Macronix

FreeRTOS RWW Driver

Application Note v1.0

Contents

1. Introduction

The growing demand for richer graphics, wider range of multimedia and other data-intensive content, is driving embedded designers to enable more sophisticated features in embedded applications. These sophisticated features require higher data throughputs and extra demands on the often limited MCU on-chip memory.
The Macronix Octa flash is used so far to provide higher data throughput and to extend the MCU on-chip memory, solving the memory size and the performance limitation. To improve the I/O performance, some Octa flash provides multiple bank architecture with simultaneous read while write operation. The sequential read operation can be occurred in any of three banks which is not been programmed or erased.
The flash driver is supported in bare-metal and also in Linux system now. To be rich in supporting flash drivers in a variety of systems, the application note describes the RWW Octa flash driver on FreeRTOS and explains how to setup RWW operation and multi-thread communications.

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 “Macronix BSP LLD Introduction” introduces and describes the architecture of LLD.
- Chapter 4 “RWW Operation Design” introduces the RWW driver design architecture.
- Chapter 5 “Revision History” describes revision history of the document.

2. Get Started

The RWW Driver is implemented on ST platform with FreeRTOS system. This section describes the requirements and procedures to setup RWW driver on FreeRTOS.

2.1 Software requirements

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

- FreeRTOS v9.0.0
- Macronix Flash BSP Low Level Driver
- This demo is based on STM32L4R9I-Discovery BSP example, we added or modified file as Table 1 and Table 2 shows.

index	new added/renamed file
1	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\app.c
2	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\app.h
3	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\mx_define.h
4	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\mxic_hc.c
5	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\mxic_hc.h
6	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\mxic_spi_nor_timer.c
7	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\mxic_spi_nor_timer.h
8	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\nor_cmd.c
9	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\nor_cmd.h
10	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\nor_ops.c
11	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\nor_ops.h
12	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\spi.c

13	eeprom_rww_octa_demo\Drivers\BSP\MXIC_NOR\spi.h
14	eeprom_rww_octa_demo\Middlewares\EEPROM\eeprom1.c
15	eeprom_rww_octa_demo\Middlewares\EEPROM\eeprom2.c
16	eeprom_rww_octa_demo\Middlewares\EEPROM\mx25lm51245g.h
17	eeprom_rww_octa_demo\Middlewares\EEPROM\eeprom.c
18	eeprom_rww_octa_demo\Middlewares\EEPROM\rwwee.h
19	eeprom_rww_octa_demo\Middlewares\EEPROM\rww.c
20	eeprom_rww_octa_demo\Middlewares\EEPROM\rwwee1.h
21	eeprom_rww_octa_demo\Middlewares\EEPROM\rwwee2.h
22	eeprom_rww_octa_demo\Projects\32L4R9IDISCOVERY\Examples\BSP\Src\performance_demo.c

Table 1 new added/renamed file

- Modified file

index	modified file
1	eeprom_rww_octa_demo\Projects\32L4R9IDISCOVERY\Examples\BSP\Src\audio_play.c
2	eeprom_rww_octa_demo\Projects\32L4R9IDISCOVERY\Examples\BSP\Src\audio_record.c
3	eeprom_rww_octa_demo\Projects\32L4R9IDISCOVERY\Examples\BSP\Src\main.c
4	eeprom_rww_octa_demo\Projects\32L4R9IDISCOVERY\Examples\BSP\Src\lcd.c
5	eeprom_rww_octa_demo\Projects\32L4R9IDISCOVERY\Examples\BSP\Src\led.c
6	eeprom_rww_octa_demo\Projects\32L4R9IDISCOVERY\Examples\BSP\Src\usart.c
7	eeprom_rww_octa_demo\Projects\32L4R9IDISCOVERY\Examples\BSP\Src\touchscreen.c

Table 2 modified file

2.2 Hardware requirements

The RWW driver is designed to work with:

- Macronix Read-While-Write Octa Flash
- STM32L4R9I Discovery board or STM32 microcontrollers with OSPI flash host controller

2.3 Setup RWW Driver on FreeRTOS

1. Initialize a hardware timer on ST board and setting the timer period equals to 100us.
2. Unzip FreeRTOS_RWW_LLD.zip file and extract to the application project.
3. Invoking RWW LLD application layer interface in multi-thread on FreeRTOS. Please refer to the user sample code and EEPROM Emulation application code in chapter 4.

3. Macronix BSP LLD Introduction

The LLD simplifies the process of developing application code in C. With LLD, users can focus on writing the high-level code required for their particular applications. Users do not have to concern themselves with the details of the special instruction sequences.
LLD is an abstract module used to hide the detailed operations of flash. User can directly call the well encapsulated API like read, write, erase and etc. to operate flash without dealing with the protocol signals, flash registers and hardware related stuff.
This LLD is designed to support Macronix's armor flash products.

3.1 BSP LLD Files

The LLD source code provided as 11 files (6 header files and 5 C source code files).

- *app.c* C file containing application function definitions.
- *app.h* header file containing application function prototypes.
- *nor_ops.c* C file containing flash operation function definitions.
- *nor_ops.h* header file containing flash operation function prototypes.
- *nor_cmd.c* C file containing flash command function definitions.
- *nor_cmd.h* header file containing flash command function prototypes.
- *spi.c* C file containing uniform interface function definitions.
- *spi.h* header file containing flash command function prototypes.
- *mxic_hc.c* C file containing HAL function definitions.
- *mxic_hc.h* header file containing HAL function prototypes.
- *mx_define.h* header file containing code customization macros.

3.2 BSP LLD Architecture

Figure 3-1 describes the directory structure of LLD. LLD directly interacts with low-level hardware and provides APIs for higher-level application. Application calls these APIs to operate flash chip. We can divide the LLD into four layers:

- **Application Layer:** This layer contains *app.c*. It provides applications for users to implement read operation, program operation, switch mode operation, and so on.
- **Flash Operation Layer:** This layer contains *nor_ops.c*, nor_*cmd.c.* It allows the programmer to easily add new functionality by invoking the lower-level command functions in the desired sequence.
- **Uniform Interface Layer:** This layer contains *spi.c* which has two mainly functions: MxSpiFlashWrite and MxSpiFlashRead. All commands are classified into these two functions.

- **Hardware Abstraction Layer (HAL)**: This layer contains *mxic_hc.c* which operates controller register to transfer command, address, dummy and data. HAL is used to adapt the LLD to the target system.

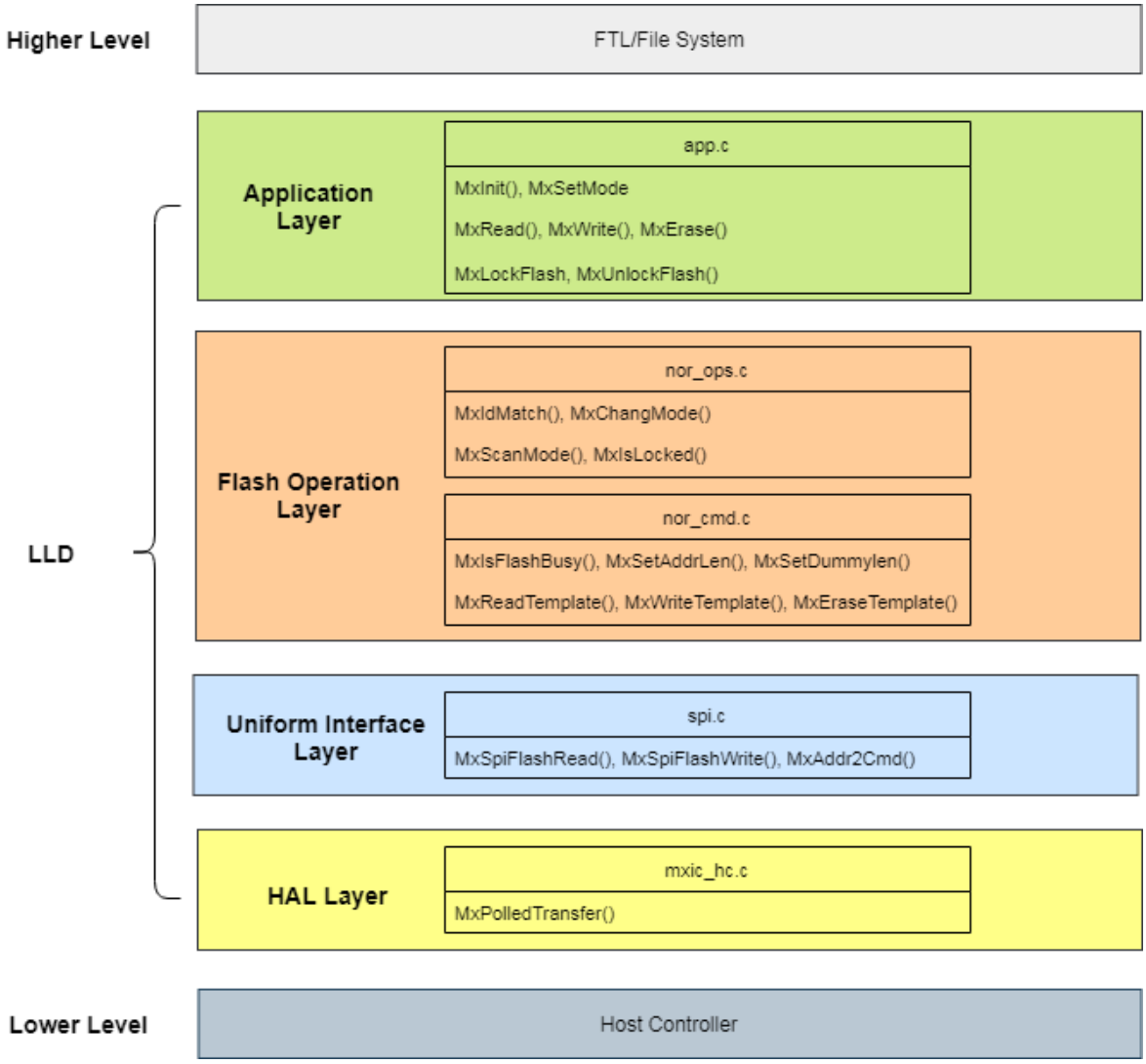


Figure 3-1 Macronix LLD Architecture

4. RWW Operation Design

The RWW driver code needs to manage multi-thread operations to maximize read-while-write probability. The RWW API which includes multi-thread management and normal LLD flash user API functions, provides a universal flash RWW operation interface for user layer.

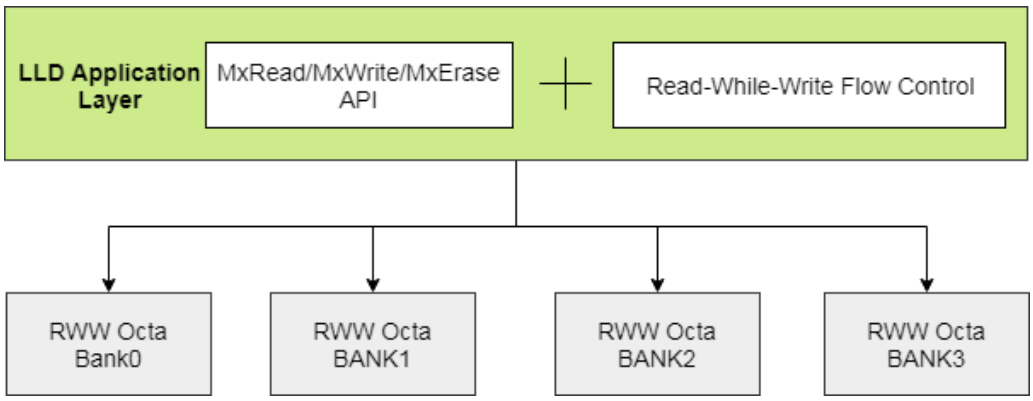


Figure 4-1 RWW Operation Control

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

1. If anyone writes any bank when the flash is still in busy state, the corresponding thread enters delayed state until the flash becomes ready. And if anyone read one bank which is different from the busy bank, the system scheduler allows the read thread to continue executing;
2. Only one thread can get the read mutex lock or write 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.

4.1 Read Operation Flow

After program/erase command is issued, auto program/erase algorithms which program/erase and verify the specified page or sector/block locations will be executed. Meanwhile, the read operation can be executing in another bank.

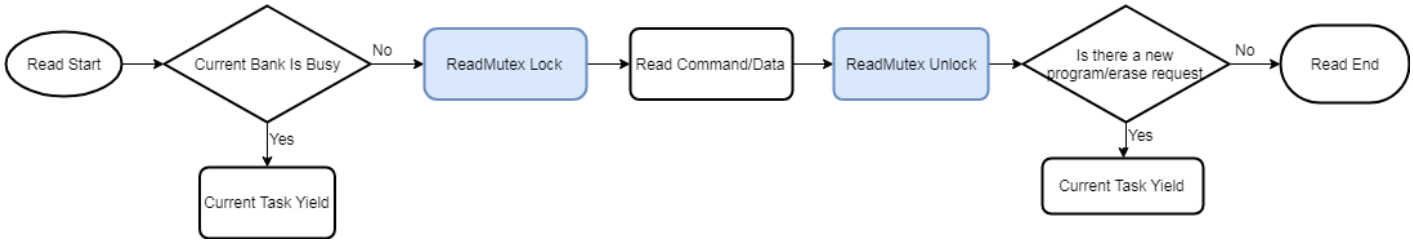


Figure 4-2 Read Flow

The ReadMutex lock protects the whole read command/data flow not to be interrupted and if there is a new program/erase request, it should be enter to blocked state.
The read operation mainly is implemented in the MxRead() function which belongs to user application layer APIs. Therefore, the read interface with RWW character is adapted to several kind types of read commands.

4.2 Write Operation Flow

The WriteMutex lock protects the whole PP command flow not to be interrupted and if the device is busy state, it should be enter to blocked state. When the program command is issued, the write thread could be waiting for the flash to be ready and release CPU resource. The strategy will help improve the system throughputs.

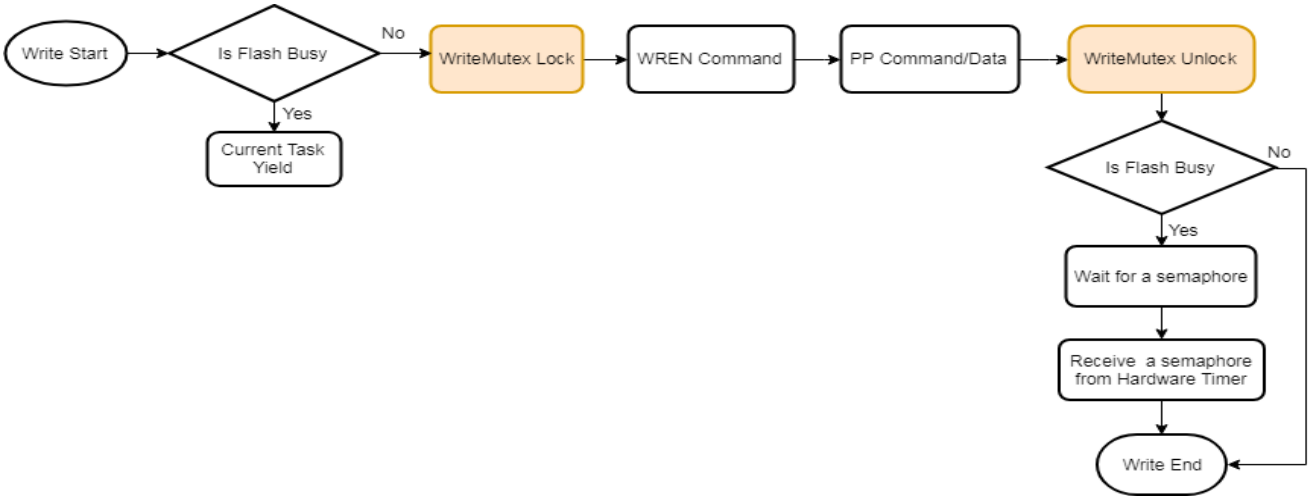


Figure4-3 Write Flow

4.3 Interruptible Write to Buffer

The multi-bank read-while-write flash memory provides an interruptible write-to-buffer sequence during programming. This sequence provides the advantage that read operations could be inserted among program data write cycles. There are three steps for the operation: issuing WRBI (Write Buffer Initial command), WRCT (Write Buffer Continue command), and WRCF (Write Confirm command) as shown in Figure 4-4.

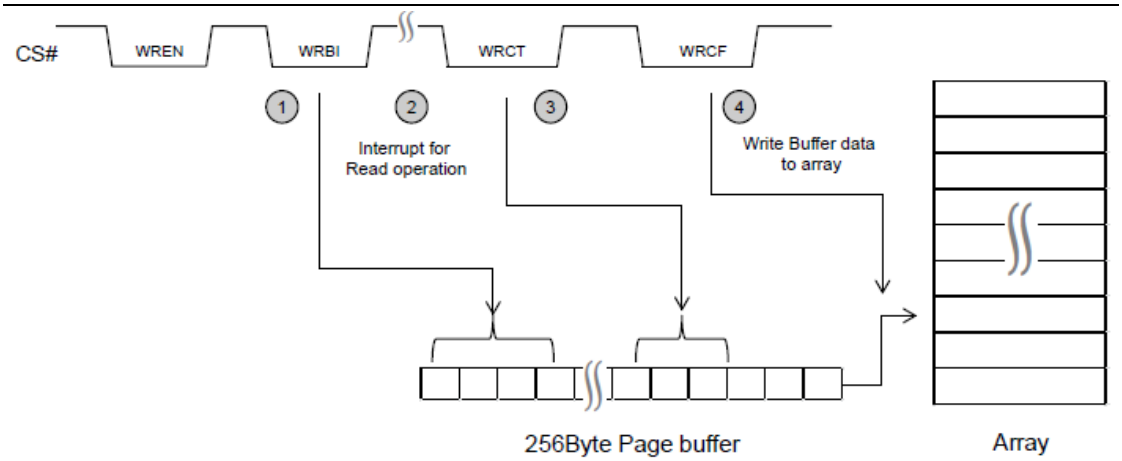


Figure4-4 Write Buffer Sequence

To trigger the interruptible write-to-buffer sequence, the system issues WREN command first to enable the WEL bit, and then issues WRBI with 4-byte address and data of 0~256 byte. After CS# has gone high, the flash memory will return to the standby status, waiting for the next command. The system could either perform read operation or issue Write Buffer Continue command (WRCT) to write more data to the page buffer. After the data writing has finished, the system will issue a Write Confirm (WRCF) command to initiate an automatic program operation to write the page buffer data into the array cell.

Multiple read or WRCT commands can be issued between the WRBI and WRCF commands. The system could also issue a WRBI directly followed by a WRCF command if there is no needs for write interrupt on the system.

4.3.1 Buffer Write

The write buffer sequence is protected by xBufferMutex lock. And the three write buffer operations is locked by xWriteMutex in multi-thread environment.

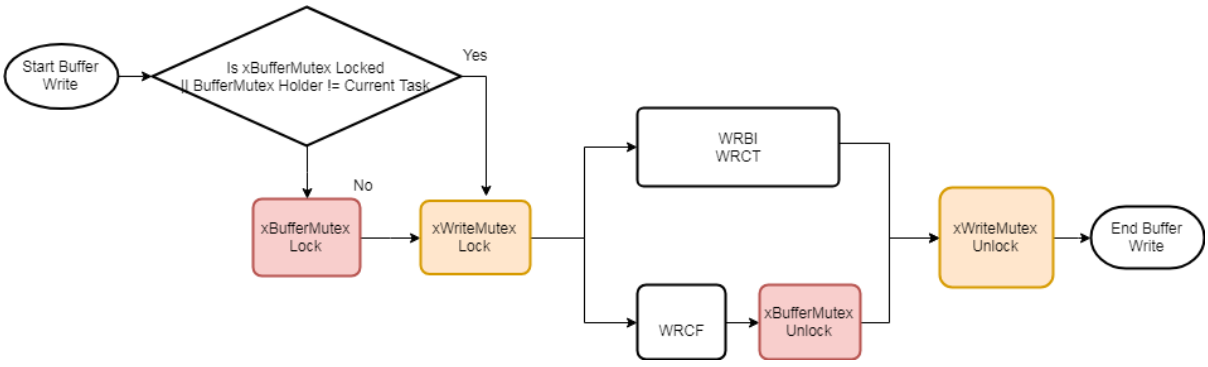


Figure4-5 Buffer Write

4.3.2 Buffer Read

The xReadMutex lock protects the whole read command/data flow not to be

interrupted as same as normal read operation.

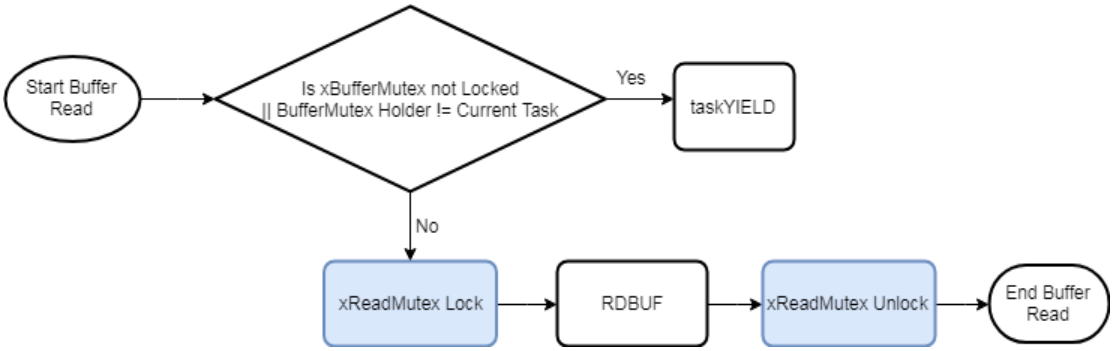


Figure4-6 Buffer Read

4.4 RWW Feature

The Flash memory array is divided into banks. The multi bank structure enables read-while-write (RWW) feature, which means read data one bank while another bank is programing or erasing. It is recommended to use Macronix Octa RWW flash to achieve better performance. User can define the "RWW_DRIVER_SUPPORT" constant in a common header file to enable the RWW feature.

4.5 RWW APIs

RWW driver operations provide the follow APIs for user application.

- int MxRead(MxChip *Mxic, u32 Addr, u32 ByteCount, u8 *Buf);

```
/*
 * Function:      MxRead
 * Arguments:     Mxic:      pointer to an mxchip structure of nor flash
device.
 *           Addr:      device address to read.
 *           ByteCount: number of bytes to read.
 *           Buf:       pointer to a data buffer where the read data
will be stored.
 * Return Value:  MXST_SUCCESS.
 *                MXST_FAILURE.
 * Description:  This function issues the Read commands to SPI Flash and
reads data from the array.
 *               Data size is specified by ByteCount.
 *               It is called by different Read commands functions like
MxREAD, Mx8READ and etc.
 */
    int MxRead(MxChip *Mxic, u32 Addr, u32 ByteCount, u8 *Buf)
```

```
{
    int status;
    int busy_stat;
    u32 len;
    u32 cnt;
    if (MxAddrSpanBank(Addr, ByteCount))
    {
        len = BANK_LEN - Addr%BANK_LEN;
        while ((busy_bank&0x7F) & (1 << BANKS(Addr)))
        {
            taskYIELD();
        }
        xSemaphoreTake(xReadMutex, portMAX_DELAY);
        status = Mxic->AppGrp._Read(Mxic, Addr, len, Buf);
        xSemaphoreGive(xReadMutex);
        while (busy_bank & 0x80)
        {
            taskYIELD();
        }
        for(cnt=len; cnt<ByteCount; cnt+=len)
        {
            len = ByteCount - cnt;
            if (len > BANK_LEN)
                len = BANK_LEN;
            while ((busy_bank&0x7F) & (1 << BANKS(Addr)))
            {
                taskYIELD();
            }
            xSemaphoreTake(xReadMutex, portMAX_DELAY);
            status = Mxic->AppGrp._Read(Mxic, Addr+cnt, len, Buf+cnt);
            xSemaphoreGive(xReadMutex);
            while (busy_bank & 0x80)
            {
                taskYIELD();
            }
        }
    }
    else
    {
        while ((busy_bank&0x7F) & (1 << BANKS(Addr)))
        {
            taskYIELD();
        }
```

```
            xSemaphoreTake(xReadMutex, portMAX_DELAY);
            status = Mxic->AppGrp._Read(Mxic, Addr, ByteCount, Buf);
            xSemaphoreGive(xReadMutex);
            if (busy_bank & 0x80)
                taskYIELD();
        }
        return status;
    }
```

- int MxWrite(MxChip *Mxic, u32 Addr, u32 ByteCount, u8 *Buf);

```
/*
 * Function:      MxWrite
 * Arguments:     Mxic:      pointer to an mxchip structure of nor flash
device.
 *            Addr:      device address to program.
 *            ByteCount: number of bytes to program.
 *            Buf:       pointer to a data buffer where the program data
will be stored.
 * Return Value:  MXST_SUCCESS.
 *                MXST_FAILURE.
 *                MXST_TIMEOUT.
 * Description:    This function programs location to the specified data.
 *                It is called by different Read commands functions like
MxPP, MxPP4B and etc.
 */
    int MxWrite(MxChip *Mxic, u32 Addr, u32 ByteCount, u8 *Buf)
    {
        int status;
        int busy_stat;
        u32 cnt, total_num, len;
        busy_bank |= 0x80;
        for(cnt=0; cnt<ByteCount; cnt+=len)
        {
            len = ByteCount - cnt;
            if ( len >    Mxic->PageSz)
            {
                len = Mxic->PageSz;
            }
            while ((MxGetStatus(Mxic) & 0x01) || delay_timing)
            {
                taskYIELD();
            }
            xSemaphoreTake(xWriteMutex, portMAX_DELAY);
```

```
            status = Mxic->AppGrp._Write(Mxic, Addr+cnt, len, Buf+cnt);
            busy_bank |= 1<<BANKS(Addr+cnt);
            xSemaphoreGive(xWriteMutex);
            vTaskPrioritySet(NULL, uxTaskPriorityGet(NULL)+1);
            xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);
            vTaskPrioritySet(NULL, uxTaskPriorityGet(NULL)-1);
        }
        busy_bank &= 0x7F;
        return status;
    }
```

- int MxErase(MxChip *Mxic, u32 Addr, u32 EraseSizeCount);

```
/*
 * Function:      MxErase
 * Arguments:       Mxic:           pointer to an mxchip structure of nor
flash device.
 *              Addr:           device address to erase.
 *              EraseSizeCount: number of block or sector to erase.
 * Return Value:  MXST_SUCCESS.
 *              MXST_FAILURE.
 *              MXST_TIMEOUT.
 * Description:   This function erases the data in the specified Block
or Sector.
 *            Function issues all required commands and polls for
completion.
 *            It is called by different Read commands functions like
MxSE, MxSE4B and etc.
 */
    int MxErase(MxChip *Mxic, u32 Addr, u32 EraseSizeCount)
    {
        int status;
        busy_bank |= 0x80;
        for(int i=0; i<EraseSizeCount; i++)
        {
            while ((MxGetStatus(Mxic)&0x01) || delay_timing)
            {
                taskYIELD();
            }
            xSemaphoreTake(xWriteMutex, portMAX_DELAY);
            status = Mxic->AppGrp._Erase(Mxic, Addr+i*SECTOR4KB_SZ, 1);
            busy_bank |= 1<<BANKS(Addr+i*SECTOR4KB_SZ);
            xSemaphoreGive(xWriteMutex);
            if ( MxGetStatus(Mxic) & 0x01 )
```

```
            {
                vTaskPrioritySet(NULL, uxTaskPriorityGet(NULL)+1);
                xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);
                vTaskPrioritySet(NULL, uxTaskPriorityGet(NULL)-1);
            }
        }
        busy_bank &= 0x7F;
        return status;
    }
```

- int MxBufferRead(MxChip *Mxic, u32 Addr, u32 ByteCount, u8 *Buf);

```
/*
 * Function:      MxBufferRead
 * Arguments:     Mxic:      pointer to an mxchip structure of nor flash
device.
 *            Addr:      device address to read.
 *            ByteCount: number of bytes to read.
 *            Buf:  pointer to a data buffer where the read data will be
stored.
 * Return Value:  MXST_SUCCESS.
 *             MXST_FAILURE.
 * Description:   This function is for read the page buffer.
 */
    int MxBufferRead(MxChip *Mxic, u32 Addr, u32 ByteCount, u8 *Buf)
    {
        int status;
        while ( uxSemaphoreGetCount(xBufferMutex) )
        {
            taskYIELD();
        }
        xSemaphoreTake(xReadMutex, portMAX_DELAY);
        status = MxRDBUF(Mxic, Addr, ByteCount, Buf);
        xSemaphoreGive(xReadMutex);
        return status;
    }
```

- int MxBufferWrite(MxChip *Mxic, u32 Addr, u32 ByteCount, u8 *Buf);

```
/*
 * Function:      MxBufferWrite
 * Arguments:     Mxic: pointer to an mxchip structure of nor flash
device.
 *            Addr:      device address to write.
```

```
 *            ByteCount: number of bytes to write.
 *            Buf: pointer to a data buffer where the write data will be
stored.
 * Return Value:  MXST_SUCCESS.
 *                MXST_FAILURE.
 * Description:   This function is for write the page buffer to implement
RWW function.
 */
    int MxBufferWrite(MxChip *Mxic, u32 Addr, u32 ByteCount, u8 *Buf)
    {
        int Status;
        busy_bank |= 0x80;
        while ((MxGetStatus(Mxic)&0x01) || delay_timing)
        {
            taskYIELD();
        }
        if (uxSemaphoreGetCount(xBufferMutex)
                || (xSemaphoreGetMutexHolder(xBufferMutex) !=
xTaskGetCurrentTaskHandle()))
        {
            xSemaphoreTake(xBufferMutex, portMAX_DELAY);
        }
        xSemaphoreTake(xWriteMutex, portMAX_DELAY);
        busy_bank |= 1<<BANKS(Addr);
        if(Mxic->WriteBuffStart == FALSE)
        {
            Mxic->WriteBuffStart = TRUE;
            Status = MxWRBI(Mxic, Addr, ByteCount, Buf);
        }
        else if(ByteCount > 0)
        {
            Status = MxWRCT(Mxic, Addr, ByteCount, Buf);
        }
        if(ByteCount == 0)
        {
            Status = MxWRCF(Mxic);
            STM32_Timer_Config(2);
            Mxic->WriteBuffStart = FALSE;
            xSemaphoreGive(xBufferMutex);
        }
        xSemaphoreGive(xWriteMutex);
        if (MxGetStatus(Mxic)&0x01)
        {
```

```
            vTaskPrioritySet(NULL, uxTaskPriorityGet(NULL)+1);
            xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);
            vTaskPrioritySet(NULL, uxTaskPriorityGet(NULL)-1);
        }
        busy_bank &= 0x7F;
        return MXST_SUCCESS;
    }
```

4.6 User Sample Code

4.6.1 Read-While-Write Operation Testbench

```
TaskHandle_t ReadThreadHandle, WriteThreadHandle, EraseThreadHandle;
static void Write_Thread(void const *argument)
{
  (void) argument;
  MxErase(&Mxic, 0x00FF0000, 1);
  MxErase(&Mxic, 0x01000000, 1);
  MxWrite(&Mxic, 0x00FFF700, PAGE_SZ*10, WrData);
  vTaskSuspend(NULL);
}

static void Erase_Thread(void const *argument)
{
  (void) argument;
  MxErase(&Mxic, 0, 2);
  vTaskSuspend(NULL);
}

static void Read_Thread3(void const *argument)
{
  (void) argument;
  for(int i=0;i<50;i++)
     MxRead(&Mxic, 0x1000000+i*PAGE_SZ, PAGE_SZ, RdData);
  vTaskSuspend(NULL);
}
int main(void)
{
  HAL_Init();
```

```
  /* Configure the System clock to 120 MHz */
  SystemClock_Config();
#ifdef RWW_DRIVER_SUPPORT
  STM32_Timer_Init();
  xBinarySemaphore = xSemaphoreCreateBinary();
  xReadMutex = xSemaphoreCreateMutex();
  xWriteMutex = xSemaphoreCreateMutex();
  xBusMutex = xSemaphoreCreateMutex();
  xBufferMutex = xSemaphoreCreateMutex();
  MxInit(&Mxic);
#endif
xTaskCreate(Erase_Thread, "THREAD_Erase", configMINIMAL_STACK_SIZE, NULL,
osPriorityNormal, &EraseThreadHandle);
xTaskCreate(Read_Thread, "THREAD_Read", configMINIMAL_STACK_SIZE, NULL,
osPriorityNormal, &ReadThreadHandle);
xTaskCreate(Write_Thread, "THREAD_Write", configMINIMAL_STACK_SIZE, NULL,
osPriorityNormal, &WriteThreadHandle);
/* Start scheduler */
  osKernelStart();

  /* We should never get here as control is now taken by the scheduler */
  for (;;);
}
```

4.6.2 Buffer Read/Write Testbench

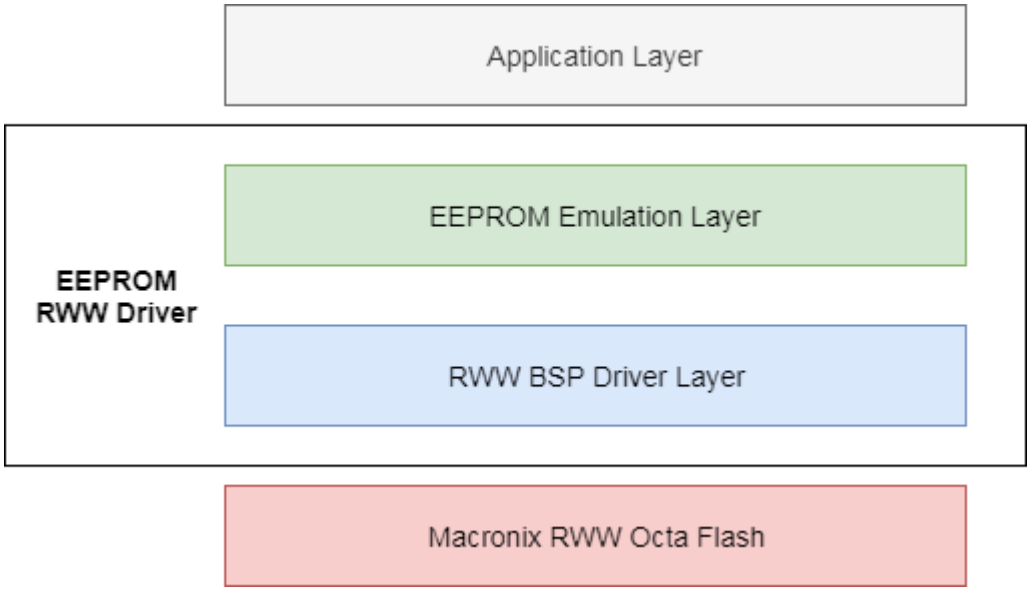
```
TaskHandle_t ReadThreadHandle, ReadThread2Handle;
static void Read_Thread(void const *argument)
{
        (void) argument;
         MxErase(&Mxic, 0, 1);
         for(int cnt=0;cnt<10;cnt++)
         {
                MxBufferWrite(&Mxic, cnt*PAGE_SZ, 100, WrData);
                MxBufferRead(&Mxic, cnt*PAGE_SZ, 100, RdData);
                MxBufferWrite(&Mxic, cnt*PAGE_SZ+100, 4, WrData+100);
                MxBufferRead(&Mxic, cnt*PAGE_SZ+100, 4, RdData+100);
                MxBufferWrite(&Mxic, 0, 0, 0);
         }
         vTaskSuspend(NULL);
}
```

```
static void Read_Thread2(void const *argument)
{
        (void) argument;

        vTaskDelay(25);
        for(int i=0;i<40;i++)
            MxRead(&Mxic, 0x1000000+i*PAGE_SZ, PAGE_SZ, RdData);
        vTaskDelay(70);
        for(int i=0;i<20;i++)
            MxRead(&Mxic, 0x1000000+i*PAGE_SZ, PAGE_SZ, RdData);
        vTaskDelay(3);
        for(int i=0;i<20;i++)
            MxRead(&Mxic, 0x2000000+i*PAGE_SZ, PAGE_SZ, RdData);
        vTaskSuspend(NULL);
}
int main(void)
{
  HAL_Init();
  /* Configure the System clock to 120 MHz */
  SystemClock_Config();
#ifdef RWW_DRIVER_SUPPORT
  STM32_Timer_Init();
  xBinarySemaphore = xSemaphoreCreateBinary();
  xReadMutex = xSemaphoreCreateMutex();
  xWriteMutex = xSemaphoreCreateMutex();
  xBusMutex = xSemaphoreCreateMutex();
  xBufferMutex = xSemaphoreCreateMutex();
  MxInit(&Mxic);
#endif
  xTaskCreate(Read_Thread, "THREAD_Buffer", configMINIMAL_STACK_SIZE, NULL,
  osPriorityNormal, &ReadThreadHandle);
  xTaskCreate(Read_Thread2, "THREAD_Read", configMINIMAL_STACK_SIZE, NULL,
  osPriorityNormal, &ReadThread2Handle);
/* Start scheduler */
  osKernelStart();

  /* We should never get here as control is now taken by the scheduler */
  for (;;);
}
```

4.7 EEPROM Emulation With RWW Driver

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.

4.7.1 EEPROM Emulation Layer



4.7.2 EEPROM RWW APIs

EEPROM emulation provides the following APIs for upper applications.

- EEPROM read

```
int mx_eeprom_read(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 **`mx_eeprom_get_param()`** function or the *MX_EEPROM_TOTAL_SIZE* constant defined in the header file "*rwwee.h*".

- EEPROM write

```
int mx_eeprom_write(uint32_t addr, uint32_t len, uint8_t
*buf);
```

```
int mx_eeprom_write_back(void);

int mx_eeprom_sync_write(uint32_t addr, uint32_t len, uint8_t
*buf);
```

Here we provide three write functions. The first function **mx_eeprom_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 **mx_eeprom_write_back()** function. The Last function **mx_eeprom_sync_write()** wraps the above two functions together to realize synchronous write.

- EEPROM flush

```
int mx_eeprom_flush(void);
```

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

- Initialization and de-initialization

```
int mx_eeprom_format(void);

int mx_eeprom_init(void);

void mx_eeprom_deinit(void);
```

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

4.7.3 EEPROM Application Code

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

```
/* Init OSPI NOR */
ret = mx_ee_rww_init();
if (ret)
  goto err;

/* Format RWWEE */
ret = mx_eeprom_format();
if (ret)
```

```
  goto err;

/* Init RWWEE */
ret = mx_eeprom_init();
if (ret)
  goto err;

/* Read RWWEE */
ret = mx_eeprom_read(addr, len, buf);
if (ret)
  goto err;

/* Update buffer */


/* Write RWWEE */
ret = mx_eeprom_write(addr, len, buf);
if (ret)
  goto err;

/* Do some more read/write operations */


/* Flush data */
ret = mx_eeprom_flush();
if (ret)
  goto err;

/* De-init RWWEE */
mx_eeprom_deinit();
```

5 Demo description

5.1 Devices

- Memory: MX25LW51245G
- Display: 390x390 DSI LCD
- Audio: Microphone/Speaker

5.2 Demo scenarios

- Recording while 1x speed playing back
- Recording while 2x speed playing back
- Performance comparison between nonRWW Driver and RWW Driver
- Performance comparison between RWW driver only and EEPROM emulation with RWW Driver

5.3 Testing result

- nonRWW Driver VS RWW Driver

Index	Item	nonRWW Driver	RWW Driver
1	Erase 512K(4K*128)	25ms*128=3.2s	25ms*128=3.2s
2	Write 512K(256*2048)	420us * 2048 = 860ms	420us * 2048 = 860ms
3	Read 4MB(256*16K)	187us*16*1024 = **3.06s**	187us*16*1024 = 3.06s
4	LCD and other Delay	0.6s	0.7s
5	Total	3.2s+860ms+**3.06s**+0.6s=7.7s	3.2s+860ms+0.7s = 4.76s

Table 3 nonRWW VS RWW

- RWW Driver VS EEPROM emulation with RWW Driver

Index	Item	RWW Driver	EEPROM emulation with RWW Driver
1	Erase	Erase 32K (25ms*8=**200ms**)	Erase 4K (25ms)
2	Write 4K(256*16)	420us * 16 = 6.72ms	420us * 16 = 6.72ms
3	Read 32K(256*16)	187us*16*8 = 24ms	187us*16*8 = 24ms
4	LCD and other Delay	950ms	840ms
5	Total	**200ms**+6.72ms+950ms=1156ms	25ms+6.72ms+840ms=872ms

Table 4 RWW VS EEPROM

6 Revision History

Date	Version	Changes
9/12/2019	1.0	First Release.