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

Lab Assignment 4

Due date: 10/11/2020, Sunday

1. Description

In this assignment, we will take a closer look of the memory subsystem on the MSP432 MCU. We will learn how to change the contents in the flash memory. We will use a 32-bit timer for timing control purpose.

2. Exercise 1

In this exercise, we will see how the compiler uses different types of memory in a program. We will also do a bit of flash memory programming.

Exercise 1.1

Create a new project (or duplicate the *empty* example project) to contain the following code:

```
#include <ti/devices/msp432p4xx/driverlib/driverlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>

char* string1  = "12345";
char string2[] = "12345";

void main(void)
{
    printf("string1: %s\n", string1);
    printf("string2: %s\n", string2);
    string1[0] = '0';
    string2[0] = '0';
    printf("string1: %s\n", string1);
    printf("string2: %s\n", string2);
}
```

This small program defines two identical strings with a slightly different method. It modifies them the same way. Build and run the program.

Explain the outputs from the program. Look up from the datasheet to find data to support your explanation. You may want to use the debugger or adding more code in the program to find out more about the two strings.

Lab Report Submission

1. Show the debug console outputs.
2. Provide explanations on the effects of the modifications.
3. Provide concrete data from the datasheet to support your explanation.
4. Provide the program listing in the appendix, with the changes you may have added.

Exercise 1.2

In this exercise, we will learn how to write or program the flash memory.

Unlike SRAM or DRAM, writing to the flash memory is a bit more involved. And it also has a pretty serious limitation; you can only change a memory cell's content from 1 to 0, but not the other way around.

By default, the entire flash memory is write protected. Before we can change its contents, we need to unprotect it first. The protection scheme is based on sectors. Each sector is 4 KB long. You unprotect the entire 4-KB sector; you cannot just unprotect a particular memory location. You can use the following DriverLib function to unprotect a sector.

```
bool FlashCtl_unprotectSector(uint_fast8_t memorySpace, uint32_t sectorMask)
```

memorySpace is the flash memory bank number and *sectorMask* indicates which sector(s) to unprotect.

To write to the memory, use the following DriverLib function.

```
bool FlashCtl_programMemory(void* src, void* dest, uint32_t length)
```

Essentially, it writes the contents pointed to by *src* to the flash memory location starting at *dest*. Consult the DriverLib User's Guide for details of the function usages.

Duplicate the project in the previous exercise. Define one *char* array, *string3*, like the following.

```
char* string1  = "12345";
char string2[] = "12345";
const char string3[1024 * 150] = {"12345"};
```

Print out the contents of all strings first. Then modify *string2* as follows.

```
string2[0] = '0';
string2[1] = '4';
```

Based on what you have learned from the previous exercise, in order to modify the other two strings, you will need to use the *FlashCtl_xxx* calls mentioned above. Duplicate the first two bytes of *string2* onto *string1* and *string3* using the following calls.

```
ROM_FlashCtl_programMemory(string2, string1, 2);
ROM_FlashCtl_programMemory(string2, string3, 2);
```

Notice that there is a *ROM_* prefix to the program function. We cannot change the flash memory while the program is also running from there. To avoid the executing program and the data to be changed being in the same sector, we use the DriverLib functions residing in the ROM to execute the write operation.

As mentioned above, before you write, you will need to unprotect the sectors where the first two bytes of string1 and string3 reside. Write a small function to convert a memory location to bank number and sector mask. Then use them on the *FlashCtl_unprotectSector()* calls. After changing the contents, print out the three strings again.

Lab Report Submission

1. Show the function to convert a memory location to bank number and sector mask.
2. Show the relevant code to unprotect and program the memory.
3. Show the debug console outputs.
4. Describe and provide an explanation on the changes of the strings.
5. Provide the program listing in the appendix.

3. Exercise 2

In this exercise, we will build a small function to delay a specific number of milliseconds (ms). It will be a useful function for timing control purposes.

Exercise 2.1

We will build the first version of the delay function.

Duplicate the *empty* example project. Add the following line to the *include* file section for using *printf()*.

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

Make sure you have the following line to show the current system clock frequency at the beginning of *main()*.

```
printf("MCLK: %u\n", MAP_CS_getMCLK());
```

We use one of the 32-bit Timer32 timers. Before we can use the timer, we need to initialize and set it up first. Add the following initialization and setup steps after the *MAP_WDT_A_holdTimer()* function call in *main()*.

```
MAP_Timer32_initModule(TIMER32_0_BASE, TIMER32_PRESCALER_1, TIMER32_32BIT,
TIMER32_FREE_RUN_MODE);
MAP_Timer32_startTimer(TIMER32_0_BASE, 0);
```

The delay function prototype is as follows.

```
void delay_ms(uint32_t count);
```

The input argument *count* is the number of ms to delay. Inside the function, use only the *MAP_Timer32_getValue()* and *MAP_CS_getMCLK()* DriverLib functions. Do not reset the timer or change the timer. You cannot assume the system running at a particular frequency. To reduce computing overheads, do not use any floating point arithmetic, either explicitly or implicitly. Do not use interrupt. It is okay to use a loop to do the job. And do not worry about the timer wrap-around situation (yet) because the timer "capacity" is much larger than what we need in this exercise.

Use the timing method in the previous labs to measure if the delay function works properly. So, you can do something like this:

```
t0 = MAP_Timer32_getValue(TIMER32_0_BASE);
delay_ms(1000);
t1 = MAP_Timer32_getValue(TIMER32_0_BASE);
```

Print out timer value difference and the measured time in µs. Measure these delay values: 5000, 2000, 1000, 5, 1, 0. The measured values should be very close to the delay values.

Lab Report Submission

1. Show the delay function. Briefly explain how it works.
2. Show the relevant code to measure the time and print the results.
3. Show the debug console outputs.
4. Provide the program listing in the appendix.

Exercise 2.2

We will use the delay function to blink the red LED (LED1) precisely at 0.5 Hz.

Duplicate the project in the previous exercise. Replace the measurement code with the code to blink the red LED. Compute the required delay for timing control. Use the basic code in the previous labs to set up and blink the LED.

Use a stopwatch to verify that the LED is blinking at the correct frequency. Take at least three measurements. Provide the data and compute the blinking frequency.

Lab Report Submission

1. Report the delay time used in the control loop.
2. Show the relevant code to do the blinking.
3. Provide the data from your stopwatch measurements. Report the calculated frequency.
4. Provide the program listing in the appendix.

Exercise 2.3

We will build a better version of the delay function that can handle a special situation in the timer.

For systems running at much higher clock frequencies than the default 3 MHz, we will quickly run into the situation that the timer wraps around back to 0xFFFFFFFF after counting down to zero. If no special handling is put in place to handle the situation, the amount of delay produced by the delay function may not be correct.

Duplicate the project in Exercise 2.1. To make the wrap-around situation happen more frequently for testing purpose, we simulate a reduced timer capacity in the delay function. Define the following constant

```
#define TIMER_MAX 0x000fffff
```

Whenever you read the current count from the timer, AND the value with *TIMER_MAX*. So, you may do something like this:

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

Basically, we reduce the timer from 32-bit to 20-bit. Only do the ANDing in the delay function, not when you measure the delay function. Modify or rewrite your delay function to handle the wrap-around situation.

Print out timer value difference and the measured time in µs. Measure these delay values in the following order: 10000, 5000, 2000, 1000, 10, 5, 2, 1, 0, 10000. Again, the measured values should be very close to the desired delay values.

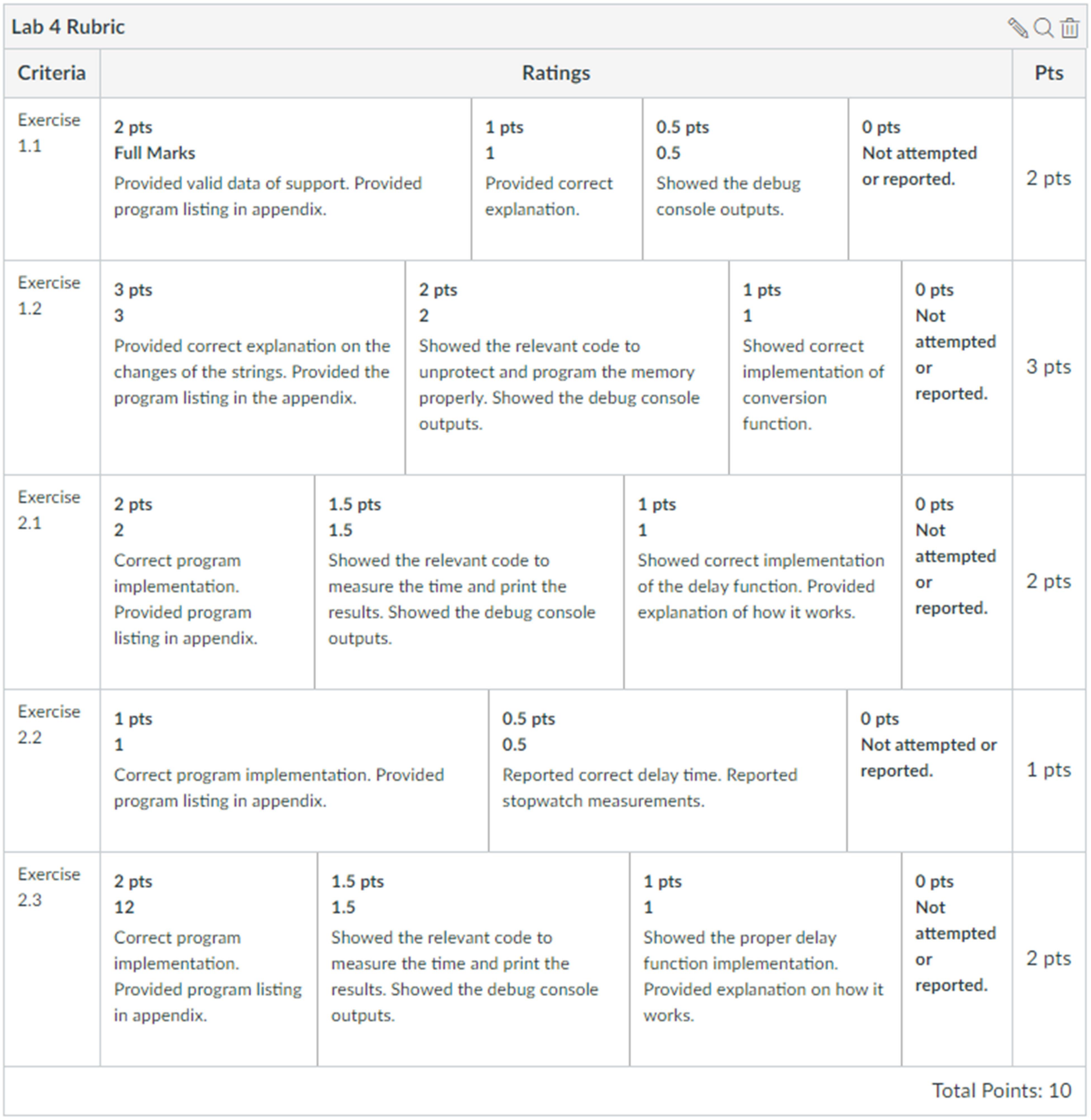
Lab Report Submission

1. Show the delay function. Briefly explain how you handle the wrap-around.
2. Show the relevant code to measure the time and print the results.
3. Show the debug console outputs.
4. Provide the program listing in the appendix.

4. 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.

5. Grading



Lab 4 Rubric

Criteria	Ratings				Pts
Exercise 1.1	**2 pts** **Full Marks** Provided valid data of support. Provided program listing in appendix.	**1 pts** **1** Provided correct explanation.	**0.5 pts** **0.5** Showed the debug console outputs.	**0 pts** **Not attempted or reported.**	2 pts
Exercise 1.2	**3 pts** **3** Provided correct explanation on the changes of the strings. Provided the program listing in the appendix.	**2 pts** **2** Showed the relevant code to unprotect and program the memory properly. Showed the debug console outputs.	**1 pts** **1** Showed correct implementation of conversion function.	**0 pts** **Not attempted or reported.**	3 pts
Exercise 2.1	**2 pts** **2** Correct program implementation. Provided program listing in appendix.	**1.5 pts** **1.5** Showed the relevant code to measure the time and print the results. Showed the debug console outputs.	**1 pts** **1** Showed correct implementation of the delay function. Provided explanation of how it works.	**0 pts** **Not attempted or reported.**	2 pts
Exercise 2.2	**1 pts** **1** Correct program implementation. Provided program listing in appendix.	**0.5 pts** **0.5** Reported correct delay time. Reported stopwatch measurements.	**0 pts** **Not attempted or reported.**		1 pts
Exercise 2.3	**2 pts** **12** Correct program implementation. Provided program listing in appendix.	**1.5 pts** **1.5** Showed the relevant code to measure the time and print the results. Showed the debug console outputs.	**1 pts** **1** Showed the proper delay function implementation. Provided explanation on how it works.	**0 pts** **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.