

TARCollege

Tunku Abdul Rahman University College

Microcontroller Peripherals
Flash Memory

Name: Wong Jang Sing

Reg. No 15 WAR 09371

Programme: RMB 2

Tutor: Dr. Poh Tze Ven

Table of Contents

CHAPTER I: INTRODUCTION	3
1.1 Objective:	3
1.2 Brief Description:	3
1.3 Flash memory Operation	6
 CHAPTER II: METHODOLOGY	 9
2.1 Unlocking Flash Control Register.....	9
2.2 Read Operation.....	9
2.3 Erase Operation	10
2.4 Programming Operation.....	10
2.5 DMA memory-to-memory transfer Operation	10
 CHAPTER III: RESULT & DISCUSSION	 12
3.1 Unlocking Flash Control Register.....	13
3.2 Read Operation.....	14
3.3 Erase Operation	16
3.4 Programming Operation.....	22
3.5 DMA memory-to-memory transfer Operation	27
3.6 Overall discussion	35
 CHAPTER IV: CONCLUSION	 36

CHAPTER I: INTRODUCTION

1.1 Objective:

Program has been written to the ARM Cortex®-M4 Microcontroller to communicate with the Embedded Flash memory interface to either perform read/program/erase operation to the Flash Memory. Result has been obtained and discussed in the following chapters.

1.2 Brief Description:

The Embedded Flash memory interface is an intermediate agent between the Cortex-M4 microcontroller and the Flash memory. Basically the microcontroller is telling the flash memory interface whether to perform read/program/erase operation to the Flash Memory. The Flash memory interface connection inside system architecture for STM32F429 is as shown below.

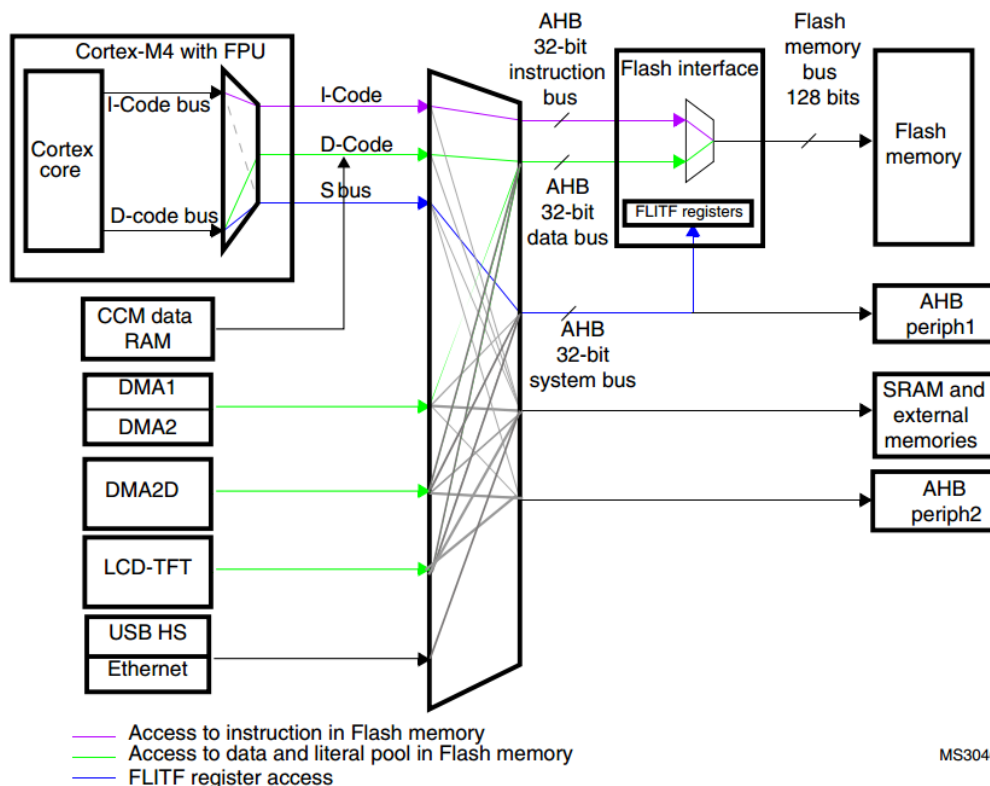


Figure 1 Flash memory interface connection inside system architecture for

The Flash memory interface manages CPU AHB (advanced high-performance bus) I-Code and D-Code accesses to the Flash memory. The I-Code and D-Code are Instruction Code and the Data Code respectively. For the I-Code, it connects the Instruction bus of the ARM microcontroller to the Flash instruction interface. While for the D-Code, it connects the D-Code bus of the ARM microcontroller to the Flash data interface. Before the flash memory interface, the data has been processed by the bus matrix. The bus matrix is as shown below.

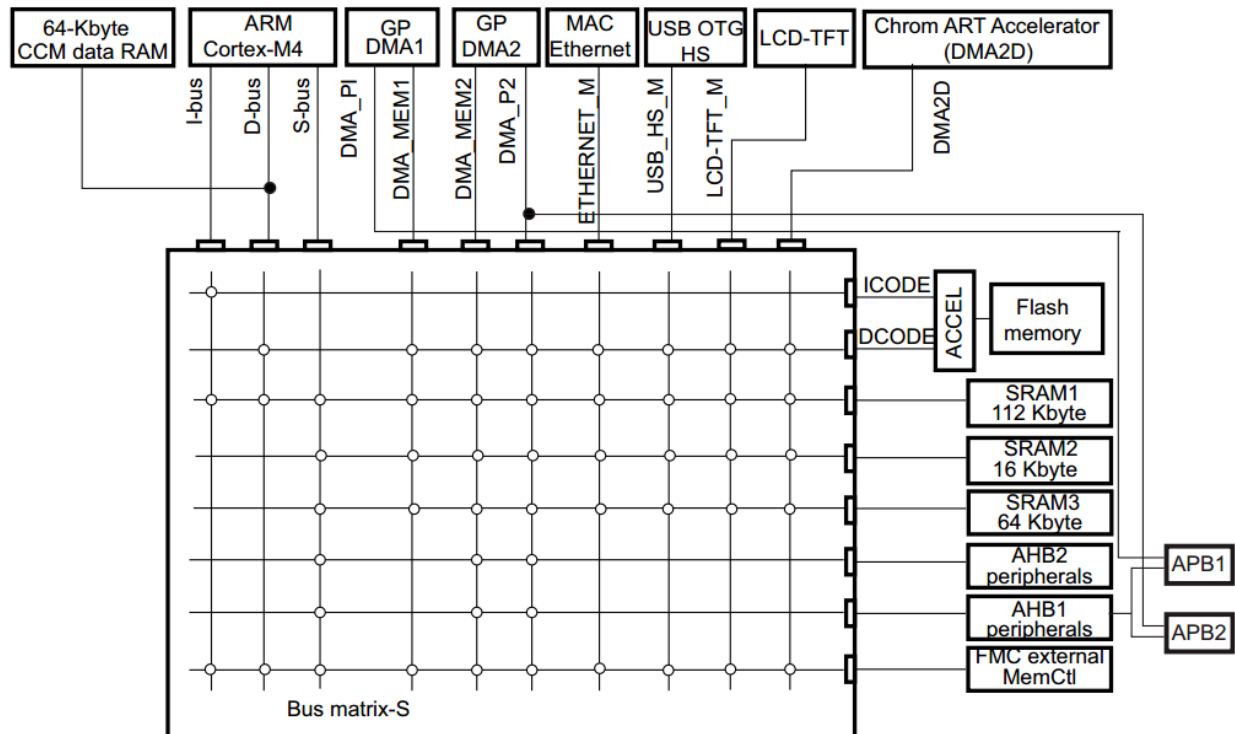


Figure 2 Bus matrix for STM32F429.

The flash memory for microcontroller STM32F429 is organized as follow:

- A main memory block divided into sectors.
- System memory
- 512 OTP (one-time programmable) bytes for user data.
- Option bytes.

The main memory block was divided into two bank with 1Mbyte. In each bank, it consists of 4 sectors with 16k bytes, 1 sector of 64k bytes and 7 sectors of 128k bytes. There are 16 additional byte in each bank which is the Option Bytes. The option bytes is to configure the read and write protection, BOR level, watchdog, dual bank

boot mode, dual bank feature, software/hardware and reset when the device is in Standby or Stop mode. The STM32F429 is a device that have 2M byte with dual bank flash memory. The flash module for 2M byte dual bank organization is as shown below.

Table 1 flash module for 2M byte dual bank organization for STM32F429

Block	Bank	Name	Block base addresses	Size
Main memory	Bank 1	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
		Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
		Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
		Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbyte
		Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
		Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
		Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
		-	-	-
		-	-	-
		-	-	-
		Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbytes
	Bank 2	Sector 12	0x0810 0000 - 0x0810 3FFF	16 Kbytes
		Sector 13	0x0810 4000 - 0x0810 7FFF	16 Kbytes
		Sector 14	0x0810 8000 - 0x0810 BFFF	16 Kbytes
		Sector 15	0x0810 C000 - 0x0810 FFFF	16 Kbytes
		Sector 16	0x0811 0000 - 0x0811 FFFF	64 Kbytes
		Sector 17	0x0812 0000 - 0x0813 FFFF	128 Kbytes
		Sector 18	0x0814 0000 - 0x0815 FFFF	128 Kbytes
			-	-
			-	-
			-	-
		Sector 23	0x081E 0000 - 0x081F FFFF	128 Kbytes
		System memory		
OTP			0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes	Bank 1		0x1FFF C000 - 0x1FFF C00F	16 bytes
	Bank 2		0x1FFE C000 - 0x1FFE C00F	16 bytes

The clock used for the Flash Memory is the HCLK, which can operate up to 180MHz maximum. The clock source used to produce the HCLK is from the external clock (CRYSTAL CLOCK). The crystal clock frequency of the device is 8MHz. The overall picture the HCLK that produced to the flash memory is as shown below.

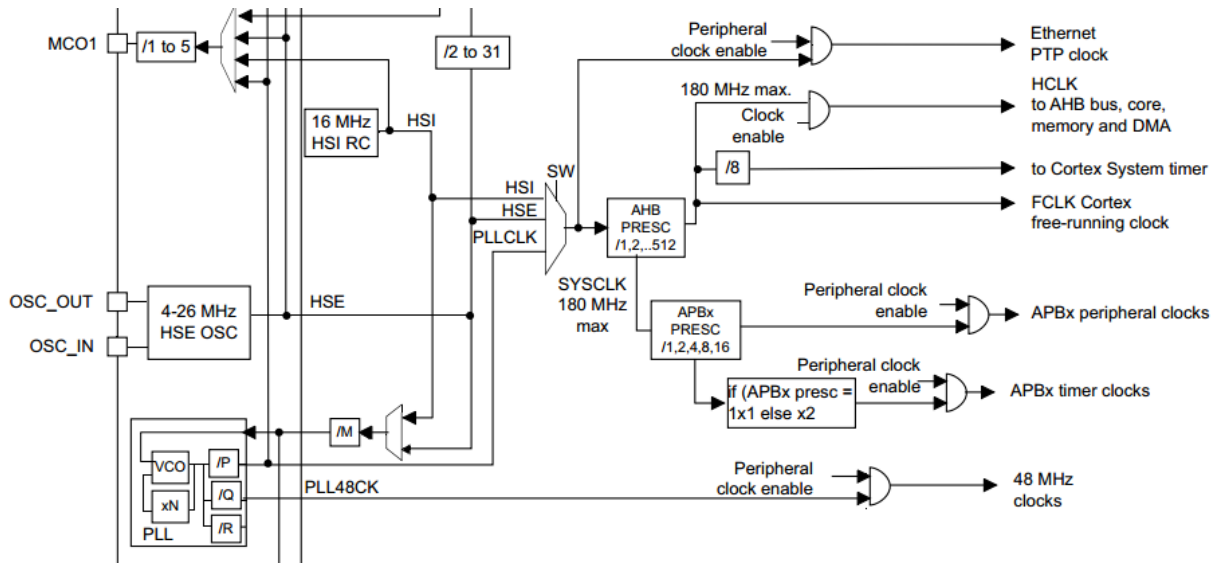


Figure 3 Clock tree for the Flash Memory.

By referring to Figure 1, there are 32-bit for instruction bus and 32-bit for data bus. So the maximum bit that able to store into the Flash is 32-bit. The Flash is unable to store a 64-bit value. These will be proven in the following chapter.

1.3 Flash memory Operation

i) Read Operation

Reading the data at the selected address of the flash memory. If the selected address was an empty address, the data of that address will be all '1' for every single bit.

ii) Erase Operation

There are three type of erase operation for the flash memory, which are sector erase, bank erase and Mass erase. After the erase operation has been done, the data in the erased address will all become '1'.

Sector Erase => to erase all the data in the selected sector.

Bank Erase => to erase all the data in the selected bank.

Mass Erase => to erase all the data in the Flash Memory.

Note that when the program written for the controller is booting in sector 1. So the sector 1 which is in bank 1 is unable to erase.

iii) Program Operation

Before programming data into an address of the flash, the particular address has to be erased only it is ready to program. Otherwise, unexpected value will be stored to the particular address selected. This happens because the flash memory of this device are only able to write a '0' to the address. When they see a '1', it will not make any changes to the memory data. Thus, violation may happens.

iv) Memory-to-memory transfer by Direct Memory Access (DMA)

With the aid of DMA, the ARM controller can now able to transfer multiple data to the flash through the streams of the DMA. Note that only DMA2 controller can be used to perform memory-to-memory transfer. The table for DMA2 request mapping is as shown below.

Table 2 DMA2 request mapping

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	ADC1	SAI1_A ⁽¹⁾	TIM8_CH1 TIM8_CH2 TIM8_CH3	SAI1_A ⁽¹⁾	ADC1	SAI1_B ⁽¹⁾	TIM1_CH1 TIM1_CH2 TIM1_CH3	
Channel 1		DCMI	ADC2	ADC2	SAI1_B ⁽¹⁾	SPI6_TX ⁽¹⁾	SPI6_RX ⁽¹⁾	DCMI
Channel 2	ADC3	ADC3		SPI5_RX ⁽¹⁾	SPI5_TX ⁽¹⁾	CRYP_OUT	CRYP_IN	HASH_IN
Channel 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
Channel 4	SPI4_RX ⁽¹⁾	SPI4_TX ⁽¹⁾	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
Channel 5		USART6_RX	USART6_RX	SPI4_RX ⁽¹⁾	SPI4_TX ⁽¹⁾		USART6_TX	USART6_TX
Channel 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
Channel 7		TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX ⁽¹⁾	SPI5_TX ⁽¹⁾	TIM8_CH4 TIM8_TRIG TIM8_COM

From the table above, there are no specified stream for flash memory. Thus, empty stream has to be chosen for the flash memory transfer such as St7-Ch0, St0-Ch1, St2-Ch2, St1-Ch3, St4-Ch3, St6-Ch3, St7-Ch3, St4-Ch4, St5-Ch5, St7-Ch6 and St0-Ch7. St stands for stream and Ch stands for channel.

These streams are the intermediate agent that enable the memory to flow from the RAM memory to the Flash Memory. The transfer of the data can be illustrate as shown in figure below.

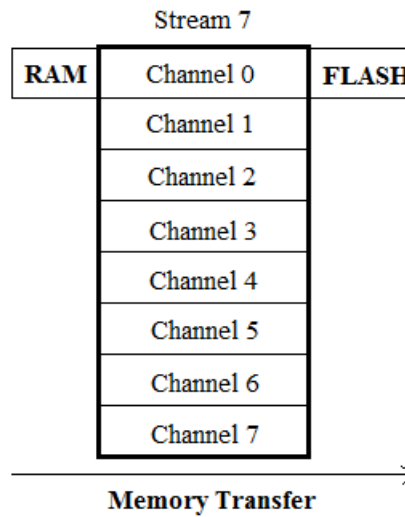


Figure 4 Memory transfer at St7-Ch0

From the Figure 4 above, it is an example of using Channel 0 of the Stream 7 to perform memory transfer from RAM memory to Flash Memory.

CHAPTER II: METHODOLOGY

In this chapter will discuss about the configuration of the flash memory. Configuration description included for unlocking Flash Control Register, read Operation, erase Operation, programming operation and DMA memory-to-memory transfer Operation

2.1 Unlocking Flash Control Register

Before performing any operation to the Flash memory interface, the flash control register (FLASH_CR) has to be unlocked by storing the key value into the flash key register (FLASH_KEYR). There are two key that has to be keyed in into the register, the sequence is as shown below.

KEY1 = 0x45670123

KEY2 = 0xCDEF89AB

The KEY1 has to be keyed into the register only followed by the KEY2. Otherwise the Flash control register will still not be unlocked.

2.2 Read Operation

For read operation of the flash memory, the wait state has to be correctly configure at the LATENCY[3:0] bit in the Flash access control register (FLASH_ACR). The number of wait states are configured depending on the clock frequency (HCLK) and the voltage supplied of the device. The number of wait state to be configure can be refer to the figure below.

Wait states (WS) (LATENCY)	HCLK (MHz)			
	Voltage range 2.7 V - 3.6 V	Voltage range 2.4 V - 2.7 V	Voltage range 2.1 V - 2.4 V	Voltage range 1.8 V - 2.1 V Prefetch OFF
0 WS (1 CPU cycle)	0 <HCLK ≤ 30	0 <HCLK ≤ 24	0 <HCLK ≤ 22	0 < HCLK ≤ 20
1 WS (2 CPU cycles)	30 <HCLK ≤ 60	24 < HCLK ≤ 48	22 <HCLK ≤ 44	20 <HCLK ≤ 40
2 WS (3 CPU cycles)	60 <HCLK ≤ 90	48 < HCLK ≤ 72	44 < HCLK ≤ 66	40 < HCLK ≤ 60
3 WS (4 CPU cycles)	90 <HCLK ≤ 120	72 < HCLK ≤ 96	66 <HCLK ≤ 88	60 < HCLK ≤ 80
4 WS (5 CPU cycles)	120 <HCLK ≤ 150	96 < HCLK ≤ 120	88 < HCLK ≤ 110	80 < HCLK ≤ 100
5 WS (6 CPU cycles)	150 <HCLK ≤ 180	120 <HCLK ≤ 144	110 < HCLK ≤ 132	100 < HCLK ≤ 120
6 WS (7 CPU cycles)		144 <HCLK ≤ 168	132 < HCLK ≤ 154	120 < HCLK ≤ 140
7 WS (8 CPU cycles)		168 <HCLK ≤ 180	154 <HCLK ≤ 176	140 < HCLK ≤ 160
8 WS (9 CPU cycles)			176 <HCLK ≤ 180	160 < HCLK ≤ 168

Figure 5 Number of wait states according to CPU clock (HCLK) frequency

2.3 Erase Operation

1. Check whether the Flash memory interface is busy by checking the BSY bit in the Flash status Register (FLASH_SR).

2. **For Sector Erase,**

- Setting the number of SNB bit in FLASH_CR to choose which sector to erase.

For Bank Erase,

- Setting MER or MER1 in FLASH_CR bit to choose which bank to erase. MER for bank1 and MER1 for bank2.

For Mass Erase,

- By setting MER and MER1 bit in FLASH_CR to erase both bank.

3. Set the STRT bit in FLASH_CR register to start erasing.
4. Wait BSY bit to be cleared.

2.4 Programming Operation

1. Check whether the Flash memory interface is busy by checking the BSY bit in the Flash Status Register (FLASH_SR).
2. Setting PG bit in Flash Control Register (FLASH_CR) to enable programming.
3. Set parallelism size (x8, x16, x32 and x64) by configuring the PSIZE[1:0] bit of the FLASH_CR register. Parallelism size shows the number of bits to be program to the flash.
4. Wait BSY bit to be cleared.

2.5 DMA memory-to-memory transfer Operation

The data transfer direction has been set to be memory-to-memory from the DIR[1:0] it of the DMA_SxCR. The memory data size MSIZE[1:0] and the peripheral data size PSIZE[1:0] has been configured depends on the number of bits of each data memory to be transferred. The priority has set to be very high at the bit PL[1:0]. The TCIE bit in DMA_SxCR has been enabled so that after the transfer been completed, it will enable the interrupt flag CTCIFx at the register DMA high interrupt flag clear register (DMA_HIFCR)

For the single memory transfer, the stream and the channel number has to be configure properly. The stream is selected by offsetting the address by using the formula below.

Address offset: Register Offset + $0x18 \times \text{stream number}$

The register offset shows the address to be offset for each specific register. The $0x18$ of address includes the configuration register (DMA_SxCR), data register (DMA_SxNDTR), peripheral address register (DMA_SxPAR), memory 0 address register (DMA_SxM0AR), memory 1 address register (DMA_SxM1AR) and FIFO control register (DMA_SxFCR). Each of the register takes $0x04$ of the address. The Channel number is selected by configuring the CHSEL[2:0] bits in the DMA_SxCR register.

For multiple memory transfer, PINC bit and the MINC bit of the DMA_SxCR has to be set. These bits has to be set to increment the address pointer of peripheral (DMA) and the address pointer of memory (Flash Memory) after each memory has been transfer.

After all configuration has been done, the DMA has been enabled by setting the EN bit in the DMA_SxCR register.

CHAPTER III: RESULT & DISCUSSION

In this chapter will show the result for all the operation that has been discussed in the previous chapter and how to configure by using the C programming language to program into the ARM Cortex®-M4 Microcontroller. The written program was compiled and debugged by the software CoCoX CoIDE version 2.0.3.

The data structure created for the embedded flash memory interface is as shown below.

```
typedef struct FLASH_TypeDef_t FLASH_TypeDef;
struct FLASH_TypeDef_t
{
    volatile uint32_t ACR;
    volatile uint32_t KEYR;
    volatile uint32_t OPTKEYR;
    volatile uint32_t SR;
    volatile uint32_t CR;
    volatile uint32_t OPTCR;
    volatile uint32_t OPTCR1;
};
```

Figure 6 Data Structure of the embedded flash memory interface

The base address for the embedded flash memory interface is 0x40023C00. Hence, macros has been created to represent the base address for the embedded flash memory interface.

```
#define FLASH_BASE_ADDRESS 0x40023C00
#define FLASH ((FLASH_TypeDef*) FLASH_BASE_ADDRESS)
```

Figure 7 macros created to represent the base address of flash

From Figure 7, the macro FLASH is now pointing to the data structure that been created in Figure 6. All the element inside the data structure is declared with the type **volatile uint32_t**. By doing that, each element of the data structure will be having the size 0x4 which will give an offset of 0x4 between to element in the data structure. All the element inside the data structure represent the existence register of the embedded flash memory interface.

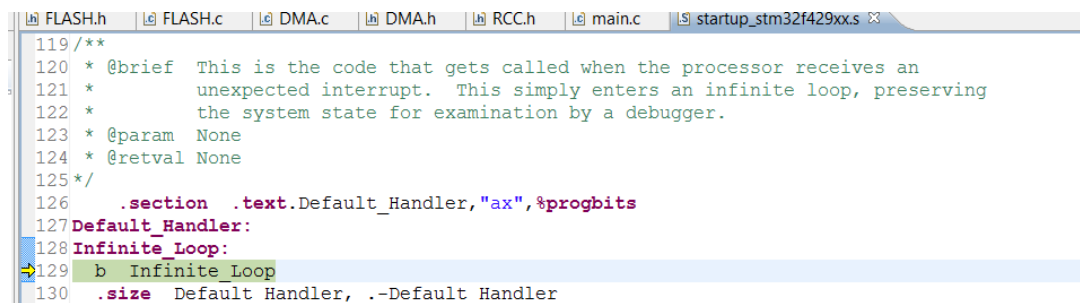
3.1 Unlocking Flash Control Register

From the previous chapter, the FLASH_CR can be unlocked by storing the sequence KEY1 and KEY2 into FLASH_KEYR register. The program function written is as shown below.

```
void unlockFlashCR() {  
    FLASH->KEYR = 0x45670123;  
    FLASH->KEYR = 0xCDEF89AB;  
}
```

Figure 8 unlockFlashCR function

The function shown in figure 8 has been called in the main and been debugged. After the flash has been unlocked, it has to be locked at the end of the program otherwise there will be error occur. The sequence has to be the same as above otherwise the FLASH_CR will be not unlocked. When the incorrect sequence has been keyed in, the compiler will be caught in the infinity loop as shown below.



The screenshot shows a code editor with several tabs: FLASH.h, FLASH.c, DMA.c, DMA.h, RCC.h, main.c, and startup_stm32f429xx.s. The active file is startup_stm32f429xx.s, showing assembly code. Lines 119-125 are comments. Line 126 is a section directive: `.section .text.Default_Handler,"ax",&progbits`. Line 127 is `Default_Handler:`. Line 128 is `Infinite_Loop:`. Line 129 is `b Infinite_Loop`, which is highlighted with a yellow arrow pointing to it, indicating a branch to the start of the loop. Line 130 is `.size Default_Handler, .-Default_Handler`.

Figure 9 infinity loop caught by the compiler

When the correct KEY sequence has been keyed into the register, the LOCK bit in the FLASH_CR will become '0' which means the control register has been unlocked.

The value in the flash register has been checked by the function checkFlashReg. The function will check the value in the SR reg, CR reg and the OPTCR reg. The function checkFlashReg is as shown below.

```
23 void checkFlashReg() {  
24     uint32_t checkSR      = FLASH->SR;  
25     uint32_t checkCR      = FLASH->CR;  
26     uint32_t checkOPTCR   = FLASH->OPTCR;  
27 }
```

Figure 10 function checkFlashReg

The register has been checked before it unlock and after it unlock. The scenarios are as shown in figures below.

```

21 checkFlashReg();
22 unlockFlashCR();
23 checkFlashReg();

```

Figure 11 unlockFlashReg function called in the main

Value of CR reg before (line 21):

Name	Value
checkSR	0x00000000
checkCR	0x80000000

Value of CR reg after (line 23):

Name	Value
checkSR	0
checkCR	0

From the result shown above, the CR has been successfully unlocked by looking at the value after of the CR reg is zero. Which means that the LOCK bit has been set to zero.

3.2 Read Operation

For the read operation, the frequency of the clock (HCLK) has been obtained by the function `getSystemClock`. The function is obtained from the Steven's project. This function is only to determine the HCLK frequency that produced to the flash memory interface. The function is a shown in the figure below.

```

12 uint32_t getSystemClock(){
13     int divM, xN, divP, divAHB;
14     int sysClock;
15     int enbAHBPrescale = RCC->RCC_CFGR & 256;
16
17     if(enbAHBPrescale == 0)
18         divAHB = 1;
19     else
20         divAHB = 1 << (((RCC->RCC_CFGR >> 4) & 7) + 1);
21
22     xN = (RCC->RCC_PLLCFGR >> 6) & 511;
23     divM = (RCC->RCC_PLLCFGR & 63);
24     divP = 1 << (((RCC->RCC_PLLCFGR >> 16) & 3) + 1);
25
26     if(((RCC->RCC_CFGR & 0xC) >> 2) == 0)
27         sysClock = INTERNAL_CLOCK / divAHB;
28     else if(((RCC->RCC_CFGR & 0xC) >> 2) == 1)
29         sysClock = CRYSTAL_CLOCK / divAHB;
30     else {
31         if(((RCC->RCC_PLLCFGR >> 22) & 1) == 0)
32             sysClock = (INTERNAL_CLOCK * xN) / (divM * divP * divAHB);
33         else
34             sysClock = (CRYSTAL_CLOCK) / (divM * divP * divAHB) * xN;
35     }
36     return sysClock;
37 }

```

Figure 12 function `getSystemClock`

The function has been called in the main to determine the clock frequency. The function been called is as shown below.

```
uint32_t HCLK=getSystemClock();
```

The value of the clock frequency will be stored at HCLK. The value shows after debugging is as shown below.

Table 3 value for the beginning of the debugging process

Name	Value
flashError	0
guard	0
i	14
latency	5
HCLK	180000000
readAdd	0x08104000
readAdd1	0x08104038
SRC_Const_Buffer	0x2002ff98
test_Buffer	0x2002ff70

From the result obtained above, the HCLK was 180 MHz. The HCLK was obtained from the external crystal clock that passed through many divider which can refer to the figure 3. The divider includes the M, P, xN and AHB prescaler.

The wait state (LATENCY) also has been checked by using the function as shown below.

```
29 uint32_t checkLatency() {
30     return FLASH->ACR & 15;
31 }
```

Figure 13 function checkLatency

From the function shown above, it mask the first five bit of the FLASH_ACR which is the LATENCY[3:0] bit. The function is called in the main to determine the number of the wait state. The function been called is as shown below.

```
int latency=checkLatency();
```

The value of the latency will be stored at latency. The value shows after debugging is also shown in table 3. The latency shown in the table was 5. Thus, by referring to the figure 5 the obtained values for frequency and the latency in table 3 was correct.

To observe whether the data has been successful modified by any operation, a pointer has to be created to point to one of the address that is located in the flash memory. The pointer created is as shown below.

```
13 WRITE_SIZE *readAdd=TARGET_ADD;
```

The WRITE_SIZE and the TARGET_ADD are the macro created to simplify the job when configuring the flash memory.

```
65 #define WRITE_SIZE uint32_t
66 #define TARGET_ADD ((WRITE_SIZE *)0x08104000)
```

The WRITE_SIZE shows the size going to be written into the flash and the TARGET_ADD is the target address chosen to perform any operation. For the readAdd, the type declared was the type which being written to the flash. The TARGET_ADD also following the same size. The result after debug is as shown below.

readAdd	0x08104000
*readAdd	0xffffffff

The first line shows the address pointed by the pointer while the second line shows the value inside the pointed address.

3.3 Erase Operation

In this section will discuss how the sector erase and the bank erase to be successfully been configured. There are unable to erase sector 0, bank 1 and to perform Mass erase for this program. That is because the current program written is stored in the sector 0 in the bank 1. If sector 0 is being erased, the program will give an error as shown below.

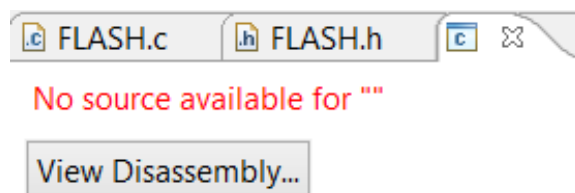


Figure 14 error message shown when erasing sector0/bank1/Mass erase

In the following section, Sector erase will choose to erase sector 13 and the bank erase will choose to erase the bank 2.

i) Sector Erase

The sector erase function is as shown below.

```
42 void sectorErase(uint32_t sectorNum){
43     while(checkBusy()) {}
44     checkFlashReg();
45     FLASH->CR &= ~FLASH_CR_SER;
46     FLASH->CR &= ~FLASH_CR_SNB;
47     FLASH->CR &= ~FLASH_CR_STRT;
48     FLASH->CR |= FLASH_CR_SER;
49     FLASH->CR |= sectorNum << FLASH_CR_SNB_bit;
50     FLASH->CR |= FLASH_CR_STRT;
51     checkFlashReg();
52     while(checkBusy()) {}
53     FLASH->CR &= ~FLASH_CR_SER;
54     FLASH->CR &= ~FLASH_CR_SNB;
55     checkFlashReg();
56 }
```

Figure 15 function sector erase

From the Figure 15 above, the function is passing in the sector number of the Flash Memory. At the beginning of the function, the while loop is to check whether there are any flash memory operation ongoing. The function checkBusy is as shown below.

```
3 int checkBusy() {
4     return ((FLASH->SR>>16) & 1);
5 }
```

Figure 16 function checkBusy

The return value for the checkBusy function is following the BSY bit in the SR register. So at the while loop shown in Figure 15, the system will keep looping if the function return '1' until the BSY bit has returning back to '0'. The while loop also has been used in the end of the function to ensure the system is ready to perform the next flash operation.

From the figure 15, the checkFlashReg function has been called again to check the data before and after configure of the flash registers. The check function at line 44 of the figure 15 is giving the result as below.

Name	Value
checkSR	0
checkCR	0

From the result observed above, the initial value for both SR and CR are '0'. The result observed after the configuration is as shown below.(the check function at line 51.)

Name	Value
checkSR	0
checkCR	0x0000008a

From the result obtained above, the value has been successfully configured into the CR register. The configuration are the SER and the STRT bit in CR has been set. The value 10001 has been stored to the SNB[4:0] bit to choose the sector 13 to be erased.

After the configuration, BSY bit has been checked. When it is not busy, it will jump out of the loop at the same time the STRT bit in the CR register has been set to '0' by the software. All the other bits of the register has been reset so that it is ready to perform the next flash operation.

The function called in the main is as shown in the figure below.

```

30 //Sector Erase
31 if(guard) {
32     sectorErase(SECTOR13);
33     flashError=checkFlashError();
34     guard=0;
35 }

```

Figure 17 sectorErase function called in the main

At line 32, the macro "SECTOR13" has been passed in to the function. The macros created for each of the SECTOR is as shown below.

```

68 //Sector in bank 1      82 //Sector in bank 2
69 #define SECTOR0        0 83 #define SECTOR12        16
70 #define SECTOR1        1 84 #define SECTOR13        17
71 #define SECTOR2        2 85 #define SECTOR14        18
72 #define SECTOR3        3 86 #define SECTOR15        19
73 #define SECTOR4        4 87 #define SECTOR16        20
74 #define SECTOR5        5 88 #define SECTOR17        21
75 #define SECTOR6        6 89 #define SECTOR18        22
76 #define SECTOR7        7 90 #define SECTOR19        23
77 #define SECTOR8        8 91 #define SECTOR20        24
78 #define SECTOR9        9 92 #define SECTOR21        25
79 #define SECTOR10       10 93 #define SECTOR22        26
80 #define SECTOR11       11 94 #define SECTOR23        27

```

Figure 18 sector macros created for both bank 1 and bank 2

Looking back to the Figure 17, the guard technique has been used. Which is the 'if' statement before performing the sectorErase. The value of the guard has been initialize to be zero so that it will not perform any operation unless the guard value has been modified manually. In every operation of the program, a break point has been placed at the 'if' statement. So if that

particular operation has been chosen to perform, the guard has to be change manually to '1'. The modifying value of the guard is as shown below.

Name	Value
flashError	0
guard	1
i	14
latency	5
HCLK	180000000
readAdd	0x08104000
readAdd1	0x08104038
SRC_Const_Buffer	0x2002ff98

From the result shown in the figure above, the guard value has been modified to '1' manually to perform the particular operation.

The sector erase operation has been illustrate by using the readAdd pointer that has been discussed in the previous section. Initially some value has been program to the address 0x08104000. The initial value has been shown below.

▲ readAdd	0x08104000
*readAdd	0x00011111

When the program reach the breakpoint in figure 17 at line 31, the value of the guard has been modified manually to enable performing sector erase. After it run though the code in figure 17 without any error, the system has to be reset only the erased value can be observed. The value after been erased is as shown below.

▲ readAdd	0x08104000
*readAdd	0xffffffff

From the result observed above, the result has been successfully been erased as the value of *readAdd has become 0xffffffff.

ii) Bank Erase

The bank erase function is as shown below.

```
58 void bankErase(int bankNum) {  
59     while(checkBusy()) {}  
60     checkFlashReg();  
61     if(bankNum==1) {  
62         FLASH->CR &= ~FLASH_CR_MER;  
63         FLASH->CR |= FLASH_CR_MER;  
64     }  
65     else{  
66         FLASH->CR &= ~FLASH_CR_MER1;  
67         FLASH->CR |= FLASH_CR_MER1;  
68     }  
69     checkFlashReg();  
70     FLASH->CR &= ~FLASH_CR_STRT;  
71     FLASH->CR |= FLASH_CR_STRT;  
72     checkFlashReg();  
73     while(checkBusy()) {}  
74     FLASH->CR &= ~FLASH_CR_MER;  
75     FLASH->CR &= ~FLASH_CR_MER1;  
76     checkFlashReg();  
77 }
```

Figure 19 function bankErase

From the Figure 19 above, the bankErase function is passing in the bank number that to be erase. The bankErase operation is most likely the same as the operation of the sectorErase, the only difference is that the enabling bit of the CR register has been changed to MER or MER rather than SER for the sector erase.

The choosing of MER and MER1 is decide on whether the bankNum is '1' or '2'. Value '1' stands for erasing bank one and value '2' stands for erasing bank two. The flash register has been checked by the checkFlashReg function. The value before and after the configuration is as shown below.

The CR register before configuration (line 60):

Name	Value
checkSR	0
checkCR	0

The CR register after configuration (line 72):

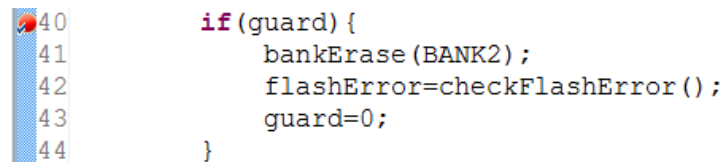
Name	Value
checkSR	0x00010000
checkCR	0x00018000

For the result obtained above, it is erasing the bank 2 of the flash memory. Note that Bank 1 is unable to be erased, the reason has already discussed in the previous chapter.

For the result obtained for the after configuration, the BSY bit of the SR register has been set. This will not cause any problem to the program, that is because the while loop mentioned previously can be solve the problem.

The bank erase will take more time than the sector erase, because for the bank erase it need time to erase all sector in the bank. Each bank consist of 12 sectors, so the bank erase will take 12 times slower than the sector erase.

The function called in the main is as shown in the figure below.



```
40
41     if(guard) {
42         bankErase(BANK2);
43         flashError=checkFlashError();
44         guard=0;
    }
```

Figure 20 bankErase function called in the main

At line 32, the macro “BANK2” has been passed in to the function. The macros created for each bank is as shown below.

```
96 #define BANK1    1
97 #define BANK2    2
--
```

The guard technique has also been used for the bankErase operation. The guard will set to ‘1’ when bankErase operation is needed to be performed.

The bank erase operation has been illustrate by using the readAdd pointer that has been discussed in the previous section. The discussion will be similar to the sector erase.

ii) Mass Erase

The mass erase operation was unable to be perform because the bank 1 was unable to be erased.

3.4 Programming Operation

The flashProgram function is as shown below.

```
117 void flashProgram(int PSIZEsel, uint64_t value, uint32_t *Address) {
118     int error=0;
119     WRITE_SIZE *write;
120
121     write=((uint32_t *) (Address));
122
123     while(checkBusy()) {}
124     checkFlashReg();
125     flashProgramConfig(PSIZEsel);
126     flashProgramEn();
127     error=checkFlashError();
128     checkFlashReg();
129     switch(PSIZEsel) {
130         case x8 :value &= 0xFF;break;
131         case x16:value &= 0xFFFF;break;
132         case x32:value &= 0xFFFFFFFF;break;
133         case x64:value &= 0xFFFFFFFFFFFFFFFF;break;
134     }
135     *write=value;
136     while(checkBusy()) {}
137     error=checkFlashError();
138     flashProgramDisable();
139 }
```

Figure 21 function flashProgram

The function is passing in three element which are the PSIZE select, the value to program and the address selected to program. For each PSIZE has been selected, the function will mask the value (with size x64) to the size been selected.

From the Figure 21 above, a pointer has been created for the programming purposes. The pointer has been pointed to the address that has to be program. The pointer will only pointed to the value when all the configuration has been done.

There are also error checking function inside the flashProgram function. The error checking function is the checkFlashError function. The function is as shown below.

```
7 int checkFlashError() {
8     int seqErr, parallelErr, alignErr, writeProtErr, OperationErr;
9
10    checkFlashReg();
11    seqErr = (FLASH->SR>>7)&1;
12    parallelErr = (FLASH->SR>>6)&1;
13    alignErr = (FLASH->SR>>5)&1;
14    writeProtErr = (FLASH->SR>>4)&1;
15    OperationErr = (FLASH->SR>>1)&1;
16
17    if(seqErr!=0 || parallelErr!=0 || alignErr!=0 || writeProtErr!=0 || OperationErr!=0)
18        return 1;
19    else
20        return 0;
21 }
```

Figure 22 function checkFlashError

When there are any error sense in the SR register, the system will be caught at the breakpoint at line 18 in Figure 22. The description of each error is as shown below.

Programming sequence error (seqErr):

This bit will be set when the FLASH_CR was not correctly been configured.

Programming parallelism error (parallelErr):

This error bit shows that the writing value parallelism size doesn't match the configuration of parallelism size set in the FLASH_CR.

Programming alignment error (alignErr):

This error bit shows that the program data has crossed the 128-bit row boundary of the Flash Memory.

The flashProgramConfig and the flashProgramEn function also has been included into the function checkFlashError. These function have been shown as below.

```
104 void flashProgramEn() {
105     FLASH->CR &= ~FLASH_CR_PG;
106     FLASH->CR |= FLASH_CR_PG;
107     checkFlashReg();
108 }
109
110 void flashProgramConfig(int PSizeSel) {
111     FLASH->CR &= ~FLASH_CR_PG;
112     FLASH->CR &= ~FLASH_CR_PSIZE;
113     FLASH->CR |= PSizeSel << FLASH_CR_PSIZE_bit;
114     checkFlashReg();
115 }
```

Figure 23 function flashProgramEn and flashProgramConfog

The function flashProgramConfig is to pass in the value of the PSize and config the value of the PSize to the CR register of the flash. In this function, the programming bit (PG) has been ensured to be disabled.

After the PSize has been configured in to the CR, the flashProgramEn only been called to enable the PG bit of the CR to start programming. The changes of the register value has been debugged as shown below.

The CR register before configuration (line 124):

Name	Value
checkSR	0
checkCR	0

The CR register after configuration (line 128):

Name	Value
checkSR	0
checkCR	0x00000201

From the result shown above, the correct value has been configured into the CR register. Also the system does not caught by the breakpoint in figure 22. These shows that there are no error while configuring the CR.

After the PG bit has been enabled the program value masked has been pointed by the write pointer and has been dereferenced. If there are any error sense by the error checking function and the system is not busy, the value is now successfully programmed into the flash memory address.

The function called in the main is as shown in the figure below.

```
48      //Flash Program
49      if (guard) {
50          flashProgram(x32, 0x23872, TARGET_ADD);
51          flashError=checkFlashError();
52          guard=0;
53      }
```

Figure 24 flashProgram function called in the main

Again, the flashProgram are using the same guard technique that is already discussed in the previous part.

The program operation has been illustrate by using the readAdd pointer that has been discussed in the previous section. Initially the selected address 0x08104000 was a empty address (0xffffffff). The initial value has been shown below.

readAdd	0x08104000
*readAdd	0xffffffff

When the program reach the breakpoint in figure 24, the value of the guard has been modified manually to enable performing flashProgram. After it run though the code in figure 24 without any error, the system has to be reset only the programed value can be observed. The value after been erased is as shown below.

readAdd	0x08104000
*readAdd	0x00023872

From the result observed above, the result has been successfully been programmed as the value of *readAdd has become 0x00023872.

The flashProgram function has been tested in different number of P_SIZE. For a different P_SIZE number chosen, the WRITE_SIZE that mentioned previously has to change according to the P_SIZE. And the P_SIZE number the input to the flashProgram function also has to change accordingly. For example, the configuration of the flashProgram has been changed as below and the macro WRITE_SIZE has been changed to uint8_t.

```
flashProgram(x8,0x23872,TARGET_ADD);
```

By doing that, it is programming the value with P_SIZE of x8. The result observed for the configuration above is as shown below.

readAdd	0x08104000
*readAdd	0x72

When the P_SIZE choose to be x8, the program value seen will only the first 8-bit of the value. Since the readAdd is declared as uint8_t by the WRITE_SIZE, there will be 8 bit can be seen in the results above. But the actual 64-bit value of the memory value will be 0xfffffffffff72. Hence, the result above proved that the program is functioning correctly.

An error has been forced to the program, the occurrence of the error will be caught by the function checkFlashError. The error been caught has been shown below.

```

7 int checkFlashError() {
8     int seqErr, parallelErr, alignErr, writeProtErr, OperationErr;
9
10    checkFlashReg();
11    seqErr    = (FLASH->SR>>7) &1;
12    parallelErr = (FLASH->SR>>6) &1;
13    alignErr   = (FLASH->SR>>5) &1;
14    writeProtErr = (FLASH->SR>>4) &1;
15    OperationErr = (FLASH->SR>>1) &1;
16
17    if(seqErr!=0 || parallelErr!=0 || alignErr!=0 || writeProtErr!=0 || OperationErr!=0)
18        return 1;
19    else
20        return 0;
21}

```

Figure 25 Error occurrence of the program

Name	Value
seqErr	0
parallelErr	1
alignErr	0
writeProtErr	0
OperationErr	0

From the result shown above, the error was produced by configuring PSIZE x32 and the WRITE_SIZE is set to uint8_t. Thus, the error will show the parallelism error in the SR register.

As mentioned in the previous chapter, the Flash is unable to store a 64-bit value. This scenarios has been proven as follow, the PSIZE has set to x64 and the WRITE_SIZE is set to uint64_t. So that a 64 bit value is ready to program to the flash memory. The configuration of the function flashProgram is as shown below.

```
flashProgram(x64, 0x10010, TARGET_ADD);
```

The result obtained by the configuration above is as shown below.

readAdd	0x08104000
*readAdd	0xffffffffffffff

From the result obtained above, the value of the address was 0xffffffffffffff. The result shows that the value 0x10010 was not successfully program to the flash. The expected result at the flash memory should be 0xffffffff10010 but the value in the memory still remain empty. This operation does not provide any error just that the value was unable to store into the flash memory.

3.5 DMA memory-to-memory transfer Operation

Before operating the DMA, the Clock has to be unreset and enabled. The data structure for the RCC is as shown below.

```
6 typedef struct RCC_t RCC_TypeDef;
7 struct RCC_t{
8     volatile uint32_t RCC_CR;
9     volatile uint32_t RCC_PLLCFGR;
10    volatile uint32_t RCC_CFGR;
11    volatile uint32_t RCC_CIR;
12    volatile uint32_t RCC_AHB1RSTR;
13    volatile uint32_t RCC_AHB2RSTR;
14    volatile uint32_t RCC_AHB3RSTR;
15    volatile uint32_t RESERVE0[1];
16    volatile uint32_t RCC_APB1RSTR;
17    volatile uint32_t RCC_APB2RSTR;
18    volatile uint32_t RESERVE1[2];
19    volatile uint32_t RCC_AHB1ENR;
20    volatile uint32_t RCC_AHB2ENR;
21    volatile uint32_t RCC_AHB3ENR;
22    volatile uint32_t RESERVE2[1];
23    volatile uint32_t RCC_APB1ENR;
24    volatile uint32_t RCC_APB2ENR;
25    volatile uint32_t RESERVE3[2];
26    volatile uint32_t RCC_AHB1LPENR;
27    volatile uint32_t RCC_AHB2LPENR;
28    volatile uint32_t RCC_AHB3LPENR;
29    volatile uint32_t RESERVE4[1];
30    volatile uint32_t RCC_APB1LPENR;
31    volatile uint32_t RCC_APB2LPENR;
32    volatile uint32_t RESERVE5[2];
33    volatile uint32_t RCC_BDCR;
34    volatile uint32_t RCC_CSR;
35    volatile uint32_t RESERVE6[2];
36    volatile uint32_t RCC_SSCGR;
37    volatile uint32_t RCC_PLLI2SCFGR;
38    volatile uint32_t RCC_PLLSAICFGR;
39    volatile uint32_t RCC_DCKCFGR;
40};
```

Figure 26 Data structure created for the RCC

The base address for the RCC is 0x40023800. Hence, macros has been created to represent the base address for the RCC. The macro created was as shown below.

```
42 #define RCC ((RCC_TypeDef*) 0x40023800)
```

With the aid of the data structure, the DMA2UnresetEnableClock function has been created. The function DMA2UnresetEnableClock created has been shown below.

```
3 void DMA2UnresetEnableClock() {
4     RCC->RCC_AHB1RSTR &= ~ RCC_AHB1RSTR_DMA2RST;    //Reseting
5     RCC->RCC_AHB1ENR  &= ~RCC_AHB1ENR_DMA2EN;
6     RCC->RCC_AHB1ENR  |= RCC_AHB1ENR_DMA2EN;         //Enable Clock
7
8     uint32_t checkAHB1ENR = RCC->RCC_AHB1ENR ;
9     uint32_t checkRCC_AHB1RSTR = RCC->RCC_AHB1RSTR;
10 }
```

Figure 27 function DMA2UnresetEnableClock

The function in the Figure 27 basically just to unreset the clock and enable it. The value of the register has been observed by the variable checkAHB1ENR and checkRCC_AHBRSTR. The observed value has been shown below.

Name	Value
checkAHB1ENR	0x00500000
checkRCC_AHBRSTR	0

From the result shown above, the DMA2EN bit in AHB1ENR register has been enabled and the AHBRSTR register has been reset.

To configure the DMA register, the data structure has to be created. The created data structure for DMA is as shown below.

```

5 typedef struct DMA_Type DMA_reg;
6 typedef struct{
7     volatile uint32_t CR;
8     volatile uint32_t NDTR;
9     volatile uint32_t PAR;
10    volatile uint32_t M0AR;
11    volatile uint32_t M1AR;
12    volatile uint32_t FCR;
13 }Stream_t;
14
15 struct DMA_Type{
16     volatile uint32_t LISR;
17     volatile uint32_t HISR;
18     volatile uint32_t LIFCR;
19     volatile uint32_t HIFCR;
20     Stream_t S0;
21     Stream_t S1;
22     Stream_t S2;
23     Stream_t S3;
24     Stream_t S4;
25     Stream_t S5;
26     Stream_t S6;
27     Stream_t S7;
28 };

```

Figure 28 Data Structure for the DMA

From the Figure 28 above, S0 to S7 stands for the existence of the stream in the DMA. The registers available for each of the stream are the element inside the first data structure in the Figure 28.

The base address for the DMA2 is 0x40026400. Note that the data structure was the same for both DMA1 and DMA2. Hence, macros has been created to represent the base address for the DMA2. The macro created was as shown below.

```

30 #define DMA2 ((DMA_reg*) 0x40026400)

```

From the discussion in previous chapter, empty stream has to be chosen to transfer the memory. The stream and the channel chosen for the transferring of memory to flash is stream 7 and channel 0.

The DMA control register of stream has been configured by the function configDMA2s7CR. The configuration of configDMA2s7CR function is as shown below.

```

10 void configDMA2s7CR(int direction,int PSize,int MSize,int PL,int CHSEL) {
11     checkDMAReg();
12
13     DMA2->S7.CR &= ~DMA_SxCR_EN;           //Stream Disable
14
15     DMA2->S7.CR |= 0x0000061e;
16
17     DMA2->S7.CR &= ~DMA_SxCR_DIR;
18     DMA2->S7.CR |= direction << DMA_SxCR_DIR_bit;
19
20     DMA2->S7.CR &= ~DMA_SxCR_PSize;
21     DMA2->S7.CR |= PSize << DMA_SxCR_PSize_bit;
22
23     DMA2->S7.CR &= ~DMA_SxCR_MSize;
24     DMA2->S7.CR |= MSize << DMA_SxCR_MSize_bit;
25
26     DMA2->S7.CR &= ~DMA_SxCR_PL;
27     DMA2->S7.CR |= PL << DMA_SxCR_PL_bit;
28
29     DMA2->S7.CR &= ~DMA_SxCR_CHSEL;
30     DMA2->S7.CR |= CHSEL << DMA_SxCR_CHSEL_bit;
31
32     checkDMAReg();
33 }

```

Figure 29 function configDMA2s7CR

Before the configuration, the stream enable bit has been ensured to be disabled. The value 0x0000061e has been stored into the control register of the stream 7. It includes the enabling bits DMEIE, TEIE, HTIE, TCIE, PINC and MINC. The DMEIE, TEIE, HTIE and TCIE in the CR register is to enable the interrupt for the status register LISR, HISR, LIFCR and HIFCR register. So that when there are any error occurred can be observed from these status register. While for the PINC and MINC, it is enabled when needed to transfer multiple memory.

The function is passing in input of direction, PSize, MSize, PL and CHSEL. These bits have already discussed in the previous chapter. The value of the CR register has been checked by the function checkDMAReg. The checkDMAReg function is as shown below.

```

4 void checkDMAReg() {
5     uint32_t checkHISR= DMA2->HISR;
6     uint32_t chcekS7CR= DMA2->S7.CR;
7 }
8

```

Figure 30 function checkDMAReg

The function has been called before and after the configuration of control register of CR in the function configDMA2s7CR. The changes of the register value has been debugged as shown below.

The CR register of Stream 7 before configuration (line 11):

Name	Value
checkHISR	0
chcekS7CR	0

The CR register of Stream 7 after configuration (line 32):

Name	Value
checkHISR	0
chcekS7CR	0x0003569e

From the result shown in the figures above, the CR is correctly been configured. After the configuration of the CR, the destination address and the source address has to be stored into the M0AR and PAR respectively. The program that store the address inside these register is as shown below.

```

35 void DMA_memcpy( uint32_t *pDstAddr, uint32_t *pSrcAddr, int uSize ){
36     DMA2->S7.PAR = (uint32_t)pSrcAddr;    // source address
37     DMA2->S7.M0AR = (uint32_t)pDstAddr;    // destination address
38     DMA2->S7.NDTR = uSize;                // Number of data items to transfer
39 }

```

Figure 31 DMA_memcpy

The inputs of the function are *pDstAddr, *pSrcAddr and uSize. The pointer pDstAddr and pSrcAddr will point to the address that passed in to the function. The address is then stored into the PAR register and the M0AR register of stream 7. For the uSize, it shows the number of memory address to be transferred from the source to destination. The uSize will be stored into the NDTR register of stream 7.

After the DMA configuration is ready, the configuration for the flash programming operation still has to be done so that the memory value are able to be transfer to the flash memory. The DMA configuration and flash programming configuration functions that been called in the main has been shown in the figure below.

```

57      //Program Flash memory by DMA transfer
58      if(guard) {
59          configDMA2s7CR(MemToMem,x32,x32,VeryHighPrio,CHANNEL0);
60          DMA_memcpy(TARGET_ADD, SRC_Const_Buffer, TRANSFER_LEN );
61          flashProgramConfig(x32);
62          flashProgramEn();
63          enableDMA();
64          while(checkBusy()) {}
65          flashError=checkFlashError();
66          flashProgramDisable();
67          clearDMAHighIntrrFlag();
68          flashError=checkFlashError();
69          guard=0;
70      }

```

Figure 32 the DMA function call in the main

The element that passing into the function configDMA2s7CR was declared as macros as shown below.

```

58 #define PeriToMem      0
59 #define MemToPeri      1
60 #define MemToMem       2
61
62 #define LowPrio        0
63 #define MediumPrio     1
64 #define HighPrio       2
65 #define VeryHighPrio   3
66
67 #define CHANNEL0       0
68 #define CHANNEL1       1
69 #define CHANNEL2       2
70 #define CHANNEL3       3
71 #define CHANNEL4       4
72 #define CHANNEL5       5
73 #define CHANNEL6       6
74 #define CHANNEL7       7

```

Figure 33 macros defined to configure the configDMA2s7CR function

Note that the function is configured as memory-to-memory transfer, PSize x32, MSize x32, very high priority and channel 0. The programming configuration is the same as the previous configuration.

The address that passed into the function DMA_memcpy are the TARGET_ADD as the destination address and the address of the first element of array SRC_Const_Buffer as the source address. The declaration of the array SRC_Const_Buffer is as shown below.

```
WRITE_SIZE SRC_Const_Buffer[]={12,23,15,10,12};
```

This array above was considered as a RAM memory. The TRANSFER_LEN has been defined as a macro to represent the amount data to be transferred. The macro declared is as shown below.

```
6 #define TRANSFER_LEN    5
```

From the macro shown above, the transfer length is 5, so there are 5 element to be transferred for the DMA, the PINC and the MINC will increment the address according to the transfer length defined.

After all the configuration has been done, the flash program bit and the DMA enable bit has been enabled to start the transferring of memory. After enabling the bit, the BSY bit of the FLASH_SR has been check to determine whether there are any operation ongoing. When BSY bit has become '0', which means the data has been completely been transferred.

After the transferring of the data, the flash has been checked whether there are any existence of errors. The flash program bit and the DMA high interrupt flag has been disabled after all the operation has been done.

While debugging for the DMA transfer, the system will be caught by the breakpoint as shown in line 58 shown in Figure 32. The guard has been set to '1' to enable the system to perform the operation set in Figure 32. After passing through all the operation shown in Figure 32 without any error, the value of address has been observed by the pointer readAdd1 after the system has been reset. The readAdd1 is a pointer which is similar to the pointer readAdd that mentioned in the previous section. The reading of the readAdd1 is as shown below.

```
26         for(i=0;i<(TRANSFER_LEN+1);i++)  
27             readAdd1++;
```

From the 'for' loop created above is to let the readAdd1 to read multiple data from multiple addresses as the i increases. One more address has been read to determine whether there are the end of the data been transferred. The result observed when debugging is as shown below.

i	0	i	1
▲ readAdd1	0x08104000	▲ readAdd1	0x08104004
*readAdd1	12	*readAdd1	23
i	2	i	3
▲ readAdd1	0x08104008	▲ readAdd1	0x0810400c
*readAdd1	15	*readAdd1	10
i	4	i	5
▲ readAdd1	0x08104010	readAdd1	0x08104014
*readAdd1	12	*readAdd1	0xffffffff

Figure 34 Result for the DMA transfer of 5 data

From the Figure shown above, the data of the readAdd1 for every address increment of 0x4 has been shown. The result above was obtained by transferring 5 element, and thus the 6th element which is in flash address (0x08104014) has been observed to be empty. From the result obtained at the figure above, the readAdd1 matches the data from the SRC_Const_Buffer memory. Therefore, the memory has been successfully transferred by the DMA memory-to-memory transfer to the Flash.

To change the amount of data to be transferred, the macro of the TRANSFER_LEN can be modified. While to change the data to be transfer can be change from the SRC_Const_Buffer array.

A string also can be transferred by the DMA transfer. The source address has been replaced by the address of the first character of the string. The string declared is as shown below.

```
18 char *str="JANG SING";
```

DMA_memcpy will now pass in the address of str rather than passing in the address of the SRC_Const_Buffer array. The function DMA_memcpy has been modified is as shown below.

```
DMA_memcpy(TARGET_ADD, ((uint32_t *)str), TRANSFER_LEN );
```

The TRANSFER_LEN has to be at least more than 9 to able to completely transfer the whole string. The WRITE_SIZE has changed to uint8_t to able the string be write into the flash correctly. The result of the transferred string is as shown below.

i	0	i	1
▲ readAdd1	0x08104000	▲ readAdd1	0x08104001
*readAdd1	'J'	*readAdd1	'A'
i	2	i	3
▲ readAdd1	0x08104002	▲ readAdd1	0x08104003
*readAdd1	'N'	*readAdd1	'G'
i	4	i	5
▲ readAdd1	0x08104004	▲ readAdd1	0x08104005
*readAdd1	''	*readAdd1	'S'
i	6	i	7
▲ readAdd1	0x08104006	▲ readAdd1	0x08104007
*readAdd1	'I'	*readAdd1	'N'
i	8	i	9
▲ readAdd1	0x08104008	▲ readAdd1	0x08104009
*readAdd1	'G'	*readAdd1	0

Figure 35 result obtained for the string transfer

From the Figure shown above, the data of the readAdd1 for every address increment of 0x1 as compared to Figure 34, that is because the WRITE_SIZE has been set to uint8_t. The 10th element shown in the Figure 35 shows the end of the string which is 'NULL', so the value 0 has been observed. Therefore, a string memory has been successfully transferred by the DMA memory-to-memory transfer to the Flash memory.

An error has been forced in to the DMA operation by commenting out the while loop that check for BSY bit of FLASH_SR.

```

58
59
60
61
62
63
64
65
66
67
68
69
70
    if(guard) {
        configDMA2s7CR(MemToMem,x32,x32,VeryHighPrio,CHANNEL0);
        DMA_memcpy(TARGET_ADD, SRC_Const_Buffer, TRANSFER_LEN );
        flashProgramConfig(x32);
        flashProgramEn();
        enabledDMA();
        //while(checkBusy()) {}
        flashError=checkFlashError();
        flashProgramDisable();
        clearDMAHighIntrrFlag();
        flashError=checkFlashError();
        guard=0;
    }

```

Figure 36 commented while loop check busy statement

The error will only been shown when the transferring length is more than 11. The error when the transferring length is more than 11 is as shown below.

```

7 int checkFlashError() {
8     int seqErr, parallelErr, alignErr, writeProtErr, OperationErr;
9
10    checkFlashReg();
11    seqErr      = (FLASH->SR>>7) & 1;
12    parallelErr = (FLASH->SR>>6) & 1;
13    alignErr    = (FLASH->SR>>5) & 1;
14    writeProtErr = (FLASH->SR>>4) & 1;
15    OperationErr = (FLASH->SR>>1) & 1;
16
17    if(seqErr!=0 || parallelErr!=0 || alignErr!=0 || writeProtErr!=0 || OperationErr!=0)
18        return 1;
19    else
20        return 0;
21}

```

Figure 37 error forced by commenting the while loop checking statement

Name	Value
seqErr	1
parallelErr	0
alignErr	0
writeProtErr	0
OperationErr	0

From the result shown above, it shows there are a sequence error occurring in the flash. This was caused by the transferring of the data. When the while loop has been commented out, the flash might be still busy performing some operation when it disables the PG bit of the FLASH_CR. Thus, this causes a violation to the flash and provides a sequence error to the FLASH_SR register.

3.6 Overall discussion

All the operations performed discussed were operating inside the while loop. So at the end of the while loop, the FLASH_CR has to be locked to avoid errors to occur. The flashLock function is as shown below.

```

38 void flashLock() {
39     FLASH->CR |= FLASH_CR_LOCK;
40 }

```

Figure 38 function flashLock

CHAPTER IV: CONCLUSION

The CoIDE project of the flash can be access through the GitHub link <https://github.com/JangSing/FLASH>. All the configuration has been correctly configured so that the FLASH are able to perform the **Read, Write** and **Erase** operation. All the result obtained has been shown in the Chapter III, the result obtained reached the expectation and the program are able to sense whether there are occurrence of either **Programming sequence error** , **Parallelism error** and **Programming alignment error**. Therefore, the Flash Memory Interface has been successfully configured to perform **Read, Write** and **Erase** operation.