Read-While-Write EEPROM Emulation Introduction (FSP Version) (RWWEE, vEE v3.0)

Rev. 1.2, 2021-04-15

Contents

Terms and Definitions

API:	Application Programming Interface
BSP:	Board Support Package
CMSIS:	Cortex Microcontroller Software Interface Standard
CR2:	Configuration Register 2
CRC:	Cyclic Redundancy Check
ECC:	Error Correction Code
EEPROM:	Electrically Erasable Programmable Read-Only Memory
HAL:	Hardware Abstraction Layer
HC:	Host Controller
HW:	Hardware
IOPS:	Input/Output Operations Per Second
LLD:	Low Level Driver
LPA:	Logical Page Address
MCU:	Microcontroller Unit
MM:	Memory Map
OSPI:	Octal SPI
PCB:	Printed Circuit Board
P/E cycle:	Program/Erase cycle
RWW:	Read-While-Write
RWWEE:	Read-While-Write EEPROM Emulation
vEE:	Virtual EEPROM
WIP:	Write In Progress
XIP:	eXecute In Place

1 Background

Many applications require storing small quantities of system related data (e.g., calibration values, device configuration, etc.) in non-volatile memory, so that it can be used or modified and reused even after power cycling the system. EEPROMs are primarily used for this purpose. EEPROMs have the ability to update individual bytes of memory million times over and the programmed locations retain the data over a long period even when the system is powered down.

To eliminate components, save silicon or PCB space and reduce system cost, many MCU vendors implement EEPROM emulation using flash memory. Unlike Flash memory, however, EEPROM does not require an erase operation to free up space before data can be rewritten. Besides, EEPROM has higher endurance than Flash memory. Hence a special software management is required to store data into Flash memory.

Generally, there are two common approaches to emulate EEPROM. The first approach is to keep a copy of the EEPROM data in a dedicated RAM buffer and periodically write the entire contents of the buffer to the program flash. The second approach is to use multiple sectors of program flash to store data using a flash translation layer with less RAM.

The first approach is only suitable for very small EEPROM size, because it may require too much RAM space as the EEPROM size increases, especially for embedded system. But, the second approach also has its inherent defect, which is that user request may be blocked for too long time when cache miss and the flash is in write or erase progress.

Therefore, we proposed this Read-While-Write EEPROM Emulation (RWWEE) to reduce the user operation latency by utilizing the Read-While-Write feature of our Octa RWW NOR flash. Testing results show that RWWEE can greatly improve IO performance, decrease operation latency and reduce CPU utilization, which makes it very suitable for multi-thread and real-time applications.

RWWEE is our third generation EEPROM emulation (vEE v3.0) specially designed for our Octa RWW NOR flash to achieve better performance. For normal NOR flash without RWW feature, it is recommended to choose our previous versions, i.e. vEE v1.0 or vEE v2.0. Please refer to **Section 5.3 vEE v1.0, v2.0 and v3.0 Comparison** for more information.

The rest of this document is organized as follows.

- Chapter 2 "Getting Started" provides quick guide and check list for user to set up development environment and run a demonstration program.
- Chapter 3 "Architecture and Mechanism" introduces the design architecture and detail mechanisms.
- Chapter 4 "Application Interface and Sample Code" describes the function and usage of EEPROM, RWW and OSPI APIs.
- Chapter 5 "Performance Analysis and Comparison" presents some performance analysis and comparison testing results.
- Chapter 6, 7 and 8 describe summary, reference documents and revision history respectively.

2 Getting Started

The RWWEE is developed and tested on Renesas RA6M4 board. User can select the same platform for quick start. The RWWEE support SPI_Nand Flash, QSPI Nor Flash, OSPI Nor Flash, OSPI RWW Flash. The onboard NOR flash does not support RWW feature. You can replace it with an Octa RWW chip (e.g. MX25LW51245G) to achieve better performance.

Although the RWWEE is very portable, there are the prerequisites. If you are using a different platform, please make sure to satisfy the following hardware and software requirements and set related configuration parameters correctly.

2.1 Hardware Requirements

Please check if your hardware environment can satisfy the following hardware requirements.

- **NOR Flash**

  RWWEE can work on SPI_Nand Flash, QSPI Nor Flash, OSPI Nor Flash, OSPI RWW Flash, User should prepare above flash and configure correct settings.

  Although our vEE can work on normal NOR flash by disabling the RWW feature, it is recommended to use Octa RWW flash to get better performance.

- **RAM size**

  RWWEE only requires tens to hundreds of statically allocated bytes according to the different configurations. Dynamic memory allocation is not required.

- **Code size**

  The exact code size depends on platform and compiler. When compiling the code using Atollic ARM tools with -0s optimization, the text section size of object file is about 3 KB. Some optional features may also affect the required ROM space size.

2.2 Software Requirements

Please check if your software environment can satisfy the following software requirements.

- **FreeRTOS embedded Operation System**

  RWWEE requires the following FreeRTOS APIs:

  ✓ `xSemaphoreCreateMutex ()`

  ✓ `vQueueDelete ()`

  ✓ `xSemaphoreTake ()`

  ✓ `xSemaphoreGive ()`

  ✓ `vTaskDelay()`

- ✓ `vTaskSetTimeOutState()`
- ✓ `xTaskCheckForTimeOut()`

If you are using a different RTOS kernel, there should have similar APIs. For detail function description, please visit the FreeRTOS official website.

- **SPI/QSPI/OSPI and HW CRC drivers**

RWWEE requires the following SPI/QSPI/OSPI APIs:

- ✓ `Check_ECC()`
- ✓ `Read()`
- ✓ `Write()`
- ✓ `Erase_Sector()`
- ✓ `Erase_Block()`
- ✓ `Erase_Chip()`

If you are using a different platform, please refer to ***Section 4.3 BSP APIs*** to check if your flash driver can satisfy RWWEE requirements.

If you prefer to use Memory Map (MM) or eXecute In Place (XIP) mode, please also wrap corresponding functionality as the above APIs.

RWWEE also requires the following CRC APIs:

- ✓ `CRC_Open()`
- ✓ `CRC_Close()`
- ✓ `CRC_Calculate()`

If your platform does not support hardware CRC, similar software CRC APIs should be provided and the CRC mutex lock may be removed from the code.

2.3 Parameters Setting

The RWWEE is developed and tested on our MX25LW51245G Octa RWW NOR flash. If using the same flash model, you can just apply the default configuration for quick start. Otherwise, please refer to ***Section 3.5 Parameters and Features*** to configure the RWWEE.

2.4 RWWEE Sample Project

After checking the development environment and configuration parameters, you can try to run the provided RWWEE Sample Project to make sure everything is right, and get more information.

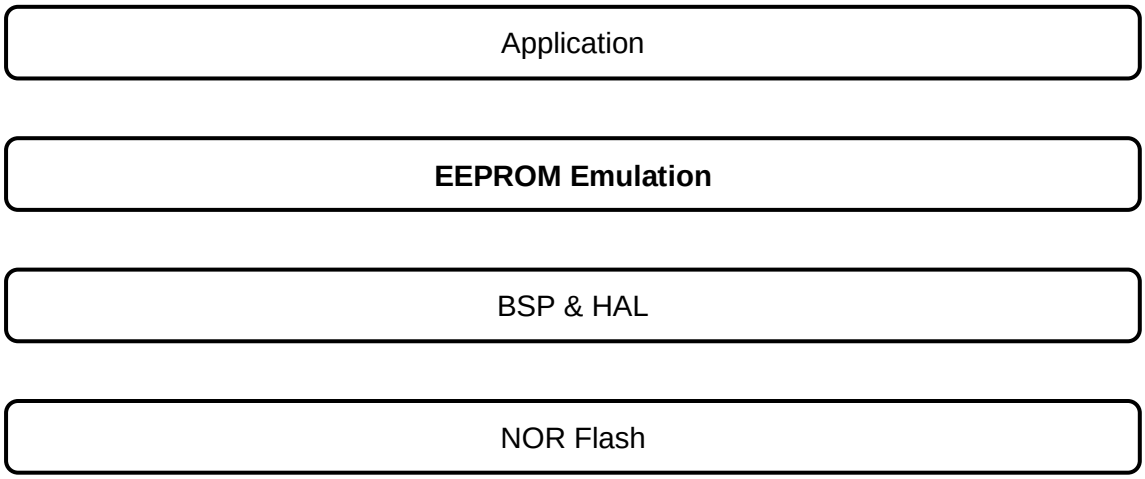
3 Architecture and Mechanism

In this chapter, we will introduce the design architecture and detail mechanisms to explain how RWWEE works. These contents are the most central part of the RWWEE.

3.1 Structure Comparison

EEPROM emulation is a flash translation layer which emulates EEPROM on flash memory. The system architecture of legacy EEPROM emulation is shown below.

Figure 3-1: Architecture of Legacy EEPROM Emulation

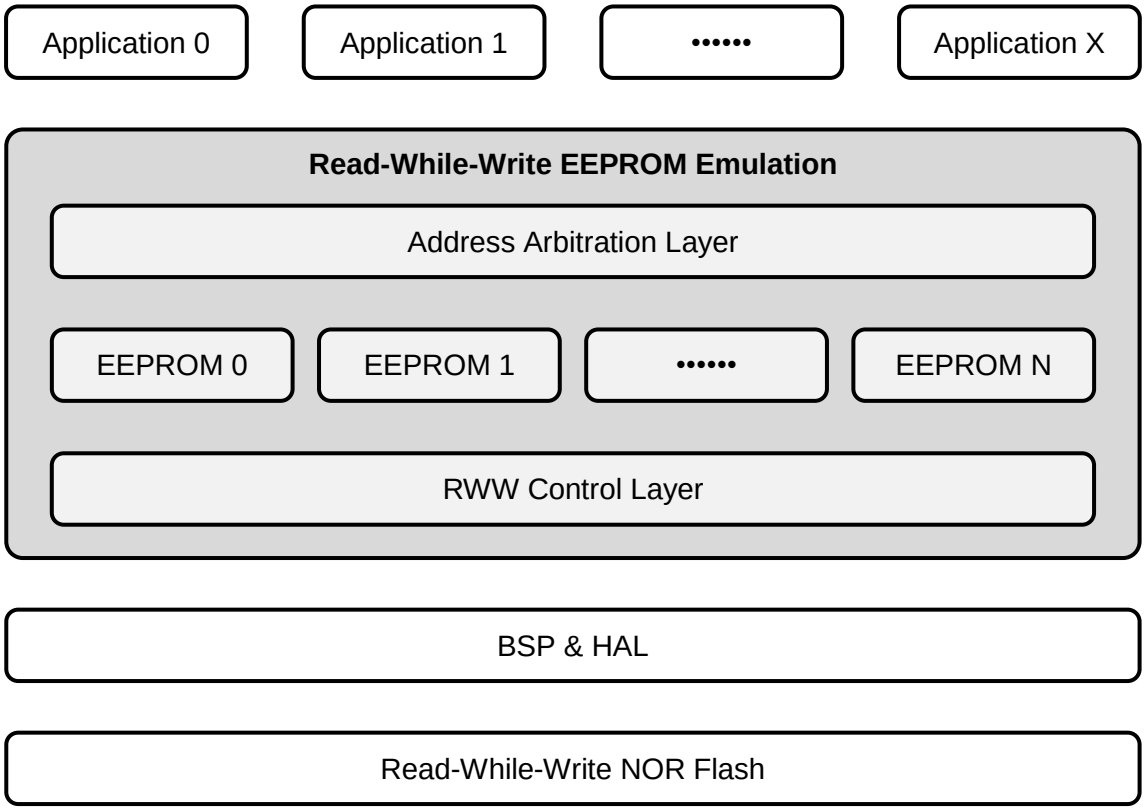


Generally, EEPROM emulation only provides read and write interface for applications, just like the actual EEPROM, and hides the flash erase operation. When there is no enough free space to store further user data, EEPROM emulation needs to erase obsoleted sectors and reclaim as free space. According to flash specifications, since program/write operations take much more time than read operation, user request latency may vary very widely.

If the flash read is available even when program or erase operation is still in progress, it can greatly improve the overall performance. Based on this, RWWEE is designed to reduce user operation latency by utilizing the Read-While-Write feature of our Octa RWW NOR Flash.

The architecture of RWWEE is shown in the figure below.

Figure 3-2: Architecture of RWW EEPROM Emulation



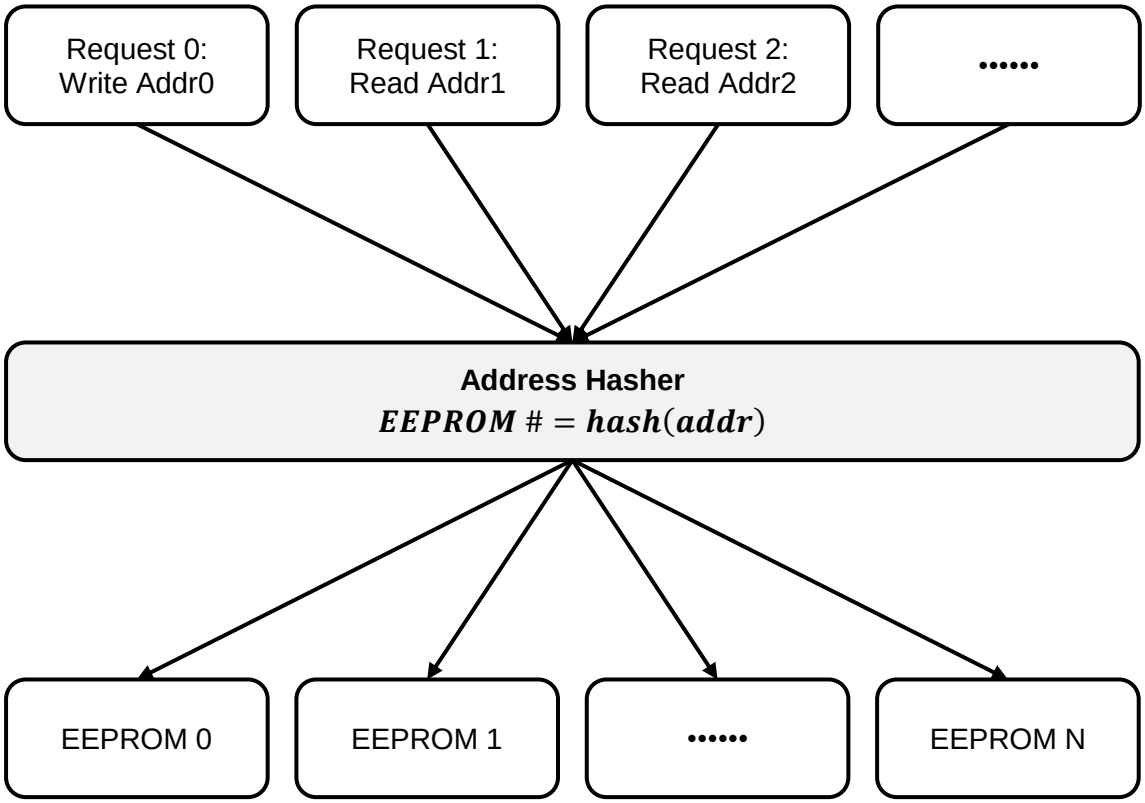
The RWWEE is composed of three layers, i.e., Address Arbitration Layer, EEPROM Emulation Layer and RWW Control Layer. The Address Arbitration Layer distributes user logical address into different banks to increase the RWW probability. The RWWEE can be viewed as multiple identical small EEPROMs and the EEPROM Emulation Layer manages each small EEPROM independently. The RWW Control Layer is responsible for multi-thread management and RWW timing control.

The RWWEE contains two source files, i.e., "*rm_vee.c*" and "*rm_rww.c*". Address Arbitration Layer and EEPROM Emulation Layer correspond to the file "*rm_vee.c*". RWW Control Layer corresponds to the file "rm_*rww.c*". Related structures and parameters are defined in the header file "*rm_vee.h*".

3.2 Address Arbitration Layer

Since flash read operation is only available in idle bank, to increase the RWW probability, we need to place user data in multiple banks. The address arbitration layer is actually a hash function which maintains a fixed mapping relationship between user logical address and each local EEPROM address, as shown in the figure below.

Figure 3-3: Address Arbitration Layer

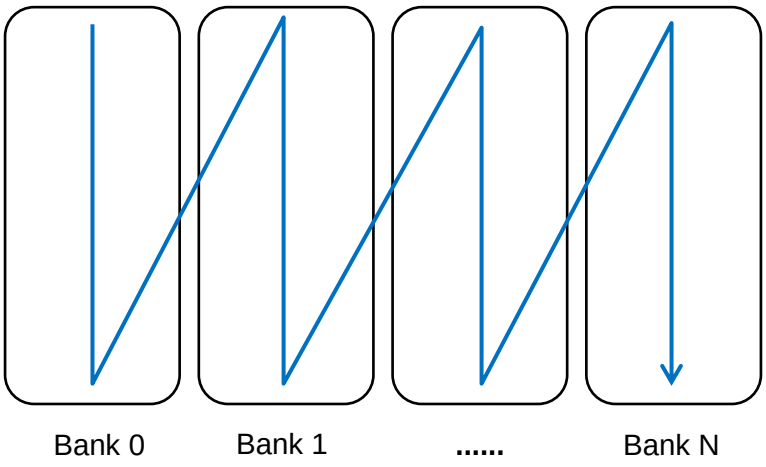


When flash program or erase operation is taking place in a bank, the other idle banks are available for flash read operation. Considering different user data patterns, we provide three kinds of hash methods in RWWEE, i.e., sequential hash, cross-bank hash and hybrid hash. User can change the definition of the *MX_EEPROM_HASH_ALGORITHM* constant to select appropriate hash method in the .xml configuration.

- **Sequential Hash**

  With this hash method, user logical address is always guaranteed to be sequential in each bank, as shown in the figure below. This method is only suitable for user data with obvious read or write pattern. For example, we can place write-oriented data in one bank and read-oriented data in the other banks, so that it can make the most of RWW feature.

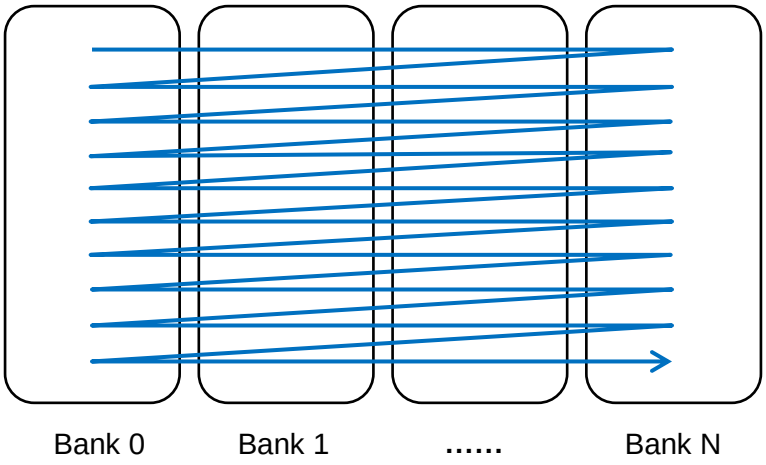
  Figure 3-4: Sequential Hash



- **Cross-bank Hash**

  If user data has no obvious read or write pattern, it is recommended to select the cross-bank hash method. It can distribute sequential logical user page address into different banks to maximize the RWW probability, as shown in the figure below.

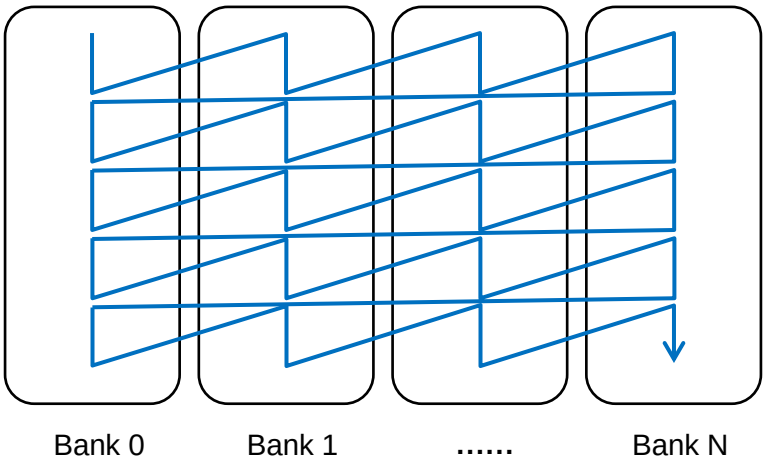
  Figure 3-5: Cross-bank Hash



- **Hybrid Hash**

  This hash method is between sequential hash and cross-bank hash. If the above two hash methods cannot bring obvious performance improvement, you may try this hash method.

  Figure 3-6: Hybrid Hash



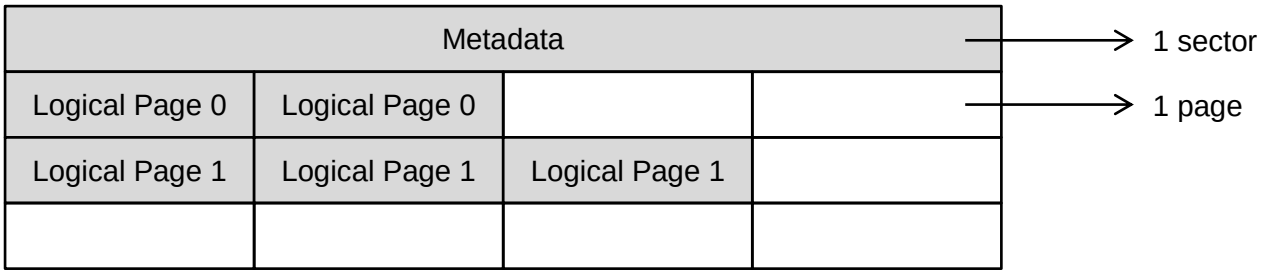
3.3 EEPROM Emulation Layer

The RWWEE can be viewed as multiple identical small subordinate EEPROMs. Or rather, we stack multiple small identical EEPROMs and get a larger EEPROM, and each small EEPROM is managed independently.

3.3.1 Physical Layout

Each small emulated EEPROM requires multiple physical sectors, including one system sector and some user sectors, as shown in the figure below. One sector is the flash erase unit and one page is the flash program unit.

Figure 3-7: Physical Layout



System sector is used to store internal metadata, including magic number, system log, etc. User sectors are used to store user data. There should have at least one free user sector at any time.

Each small emulated EEPROM requires a physically contiguous space with the same size. If the flash supports RWW feature, it is better to place multiple EEPROMs in different banks. For example, user can reserve several blocks for RWWEE in each bank respectively, as highlighted in the figure below. The offset can be different in each bank, but the reserved space size must be the same in each bank.

Figure 3-8: Flash Partition

Bank (16MB)	Block (64KB)	Sector (4KB)	Address Range	
0	0	0	0000000h	0000FFFh
		7	0007000h	0007FFFh
		8	0008000h	0008FFFh
		15	000F000h	000FFFFh
	1	16	0010000h	0010FFFh
		7	0017000h	0017FFFh
		8	0018000h	0018FFFh
		31	001F000h	001FFFFh
	≀			
	255	4080	0FF0000h	0FF0FFFh
		4087	0FF7000h	0FF7FFFh
		4088	0FF8000h	0FF8FFFh
		4095	0FFF000h	0FFFFFFh

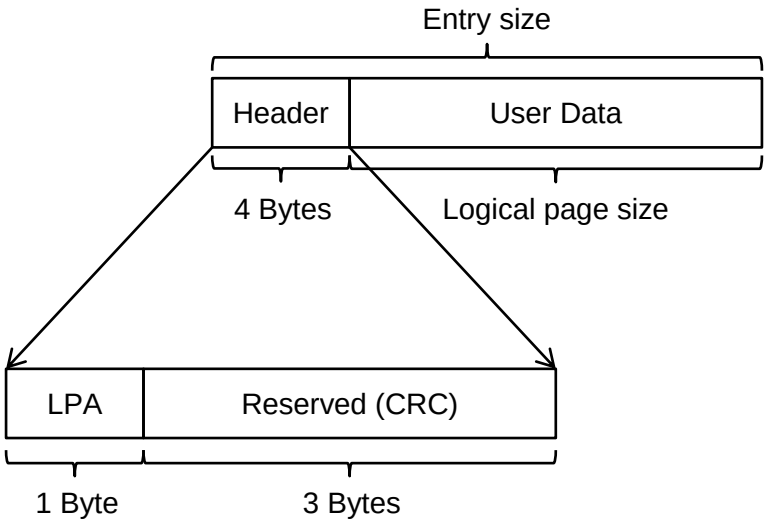
Bank (16MB)	Block (64KB)	Sector (4KB)	Address Range	
1	256	4096	1000000h	1000FFFh
		4103	1007000h	1007FFFh
		4104	1008000h	1008FFFh
		4111	100F000h	100FFFFh
	257	4112	1010000h	1010FFFh
		4103	1017000h	1017FFFh
		4104	1018000h	1018FFFh
		4127	101F000h	101FFFFh
	≀			
	511	8176	1FF0000h	1FF0FFFh
		8183	1FF7000h	1FF7FFFh
		8184	1FF8000h	1FF8FFFh
		8191	1FFF000h	1FFFFFFh

Bank (16MB)	Block (64KB)	Sector (4KB)	Address Range	
2	512	8192	2000000h	2000FFFh
		8199	2007000h	2007FFFh
		8200	2008000h	2008FFFh
		8207	200F000h	200FFFFh
	513	8208	2010000h	0010FFFh
		8199	2017000h	2017FFFh
		8200	2018000h	2018FFFh
		8223	201F000h	201FFFFh
	≀			
	767	12272	2FF0000h	2FF0FFFh
		12279	2FF7000h	2FF7FFFh
		12280	2FF8000h	2FF8FFFh
		12287	2FFF000h	2FFFFFFh

Bank (16MB)	Block (64KB)	Sector (4KB)	Address Range	
3	768	12288	3000000h	3000FFFh
		12295	3007000h	3007FFFh
		12296	3008000h	3008FFFh
		12303	300F000h	300FFFFh
	769	12304	3010000h	3010FFFh
		12295	3017000h	3017FFFh
		12296	3018000h	3018FFFh
		12319	301F000h	301FFFFh
	≀			
	1023	16368	3FF0000h	3FF0FFFh
		16375	3FF7000h	3FF7FFFh
		16376	3FF8000h	3FF8FFFh
		16383	3FFF000h	3FFFFFFh

The read and write unit of EEPROM emulation is an entry. Typically, entry size can be equal to physical page size. As shown in the figure below, each entry contains two parts, i.e., header and user data. In default, header occupies 4 bytes. One byte is for Logical Page Address (LPA) and the other three bytes is for redundant data. If the entry size is 64 bytes, then the logical page size is 60 bytes.

Figure 3-9: Data Packet Format



User can change the entry size in the header file "rm_*rww.h*". If the user average read/write length is much smaller or larger than logical page size, it may lead to poor performance. Therefore, please make sure that the logical page size and average read/write length are approximately equal or at least on the same order of magnitude.

3.3.2 User Data Management

In this section, we will introduce the user data update process. Each logical user page is mapped to an exclusive subordinate EEPROM according to the defined hash method, and each small EEPROM is managed independently. Here we use an example to explain the user data update process.

First, there is no user data and all user sectors are free, as shown below.

Figure 3-10: Initial State of RWWEE

Metadata			
Free Page			

Second, user writes logical page 0. EEPROM emulation packs user data as an entry and writes it to the first physical page of a free user sector.

Figure 3-11: User Data Update Process – Step 1

Metadata			
Logical Page 0			

Third, user writes logical page 1. EEPROM emulation packs user data as an entry and writes it to the first physical page of another free user sector.

Figure 3-12: User Data Update Process – Step 2

Metadata			
Logical Page 0			
Logical Page 1			

Fourth, user updates logical page 1. EEPROM emulation writes it to the next free physical page of the same user sector.

Figure 3-13: User Data Update Process – Step 3

Metadata			
Logical Page 0			
Logical Page 1	Logical Page 1		

Fifth, user updates logical page 1 again. EEPROM emulation continues to write to the same user sector.

Figure 3-14: User Data Update Process – Step 4

Metadata			
Logical Page 0			
Logical Page 1	Logical Page 1	Logical Page 1	

Then, user updates logical page 1 again and the corresponding user sector is full now.

Figure 3-15: User Data Update Process – Step 5

Metadata			
Logical Page 0			
Logical Page 1	Logical Page 1	Logical Page 1	Logical Page 1

After receiving another logical page 1 update request, since current user sector is full, EEPROM emulation needs to switch to another free sector and the current user sector is obsoleted, as shown below.

Figure 3-16: User Data Update Process – Step 6

Metadata			
Logical Page 0			
Logical Page 1	Logical Page 1	Logical Page 1	Logical Page 1
Logical Page 1			

At appropriate time, EEPROM emulation needs to erase the obsoleted sector and reclaim as free sector.

Figure 3-17: User Data Update Process – Step 7

Metadata			
Logical Page 0			
Logical Page 1			

The user data update process seems no obvious different than other EEPROM emulations. Actually, we postpone the flash write and erase operation to increase the RWW probability. For example, we always do the flash erase operation at the end of user request and just send the erase command without waiting flash ready, so that user can read the other idle banks while flash erase is still in progress.

3.4 RWW Control Layer

As mentioned above, RWWEE contains multiple small EEPROMs distributed in different banks and each small EEPROM is managed independently. Therefore, we need an RWW control layer to manage multi-thread operations to maximize RWW probability. The RWW control layer wraps BSP flash APIs and provide a universal flash RWW operation interface for upper layer.

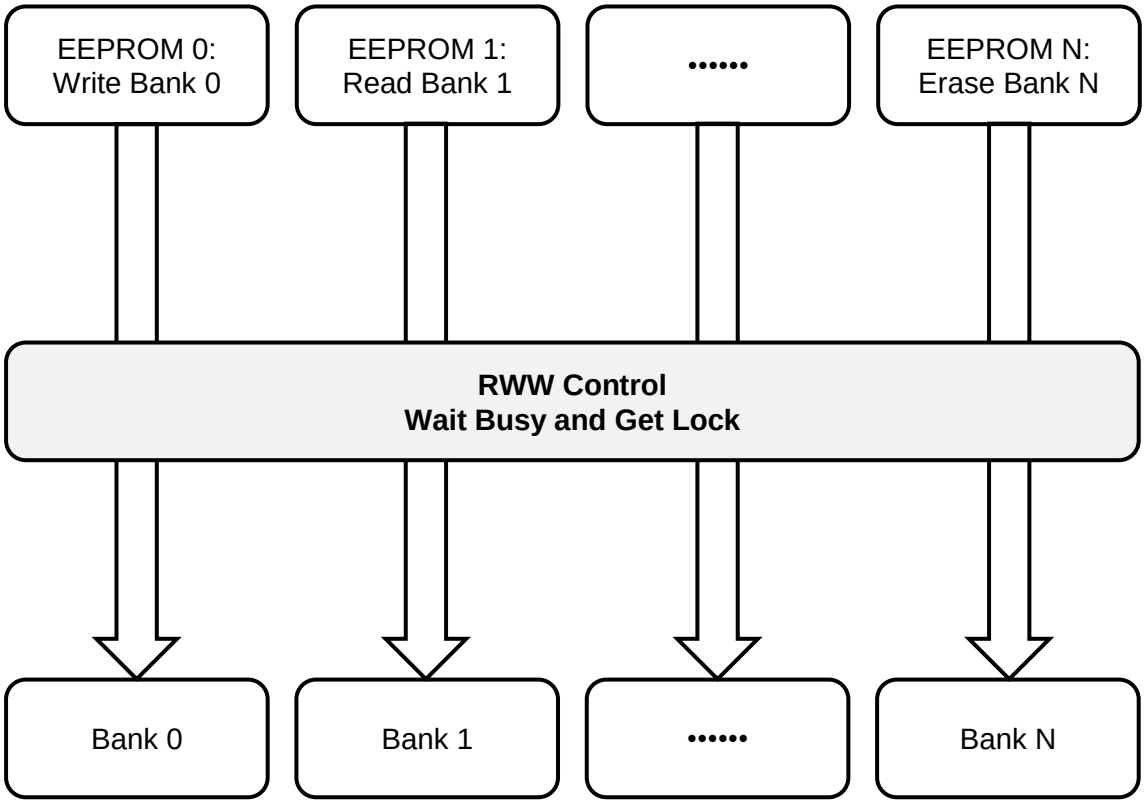
When multiple threads require to access the flash at the same time, the management strategy of the RWW control layer is as follows.

1. If anyone reads the busy bank or writes any bank when the flash is still in busy state, the corresponding thread enters delayed state until the flash becomes ready;

2. Only one thread can get the device mutex lock at any time, and the other threads enter blocked state;

With this management strategy, we can increase the RWW probability and reduce CPU utilization.

Figure 3-18: RWW Control Layer

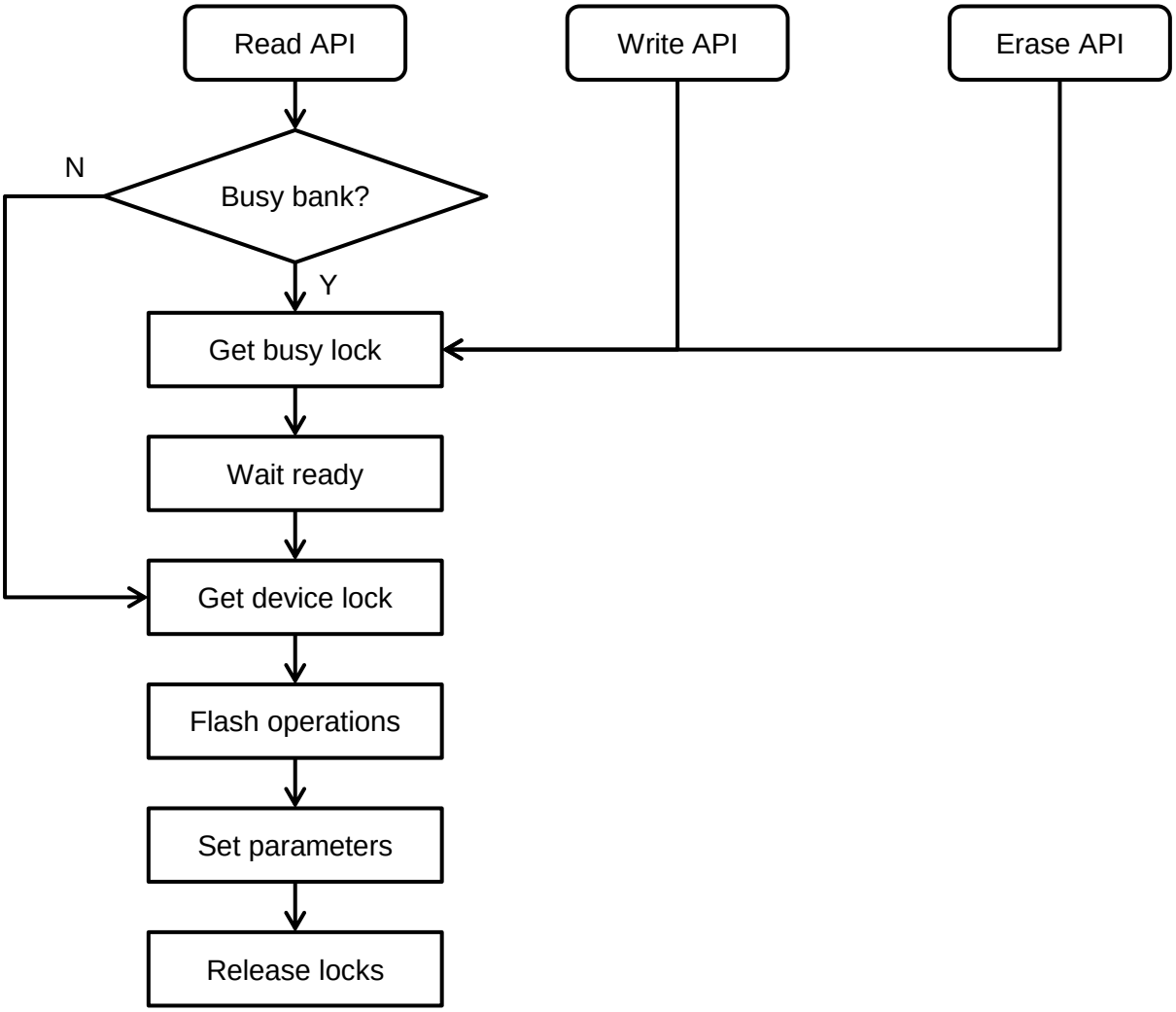


The flow chart of read, write and erase operation of the RWW control layer is shown in the figure below. There are two mutex locks used to control multi-thread and RWW operations. The “Busy Lock” blocks any request that does not meet RWW requirement. The “Device Lock” is responsible to schedule requests meeting RWW requirement and avoid access conflict.

Suppose all user threads have the same priority. With the RWW control layer, user requests meeting RWW requirement have higher possibility to be executed first. However, user thread with higher priority may override this RWW control logic due to the mutex mechanism.

Although multi-thread operations can help to improve the RWW probability, single thread operations can also enjoy great benefits from the RWW feature. RWWEE requires OSPI driver to return immediately after issuing corresponding commands without waiting flash ready, so that upper applications can utilize this flash busy time to read user data from the other idle banks or do something else.

Figure 3-19: Flow Chart of RWW Control



Besides, the RWW control layer provides a universal flash RWW interface for upper applications, such as file system. To benefit from the RWW feature and avoid access conflict, for other applications requiring accessing the flash memory, it is recommended to use the provided universal RWW interface instead of the original BSP or LLD interface.

3.5 Parameters and Features

Because of its high configurability, RWWEE is well adapted to the needs of different customers. We define some critical parameters in the .xml configuration file. User can configure these parameters according to specific requirements and environment.

3.5.1 Flash RWW Feature

Although RWWEE can work on normal flash, it is recommended to use our Octa RWW flash to achieve better performance. User can choose ‘’SPI RWW NOR FLASH’’ in configuration and then source code will auto define the “*MX_FLASH_SUPPORT_RWW*” constant in a common header file to enable the RWW feature.

3.5.2 Flash Specifications

User should check the following flash specification parameters defined in the header file "*rm_vee.h*" and "rm_vee_cfg.h" according to flash datasheet.

Table 3-1: Flash Specifications Parameters

Parameters	Description
MX_FLASH_CHUNK_SIZE	Flash ECC chunk size
MX_FLASH_PAGE_SIZE	Flash page size
MX_FLASH_SECTOR_SIZE	Flash sector size
MX_FLASH_BLOCK_SIZE	Flash block size
MX_FLASH_TOTAL_SIZE	Flash total size
MX_FLASH_BANK_SIZE	Flash bank size
MX_FLASH_BANK_SIZE_MASK	~(MX_FLASH_BANK_SIZE - 1)
MX_FLASH_BANKS	The number of flash banks
MX_FLASH_PAGE_WRITE_TICKS	Typical page write time (tick)
MX_FLASH_SECTOR_ERASE_TICKS	Typical sector erase time (tick)
MX_FLASH_BLOCK_ERASE_TICKS	Typical block erase time (tick)

3.5.3 Physical Layout

As mentioned above, RWWEE can utilize the flash RWW feature to greatly improve overall performance by distributing user data into multiple banks. Therefore, it requires multiple continuous physical spaces in different banks.

First, please set the following parameters to determine the size of EEPROM area of each bank in the header file ""rm_vee_cfg.h". In RWWEE, we use the term "cluster" to indicate a continuous physical area containing a number of physical sectors. Typical, user can set the cluster size equal to the physical block size.

Table 3-2: Flash Layout Parameters

Parameters	Symbol	Description
MX_EEPROM_BANKS	n_{bank}	The number of banks used for RWWEE
MX_EEPROM_CLUSTERS_PER_BANK	$n_{cluster}$	The number of continuous clusters used for RWWEE in each bank
MX_EEPROM_SECTORS_PER_CLUSTER	n_{sector}	The number of sectors per cluster

Second, please set the start physical address of EEPROM area of each bank in the source file "*rm_vee.c*", as shown below. The offset of EEPROM area in each bank can be different.

```
/* EEPROM physical address offset in each bank */
static uint32_t bank_offset[MX_EEPROMS] =
{0x00200000, 0x01200000, 0x02200000, 0x03200000};
```

At last, please set the following parameters to determine the internal data layout in the header file "rm_*vee.h*".

Table 3-3: EEPROM Layout Parameters

Parameters	Symbol	Description
MX_EEPROM_HEADER_SIZE	S_{header}	User data header size. Please just use the default value 4.
MX_EEPROM_ENTRY_SIZE	S_{entry}	The flash read/write unit size. This value should be approximately equal to the average user read/write length.
MX_EEPROM_FREE_SECTORS	n_{free}	The number of additional free user sectors in each cluster. The default value is 1.

Then, the user-visible EEPROM size can be calculated by:

$$EEPROM\ Size = (S_{entry} - S_{header})(n_{sector} - n_{free} - 1)n_{cluster}n_{bank}$$

3.5.4 EEPROM Endurance

After determining all the parameters in the previous section, the average write cycle of RWWEE can be calculated by:

$$Write\ Cycle = ec_{sector}\frac{S_{sector}(n_{sector} - 1)}{S_{entry}(n_{sector} - n_{free} - 1)},$$

where ec_{sector} indicates the typical P/E cycle of single sector.

User can adjust related influence factors to get appropriate endurance strength. For example, if $ec_{sector} = 100000$, $S_{sector} = 4096$, $S_{entry} = 256$, $n_{sector} = 16$ and $n_{free} = 1$, then the average endurance is:

$$Write\ Cycle = 100000 \times \frac{4096 \times (16 - 1)}{256 \times (16 - 1 - 1)} = 1714285$$

3.5.5 Flash Partition

Since the flash may be used for multiple purposes and RWWEE only occupies very small flash space, it is necessary to partition the flash according different applications. In this section, we offer some flash layout examples to user for reference. The flash model is MX25LW51245G. Some flash parameters are defined as below.

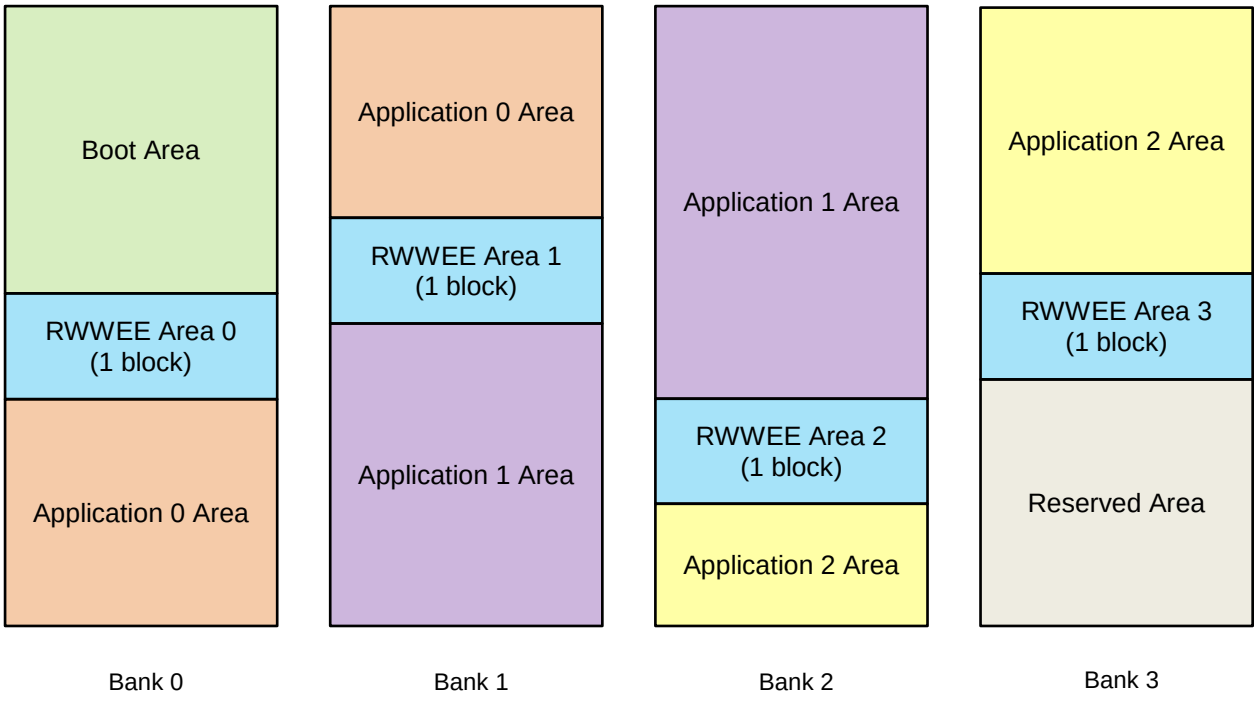
$S_{sector} = 4096$

$ec_{sector} = 100000$

Example I:

User can reserve one physical block for RWWEE in each bank respectively, as shown in the figure below.

Figure 3-20: Flash Partition – Example I



Related parameters are defined as below.

$n_{bank} = 4$

$n_{cluster} = 1$

$n_{sector} = 16$

$n_{free} = 1$

$S_{entry} = 256$

$S_{header} = 4$

Then we can calculate the RWWEE size:

$EEPROM\ Size$

$= (S_{entry} - S_{header})(n_{sector} - n_{free} - 1)n_{cluster}n_{bank}$

$= (256 - 4) \times (16 - 1 - 1) \times 1 \times 4$

$= 14112\ B$

$\approx 13.78\ KB$

The average endurance of RWWEE is:

$Write\ Cycle$

$$= ec_{sector} \frac{S_{sector}(n_{sector} - 1)}{S_{entry}(n_{sector} - n_{free} - 1)}$$

$$= 100000 \times \frac{4096 \times (16 - 1)}{256 \times (16 - 1 - 1)}$$

≈ 1714285

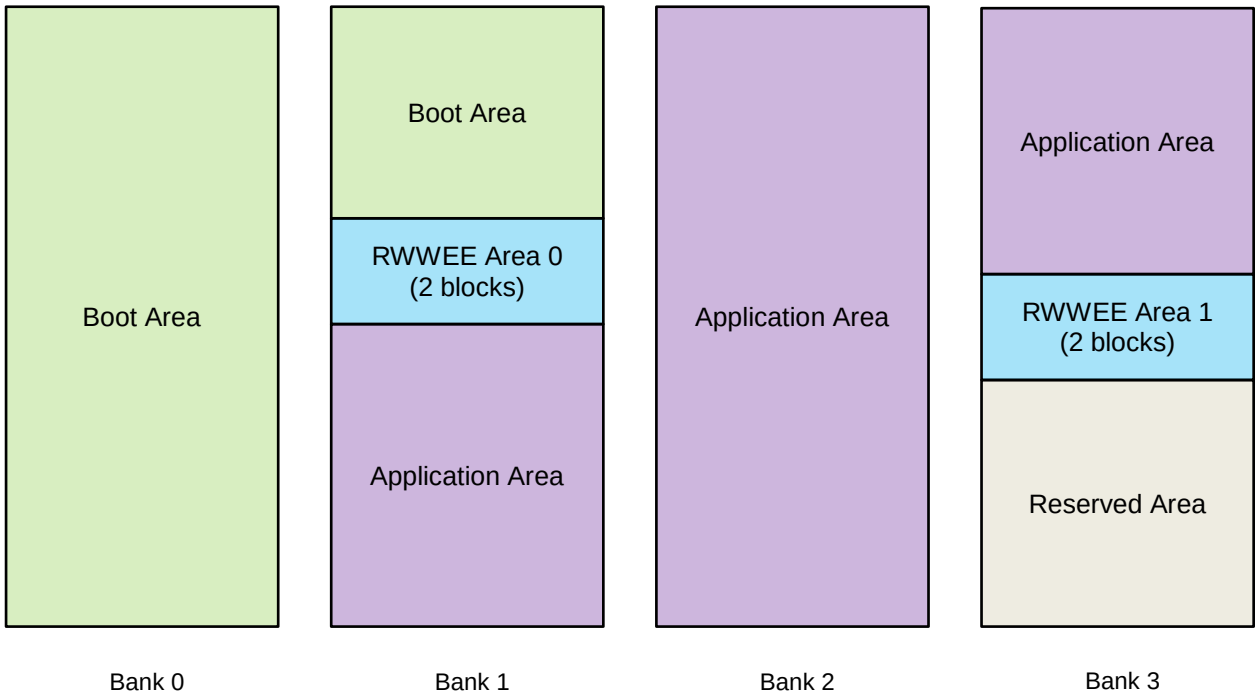
$\approx 1714\ K$

$\approx 1.7\ M$

Example II:

User can reserve two physical blocks for RWWEE in two different banks respectively, as shown in the figure below. Since only two banks are used for RWWEE, the I/O performance may be lower than that of Example I.

Figure 3-21: Flash Partition – Example II



Related parameters are defined as below.

$n_{bank} = 2$

$n_{cluster} = 1$

$n_{sector} = 32$

$n_{free} = 1$

$S_{entry} = 128$

$S_{header} = 4$

Then we can calculate the RWWEE size:

$$EEPROM\ Size$$

$$= (S_{entry} - S_{header})(n_{sector} - n_{free} - 1)n_{cluster}n_{bank}$$

$$= (128 - 4) \times (32 - 1 - 1) \times 1 \times 2$$

$$= 7440\ B$$

$$\approx 7.27\ KB$$

The average endurance of RWWEE is:

$$Write\ Cycle$$

$$= ec_{sector} \frac{S_{sector}(n_{sector} - 1)}{S_{entry}(n_{sector} - n_{free} - 1)}$$

$$= 100000 \times \frac{4096 \times (32 - 1)}{128 \times (32 - 1 - 1)}$$

$$\approx 3306666$$

$$\approx 3229\ K$$

$$\approx 3.15\ M$$

3.5.6 Hash Algorithm

As described in ***Section 3.2 Address Arbitration Layer***, we provide three kinds of hash methods to adapt to different user data patterns. User can change the definition of the *MX_EEPROM_HASH_AlGORITHM* constant in the header file ““rm_vee_cfg.h”, as shown below.

```
#define MX_EEPROM_HASH_CROSSBANK        0    /* Page hash */
#define MX_EEPROM_HASH_HYBRID           1    /* Block hash */
#define MX_EEPROM_HASH_SEQUENTIAL       2    /* Bank hash */

/* Address hash algorithm */
```

```
#define MX_EEPROM_HASH_AlGORITHM        MX_EEPROM_HASH_CROSSBANK
```

3.5.7 Data Integrity

To prevent data corruption during power cycling and other situations, we design some checking mechanisms to protect user data integrity. User can enable corresponding mechanisms according to specific requirement and environment. These features are defined in the header file ““rm_vee_cfg.h”.

Table 3-4: Data Integrity Related Parameters

Mechanisms	Description
MX_EEPROM_PC_PROTECTION	Handle insufficient erase caused by sudden power cycling. (This feature may influence overall performance because we need to write journal to the system sector after each erase operation.)
MX_EEPROM_CRC_HW	Calculate 16-bit CRC for user data and store in the entry header.
MX_EEPROM_READ_RETRIES	The number of read retries.
MX_EEPROM_READ_ROLLBACK	Try to recover the older version of user data if any read error happens.
MX_EEPROM_ECC_CHECK	Check flash ECC status after each read operation. This feature has conflicts with RWW feature.

3.5.8 Wear Leveling

To extend the flash service life, wear leveling mechanism will be triggered when the erase count difference reaching a user-defined threshold. User can change the threshold value in the header file “rm_vee_cfg.h”, as shown below. This feature relies on the background task. If the background thread is not enabled, user can call the background function **`mx_eeprom_background`**() before CPU enters sleep mode.

```
/* Wear leveling interval */
#define MX_EEPROM_WL_INTERVAL           10000
```

3.5.9 Background Task

To improve the response time of user request, RWWEE postpones some time-consuming tasks to the background thread. User can enable this feature by defining the *MX_EEPROM_BACKGROUND_THREAD* constant and setting related parameters in the header file “rm_*rww.h*”, as shown below.

```
/* Background thread */
#define MX_EEPROM_BACKGROUND_THREAD

#ifdef MX_EEPROM_BACKGROUND_THREAD
#define MX_EEPROM_BG_THREAD_PRIORITY   osPriorityLow /* Background thread priority */
#define MX_EEPROM_BG_THREAD_STACK_SIZE 256       /* Background thread stack size */
#define MX_EEPROM_BG_THREAD_DELAY      10000     /* Background thread delay (ms) */
#define MX_EEPROM_BG_THREAD_TIMEOUT    10        /* Background thread timeout (ms) */
#endif
```

User can also call the background function **`mx_eeprom_background`**() before CPU enters sleep mode.

4 Application Interface

In this chapter, we will introduce related APIs of each layer. Generally, upper applications may only use the EEPROM APIs to read and write user data. For other applications requiring accessing flash bypass RWWEE, it is recommended to use the provided generic RWW APIs to enjoy performance improvement. We also provide the corresponding OSPI driver implementation here to help user adjust their BSP APIs to satisfy RWWEE requirements.

4.1 EEPROM APIs

EEPROM emulation provides the following APIs for upper applications.

- EEPROM read

```
fsp_err_t RM_VEE_Read (mxic_vee_ctrl_t * const p_api_ctrl, uint32_t addr, uint32_t
    len, uint8_t *buf);
```

  There is no special restriction when using the EEPROM read function. User can get the EEPROM size through the **RM_VEE_GetParam()** function or the *MX_EEPROM_TOTAL_SIZE* constant.

- EEPROM write

```
fsp_err_t RM_VEE_Write (mxic_vee_ctrl_t * const p_api_ctrl, uint32_t addr, uint32_t
len, uint8_t *buf);

fsp_err_t RM_VEE_WriteBack (mxic_vee_ctrl_t * const p_api_ctrl);

fsp_err_t RM_VEE_SyncWrite (mxic_vee_ctrl_t * const p_api_ctrl, uint32_t addr,
uint32_t len, uint8_t *buf);
```

  Here we provide three write functions. The first function **RM_VEE_Write()** just write user data to internal buffer and return back. The data buffer will be flushed to flash in the next user write call, or in background thread, or when user calls the **RM_VEE_WriteBack()** function. The Last function **RM_VEE_SyncWrite()** wraps the above two functions together to realize synchronous write.

- EEPROM flush

```
fsp_err_t RM_VEE_Flush (mxic_vee_ctrl_t * const p_api_ctrl);
```

  This function flushes both user data buffer and internal metadata to the flash. Remember to call this function before shutting down the system.

- Background task

```
void RM_VEE_Background (mxic_vee_ctrl_t * const p_api_ctrl);
```

  This is the background task used to do some time-consuming works during CPU idle cycles. User can also call this function at appropriate time, for example before CPU sleeps.

- Initialization and de-initialization

```
fsp_err_t RM_VEE_Format (mxic_vee_ctrl_t * const p_api_ctrl);

void RM_VEE_GetParam (struct eeprom_param *param);

fsp_err_t RM_VEE_Open (mxic_vee_ctrl_t * const p_api_ctrl, mxic_vee_cfg_t const *
const p_cfg);

void RM_VEE_Close (mxic_vee_ctrl_t * const p_api_ctrl);
```

  As their names imply, these functions are typically used on initialization or cleanup. Remember to call **RM_VEE_Close**() function before system shutdown to prevent insufficient flash write or erase.

User can refer to the following sample code to operate the EEPROM emulation.

```
/* Format RWWEE */
ret = RM_VEE_Format(g_mx_vee0.p_ctrl);
if (ret)
  goto err;

/* Init RWWEE */
ret = RM_VEE_Open(g_mx_vee0.p_ctrl);
if (ret)
  goto err;

/* Read RWWEE */
ret = RM_VEE_Read(g_mx_vee0.p_ctrl, addr, len, buf);
if (ret)
  goto err;

/* Update buffer */
;

/* Write RWWEE */
ret = RM_VEE_Write(g_mx_vee0.p_ctrl, addr, len, buf);
if (ret)
  goto err;

/* Do some more read/write operations */
;

/* Flush data */
ret = RM_VEE_Flush(g_mx_vee0.p_ctrl);
if (ret)
  goto err;

/* De-init RWWEE */
RM_VEE_Close(g_mx_vee0.p_ctrl);
```

5 Performance Analysis and Comparison

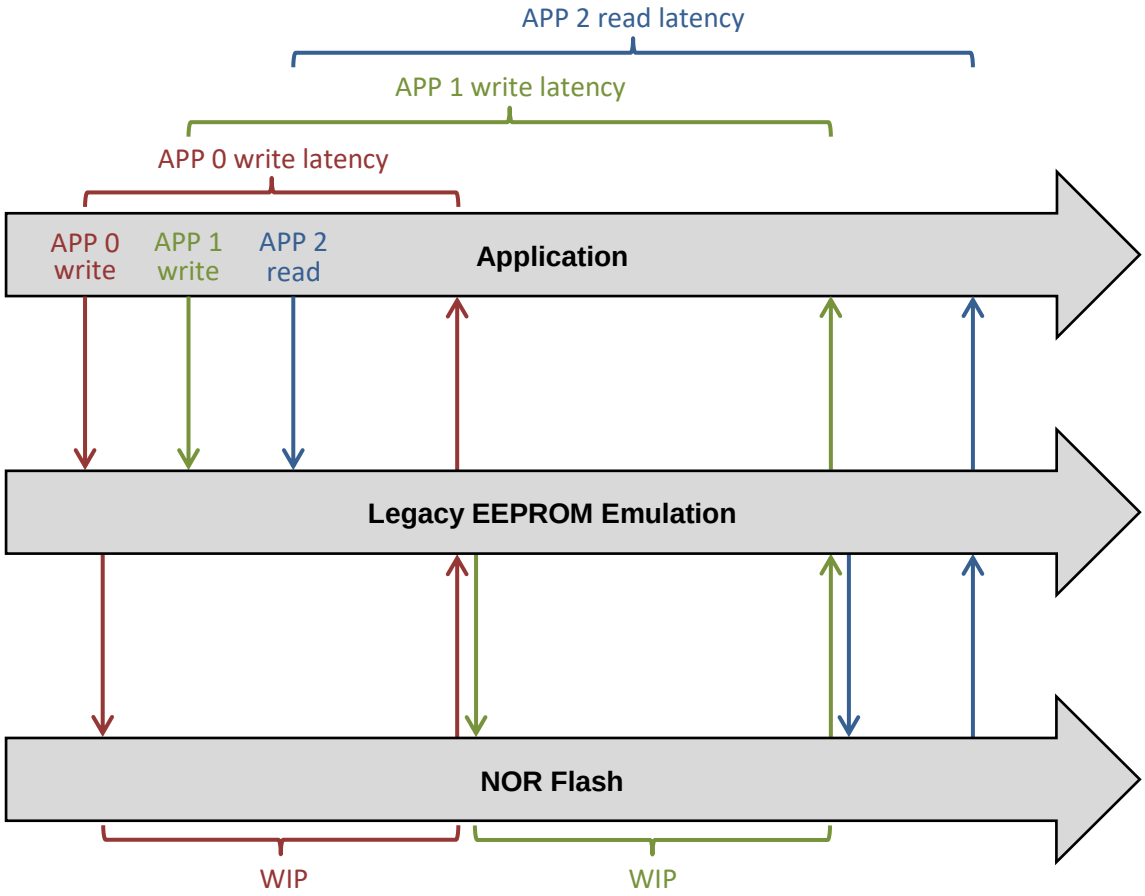
In this chapter, we try to explain why RWWEE can achieve better performance than other solutions, from both theoretical analysis and actual testing. We also provide an overall comparison of vEE v1.0, v2.0 and v3.0 to help user to select the most appropriate solution.

5.1 RWW Latency Analysis

Before presenting the performance comparison testing results, we try to explain why RWWEE can reduce the user operation latency.

The latency analysis of legacy EEPROM emulation is shown in the figure below. Suppose there are three threads requiring accessing the emulated EEPROM. The EEPROM emulation executes and responds application requests in the order of requests arrival. Therefore, current application thread needs to wait until the execution of previous request completes and the flash becomes ready.

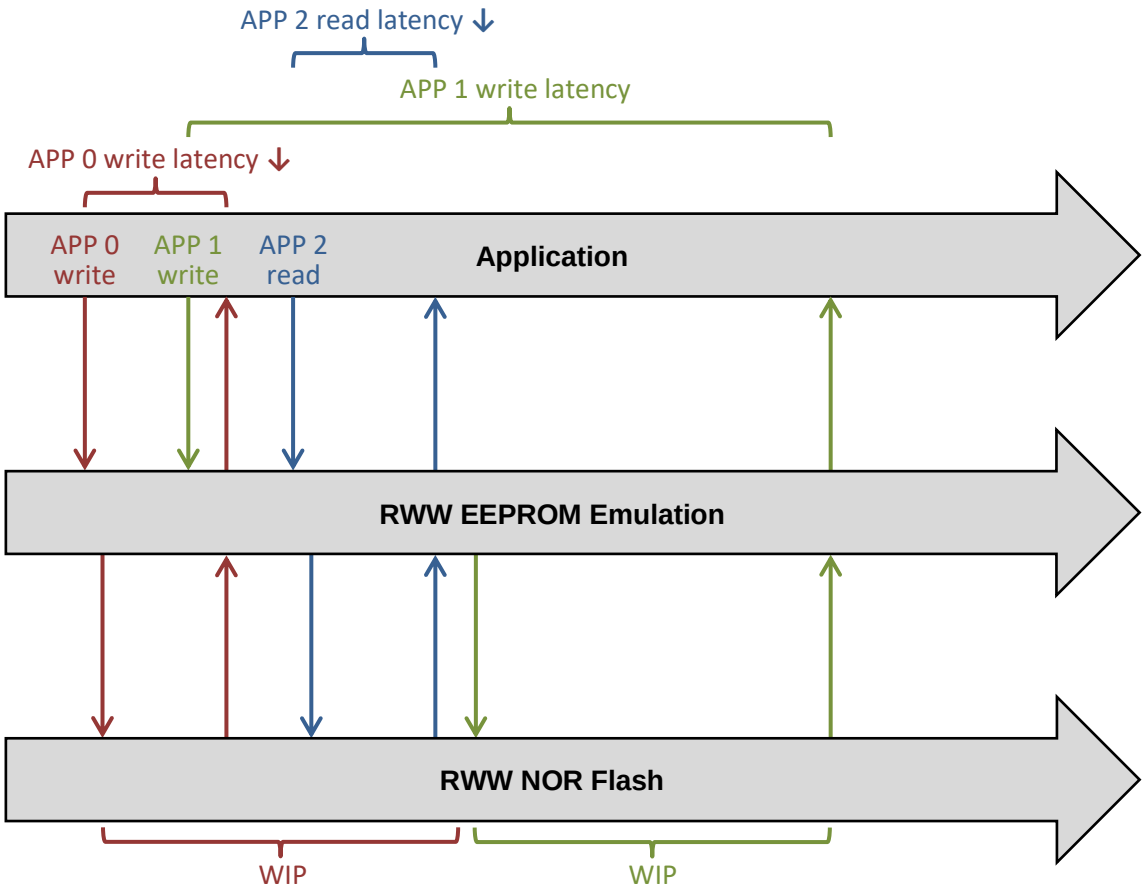
Figure 5-1: Latency Analysis of Legacy EEPROM Emulation



Unlike legacy EEPROM emulation, RWWEE can still accept and execute some read requests while the flash is in program/erase process. As shown in the figure below, APP 2 can read the RWWEE when the write request of APP 0 is still in progress. The response time of RWWEE is much shorter than that of legacy EEPROM emulation. The testing results shown in the next section can support the theoretical analysis.

However, not all read requests can be served in this situation. If the storage location of APP 0 data and APP 1 data is in the same bank, APP 2 needs to queue to access the EEPROM. Therefore, please select appropriate address hash method according to your requirements and environment.

Figure 5-2: Latency Analysis of RWW EEPROM Emulation



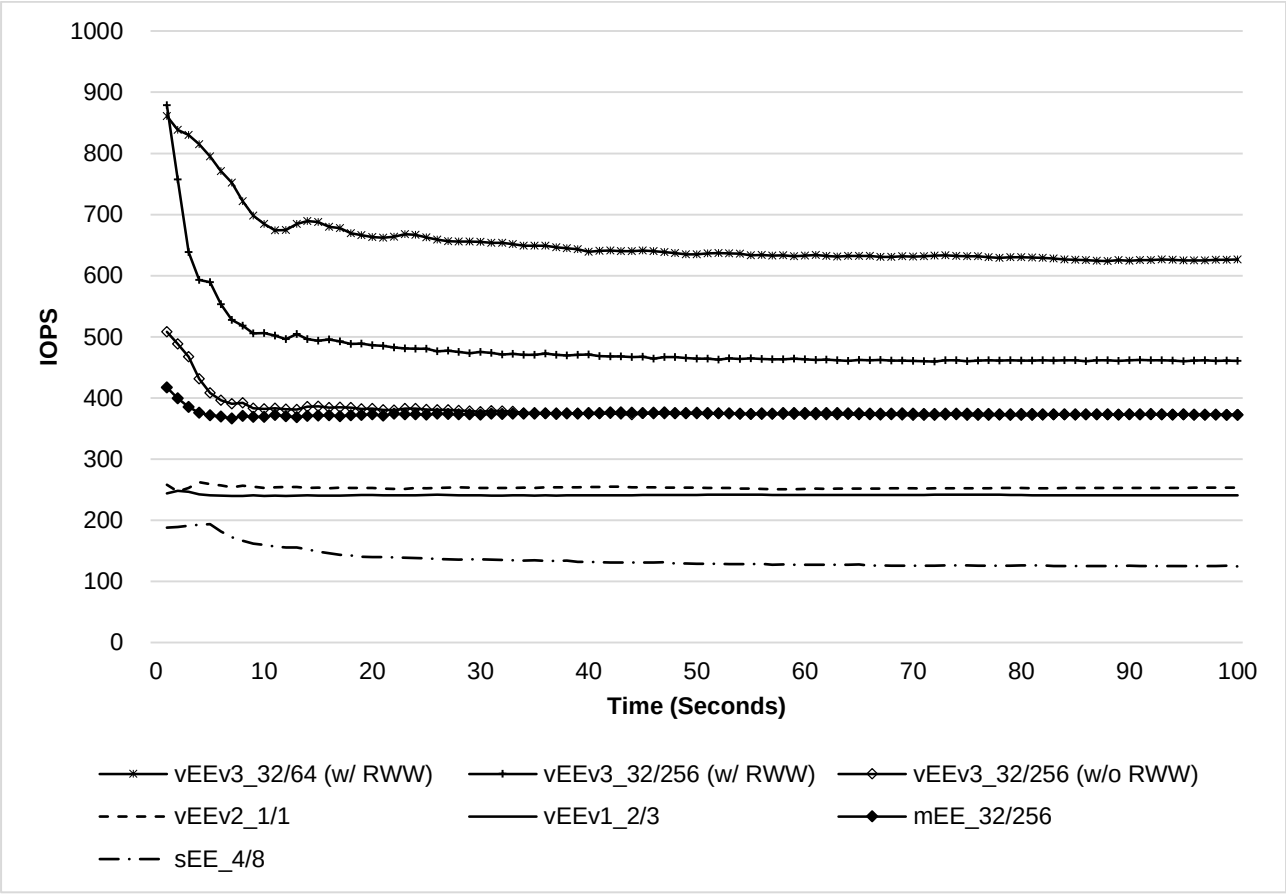
5.2 Performance Testing Results

We have evaluated the performance of several different EEPROM emulations, including vEE v1.0, vEE v2.0, vEE v3.0 (RWWEE), mEE and sEE. The vEE v1.0, vEE v2.0 and vEE v3.0 are all developed by Macronix. The mEE and sEE are both open source EEPROM emulations from the Internet.

Since most other EEPROM emulations do not support multi-thread operations, the following test data is collected under single-thread testing environment. Even so, testing results show that RWWEE achieves much better performance improvement than other solutions.

The single thread random read/write IOPS comparison is shown in the figure below. For each label in the chart legend, the number on the left-hand side of the slash (/) sign indicates the average application read/write length, and the number on the right-hand side of the slash (/) sign indicates the read/write unit size of EEPROM emulation.

Figure 5-3: Single Thread Random Read/Write IOPS Comparison (Cumulative Average)



As we can see from the figure above, the RWW feature can bring great IOPS improvement. For vEE v3.0 (RWWEE), the smaller difference between average application read/write length and EEPROM emulation read/write unit size can bring higher IOPS performance. The reason why IOPS decreases over time is that the EEPROM is empty at the beginning of the testing and RWWEE just resets the user data buffer and returns to applications without accessing the flash memory if the address mapping is empty. Besides, compared with vEE v1.0, vEE v2.0 also has higher IOPS.

We also measured the random read/write latency of vEE v3.0 and mEE, as shown in the two figures below. The average application read/write length is set to be 32 bytes, and the read/write unit size of EEPROM emulation is set to be 256 bytes. These higher latency values around 30 milliseconds are caused by internal sector erase operation, while these latency values below 10 milliseconds represent flash read and write operations.

Figure 5-4: Single Thread Random R/W Latency of vEE v3.0 (32/256)

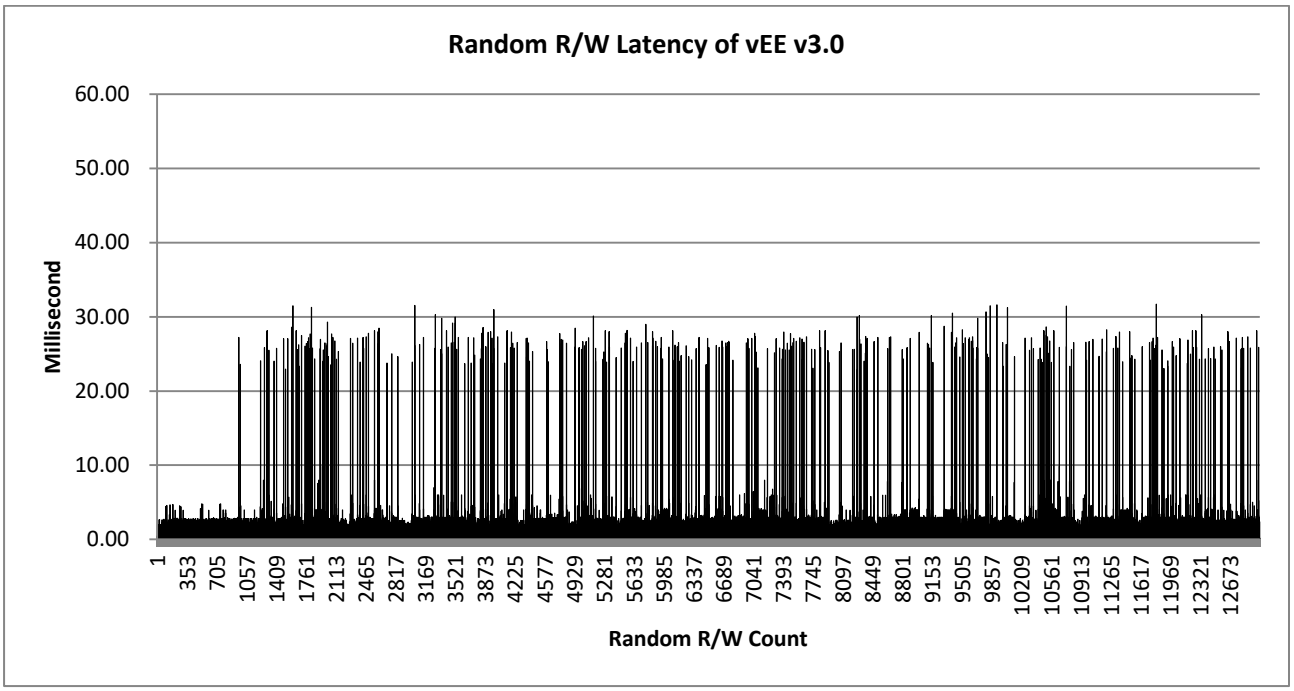
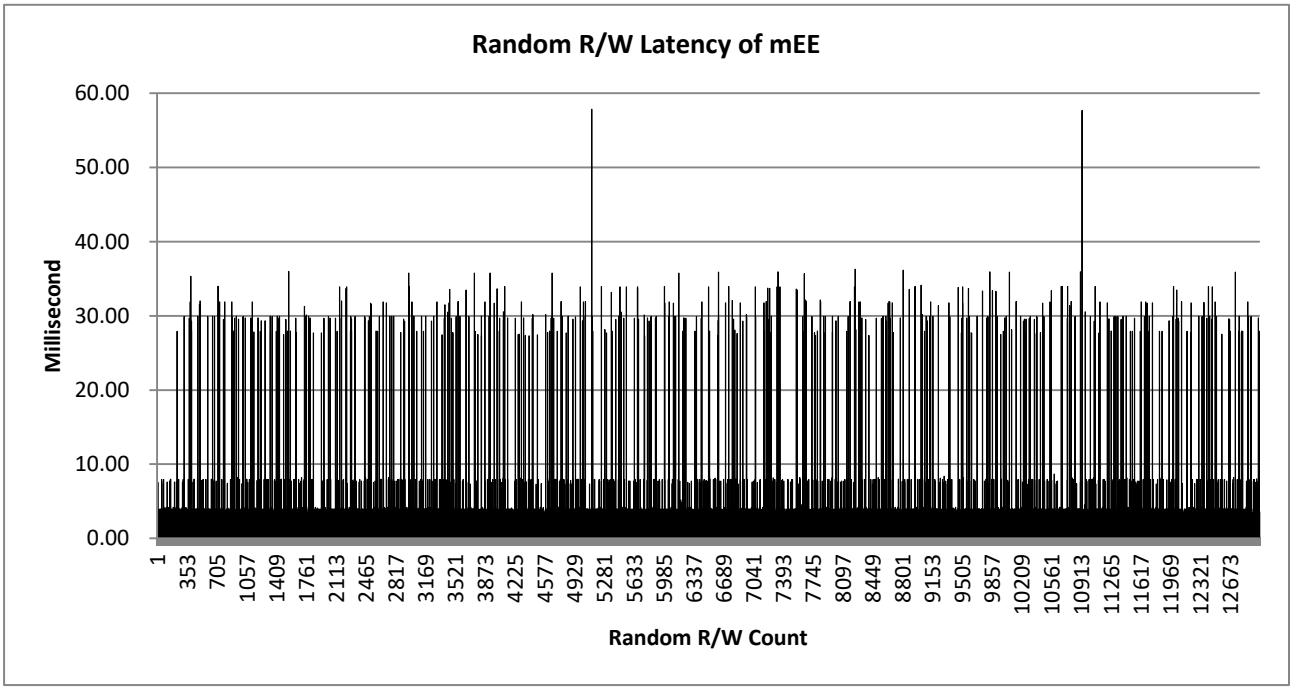


Figure 5-5: Single Thread Random R/W Latency of mEE (32/256)



As we can see from the two figures above, the latency of RWWEE decreases by about 20%. In the IOPS testing, the difference between vEEv3_32/256 (w/ RWW) case and mEE_32/256 case is also about 20%. Therefore, we can draw a conclusion that RWWEE greatly improves the overall performance by utilizing the flash RWW feature to reduce application request latency. Besides, RWWEE can greatly reduce the CPU utilization, because the time of polling flash ready decreases a lot.

5.3 vEE v1.0, v2.0 and v3.0 Comparison

Here we provide an overall comparison of vEE v1.0, v2.0 and v3.0 to help user to select the most appropriate solution, as shown in the table below.

Table 5-1: vEE v1.0, v2.0 and v3.0 Comparison

Item	vEE v1.0	vEE v2.0	vEE v3.0
Code Size	1.05 KB	1.68 KB	2.98 KB
Cache/Buffer Size	4 KB	0 KB	100 ~ 400 B per Bank
Data Packet Format (header + data)	1B + 1B, 2B + 1B	1b + 1B	4B + 4B, 4B + 12B, 4B + 28B, 4B + 60B, ……
EEPROM Size and Endurance (64 KB Flash Space)	…… 2 KB (15 X) 1 KB (31 X) 512B (63 X) ……	…… 4 KB (14 X) 2 KB (28 X) 1 KB (56 X) ……	…… 3.45 KB (17.14 X) 1.70 KB (34.29 X) 896 B (68.57 X) ……
IOPS (Default Configuration)	241	254	626/461
New Feature	Power Cycling	Low RAM Consumption	RWW: High Performance Multi-thread Operations Low CPU Utilization

If the MCU ROM size is very anxious, vEE v1.0 or v2.0 may be considered. If the RAM resource is very limited, we recommend you to try the vEE v2.0. If ROM and RAM resources are relatively rich in your environment, vEE v3.0 (RWWEE) may be the best choice.

6 Summary

RWWEE is our third generation EEPROM emulation (vEE v3.0). It is specially designed for our Octa RWW NOR Flash to overcome the shortcomings of legacy EEPROM emulation. Compared with our previous two generations, RWWEE can greatly improve the overall performance by taking the advantage of the flash RWW feature to reduce application operation latency. The performance testing results proved that RWWEE can achieve much better improvement.

However, RWWEE also has some defects, for example there is some probability that current operations cannot meet RWW requirements and application may still need to wait until the flash memory becomes ready. Besides, compared with vEE v1.0 and v2.0, the code size is relatively large. We hope to further improve the EEPROM emulation in the future.

Thanks for your reading and have a nice day.

7 Reference Documents

Table 7-1 shows the related datasheet and application note versions used in this application note. For the most current Macronix specification, please refer to the Macronix Website at http://www.macronix.com.

Table 7-1: Datasheet Version

Datasheet	Source	Date Issued	Version
MX25LW51245G	Macronix Website	April 14, 2017	Rev. 0.02
vEE v1.0	Macronix Website	December 10, 2010	Rev. 0.00
vEE v2.0	Macronix Website	November 30, 2018	Rev. 1.00

8 Revision History

Table 8-1: Revision History

Revision No.	Description	Page	Date
Rev. 1.2	Initial Release	ALL	April 15, 2021

Except for customized products which have been expressly identified in the applicable agreement, Macronix's products are designed, developed, and/or manufactured for ordinary business, industrial, personal, and/or household applications only, and not for use in any applications which may, directly or indirectly, cause death, personal injury, or severe property damages. In the event Macronix products are used in contradicted to their target usage above, the buyer shall take any and all actions to ensure said Macronix's product qualified for its actual use in accordance with the applicable laws and regulations; and Macronix as well as its suppliers and/or distributors shall be released from any and all liability arisen therefrom.

For the contact and order information, please visit Macronix's Web site at: http://www.macronix.com.