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

Lab Assignment 2

Due date: 09/22/2024, Sunday

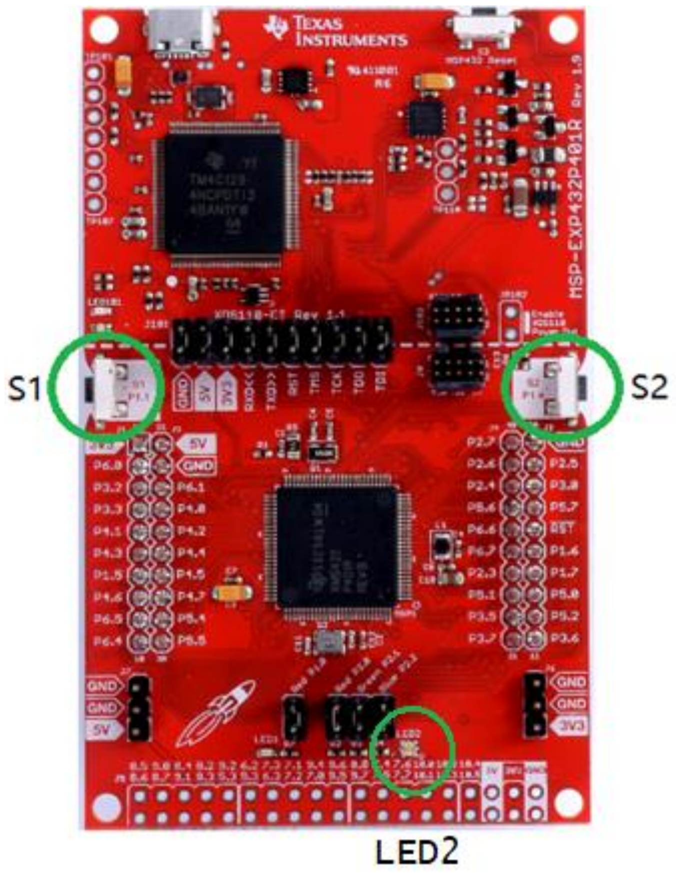
1. Overview

In this assignment, we will be familiarized with the bit-banding feature of the MSP432 MCU for controlling external devices. We will also use a 32-bit timer to measure time precisely so that we can characterize some events accurately.

2. Exercise 1 Bit-Banding

In this exercise, we will use different methods to use a push button on the LaunchPad to control an LED with and without using the DriverLib functions. In particular, we will learn how to use the bit-banding feature of the MCU to streamline the control of the device.

In this exercise, we are going to use the left push button (S1) on the LaunchPad to turn on or off a blue LED.



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 we 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, an 8-bit byte in the MCU's address 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, the same example project we used in Lab 1 (on Windows, the project is the ccs version in a folder like C:\ti\simplelink_msp432p4_sdk_3_40_01_02\examples\nortos\MSP_EXP432P401R\driverlib\gpio_toggle_output). From the schematic in the user's guide of the LaunchPad (Section 6), we 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; a value of 1 turns on the LED whereas 0 turns it off.

To toggle the control bit, i.e., to change the state of the LED, DriverLib provides the following function.

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

The function reverses the state of the control bit in the port. If we call this function repeatedly at a certain interval, the LED will in effect blink at a particular frequency.

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. To set the control bit high, i.e., to turn on the LED, DriverLib provides the following function.

```
MAP_GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0)
```

To set the control bit low, i.e., to turn off the LED, the following function can be used.

```
MAP_GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN0)
```

There are two push buttons on the LaunchPad. The port number and pin number of the push buttons can be found in the LaunchPad's schematic. We can read the state of a button by using the *MAP_GPIO_getInputPinValue()* DriverLib function. As with the output control pins, we will need to set up the input pins first. Since it will be necessary to use a pull-up resistor to read the input state properly, you will need to use the *MAP_GPIO_setAsInputPinWithPullUpResistor()* function.

Consult the "General Purpose Input/Output (GPIO)" chapter in the DriverLib User's Guide for details on the GPIO functions.

Exercise 1.1 DriverLib

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

Import (and rename) the *empty* skeleton project (the one you used in Lab 1, Exercise 2) 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) and the S1 push button. Modify *main()* accordingly to set up three output pins and one input pin.

Add the following line to the *include* file section for using the *printf* and other library functions.

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

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* control loop is entered. In the forever-running control loop, we will use S1 to toggle the state of the blue LED. That is, when we first press S1, the system will turn on the blue LED, and when we press S1 again, it will turn it off. The control loop just repeats this process without exiting. Note that if we press and hold S1, the LED state changes only at the moment S1 is pressed; the LED state does not change when S1 is being held. To simplify control, no button debouncing action needs to take place. All the control logic of this application is contained in the forever control loop; no interrupt or other mechanisms shall be used.

Take a short video clip of the control operations of the LED, demonstrating the fulfillment of the stated requirements above. Show short and long pressing of button. Place the video file where it can be accessed remotely.

Lab Report Submission

1. List the relevant code to do the setup, state initialization and control of the LED and the push button (the devices under control in this application).
2. There should be a difference in power consumption of the LaunchPad between the two states of S1 (pressed or not pressed). Which state will consume more power? Justify your answer in reference to the LaunchPad’s circuitry.
3. 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.
4. List the entire program in the appendix.

Exercise 1.2 Direct Port Access

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

Duplicate (and rename) the project in the previous exercise. To duplicate a project on CCS, right-click on the project on the project pane, copy it, then paste it back.

We need to find the memory address that corresponds to the port of the LEDs and the push button. We 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’s 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 change the output state to drive the LED. Repeat the same procedure for the push button to get the port address.

To simplify the device setup, let's keep the DriverLib functions as in Ex. 1.1 to set up and initialize the states of the LEDs, before entering the *while* loop to control the LED. In the *while* loop, **do not** use any DriverLib functions. Create two **byte** (`uint8_t`) pointers to achieve that instead, one for the LED and another for the push button. 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, we 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. Similarly, when we read from the push button port, we need to pick out the correct bit for S1.

For the MSP432 MCU, one port typically consists of 8 bits, i.e., there can be up to 8 devices associated with it. In the table, there are several addresses associated with a port for different purposes. We just need the one that we can use to read/write the state of the input/output pin; we can ignore other addresses.

Tip: On the MSP432 MCU, the ports can be accessed either as 8-bit bytes or 16-bit words. If we try to access them as a 32-bit word, the results are not predictable because such access method may not be supported in the hardware (at least it is not documented). So, do not create a 32-bit word pointer to access an I/O port. And keep in mind that all the addresses (or pointer values) are 32-bit (nature of the MCU), regardless of the data type.

Lab Report Submission

1. Report the port addresses and bit positions that are used to control the blue LED and to read the push button's state.
2. List the relevant code to control the blue LED.
3. Explain the operations in your control loop.
4. List the entire program in the appendix.

Exercise 1.3 Bit-banding Access

In this exercise, we will use the direct memory write method using bit-banding to control 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 address space. Using the locations in the bit-band alias region allows us to deal with a specific bit in a port.

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

```
alias_addr = (port_addr - peripheral_region_addr) * 32 + bit_position * 4 +
             alias_region_addr,
where
port_addr = port address of interest in the MCU's address space,
peripheral_region_addr = starting address of the entire peripheral region,
bit_position = bit position of interest in the port, and
alias_region_addr = starting address of the entire peripheral alias region.
```

We can find the value of *`alias_region_addr`* in Table 1-7 of the MCU technical reference manual. The mechanism of bit-banding is also described in Section 1.4.5 of the MCU technical reference manual.

Duplicate (and rename) the project in the previous exercise. In your program, translate the port addresses and the bit positions in the previous exercise to alias addresses. Print the computed addresses to the debug console. Writing to an alias address can only change one specific bit in the port. Similarly, reading from an alias address reflects only one bit of information. With this, change the way you control the blue LED in the *while* loop. The operations of control should be much simplified from what was done in the previous exercise. There should not be any bit-wise operations needed.

When we use a pointer to access the alias region, make sure it is either a 16-bit or 8-bit pointer. Using a 32-bit pointer will result a memory access error, and the program will crash. We can use width-specific data types like `uint8_t` or `uint16_t` for access to the alias region.

Lab Report Submission

1. Show the debug console outputs: the computed alias addresses for the LED and push button.
2. List the relevant code to control the blue LED.
3. Explain the operations in the control loop. What are the advantages of the control method in this exercise, compared to the one in the previous exercise?
4. List the entire program in the appendix.

3. Exercise 2 Time Measurement

In this exercise, we will do some time measurement of how long the push button is pressed. We will use a high-precision 32-bit counter/timer in the MCU and several DriveLib functions.

Duplicate the project for Exercise 1.1 (where we turned on and off an LED with a push button). 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, we use the *MAP_Timer32_getValue()* function like the following.

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

Use the *MAP_Timer32_getValue()* function in the control loop to measure repeatedly how long the push button is pressed. Basically, we read the counter values when the button is detected pressed and when it is detected released. The difference would be the duration in counter count. Use the system clock frequency and the count difference to compute the duration of the push button being pressed in milliseconds (use the formula $T = 1/F$, where T is the period and F is the frequency). We use *printf* to display the computed time (in ms) after the button is released. And do not worry about the timer wrap-around situation because the timer "capacity" is much larger than what we need for this exercise.

Lab Report Submission

1. Show a sample of the debug console outputs. Show both long and short presses.
2. List the relevant code that does the measurement.
3. List the entire program listing in the appendix.

4. Exercise 3 Frequency Measurement

In this exercise, we will do some measurement of how fast (in Hz) the LED is blinking in a previous exercise. Basically, we will the same measurement methodology in the previous exercise.

Duplicate the project in Exercise 3 of Lab 1. Add the necessary *include* file for the library function *printf* and the setup code for the 32-bit timer.

Make sure the *for* loop count in the *while* loop set to 100,000 so that we can see the LED is blinking slowly. Use the *MAP_Timer32_getValue()* function to measure how long the delay *for* loop generates. Basically, we read the counter values before and after the loop. The difference would be the duration in counter count. Use the system clock frequency to compute the duration and the blinking frequency (use the formula F = 1/T, where T is the period and F is the frequency). Note that blinking consists of two operations: turning the LED on and off; but each *while* loop iteration only does ON or OFF, not both. Therefore, the frequency is computed from two time durations. And do not worry about the timer wrap-around situation because the timer “capacity” is much larger than what we need for this exercise.

At the very end of the *while* loop, use *printf* to print the counter value difference, the duration in real time in milliseconds and the blinking frequency. The program should repeatedly display those results on a single line on the debug console.

Lab Report Submission

1. List the relevant code that does the measurement.
2. Show a sample (a few lines) of the debug console outputs.
3. All measurement methods carry some uncertainty or error. Discuss a major source of error of the method used to measure the blinking frequency.
4. List the entire program 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.