San José State University
Department of Computer Engineering
CMPE 146-01, Real-Time Embedded System Co-Design, Fall 2024

Lab Assignment 3

Due date: 10/06/2024, Sunday

1. Overview

In this assignment, we will be familiarized with the CRC-32 accelerator on the MSP432 MCU that can be used to ensure data integrity for communication or data storage. We will see how it can help in terms of speed and effectiveness. We will also be familiarized with the interrupt and DMA mechanisms. We will use interrupt to control a device and a special data structure for reliable communication between an ISR and an application task. We will characterize the performance of the DMA mechanism.

2. Exercise 1 CRC-32

In this exercise, we will use different methods to compute the checksum of a block of data. We will use the CRC-32 accelerator on the MSP432 MCU and a software function (not using dedicated hardware) to generate the CRC checksum. We will also use a different algorithm to generate a different type of checksum using a software function. We will compare the methods' speed and effectiveness.

Exercise 1.1 Simple checksum

CRC is just one of many checksum generation algorithms that we can use. Another commonly used algorithm to compute a checksum is simply adding all the data together; the sum of all data is the checksum. We will write a version of this simple method.

Import (and rename) the *empty* skeleton project on CCS. Add the following line to the *include* file section for using the *printf* and other library functions.

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

Write a small function to implement the simple checksum algorithm. Here is the prototype:

```
uint32_t compute_simple_checksum(uint8_t* data, uint32_t length)
```

The function just adds all the data in the input data array together to form a 32-bit word. Since the input data type is byte and the sum is a 32-bit word, simply adding the byte values will produce an imbalance on the weights (or significance) among the four bytes in the final 32-bit checksum word. The least significant byte would reflect more changes than other bytes because of the simple addition method of adding a byte to a 32-bit word. To address the imbalance issue, we will add the input bytes to different byte positions in the 32-bit checksum word. The first byte goes to the least significant byte (Byte 0), the second byte goes to the next higher-order byte (Byte 1), the third byte goes to Byte 2, and so on. After Byte 3, it is wrapped back to Byte 0. The carry-over, if any, from the addition will be added to the next higher-order byte. After all input data are added, reverse all the bits in the sum value, i.e., one becomes zero and zero becomes one.

Declare a byte array in *main.c* for testing the function, like the following.

```
static uint8_t myData[10240];
```

In *main()*, call the function and print the checksum returned in 8-digit hexadecimal format. We will use a specific data pattern for this exercise. Set a breakpoint at the line calling *compute_simple_checksum()* and change the breakpoint's properties so that CCS will read the data from a file, *Lab3_Ex1_data.dat*, which comes with this assignment on Canvas. There is a YouTube video here that shows us how to change the breakpoint properties. Basically, set the "Action" field to "Read Data from File," the "File" field to the full path of the data file, the "Start Address" field to "myData," and the "Length" field to "2560."

The default behavior of a breakpoint is to halt CPU execution, and CCS will wait for user's command to continue. By changing the breakpoint's "Action" field to "Read Data from File" from the default "Remain Halted," the execution will be paused and then continue immediately after CCS finishes reading data from the file and placing them in the MCU's memory. This way, we can do more extensive debugging or testing with different data patterns.

Your program should print out some data values for ensuring you are using the correct data. For example, the first four bytes are 0x0f, 0x62, 0x29, 0x29. The last bytes are 0xdc, 0x55, 0x39, 0xaa. You cannot make any assumptions about the input arguments: data's starting address and the size of the data block.

Lab Report Submission

1. List the function.
2. Show the debug console outputs.
3. Propose a method to speed up the computation of the function. (You do not need to implement it.)
4. List the entire program listing in the appendix.

Exercise 1.2 Speedup

In this exercise, we will use the CRC-32 accelerator to compute the CRC checksum of a block of data. We will import the example project, C:\ti\simplelink_msp432p4_sdk_3_40_01_02\examples\nortos\MSP_EXP432P401R\driverlib\crc32_32-bit_signature_calculation\ccs (on Windows), and modify it for this exercise. We will only need to modify the main C file, *crc32_32-bit_signature_calculation.c*. We also need to add the *compute_simple_checksum()* function from the previous exercise.

The DriverLib function, *MAP_CRC32_setSeed()* sets the seed for generating checksum. Then we send the data bytes one by one using *MAP_CRC32_set8BitData()*. After all data are provided to the unit, we can read computed checksum with *MAP_CRC32_getResultReversed()*.

At this point, we should have three methods to compute a 32-bit checksum of a data block. The first two are given in the example project. One is using the DriverLib functions *MAP_CRC32_xxx()*, which use the hardware accelerator. The second method (a pure software method) uses the given *calculateCRC32()* function, located after *main()*. The third one is using *compute_simple_checksum()* that you just wrote for the previous exercise. The results from the first two must be identical because exactly the same algorithm is implemented. (Don't forget to reverse all the bits after calling *MAP_CRC32_getResultReversed()*, just like what the example code does.)

We are going to see how fast each method computes the checksum. Change the declaration and the array size of *myData* to be much bigger, like 10240 (10K), as follows.

```
static uint8_t myData[10240];
```

We will fill the array with data from the same file (*Lab3_Ex1_data.dat*) we used in the previous exercise. So, set a breakpoint at a proper line before any checksum computation is done in *main()* and change the breakpoint properties like before. The program should also print out some data values for ensuring the correct data are being used.

Use the time measurement method that we have used in Lab 2. Measure how long the three methods take to compute the checksum. Print out the time in µs (microseconds) and the checksum for each method.

For the CRC-32 checksum methods (not the one you wrote), compute the speedup of using the accelerator over the pure software method. Print the result. We define the speedup as the ratio of two speeds, the reference speed being from the pure software method.

All the measurements should be done in *main()*, not in any functions being called. Do not make any changes to *calculateCRC32()* (the function provided in the example project) and *compute_simple_checksum()* (the function that you wrote).

Lab Report Submission

1. List the code that does the measurements.
2. Show the debug console outputs: times in µs, checksums in hex and speedup.
3. Looking at the CRC speedup result, which method is faster? What are the major contributing factors to the speedup? Name and describe two.
4. List the entire program listing in the appendix.

Exercise 1.3 Simulate data corruption

We are going to make some small changes to the data and see how the checksum changes. Use the same method as before to set up the data block from *Lab3_Ex1_data.dat*.

Duplicate the project of the previous exercise. We will just use two methods: *MAP_CRC32_xxx()* and *compute_simple_checksum(),* i.e, the hardware method and the method that you wrote. Assuming no prior knowledge of the data values in the array, reverse Bit 7 of the value in *myData[8000]*. Compute the checksums and print them. Then, reverse Bit 7 of the value in *myData[9016]*. Compute the checksums and print them.

Your program shall produce three sets of results because there are three different data sets: original without any changes, original with one bit “corrupted” and original with two bits “corrupted.” For each set, compute the checksums with the two different methods. Then print the results. Since the checksum algorithms are different, the computed checksums should also be different. Since we are not interested in speed in this exercise, you should remove the code for timing and computing speedup.

Lab Report Submission

1. List the code that does the above actions.
2. Show the debug console outputs, showing at least the methods and the checksums in hex.
3. You have three checksum results from each method: one before any changes are done and two after a datum is modified. Describe the results and explain what you see regarding the changes (if any) on the checksums.
4. Based on what you observed, which checksum method is better and why?
5. List the entire program in the appendix.

3. Exercise 2 Interrupt

To provide fast and predictable response to an external event, we quite often make use of the interrupt mechanism on the MCU. In this exercise, we will use interrupt to control an LED with a push button and to measure the duration when the button is pressed.

The example project, C:\ti\simplelink_msp432p4_sdk_3_40_01_02\examples\nortos\MSP_EXP432P401R\driverlib\gpio_input_interrupt\ccs (on Windows), shows us how to set up the device to generate interrupt from the push button through the GPIO module. We will follow the same way for this exercise.

Here is the interrupt setup for the GPIO module:

```
MAP_GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN1);
MAP_GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN1);
```

Here is the setup for the interrupt controller:

```
MAP_Interrupt_enableInterrupt(INT_PORT1);
MAP_Interrupt_enableMaster();
```

The interrupt service routine (ISR), *PORT1_IRQHandler()*, handles interrupts from the port. We need the following calls to find out which pin triggers the interrupt and to do some housekeeping.

```
MAP_GPIO_getEnabledInterruptStatus(GPIO_PORT_P1);
MAP_GPIO_clearInterruptFlag(GPIO_PORT_P1, status);
```

In the example project, there are two DriverLib calls: *MAP_SysCtl_enableSRAMBankRetention()* and *MAP_PCM_gotoLPM3()*. We will ignore them for this exercise as they are for power saving purpose.

Exercise 2.1 Control

In this exercise, we will control the green LED using interrupt from the push button S1. The interaction between the push button and the green LED is that the green LED is turned off when the push button is pressed; it is turned on when the button is released. We will use interrupt to do the work, not with constant polling.

Import (and rename) the *empty* skeleton project. Modify the main C file with the setup method described above to use the interrupt. We can also make use of the ISR in the example *gpio_input_interrupt* project. We can use the same methods we used to set up the LEDs and push button in the previous lab. Make sure to initialize the devices to their proper known states.

For this exercise, *main()* only sets things up. The forever control loop should be empty (does nothing). All the control will be done in the ISR, *PORT1_IRQHandler()*.

Note that the example project's setup only generates interrupt when the button is pressed, i.e., the logic level at the pin goes from High to Low. Releasing the button does not generate any interrupt. We will need to add additional DriverLib calls to make that happen for this exercise. When the push button is not pressed (the default position), the input state is High. Therefore, we want to receive the first signal (interrupt) when it changes from High to Low. In *main()*, we will need to enable the High-to-Low transition to generate the interrupt. We can use the following DriverLib function call.

```
MAP_GPIO_interruptEdgeSelect(GPIO_PORT_P1, GPIO_PIN1,
GPIO_HIGH_TO_LOW_TRANSITION);
```

In the ISR, once the High-to-Low transition interrupt is received, it turns off the LED immediately. Before the ISR returns, it must direct the GPIO module to detect the Low-to-High transition when the push button is released. Similarly, when the Low-to-High transition interrupt is received, the ISR turns on the LED immediately, and then it redirects the GPIO module to detect the High-to-Low transition when the push button is pressed again.

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. There should not be any loop in the ISR. You can have *printf* in it, but only for debugging purpose. After the program is working to your satisfaction, remove any debugging code.

Take a short video clip of the control operation of the LED. Place the video file where it can be accessed remotely.

Lab Report Submission

1. Compared to the polling method using bit-banding (Exercise 1.3 in Lab 2) to control the LED in the previous lab, does the interrupt method used in this exercise provide a quicker response to the push button actions? Justify your answer.
2. Include a link to the video clip. Do not submit the video file to Canvas. Make sure the video file can be opened by just clicking on the link.
3. Place the entire program listing in the appendix.

Exercise 2.2 Measurement

In this exercise, we will add the time measurement of the duration of the push button being pressed to the previous exercise. We will use the 32-bit timer/counter for the measurement, similar to the way measurement is done in the previous lab.

In the ISR, it will record the counter value (the "timestamp") when the ISR is invoked and the state of the push button. These two pieces of information are stored in a queue. The control loop in *main()* will read the records and compute the duration of time when the push button was pressed.

As mentioned above, the ISR should be short and fast. It cannot do too much work. Typically, we defer further processing of data or events recorded to a higher-level task. In our case, the task is the control loop in *main()*. In *main(),* the forever control loop checks if there is a new record in the queue. When there is, it retrieves the record from the queue, computes and prints the time duration in ms. It repeats this process continually in the loop. A communication protocol needs to be established between *main()* and the ISR so that information can be passed from ISR to *main()* without creating any race conditions.

To implement an effective and reliable communication protocol, we will use a circular queue to hold the records. We can use the following code (to be placed before *main()*) to define the circular queue and the variables to use it.

```
#define QUEUE_SIZE 10
struct record {
    uint32_t timestamp;
    uint8_t state;
};
struct record queue[QUEUE_SIZE];
volatile int read_index;
volatile int write_index;
```

The data structure *struct record* contains *timestamp* to store the 32-bit counter value and *state* to store the state of the push button when the ISR is invoked. The array *queue* can hold *QUEUE_SIZE* records.

The index *write_index* contains the queue position where the last record was stored. When the ISR is invoked, it puts a new record in the **next** position. Then it advances the index.

In the forever control loop in *main()*, the index *read_index* contains the queue position where the last record was retrieved. When the two indexes, *read_index* and *write_index*, are not equal, that means that there is at least one new record that has not been retrieved or processed yet. Initially, before the control loop is entered, both indexes are set to zero, meaning there is no new record in the queue yet. When the control loop detects that the indexes are not equal, it retrieves the next record following the one referenced by *read_index*. Then it advances *read_index*.

Note that since it is circular queue, when *read_index* or *write_index* is advanced beyond the last element of the data array, it is wrapped back to the beginning of the array. Both indexes must always address a valid element in the array.

When a new record is available, the control loop checks if the button is released or not. Note that the checking is done on the record, not by reading the port again. If there is a record indicating the push button has been released, it computes the time duration between its timestamp and the previous one. Then it prints the duration in milliseconds to the debug console. Also print the value of *read_index* on the same line.

Notice that the ISR only stores data in the queue and writes to the index *write_index*. The control loop only reads data from the queue and writes to the index *read_index*. There is no instance when both the control loop and the ISR try to access the same record, thus preserving the integrity of the records in the queue.

In a typical application, the control loop in *main()* has other things to do, besides displaying the time measurement results. Let's simulate other tasks by adding a dummy delay at the beginning of the control loop. We can use the *__delay_cycles()* function for that purpose.

```
__delay_cycles(200 * 3000);     // Delay 200 ms at 3 MHz
```

The function accepts an argument of number of clock cycles to delay. The code above will force the CPU to idle (do nothing) for exactly 200 ms in the control loop. The function uses processor instructions to waste 600,000 (200 x 3000) clock cycles (MCLK), which generally is not a good practice. The argument must be constant; it cannot be a variable.

It is possible that the control loop (the foreground high-level task) in *main()* cannot process new records faster than the ISR produces them. Since the queue has a finite size, the queue may be full at some point and the ISR has no empty slot in the array to insert a new record. The ISR shall check for such condition. If it does happen, the ISR shall print an error message to the debug console and return immediately without changing anything in the queue and *write_index*..

Lab Report Submission

1. Show a sample (at least the first seven lines) of the debug console outputs. Show results from both short and long presses. The outputs should include the measured time in ms and the read index.
2. The array *queue* holds a total of 10 records. Is the queue long enough for this application? Justify your answer.
3. Place the entire program listing in the appendix.

4. Exercise 3 DMA

Using software to transfer a block data to the CRC-32 accelerator can be inefficient in terms of throughput and power consumption. With DMA, the processor is not involved with the actual data transfer, so the entire transfer process can be much faster and save more energy.

Exercise 3.1 Speedup

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, C:\ti\simplelink_msp432p4_sdk_3_40_01_02\examples\nortos\MSP_EXP432P401R\driverlib\dma_crc32_transfer_calculation\ccs (on Windows), 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 the base address of such data structure. 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. There are two control data structures: primary and alternate. Here, we are not using the Ping-Pong cycle type, so we just use the primary one, which is indicated by *UDMA_PRI_SELECT*. The data width of transfer is 8-bit (*UDMA_SIZE_8*). We increase the source address by one byte (8 bit, *UDMA_SRC_INC_8*). We do not increment the destination peripheral data register (*UDMA_DST_INC_NONE*) as there is only one data register we are writing to. We set the arbitration size to 1024 (*UDMA_ARB_1024*), meaning the DMA controller only tries to serve other channels' requests only after 1024 transfers are done.

MAP_DMA_setChannelTransfer() sets the transfer mode (*UDMA_MODE_AUTO*), source address (*data_array*), destination address (*&CRC32->DI32*) and size of transfer (1024).
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 cycle is done, the DMA controller sends a signal to the interrupt controller. We 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 cycle 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 in the previous exercise that copies all the data to the accelerator with software. The 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);
```

For this exercise, modify the main C file to display the checksums in hex using the *hardware* and *dma* methods. There should be two separate functions in your program: *main()* and the ISR. Only display the results in *main()*. For the *dma* method, *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 will not enter sleep mode. Once it gets the notification, it should exit the *while* loop and print the checksum.

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 counter, 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. Show the outputs in the debug console, which should contain at least the checksums in hex, the measured times (of both methods) in µs and the speedup.

2. Place the entire program listing in the appendix.

Exercise 3.2 Different data block sizes

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, most likely, there are bugs in the program.

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

Lab Report Submission

1. Show the outputs in the debug console.
2. With the numerical data collected, show a plot of speedup versus block size.
3. Describe the speedup results as the block size is increased. Provide an explanation on why the observed results are produced. (Hint: The *dma* method is not as useful in some situations as others.)
4. Place the entire program listing in the appendix.

5. 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 program source code in your report. Do not use screenshots to show your codes. In the report body, if requested, 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 for the report.

In your report, show the debug console outputs (when requested) in text format; copy and paste them to your report. In general, do not use screenshot to show any text outputs or data; otherwise, points will be deducted.

When presenting a screenshot in the report, only present the relevant information. You may need to crop them from the initial screenshot. Please use proper scale so that the contents can be read easily.

Your report should have several sections. The first section is the information header that contains at least your name, course name and lab assignment number. Each exercise should have its own section where you discuss the results of the exercise. At the end, if requested, there should be the appendix that contains the complete source code for each exercise. The source code may be used to reproduce your reported results during grading. So, make sure they do produce the results you reported.

6. Grading Policy

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

If your program (listed in the appendix) fails to compile, you will get zero point for the exercise.

If your program's outputs do not match what you reported, you will get zero point for the exercise.

Points may be deducted for incorrect statements, typos, non-readable texts on screenshots, lack of description on the graphs/pictures shown, etc. So, proofread your report before submission.

Pay attention to the required program behaviors described in the assignment. Points will be deducted for not producing the required outcomes.