#define ADDR_BLOCK_HI (0f800h)
```

3.2.3 API_BLOCK_LO

This specifies the lower address flash block to be used for the virtual EEPROM storage area. The value is should be assigned to must match the valid values define in the flashAPI.h file. The “LO” portion of the name no longer has any mean as it did in previous versions.

Example:

```
#define API_BLOCK_LO    BLOCK_B
```

3.2.4 ADDR_BLOCK_LO

This specifies the address value of the beginning of the API_BLOCK_LO flash block to be used for the virtual EEPROM storage area. The address value must be expressed in hex with a leading 0 and an ‘h’ at the end.

Example:

```
#define ADDR_BLOCK_LO (0f000h)
```

3.2.5 BLOCK_SIZE

This specifies the individual size of each flash block being used for the virtual EEPROM storage area. Note that both virtual EEPROM flash blocks must be the same size. This value is expressed in Words (2 Bytes).

Example:

```
#define BLOCK_SIZE (1024)   /* 2k flash blocks */
```

3.2.6 NV_FAR

On some M16C devices, one of the virtual EEPROM storage area flash blocks may be located in the *far* (above 64KB addresses) area of the device. Accessing far memory in the M16C requires more code and time than accessing *near* (below 64KB addresses) memory. In order to create more efficient and faster code, the following #if should be set to 1 (near memory only) unless one of the virtual EEPROM storage area flash blocks is at or above the address 0x10000.

Example of near only addresses:

```
#if 1
#define NV_FAR _near  /* if both flash blocks are "near", use this
                       (<64k on M16C, or anywhere on M32C, R8C) */
#else
#define NV_FAR _far   /* if either flash block is "far", use this
                         (>64k on M16C) */
#endif
```

3.2.7 MAX_RECORD_LABEL

This value sets the maximum number for virtual EEPROM records that can be saved at one time. Since the record labels must be a value between 0 and MAX_RECORD_LABEL, the maximum number of records you can have at any one time will be MAX_RECORD_LABEL + 1.

An example of allowing a maximum of 9 records at once (labeled 0 – 8) would be:

```
#define MAX_RECORD_LABEL 8
```

3.3 Default Configuration Values

The following table shows the default values for each of these parameters for the MCUs that support the Virtual EEPROM API. If the device you are using is in this table, it is recommended to use these values for you implementation.

	API_BLOCK_HI	ADDR_BLOCK_HI	API_BLOCK_LO	ADDR_BLOCK_LO	BLOCK_SIZE	NV_FAR
M16C/26	BLOCK_A	(0f800h)	BLOCK_B	(0f000h)	(1024)	#if 1
M16C/26A	BLOCK_A	(0f800h)	BLOCK_B	(0f000h)	(1024)	#if 1
M16C/28	BLOCK_A	(0f800h)	BLOCK_B	(0f000h)	(1024)	#if 1
M16C/29	BLOCK_A	(0f800h)	BLOCK_B	(0f000h)	(1024)	#if 1
M16C/62P	BLOCK_1	(0fe000h)	BLOCK_A	(0f000h)	(2048)	#if 0
M16C/6N4	BLOCK_1	(0fe000h)	BLOCK_A	(0f000h)	(2048)	#if 0
M16C/6N5	BLOCK_1	(0fe000h)	BLOCK_A	(0f000h)	(2048)	#if 0
M32C/85	BLOCK_1	(0ffe000h)	BLOCK_A	(0f000h)	(2048)	#if 0
R8C_12	BLOCK_B	(02800h)	BLOCK_A	(02000h)	(1024)	#if 1
R8C_13	BLOCK_B	(02800h)	BLOCK_A	(02000h)	(1024)	#if 1
R8C_15	BLOCK_B	(02800h)	BLOCK_A	(02400h)	(512)	#if 1
R8C_17	BLOCK_B	(02800h)	BLOCK_A	(02400h)	(512)	#if 1

3.4 Modifying the memory sections

R8C/M16C/M32C starter kits are distributed with NC30 compiler start-up files configured for the device for that kit. Some startup files have the Data flash area defined as a constant data section for ‘C’ programming. It is necessary to remove that section definition from the startup file in order to use the Virtual EERPOM source files.

The following is an example of how to remove the constant data section from the section definition file **sect30.inc**. Simply comment out the section definitions by placing a semi-colon (;) at the beginning of each line.

```
;---------------------------------------------------------------
; Near ROM data area. For "near const".
; Near ROM is all ROM below adress 10000h
; Data flash is located from 0F000h to 0FFFFh.  To access this
; flash the user program must enable access (set pm10 to 1)
;---------------------------------------------------------------
;  .section    rom_NE,ROMDATA
;  .org        0F000H      ; Data flash - 2KB X 2
;rom_NE_top:
;
;  .section    rom_NO,ROMDATA
;rom_NO_top:
```

4. Using the Flash API Functions

4.1 Writing data to the Virtual EEPROM

In order to write data to the virtual EERPOM area, the API requires that you first fill out a ‘C’ structure of type “NV_type”. This data type is defined by the virtual EERPOM header file “virtee_variable.h”. Once a NV_type structure has been filled out with the appropriate information, a pointer to that structure is then passed as a parameter in the virtual EEPROM API.

The values of the individual structure members are as follows:

NV_type.record_label

This member must be filled in with an integer value that will represent which virtual location your data will be held in. Valid values for this member are any value between 0x00 and MAC_RECORD_LABEL.

NV_type.data_size

This member must be filled in with the size of the data you will be writing to the virtual EERPOM area. This value represents the size of your data in WORDs minus 1. For example, if you are writing 10 bytes of data, the value of data_size would be 4 (10/2 – 1 = 4).

It is important to understand that the size of the data you write to a specific EEPROM location (record) does not have to remain constant. For example, if you had previously stored a 20 byte string to record location 7, it is perfectly OK to next time overwrite record 7 with a 30 bytes string. The virtual EEPROM API will dynamically adjust for varying data sizes and only utilize the flash space need to hold that value.

NV_type.pData

This member must be filled in with a pointer to the buffer containing your data to be saved.

NV_Write() Function Return Values

The NV_Write function returns a value to report the success of the operations. The values are as follows:

0: Operation completed successfully

1: The API could not perform the write.

Example of a typical write operation:

```
void write_function( void )
{
    NV_type my_data;
    char my_string[12] = “Hello World”;
    unsigned char result;

    my_data.record = 7;                          /* Save to record location 7 */
    my_data.size = (12/2 – 1);                    /* Size is number of WORDs – 1 */
    my_data.pData = (unsigned char *) my_string;  /* Had to cast as unsigned char pointer
                                                      to avoid compiler warning */
    result = NV_Write(&my_data));                 /* Write data to virtual EE */
    if (result != 0)
    {
        /* ERROR, could not write! */
    }
}
```

4.2 Reading data from the Virtual EEPROM

In order to read data from the virtual EERPOM area, the API uses the same structures that were created for storing the data. You first create and fill out a ‘C’ structure of type “NV_type”. This data type is defined by the virtual EERPOM header file “virtee_variable.h”. Once a NV_type structure has been filled in with the appropriate information, a pointer to that structure is then passed as a parameter in the virtual EEPROM API.

The values of the individual structure members are as follows

NV_type.record_label

This member must be filled in with an integer value that will represent which virtual location your data is being held in. Valid values for this member are any value between 0x00 and MAX_RECORD_LABEL.

If you attempt to read a record location that doesn't exist, the NV_Read function will return a value of 1.

NV_type.data_size

This member does not need to be initialized before the API call.

After the NV_Read function returns, this member will contain the size of the data that was just read from the record location specified. This value represents the size of your data in WORDs minus 1. For example, if you just read 10 bytes of data, the value of data_size will be 4 (10/2 – 1 = 4).

NV_type.pData

This member does not need to be initialized before the API call.

After the NV_Read function returns, this member will contain a pointer to the data for the record location specified. The memory location that pData will be pointing to will be the actual location of the data as is lies in the flash memory area.

NV_Read() Function Return Values

The NV_Read function returns a value to report the success of the operations. The values are as follows:

0: Operation completed successfully

1: Specified record not found

2: Virtual EEPROM flash area is in an invalid state.

Example of a typical read operation:

```
void write_function( void )
{
    NV_type my_data;
    char my_string[12];
    unsigned char result, my_size;
    int i;

    my_data.record = 7;                     /* Read from record location 7 */
    result = NV_Read(&my_data));             /* Write data to virtual EE */
    if (result != 0)
    {
        /* ERROR, could not read that record */
    }
    else
    {
        my_size = (my_data.size + 1) * 2;     /* Convert value to size in bytes */
        for(i=0;i<my_size;i++)
            my_string[i] = my_data.pData[i]; /* Copy data into our buffer */
    }
}
```