

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

Lab Assignment 2

Due date: Sunday, 9/13/2020

1. Description

In this assignment, you will be familiarized with the bit-banding feature of the MSP432 MCU for controlling external devices, the built-in 32-bit timer for timing purposes, and the integrated CRC-32 accelerator to speed up specific computation.

2. Exercise 1. Bit-Banding

In this exercise, you will use different methods to blink an LED without using the DriverLib functions. In particular, you will learn how to use the bit-banding feature of the MCU to streamline the control of the device.

On the LaunchPad, LED2 is a lighting device that actually consists of three separate LEDs: red, green and blue. They are closely packed together, so you can generate different “intensity” levels on individual LEDs to produce a wide range of color effects. For this exercise, we only focus on controlling the blue LED.

The three LEDs are controlled by three bits in a port, a memory location in the MCU’s memory space. DriverLib provides functions to set up and drive the control bits. For this exercise, we will also use direct memory read/write operations, to do exactly the same thing.

Let’s look at the example project, [Software/SimpleLink MSP432P4 SDK .../Examples/Development tools/MSP432P401R LaunchPad - Red.../DriverLib><gpio_toggle_output/No RTOS/CCS Compiler/gpio_toggle_output](#), the one you used in Lab 1. From the schematic in the user’s guide of the LaunchPad, you will find LED1 is controlled by Bit 0 in Port 1. To set up the control bit, the following function in *main()* is used.

```
MAP_GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0)
```

The function simply sets up Bit 0 in Port 1 as an output “pin” to drive the LED.

To toggle the control bit, i.e., to blink the LED, DriverLib provides the following function.

```
MAP_GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0)
```

Quite often, it is desirable to set the pin to a known state, high or low. There are DriverLib functions to set the control bit high or low. Consult the “General Purpose Input/Output (GPIO)” chapter in the DriverLib User’s Guide for details on those and other GPIO functions.

Exercise 1.1

In this exercise, we will blink the blue LED in LED2 using DriverLib.

Import (and rename) the *gpio_toggle_output* example project on CCS. Refer to the schematic to find out what the port number and bit numbers are for all three LEDs (red, green and blue). Modify *main()* accordingly to set up three output pins.

It is always a good practice to bring all the LEDs to a known state before changing any of them. Otherwise, the outcomes of the execution of the program may become unpredictable. To do that, turn off all three LEDs by setting the output pins low before the *while* loop is entered. Change the loop count in the *for* loop to 50000 (from 5000) so that the blinking can be easily recognized.

Lab Report Submission

1. List the relevant code to do the setup and control of the LEDs.
2. List the entire program listing in the appendix.

Exercise 1.2

In this exercise, we will use the direct memory read/write methods to blink the blue LED without using DriverLib.

Duplicate (and rename) the project in the previous exercise. CCS does not provide function (yet) to duplicate an existing project in the project pane. You can do so by importing the *empty* example project and replace the main source file with contents from the previous exercise.

We need to find the memory address that corresponds to the port of the LEDs. You can find the address by looking up from the MCU datasheet. Table 6.1 contains the address ranges of all peripherals. The peripheral for controlling the LEDs is referred to as “Port Module.” Table 6-21 contains the offset addresses of all ports. Look at the LaunchPad schematic. Find out what the port number and bit number are referred to the blue LED. Then look up from the table the address for the port that would drive the LED.

In the *while* loop, do not use any DriverLib functions to toggle the LED. Create a memory pointer to achieve that instead. Note that the port (mapped as a memory location) also drives the other two LEDs (red and green). Therefore, in order to change the output state of just the blue LED, you will need to read all states from the port first. Change only the bit for the blue LED. Then write the states back to the port.

Lab Report Submission

1. List the relevant code to toggle the blue LED.
2. Report the port address and bit position that are used to control the blue LED.
3. Explain the operations that you use to toggle the blue LED.
4. List the entire program listing in the appendix.

Exercise 1.3

In this exercise, we will use the direct memory write method using bit-banding to blink the blue LED without using DriverLib.

A peripheral port's bits are mapped as individual memory words in the bit-band alias region of the MCU's memory space. Using the locations in the bit-band alias region allows us to focus on the control bit of interest, without worrying about the surrounding bits in the same port. In the previous exercise, when you write to a port, you have to make sure that other bits in a port that you are changing maintain their states. With bit-banding, you do not need to worry about those bits.

Given the port address and the bit position, you can get the aliased memory address in the bit-band region using the following formula.

$$\text{alias_addr} = (\text{port_addr} - \text{peripheral_region_addr}) * 32 + \text{bit_position} * 4 + \text{alias_region_addr},$$

where
 port_addr = port address of interest in the MCU's memory space,
 peripheral_region_addr = starting address of the peripheral region,
 bit_position = bit position of interest in the port, and
 alias_region_addr = starting address of the alias region.

The mechanism of bit-banding is also described in Section 1.4.5 of the MCU technical reference.

Duplicate (and rename) the project in the previous exercise. Translate the port address and the bit position in the previous exercise to a single alias address. Writing to this address can only change one particular bit in the port. With this, change the way you toggle the blue LED in the *while* loop. The operations of control should be simplified from what was done in the previous exercise.

Lab Report Submission

1. List the relevant code to toggle the blue LED.
2. Report the alias address used to control the blue LED.
3. Explain the operations that you use to toggle the blue LED.
4. List the entire program listing in the appendix.

3. Exercise 2. 32-Bit Timer

In this exercise, you will do some time measurement of how fast the LED is blinking in the previous exercise. You will use a 32-bit counter/timer in the MCU and several DriveLib functions.

Duplicate the project in Exercise 1.3. Add the following line to the *include* file section for the library function *printf*.

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

Add the following lines to the beginning of *main()* in the main source file.

```
MAP_Timer32_initModule(TIMER32_0_BASE, TIMER32_PRESCALER_1, TIMER32_32BIT,
TIMER32_FREE_RUN_MODE);
MAP_Timer32_startTimer(TIMER32_0_BASE, 0);
printf("%u\n", MAP_CS_getMCLK());
```

MAP_Timer32_initModule() sets up the counter. *MAP_Timer32_startTimer()* starts the counter. Note that the counter is a count-down counter. It starts from 0xFFFFFFFF. On each input clock pulse, it decrements the counter value by one. After it reaches 0, it starts from 0xFFFFFFFF again.

`MAP_CS_getMCLK()` returns the frequency of the system clock, which triggers the counter to decrement. By default, the frequency is set to 3 MHz during system startup. So, the `printf` call should show such frequency value.

To read the counter value, you can use the `MAP_Timer32_getValue()` function like the following.

```
uint32_t t0 = MAP_Timer32_getValue(TIMER32_0_BASE);
```

Make sure the *for* loop count in the *while* loop set to 100000 so that you can see the LED is blinking slowly. Use the `MAP_Timer32_getValue()` function to measure how long each iteration of the *while* loop takes. Basically, you read the counter values at the beginning and at the end of the loop. The difference would be the duration in counter count. Use the system clock frequency to compute the duration to real time in milliseconds. You can use `printf` to display the time at the very end of the *while* loop. So, the program will repeatedly display the times on the debug console.

Based on the times shown, compute the blinking frequency manually. Note that blinking consists of two operations: turning the LED on and off. Each loop iteration only does ON or OFF, not both. Therefore, the frequency is computed from two time durations.

Lab Report Submission

1. List the relevant code that does the measurement.
2. Show a sample of the debug console outputs.
3. Report the frequency of blinking in Hz.
4. List the entire program listing in the appendix.

4. Exercise 3. CRC-32

In this exercise, you will use the CRC-32 checksum generator to compute the CRC checksum of a block of data. On the CCS Resource Explorer Offline, you will import the example project, [Software/SimpleLink MSP432P4 SDK.../Examples/Development Tools/MSP432P401R LaunchPad - Red 2.x \(Red\)/DriverLib/crc32_32-bit_signature_calculation/No RTOS/CCS Compiler/crc32_32-bit_signature_calculation](#), and modify it for this exercise. You will only need to modify the main C file, `crc32_32-bit_signature_calculation.c`.

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

Exercise 3.1

Change the array size of `myData` to be much bigger, like 10240 (10K). Add a function or some code near the beginning of `main()` to initialize the array with random numbers. You can use the C library function `rand()` or some other way you prefer. You also need to make the numbers odd and even alternatively, for example, 12, 15, 34, 255, 70, ... So, this is the block of data we are going to compute the checksums on. You may want to use `printf` to print out some beginning numbers to convince yourself that the data pattern is correct.

Lab Report Submission

1. List the code to generate the data.

2. Briefly explain how you create the required data pattern.

Exercise 3.2

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

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

The function will just add all the data in the data array together into a 32-bit word. Upon returning from the function, reverse all the bits in the sum value, i.e., one becomes zero and zero becomes one.

Lab Report Submission

1. List the function.
2. Briefly explain how it works.

Exercise 3.3

At this point, you should have three methods to compute a 32-bit checksum of the 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. The results from the first two should be identical. (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. Use the time measurement method that you have used in Exercise 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 checksum methods (not the one you wrote), compute the speedup of using the accelerator over the pure software method. Print the result.

Lab Report Submission

1. List the code that does the measurements.
2. Show the debug console outputs: times, checksums and speedup.
3. List the entire program listing in the appendix.

Exercise 3.4

We are going to make some small changes to the data and see how the checksum changes. We will just use two methods: *MAP_CRC32_xxx()* and *compute_simple_checksum()*, i.e, the hardware method and the method that you wrote. Pick a data point in the data block, say, *myData[20]*. Reverse the least significant bit. Compute the checksums and print them. Then, reverse the least significant bit of the following byte, *myData[21]*. Compute the checksums and print them.

Lab Report Submission




1. List the code that does the above actions.
2. Show the debug console outputs: times, checksums.

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 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 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.

6. Grading

Lab 2 Rubric										  
Criteria		Ratings								Pts
Exercise 1.1		1 pts Full Marks Correct setup and control of 3 LEDs. Program listing in appendix.					0 pts No Marks Not attempted or reported.			1 pts
Exercise 1.2		1 pts Full Marks Correct implementation. Program listing in appendix.		0.6 pts Operations Described correctly the operations of control.		0.3 pts Port address Used correct memory address and bit position for the blue LED.			0 pts No Marks Not attempted or reported.	1 pts
Exercise 1.3		1 pts Full Marks Correct implementation. Program listing in appendix.		0.6 pts Operations Described correctly the operations of control.		0.3 pts Alias address Used correct alias address for control.		0 pts No Marks Not attempted or reported.	1 pts	
Exercise 2		2 pts Full Marks Correct implementation. Program listing in appendix.		1.5 pts Frequency Reported correct blinking frequency.	1 pts Measurement Correct measurement method.	0.5 pts Listing and outputs Listed the relevant code that does the measurement. Showed the debug console outputs.			0 pts No Marks Not attempted or reported.	2 pts
Exercise 3.1		1 pts Full Marks Correct implementation.		0.6 pts Method Correct method.		0.3 pts Data generation Listed the code to generate the data.		0 pts No Marks Not attempted or reported.		1 pts
Exercise 3.2		1 pts Full Marks Correct implementation.		0.6 pts Method Correct method.		0.3 pts Function Listed the function.		0 pts No Marks Not attempted or reported.		1 pts
Exercise 3.3		2 pts Full Marks Correct implementation. Program listing in appendix.		1.5 pts Outputs Showed correct outputs.	1 pts Measurement Showed correct measurement methods.		0.5 pts Measurement Listed code that does measurements. Show debug console outputs.		0 pts No Marks Not attempted or reported.	2 pts
Exercise 3.4		1 pts Full Marks Explained correctly which method is better. Program listing in appendix.			0.7 pts Outputs Produced correct outputs.	0.3 pts Listing Listed relevant code that changed the data pattern. Showed debug console outputs.			0 pts No Marks Not attempted or reported.	1 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.