San José State University
Department of Computer Engineering
CMPE 146-03, Real-Time Embedded System Co-Design, Fall 2020

Lab Assignment 3

Due date: 9/27/2020, Sunday

1. Description

In this assignment, we will be familiarized with the DMA and interrupt mechanisms on the MSP432 MCU.

2. Exercise 1

In this exercise, we will use DMA to transfer a block of data to the CRC-32 accelerator to compute the CRC checksum. We will import the example project, Software/SimpleLink MSP432P4 SDK…/Examples/Development Tools/MSP432P401R LaunchPad - Red 2.x (Red)/DriverLib/dma_crc32_transfer_calculation/No RTOS/CCS Compiler/dma_crc32_transfer_calculation, and modify it for this exercise. Only the main C file, *dma_crc32_transfer_calculation.c*, should be modified.

The example project uses the DMA controller to transfer all the data to the CRC-32 accelerator. The processor sets up the DMA controller and uses software trigger to initiate the transfer. When the transfer is completed, an interrupt is generated to notify the processor and the processor can read the checksum from the result register in the CRC-32 accelerator.

Before we can use the DMA controller, we will need to set it up first with the following two DriverLib functions calls:

```
MAP_DMA_enableModule();
MAP_DMA_setControlBase(controlTable);
```

MAP_DMA_enableModule() enables the controller to be used. The controller needs some memory space to use as control data structure for its operations; *MAP_DMA_setControlBase()* defines where such data structure is. Note that the data structure must be aligned on a 1024-byte boundary.

The controller supports several channels and different operation modes for data transfer. To set up for the type of data transfer desired, use the following three functions:

```
MAP_DMA_setChannelControl(UDMA_PRI_SELECT, UDMA_SIZE_8 | UDMA_SRC_INC_8 |
UDMA_DST_INC_NONE | UDMA_ARB_1024);
MAP_DMA_setChannelTransfer(UDMA_PRI_SELECT, UDMA_MODE_AUTO, data_array, (void*)
(&CRC32->DI32), 1024);
MAP_DMA_assignInterrupt(DMA_INT1, 0);
```

MAP_DMA_setChannelControl() selects the control data structure, sets up the data size (8, 16 or 32 bits), determines how the source and destination address increment, and the arbitration size. *MAP_DMA_setChannelTransfer()* sets the transfer mode, source address, destination address and size of transfer. *MAP_DMA_assignInterrupt()* assigns a DMA channel to be handled by a specific interrupt handler.

In the example project, the interrupt service routine (ISR) is *DMA_INT1_IRQHandler()*, which is located after *main()*.

When a DMA transfer is done, the DMA controller sends a signal to the interrupt controller. Use the following two functions to set up the interrupt controller.

```
MAP_Interrupt_enableInterrupt(INT_DMA_INT1);
MAP_Interrupt_enableMaster();
```

MAP_Interrupt_enableInterrupt() enables the DMA interrupt. *MAP_Interrupt_enableMaster()* allows the processor to respond to interrupts.

To initiate a DMA transfer with a software signal, use the following functions:

```
MAP_DMA_enableChannel(0);
MAP_DMA_requestSoftwareTransfer(0);
```

By default, all DMA channels are disabled. To enable one, use *MAP_DMA_enableChannel()*. After the transfer is done, it is automatically disabled. So, we will need to enable it each time when a transfer is needed. *MAP_DMA_requestSoftwareTransfer()* sends a signal to the DMA controller to start the transfer.

Add the following line to the *include* file section.

```
#include <stdio.h>
```

Add the following line to define the “seed” for computing a CRC-32 checksum before *main()*.

```
#define CRC32_SEED             0xFFFFFFFF
```

Remove the *dataarray.c* file from the project. Instead of using an external data array, declare *data_array* like the following before *main()*:

```
uint8_t data_array[1024];
```

To ensure we are using the DMA correctly, we are also going to produce the same CRC-32 checksum using the method that copies all the data to the accelerator with a *for* loop. The following snippet can be used for that purpose:

```
MAP_CRC32_setSeed(CRC32_SEED, CRC32_MODE);
int ii;
for (ii = 0; ii < sizeof(data_array); ii++)
    MAP_CRC32_set8BitData(data_array[ii], CRC32_MODE);
uint32_t hwCRC = MAP_CRC32_getResult(CRC32_MODE);
printf("hwCRC=%08x\n", hwCRC);
```

This method does not use DMA or interrupt at all. Place the code after the *MAP_WDT_A_holdTimer()* call in *main()*. Let’s call this method the *hardware* method, and the other method as *dma*.

To produce the same checksum with the *dma* method, we will need to reinitialize the result register in the CRC-32 accelerator before sending the data there. So, add the following line before the *MAP_DMA_requestSoftwareTransfer()* call:

```
MAP_CRC32_setSeed(CRC32_SEED, CRC32_MODE);
```

In the ISR, *DMA_INT1_IRQHandler()*, print the checksum, like what was done in the snippet above. (It is usually a very bad practice to use *printf* in an ISR. We are breaking the rule for now just to make sure we have the correct framework to build on.)

Build and run the program. The two checksums displayed must be identical.

Lab Report Submission

1. List *main()* and the ISR.
2. Show the outputs in the debug console.
3. Place the entire program listing in the appendix.

3. Exercise 2

In this exercise, we are going to use interrupt the proper way. To provide fast response to an interrupt (an event that needs immediate attention), we should only do what is absolutely necessary in an ISR, so as to keep the routine as short and fast as possible. Typically, we defer further processing of data or event to a higher-level task. In our case, the task is *main()*. So, we will print the checksum and other results in *main()*, not in the ISR. The *dma* method should be a more efficient way of transferring data than the *hardware* method. We will do some measurements to compare the speed of the *hardware* and *dma* methods.

Duplicate the project in Exercise 1 (you can use the method mentioned in Lab 2). There are two separate program routines in the project: *main()* and the ISR. *main()* sets things up to initiate the DMA transfer and then waits for it to finish. When the transfer is completed, the ISR is invoked by the processor. The ISR then notifies *main()* that the transfer is completed. We will use a simple flag so that *main()* and the ISR can communicate. We can implement such flag by declaring a variable like this:

```
volatile int dma_done;
```

Before *main()* triggers the DMA transfer, it sets the *dma_done* flag to 0 and enters a *while* loop to wait for its state to change. The ISR sets the *dma_done* flag to 1 to indicate the completion of transfer. Once *main()* gets the notification, it can proceed to do other things.

Remove the *MAP_PCM_gotoLPM0()* call in the *while* loop in *main()* so that the processor won't enter sleep mode. Once it gets the notification, it should exit the *while* loop and print checksum.

Remove the *printf* call in the ISR. A *printf* call (even a short one) takes up a lot of time (could be tens of milliseconds or more). Typically, we only place it in an ISR for debugging purpose. After the ISR reads the checksum from the result register and place it in a global variable, it sets the *dma_done* flag and then exits. The whole ISR should be very short.

Make sure both methods produce the same checksum. Both methods use the same accelerator with the same data block; the only difference is the way they send the data to the accelerator.

With the 32-bit timer (used in the previous lab), measure the time in µs the *hardware* method takes. Then measure the time the *dma* method takes. You cannot assume the system running at certain clock frequency; use *MAP_CS_getMCLK()* to get the current system clock frequency for the measurements. Note that the time measurement should not include any *printf* calls as they are not part of the checksum computation; they are for our information only. All the measurements should be done in *main()*. Compute the speedup of the *dma* method over the *hardware* method.

Lab Report Submission

1. List the relevant code of both computing methods in *main()* and the ISR.
2. Show the outputs in the debug console, which should contain at least the checksums, the measured times (of both methods) and the speedup.
3. Place the entire program listing in the appendix.

4. Exercise 3

In this exercise, we will do measurements with different block sizes, all less than or equal to 1024 bytes. Declare a global size array like the following.

```
int size_array[] = {2, 4, 16, 32, 64, 128, 256, 786, 1024};
```

Duplicate the project in the previous exercise. Add a *for* loop to *main()* so that it can do the aforementioned measurements a number of times. In each loop, display the block size. Use the same data block, *data_array*. Make sure to change the size parameters within the loop accordingly (for example, the parameter to be set in the DMA controller). Keep in mind that certain DriverLib setup function calls, for example, *MAP_DMA_enableModule()*, needs to be called only once for the entire program. So, don't place those functions inside the big *for* loop; otherwise, the measurements won't be accurate. Remember the checksums from both methods must be identical. If they are not, there are bugs in the program.

The ISR should remain the same. All the necessary changes are done in *main()*.

Lab Report Submission

1. List the code of the outermost *for* loop.
2. Show the outputs in the debug console.
3. Describe the speedup results as the block size is increased. Explain your observation. (Hint: The *dma* method is not as useful in some situations as others.)
4. Place the entire program listing in the appendix.

5. Exercise 4

The method implemented so far can only transfer a data block of 1024 bytes or less with DMA. In this exercise, we will do some enhancement so that a much larger block can be transferred. If the block size is larger than 1024, we will set up another DMA transfer in the ISR immediately. So, multiple interrupts will occur before the entire transaction is completed.

In *main()*, if the data block size is larger than 1024, we will set up the DMA controller to transfer the first 1024 bytes. When the ISR is invoked, it will check if the entire block of data has been transferred. If not, it sets up another DMA transfer. Again, if more than 1024 bytes remains, it sets up to transfer only 1024 bytes, and lets the ISR set up to transfer the rest when the next interrupt occurs. The DMA transfers are all independent, so the ways to set up the DMA controller should be very similar in *main()* and in the ISR. To keep the ISR overhead low, use only the DriverLib functions that are absolutely necessary.

Since the ISR needs to know how much data left to be transferred, we can use a global variable to contain the size. *main()* can initialize it when it sets up the first DMA transfer. Then the ISR updates it when it handles the interrupt. *main()* shall not involve in any DMA transfer, except the very first one. As in the previous exercise, it clears the *dma_done* flag before triggering the first DMA transfer and waits for the flag's state to change. The ISR sets the *dma_done* flag only after the entire data block is transferred.

We are going to do the same measurements as in the previous exercise. Modify the global size array to the following.

```
int size_array[] = {512, 1024, 1030, 1824, 2048, 2049, 2303, 10240};
```

Make *data_array* much bigger like the following so that it can accommodate all different sizes:

```
uint8_t data_array[10240];
```

Lab Report Submission

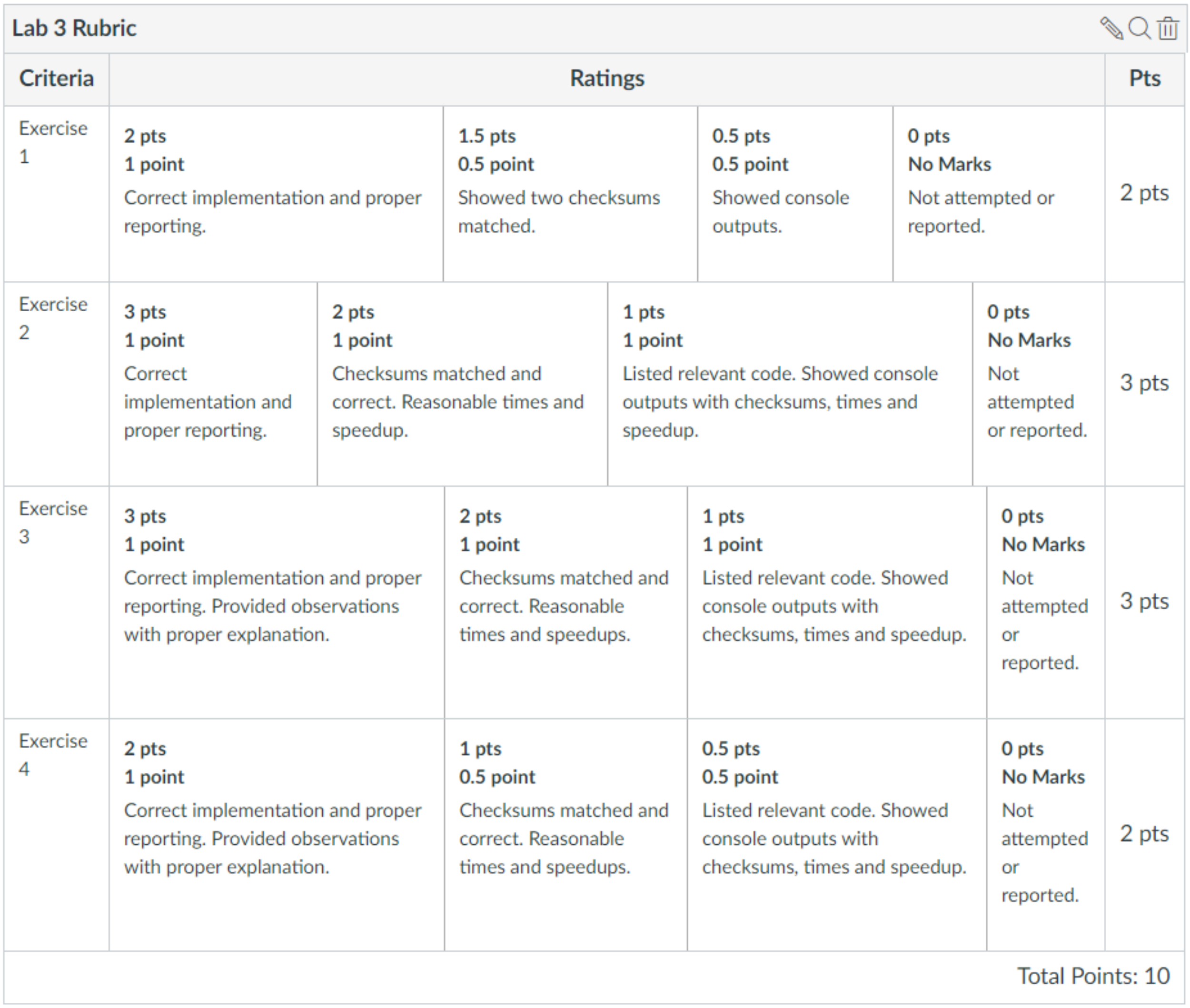
List *main()* and the ISR. Show a screenshot of the outputs in the debug console, which should include at least the block size, measured times (of both methods) and speedup. Describe the speedup results as the block size is increased. How do the results look differently from the previous exercise? Explain your observation.

1. List the code of the outermost *for* loop and the ISR.
2. Show the outputs in the debug console.
3. Describe the speedup results as the block size is increased. Explain your observations.
4. Place the entire program listing in the appendix.

6. Submission Requirements

Write a formal report that contains at least what are required to be submitted in the exercises. Submit the report in PDF to Canvas by the deadline. Include source code in your report. As for the report contents, do not use screenshots to show your codes. In the report body, list the relevant codes or screenshots only for the exercise you are discussing. Put the entire program listing in the appendix. Use single-column format.

7. Grading



Lab 3 Rubric

Criteria	Ratings				Pts
Exercise 1	**2 pts** **1 point** Correct implementation and proper reporting.	**1.5 pts** **0.5 point** Showed two checksums matched.	**0.5 pts** **0.5 point** Showed console outputs.	**0 pts** **No Marks** Not attempted or reported.	2 pts
Exercise 2	**3 pts** **1 point** Correct implementation and proper reporting.	**2 pts** **1 point** Checksums matched and correct. Reasonable times and speedup.	**1 pts** **1 point** Listed relevant code. Showed console outputs with checksums, times and speedup.	**0 pts** **No Marks** Not attempted or reported.	3 pts
Exercise 3	**3 pts** **1 point** Correct implementation and proper reporting. Provided observations with proper explanation.	**2 pts** **1 point** Checksums matched and correct. Reasonable times and speedups.	**1 pts** **1 point** Listed relevant code. Showed console outputs with checksums, times and speedup.	**0 pts** **No Marks** Not attempted or reported.	3 pts
Exercise 4	**2 pts** **1 point** Correct implementation and proper reporting. Provided observations with proper explanation.	**1 pts** **0.5 point** Checksums matched and correct. Reasonable times and speedups.	**0.5 pts** **0.5 point** Listed relevant code. Showed console outputs with checksums, times and speedup.	**0 pts** **No Marks** Not attempted or reported.	2 pts
					Total Points: 10

Grading Policy

The lab's grade is determined by the report. If you miss the submission deadline for the report, you will get zero point.