ECE 153B – Winter 2024
Cyclic Redundancy Check

Lab 6 – CRC Implementation by Software, Hardware and DMA

Deadline: **March 8, 2024, 10:00 PM**

Objectives

Understanding the basic concept of using Cyclic Redundancy Check (CRC) to detect data errors

1. Using software to compute CRC
2. Programming the CRC module to calculate the CRC.
3. Programming DMA controller and CRC module to calculate the CRC.

Grading

Part	**Weight**
Part A	30 %
Part B	30 %
Part C	30 %
Checkoff Questions	10 %

Submit your code on GradeScope by the specified deadline. In the week following the submission on GradeScope, you will demo/checkoff your lab to the TA.

Please note that we take the Honor Code very seriously, do not copy the code from others.

Necessary Supplies

- STM32L4 Nucleo Board
- Type A Male to Mini B USB Cable

Contents

1 Lab Overview

A. Calculate CRC using software and measure the time to calculate

B. Calculate CRC using hardware and measure the time to calculate

C. Calculate CRC with DMA and measure the time to calculate

2 Introduction to CRC

The Cyclic Redundancy Check (CRC) is an efficient error detection technique widely used in networking and storage. CRC only detects errors but does not correct errors. CRC has been widely used to ensure data transmission or storage integrity.

2.1 Polynomial Division in CRC

CRC is based on polynomial arithmetic. A polynomial is an expression in the following form, in which a_n, a_{n-1}, ..., and a_0 are coefficients.

$$a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} ... + a_1 x + a_0$$

CRC treats binary data as a polynomial over GF(2), i.e., each coefficient is either zero or one. For example, 11, which is `1011` in binary, has the polynomial form of

$$G_{11}(x) = 1x^3 + 0x^2 + 1x^1 + 1x^0 = x^3 + x + 1$$

$$G_{11}(2) = 2^3 + 2 + 1 = 11$$

CRC performs division between two polynomials and computes the remainder of the division. The dividend is the given data bitstream to be checked, and the divisor is the pre-defined generator polynomial. The remainder is called CRC, which is used for error detection.

For example, a division of 11 over 6, in integer field $\mathbb{Z}$ is:

$$\begin{array}{rrrrr}
 & x - & 1 & & \\ \cline{2-5}
x^2 + x \big) & x^3 + & 0 + & x + 1 \\
 & x^3 + & x^2 + & 0 \\ \cline{2-5}
 & & -x^2 + & x + 1 \\
 & & -x^2 - & x + 0 \\ \cline{3-5}
 & & & 2x + 1
\end{array}$$

In practice, we perform division calculation in boolean field $\mathbb{B}$, where $1 + 1 = 0$, $0 - 1 = 1$. This is effectively taking only the last bit in the 2's complement representation of the number. In this form, the long division above becomes:

$$\begin{array}{rrrr}
 & x + & 1 & \\ \cline{2-4}
x^2 + x \big) & x^3 + & 0 + x + 1 \\
 & x^3 + x^2 + 0 & \\ \cline{2-4}
 & & x^2 + x + 1 \\
 & & x^2 + x + 0 \\ \cline{3-4}
 & & 1
\end{array}$$

We can see that in boolean field, both addition and substraction can be replaced by XOR operation. This further simplifies our calculation.

For dividend of x bits and divisor of y bits, we can see that we performs up to $x - y + 1$ operations, and the remainder is at most $y - 1$ bits.

2.2 CRC Standards

A few fixed bits are often added at the beginning of the data bitstream because 0-bits may be inserted into the beginning due to clocking errors. If no non-zero bits are inserted at the beginning, such stuck-to-zero errors might go undetected.

The most popular CRC is the standard **CRC-32**, which has used many applications such as IEEE 802.3 (Ethernet), MPEG-2, gzip, POSIX CKSUM, PNG, and Serial ATA. It uses the following polynomial generator as the divisor to compute a 32-bit checksum. Its corresponding value is `0x04C11DB7`. For Ethernet, the standard CRC-32 can detect up to three independent bit errors when the maximum transmission unit is set to 1500 bytes. CRC-32 is the default CRC calculation on STM32L4.

$$G(x) = x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	0	0	1	1	1	0	1	1	0	1	1	0	1	1	1

We can see that `0000_0100_1100_0001_0001_1101_1011_0001` is exactly `0x04C11DB7`.

2.3 STM32 Implementation

In STM32, CRC is implemented using 33 bit divisor instead, where for any given 32-bit divisor, STM32 CRC algorithm prepends a bit of 1 before MSB. Thus, the default divisor is actually `0x104C11DB7`.

Then, for input size of n, STM32 appends n bit of 0 after the input data. So the input data is now 64 bits. The operation to perform is 64-33+1=32 shift operations, and the remainder has 33-1=32 bits.

Before operation, the algorithm takes the previous CRC value and XOR'ed it with the input data, to inherit the CRC and allows CRC checks at arbitrary point of the data stream. Since CRC values have 32 bits, it's unlikely that it will have coincidental match.

There will be a non-zero initial CRC value, to prevent stuck-at-zero error. If both the initial CRC and the data input is zero, the resulting CRC will also be zero. This is undesirable. The default implementation use `0xFFFFFFFF` as the initial CRC value.

3 Introduction to DMA

3.1 DMA Setup

DMA allows communication between memory and peripherals directly, without involvement of CPU. It allows configuration of peripheral data address and memory data address, then perform transaction between them.

For operations between peripherals and memory, peripherals first sends a request signal. DMA will send an acknowledgement signal once the request is fulfilled. Then the peripheral will release the request.

Here are a few things to note during DMA setup:

1. **Access**: All DMA configurations should be programmed only when the DMA channel is disabled and DMA clock is enabled.
2. **Addresses, Data Size, and Transfer Direction**: Peripheral data address and memory data address should be programmed into the data address registers. Then we need to specify their corresponding data size (8-bit, 16-bit, or 32-bit) and transfer direction (peripheral-to-memory or memory-to-peripheral).
   - In memory-to-memory mode, we still need to configure things above, where one memory address is treated as a peripheral that does not require handshake, and will be accessed directly.
   - When data width of source and destination are different, the alignment and endianness are defined in Table 36 in the manual (see Appendix A).
3. **Address Increment**: Both peripheral data address and memory data address can be optionally configured to increment after each transaction.
   - Usually the peripheral data address is not incremented, but the memory data address should.
   - The increment amount depends on the configured data size.
   - If an address register is configured to increment, the peripheral / memory data address register still keeps the initial value. The current address is not accessible through software.
4. **Number of Data**: There is a number of data register to keep track of the remaining count of transfer to perform. It will decrement after each operation.
5. **Circular Mode**: It is possible to configure the DMA to be in circular mode, where the data address registers and number of data register will reset to their initially configured value. Note that this mode is not available in memory-to-memory transfer mode.
6. **Priority**: Since the CPU, DMA, and peripherals share the same bus matrix, DMA transactions can block the CPU and other DMA channels. Thus, we need to have priorities in DMA. There are 4 configurable levels of DMA, represented by 2 priority bits in the CCR register.

3.2 DMA Interrupts

After configuring DMA setup, we need to configure DMA interrupts. There are 4 interrupts for each DMA channel:

1. The global interrupt. Its flag will raise when any of the following interrupts fire.
2. The Transfer Complete (TC) interrupt. It fires when the transfer is completed.
3. The Half Transfer (HT) interrupt. It fires when the transfer is halfway done.
4. The Transfer Error (TE) interrupt. It fires when the DMA attempts to read or write without permission. When it fires, the DMA channel will be disabled at the same time.

We can program the DMA channel to enable or disable any of the TC, HT, or TE interrupts. The global interrupt cannot be disabled. Thus, remember to clear the global interrupt flag as well when you are handling other DMA interrupts.

3.3 DMA Requests from Peripherals

For DMA to communicate with peripherals, the peripherals need to be able to send and clear DMA requests. Each peripheral that supports DMA requests have mappings to map each request to a specific channel and request number.

Each DMA channel supports 8 different request numbers, but only one request number can be selected at a time for each channel. Note that there could be multiple different peripheral requests that are mapped to the same request number of the same DMA channel. Be aware of this when programming DMA.

The exact mapping for peripheral DMA requests can be referenced in Table 38 and 39 in the manual, and in Appendix B.

4 Part A - Software CRC Calculation

In this part, you are going to implement CRC in software and measure the time to calculate. You are provided with the data stream in `data.c`, which can be used by including `CRC.h`.

4.1 Workspace Setup

Before implementing CRC, we need to setup system clock, timers for timing the CRC operation, LED, and UART to communicate with termite. Copy your code from previous labs to fill the blanks in `main/LED.c` and `setup/UART.c`.

You can notice that there are 2 source groups. Source groups will be compiled together. Since you will have more files in this lab than previous labs, having source groups helps to keep your workspace organized. We keep the setup files, clock, timer, and UART code in the `setup` source group, and LED, main, and CRC code in `main` source group.

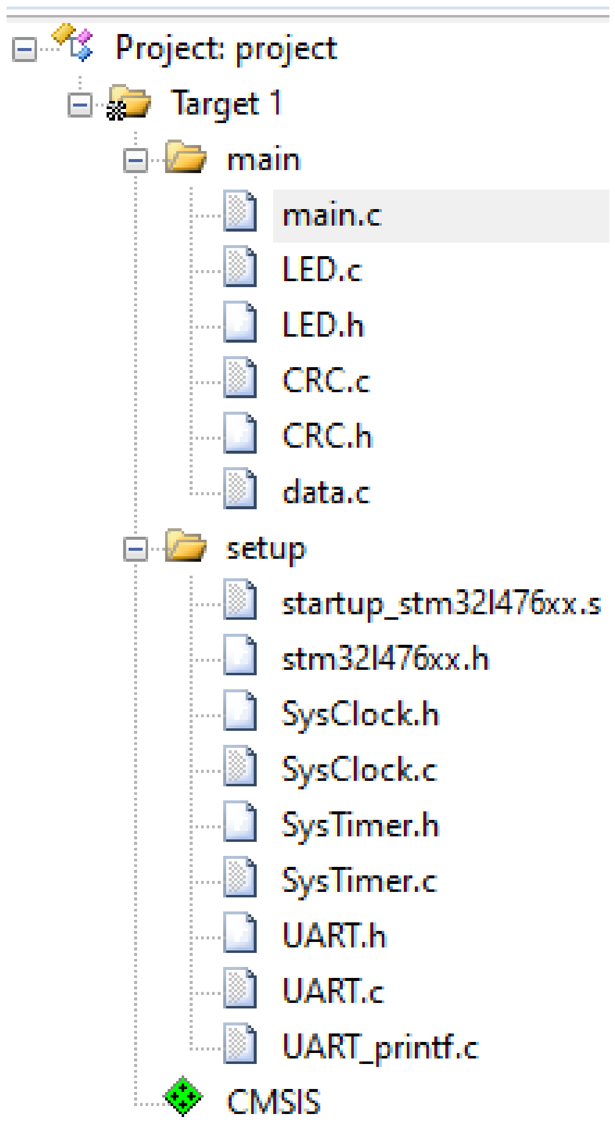

Figure 1: Source groups in lab template

4.2 Use SysTick as Timer

Use a system clock of 80MHz and lab 4A's implementation of UART2. Timer should have a `delay()` function as usual, and 2 more functions: `startTimer()` and `endTimer()`.

Now we want the SysTick to handle 2 separate tasks: create delays and measure time intervals. Fortunately, we won't need them at the same time, so we don't need to worry about potential conflicts between them.

The implementation will keep SysTick disabled, and only enable SysTick when it's performing a task. Before enabling the SysTick, initialize its `LOAD` to a desirable value and `VAL` to 0. You probably want different values in them for ease of calculation.

1. Select your clock source in `SysTick_Init()`. It determines your clock frequency.
2. Implement `SysTick_Handler()`. It should be the same as in previous labs.
3. Implement `delay()`. Now, instead of just waiting for the counters, you need to do the following:
   a) Reset the counter
   b) Reset `VAL` to 0 for SysTick
   c) Set `LOAD` to a desirable value for SysTick
   d) enable SysTick
   e) busy waiting for counter to reach desired value
   f) disable SysTick for future use.
4. Implement `startTimer()` and `endTimer()`. It works similar to delay, but without the busy waiting.
   a) In `startTimer()`:
      i. Reset the counter
      ii. Reset `VAL` to 0 for SysTick
      iii. Set `LOAD` to a desirable value for SysTick
      iv. Enable SysTick to start timing
   b) In `endTimer()`:
      i. Disable SysTick first to pause the timer
      ii. Read the values from VAL and the counter
      iii. Calculate the time using both numbers

Note that your load value will depend on your choice of clock source.

For `delay()`, you will want to select a load value corresponding to 1 ms interrupt frequency.

For `startTimer()`, you will want to select some values that allows you to more precisely measure bitwise operations instead of multiplication when calculating the time using both `VAL` and counter.

4.3 Software CRC Implementation

Now you can start implementing CRC in software, in `CRC.c`. Follow the flowchart in Figure 2 should be enough. Note that the initial CRC, the input data, and the 32-bit polynomial are all provided to you.

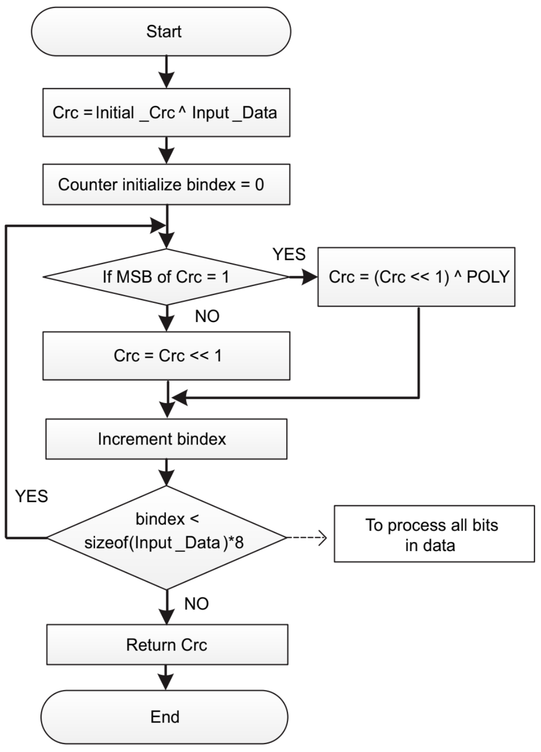
Start
Crc = Initial _Crc ^ Input _Data
Counter initialize bindex = 0
If MSB of Crc = 1
YES
Crc = (Crc << 1) ^ POLY
NO
Crc = Crc << 1
Increment bindex
YES
bindex < sizeof(Input _Data)*8
To process all bits in data
NO
Return Crc
End

Figure 2: STM32 flow chart for CRC calculation

Then, test it using your main function. There are still a few things to complete in `main.c`:

1. Initialize SysClock, SysTick, LED, and UART2. Then in the while loop:
2. Toggle the LED to indicate that it's running.
3. Start the timer.
4. Initialize a variable to hold the CRC with the default initial CRC value, from `CRC.h`.
5. Iterate through `DataBuffer` (with size of `BUFFER_SIZE`) and feed values into CRC calculation function you just implemented. Use your CRC variable as the initial CRC, and use the provided polynomial defined in `CRC.h`.
6. Stop the timer and read the time span of the calculation.
7. Compare the resulting CRC with the expected value. If it's different, turn off the LED to show status and exit the while loop.
8. Print the time span in microseconds to Termite.
9. Wait 1 second before continue to prevent log-spam in Termite.

Then run the code, making sure that your CRC calculation is correct and that SysTick gives a correct 1 second delay.

5 Part B - Hardware CRC Calculation

5.1 Introduction to hardware CRC block

In this part, you need to use the hardware CRC block to perform the CRC calculation. This is done by writing the data into CRC data register. The CRC block takes 1 AHB clock cycle for every byte of input data: It takes only 4 clock cycles to process a 32-bit input data. This means we can continuously write to the data register without waiting on any wait states.

Also, the CRC block's data register is very special: when you write to the data register, you add more data inputs to calculate the CRC value, but when you read from the data register, the data you get is actually the calculated CRC. Thus, to get the CRC value, simply read from the data register.

5.2 Lab Exercise

To initialize the CRC block, we need to do the following in `CRC_Init`:

1. Enable the clock for CRC block
2. Configure the control register to use 32 bit words
3. Load the polynomial into the CRC polynomial register
4. Load the initial value into the initial CRC value register

To compute the CRC using the CRC block, we need to do the following in `CRC_CalcBlockCRC`:

1. Write each word into the CRC data register
2. Read the CRC value from the data register

After that, in the while loop in `main.c`, we need to:

1. Toggle the LED to show that the CRC calculation is going
2. Start timer
3. Reset the CRC in the control register
4. Compute CRC
5. Stop timer and store the time to calculate
6. Check if calculated CRC match the expected value. If not, turn off the LED and exit.
7. Print the time to calculate in microseconds
8. Delay 1 second

Then, measure the time to calculate the CRC, and compare it to the time measured in part A. How many time faster is it?

6 Part C - DMA CRC Calculation

In this part, you will still use the hardware CRC calculation block, but instead of writing every word into the CRC data register, we will configure direct memory access (DMA) instead. It will free up the CPU for other tasks while the CRC block is calculating.

First, we will need to initialize DMA in `DMA_Init`:

1. Enable the clock for DMA
2. Wait 20us for DMA to finish setting up

Then in DMA channel 6:

3. Disable the channel
4. Set Memory-to-memory mode
5. Set channel priority to high
6. Set peripheral size to 32-bit
7. Set memory size to 32-bit
8. Disable peripheral increment mode
9. Enable memory increment mode
10. Disable circular mode
11. Set data transfer direction to Memory-to-Peripheral

Then we need to set the data source and destination. Remember that they are pointers to memory addresses, but you need to cast them to `uint32_t` in order to store them in DMA.

12. Set the data source to data buffer provided in `CRC.h`
13. Set the data destination to the data register of the CRC block

Then we need to setup the DMA interrupts:

14. Disable half transfer interrupt
15. Disable transfer error interrupt
16. Enable transfer complete interrupt
17. Set interrupt priority to 0 in NVIC
18. Enable interrupt in NVIC

Secondly, we need to complete the DMA interrupt in `DMA1_Channel6_IRQHandler`:

1. Clear NVIC interrupt flag
2. Check Transfer Complete interrupt flag. If it occurs, clear the flag and mark computation as completed by calling `computationComplete`.
3. Clear global DMA interrupt flag.

Lastly, in `main.c`, we need to complete the following:

In `computationComplete`:

1. Store the calculated CRC value in a volatile static variable
2. Mark the computation as completed

In `main`:

1. Toggle the LED to show that the CRC calculation is running
2. Start timer
3. Mark the computation as not complete
4. Reset the CRC in the control register
5. Set number of data to transfer in `CNDTR` in DMA
6. Enable DMA channel to start data transfer and CRC computation
7. Wait till the computation is completed
8. Disable DMA channel
9. Stop timer and store the time to calculate
10. Check if calculated CRC matches the expected value. if not, turn off the LED and exit.
11. Print the time to calculate in microseconds
12. Delay 1 second

Then, run the program and compare its run time with the results you get in part B. How many times faster is it?

7 Checkoff Requirements

Demonstrate and report the computation time in 3 parts. The number should be reasonably close to the standard. Be aware that you should use the 80MHz clock provided in the template.

8 References

[1] STM32L4x6 Advanced ARM-based 32-bit MCUs Reference Manual

[2] "What is the difference between full-stepping, the half-stepping, and the micro-drive?," Automate. [Online]. Available: https://www.automate.org/case-studies/what-is-the-difference-between-full-stepping-the-half-stepping-and-the-micro-drive.

[3] Yifeng Zhu, "Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C", ISBN: 0982692633

9 Appendix A - Alignment and Endianness

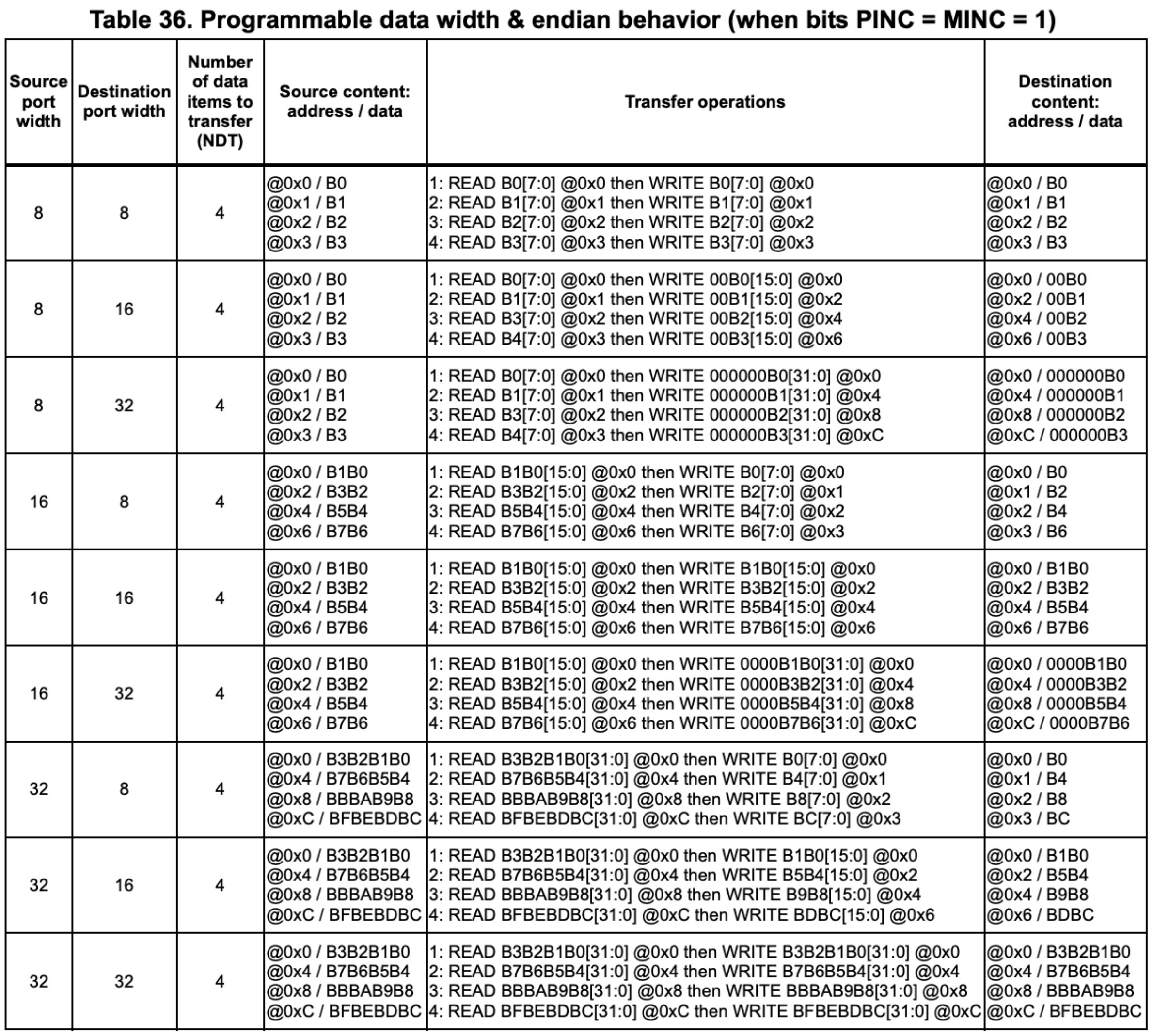

Table 36. Programmable data width & endian behavior (when bits PINC = MINC = 1)

Source port width	Destination port width	Number of data items to transfer (NDT)	Source content: address / data	Transfer operations	Destination content: address / data
8	8	4	@0x0 / B0 @0x1 / B1 @0x2 / B2 @0x3 / B3	1: READ B0[7:0] @0x0 then WRITE B0[7:0] @0x0 2: READ B1[7:0] @0x1 then WRITE B1[7:0] @0x1 3: READ B2[7:0] @0x2 then WRITE B2[7:0] @0x2 4: READ B3[7:0] @0x3 then WRITE B3[7:0] @0x3	@0x0 / B0 @0x1 / B1 @0x2 / B2 @0x3 / B3
8	16	4	@0x0 / B0 @0x1 / B1 @0x2 / B2 @0x3 / B3	1: READ B0[7:0] @0x0 then WRITE 00B0[15:0] @0x0 2: READ B1[7:0] @0x1 then WRITE 00B1[15:0] @0x2 3: READ B3[7:0] @0x2 then WRITE 00B2[15:0] @0x4 4: READ B4[7:0] @0x3 then WRITE 00B3[15:0] @0x6	@0x0 / 00B0 @0x2 / 00B1 @0x4 / 00B2 @0x6 / 00B3
8	32	4	@0x0 / B0 @0x1 / B1 @0x2 / B2 @0x3 / B3	1: READ B0[7:0] @0x0 then WRITE 000000B0[31:0] @0x0 2: READ B1[7:0] @0x1 then WRITE 000000B1[31:0] @0x4 3: READ B3[7:0] @0x2 then WRITE 000000B2[31:0] @0x8 4: READ B4[7:0] @0x3 then WRITE 000000B3[31:0] @0xC	@0x0 / 000000B0 @0x4 / 000000B1 @0x8 / 000000B2 @0xC / 000000B3
16	8	4	@0x0 / B1B0 @0x2 / B3B2 @0x4 / B5B4 @0x6 / B7B6	1: READ B1B0[15:0] @0x0 then WRITE B0[7:0] @0x0 2: READ B3B2[15:0] @0x2 then WRITE B2[7:0] @0x1 3: READ B5B4[15:0] @0x4 then WRITE B4[7:0] @0x2 4: READ B7B6[15:0] @0x6 then WRITE B6[7:0] @0x3	@0x0 / B0 @0x1 / B2 @0x2 / B4 @0x3 / B6
16	16	4	@0x0 / B1B0 @0x2 / B3B2 @0x4 / B5B4 @0x6 / B7B6	1: READ B1B0[15:0] @0x0 then WRITE B1B0[15:0] @0x0 2: READ B3B2[15:0] @0x2 then WRITE B3B2[15:0] @0x2 3: READ B5B4[15:0] @0x4 then WRITE B5B4[15:0] @0x4 4: READ B7B6[15:0] @0x6 then WRITE B7B6[15:0] @0x6	@0x0 / B1B0 @0x2 / B3B2 @0x4 / B5B4 @0x6 / B7B6
16	32	4	@0x0 / B1B0 @0x2 / B3B2 @0x4 / B5B4 @0x6 / B7B6	1: READ B1B0[15:0] @0x0 then WRITE 0000B1B0[31:0] @0x0 2: READ B3B2[15:0] @0x2 then WRITE 0000B3B2[31:0] @0x4 3: READ B5B4[15:0] @0x4 then WRITE 0000B5B4[31:0] @0x8 4: READ B7B6[15:0] @0x6 then WRITE 0000B7B6[31:0] @0xC	@0x0 / 0000B1B0 @0x4 / 0000B3B2 @0x8 / 0000B5B4 @0xC / 0000B7B6
32	8	4	@0x0 / B3B2B1B0 @0x4 / B7B6B5B4 @0x8 / BBBAB9B8 @0xC / BFBEBDBC	1: READ B3B2B1B0[31:0] @0x0 then WRITE B0[7:0] @0x0 2: READ B7B6B5B4[31:0] @0x4 then WRITE B4[7:0] @0x1 3: READ BBBAB9B8[31:0] @0x8 then WRITE B8[7:0] @0x2 4: READ BFBEBDBC[31:0] @0xC then WRITE BC[7:0] @0x3	@0x0 / B0 @0x1 / B4 @0x2 / B8 @0x3 / BC
32	16	4	@0x0 / B3B2B1B0 @0x4 / B7B6B5B4 @0x8 / BBBAB9B8 @0xC / BFBEBDBC	1: READ B3B2B1B0[31:0] @0x0 then WRITE B1B0[15:0] @0x0 2: READ B7B6B5B4[31:0] @0x4 then WRITE B5B4[15:0] @0x2 3: READ BBBAB9B8[31:0] @0x8 then WRITE B9B8[15:0] @0x4 4: READ BFBEBDBC[31:0] @0xC then WRITE BDBC[15:0] @0x6	@0x0 / B1B0 @0x2 / B5B4 @0x4 / B9B8 @0x6 / BDBC
32	32	4	@0x0 / B3B2B1B0 @0x4 / B7B6B5B4 @0x8 / BBBAB9B8 @0xC / BFBEBDBC	1: READ B3B2B1B0[31:0] @0x0 then WRITE B3B2B1B0[31:0] @0x0 2: READ B7B6B5B4[31:0] @0x4 then WRITE B7B6B5B4[31:0] @0x4 3: READ BBBAB9B8[31:0] @0x8 then WRITE BBBAB9B8[31:0] @0x8 4: READ BFBEBDBC[31:0] @0xC then WRITE BFBEBDBC[31:0] @0xC	@0x0 / B3B2B1B0 @0x4 / B7B6B5B4 @0x8 / BBBAB9B8 @0xC / BFBEBDBC

10 Appendix B - Peripheral DMA Request Mappings

Table 38. Summary of the DMA1 requests for each channel

Request. number	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7
0	ADC1	ADC2	ADC3	DFSDM0	DFSDM1	DFSDM2	DFSDM3
1	-	SPI1_RX	SPI1_TX	SPI2_RX	SPI2_TX	SAI2_A	SAI2_B
2	-	USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
3	-	I2C3_TX	I2C3_RX	I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
4	TIM2_CH3	TIM2_UP	TIM16_CH1 TIM16_UP	-	TIM2_CH1	TIM16_CH1 TIM16_UP	TIM2_CH2 TIM2_CH4
5	TIM17_CH1 TIM17_UP	TIM3_CH3	TIM3_CH4 TIM3_UP	TIM7_UP. DAC2	QUADSPI	TIM3_CH1 TIM3_TRIG	TIM17_CH1 TIM17_UP
6	TIM4_CH1	-	TIM6_UP DAC1	TIM4_CH2	TIM4_CH3	-	TIM4_UP
7	-	TIM1_CH1	TIM1_CH2	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM15_CH1 TIM15_UP TIM15_TRIG TIM15_COM	TIM1_UP	TIM1_CH3

Table 39. Summary of the DMA2 requests for each channel

Request. number	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7
0	-	-	ADC1	ADC2	ADC3	-	-
1	SAI1_A	SAI1_B	SAI2_A	SAI2_B	-	SAI1_A	SAI1_B
2	UART5_TX	UART5_RX	UART4_TX	-	UART4_RX	USART1_TX	USART1_RX
3	SPI3_RX	SPI3_TX	-	TIM6_UP DAC1	TIM7_UP DAC2	-	QUADSPI
4	SWPMI_RX	SWPMI_TX	SPI1_RX	SPI1_TX	-	LPUART_TX	LPUART_RX
5	TIM5_CH4 TIM5_TRIG TIM5_COM	TIM5_CH3 TIM5_UP	-	TIM5_CH2	TIM5_CH1	I2C1_RX	I2C1_TX
6	AES_IN	AES_OUT	AES_OUT	-	AES_IN	-	-
7	TIM8_CH3 TIM8_UP	TIM8_CH4 TIM8_TRIG TIM8_COM	-	SDMMC1	SDMMC1	TIM8_CH1	TIM8_CH2