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

Lab Assignment 4

Due date: 10/20/2024, Sunday

1. Overview

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 flash memory. We will build a simple application that uses flash memory to keep track of the power-on reset events. We will also compare the access times of SRAM and flash memory.

2. Exercise 1 Flash memory

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 String modifications

Import (and rename) the *empty* skeleton project. Modify the main C file to contain the following code:

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

char* string1  = "123123";
char string2[] = "123123";

void main(void)
{
    MAP_WDT_A_holdTimer();
    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 declares two identical strings with slightly different methods. It modifies them the same way, changing the first character of the strings. Build and run the program, and examine the debug console outputs. Explain why the program produces the outputs observed. Consult the MCU datasheet to find the information to support your explanation. You may want to use the debugger or add more code in the program to find out more about the two strings. (Hint: It is related to the memory organization on the MCU and where

the two strings are physically stored. It has nothing to do with compiler versions in this case. If you cannot come up with anything, move on to the next exercise and revisit this exercise later.)

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 have added, if any.

Exercise 1.2 Program flash memory

In this exercise, we will learn how to program (or write to) the flash memory on MSP432.

Unlike SRAM or DRAM, programming flash memory is quite involved. It is not simply to generate a memory-write transaction in the bus; there is a sequence of operations involved. Flash memory has a pretty serious limitation: you can only change a memory cell's content from 1 to 0, but not the other way around. Furthermore, you cannot write to it unlimited number of times.

By default, the entire flash memory on the MCU 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. We unprotect the entire 4-KB sector; we cannot just unprotect a particular memory location. We 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 and *sectorMask* indicates which sector(s) to unprotect. Consult the DriverLib User's Guide for details of the function usages.

To program 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. Declare one *char* array, *string3*, exactly as the following.

```
char* string1  = "123123";
char string2[] = "123123";
const char string3[1024 * 160] = {"123123"};
```

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

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

In order to modify the other two strings, *string1* and *string3*, we will need to use the *FlashCtl_xxx* DriverLib functions. 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 (it is just not safe to do that). 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 program operation. **Warning**: Do not use any erase function in this lab assignment. The function is not necessary anyway, and more importantly, if it is not done properly, the board may become inoperable and may not be recovered from the error.

As mentioned above, before we program, we 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. Note that since we are changing just two bytes, which should be in the same sector, the sector mask should have just one single bit turned on.

Use the function prototype as shown below.

```
bool get_flash_bank_sector(uint32_t mem_address, uint32_t* bank_number,
uint32_t* sector_mask);
```

The function returns *true* if there is no error; otherwise, it returns *false*, indicating some error condition. The bank number and sector mask are returned through two pointer arguments. In *main()*, print the bank number and sector mask returned by the function in hexadecimal. (Note: There is a *MAP_FlashCtl_getMemoryInfo()* function in DriverLib, but it does not return the exact information we need. Do not use it for this exercise.)

Lab Report Submission

1. List 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, including at least the bank numbers and sector masks in hex.
4. Describe and provide an explanation on the changes of the strings.
5. Provide the program listing in the appendix.

3. Exercise 2 Non-volatile power-on-reset counter

In this exercise, we will build a small program to limit how many times the system can function. If the user has powered on the board for a few times, the board will no longer do anything.

The function here is just blinking the green LED at precisely 1 Hz. (It can be something more useful in real-world situations.) The system basically uses the non-volatile flash memory to keep track of how many times the program has been restarted upon the LaunchPad being powered on. If the board (system) has been powered on more than **three** times (a small number here just for easier testing), the program will not blink the green LED anymore. Instead, it turns on the red LED and enters an infinite forever dummy loop (for a real-world application, this can be some alternate actions).

Duplicate the project in the previous exercise and modify the main C file accordingly. Add in the flash memory programming technique in Exercise 1 to implement a small "counter" to keep track of how many times the program has executed since it was downloaded from CCS. Do not arbitrarily pick some location in the flash memory space for the counter; let the compiler do the memory allocation. We can use a 32-bit unsigned word allocated (by the compiler) in the flash memory, similar to the way where we store the strings in the previous exercise.

No erase operation shall be used. Use the fact that when we program a flash memory location, we can only change bits from 1 to 0, but not from 0 to 1. Initially, the "counter" contains a constant value when the program is built. Each time the program starts up, it will check if such counter has expired or not. If it has, it turns on the red LED (to indicate the board is no longer working) and enter the idle loop. Otherwise, it "decrements" the counter, then it proceeds forward to control the LED.

To blink the LED, we can use the method in the *gpio_toggle_output* example project. Instead of using the *for* loop to determine the timing, we can replace it with the *__delay_cycles()* function that we used in previous lab. We provide the proper value in the argument to the function that corresponds to the desired blinking frequency.

To test the program, we will need to disconnect the LaunchPad USB cable from the laptop/PC to remove the power. After building and downloading the program on CCS, do not proceed to execute the program. Disconnect the cable, wait for a couple of seconds, then reconnect the cable. We should see the green LED blinking. If we disconnect and reconnect again, the LED will still blink. However, the same action on and after the 4th power-on, the system will not blink the green LED, but it will turn on the red LED permanently.

We only set the number of times the system can function upon power-on to a small number, just for easier testing. However, the countdown counter you devise shall be able to handle a much larger number, like at least 30 times. So, devise a scheme that you can just change one single constant integer in your program, and it will still produce the desired outcomes.

Take a short video clip of at least 5 power-on cycles. Place the video file where it can be accessed remotely.

Lab Report Submission

1. List relevant code to manipulate the "counter" and briefly explain the implementation of the countdown. Report the countdown sequence in your test.
2. If you are to make the system work (blink) for 13 times (instead of three) of power-on cycles, exactly how would you change the program?
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. Provide the program listing in the appendix.

4. Exercise 3 SRAM function

In this exercise, we will investigate the performance of running a small function in the SRAM and flash memory. Our entire application is typically stored in the flash memory. The CPU fetches instructions from the flash memory or ROM. However, in some situations, it will be advantageous to execute instructions from the SRAM. We will see how it can be done with CCS.

To place a function in the SRAM for execution, we can use the pragma directive as shown in the following code snippet.

```
#pragma CODE_SECTION(sram_function, ".TI.ramfunc")
void sram_function()
{
    register int i;
    register int count = 100000;
    for (i=0; i<count; i++)
    {
    }
}
```

The pragma directs the compiler (and linker) to place the function *sram_function()* in the SRAM for execution. It specifies the function name and the memory section. A program consists of several memory allocation sections. Program instructions and data are typically placed in different sections. The *.TI.ramfunc* section is a special section where the functions there are copied from the flash memory to SRAM upon system startup (before *main()* is entered). The application will then use the functions in the SRAM, rather than those in the flash memory.

Import (and rename) the *empty* example project for this exercise. Make sure to include <*stdio.h*> for *printf* and other library functions. Place the code snippet above in the main C file. Duplicate the code but without the pragma. Rename the function. Now, we have two identical functions, one in the SRAM and another in the flash memory, to execute during run time. The function is written so that in the *for* loop, there are no data memory accesses at all. That is, while the loop is being executed, the memory access happens only on either the SRAM or flash memory, but not on both.

Measure the execution times of both functions in *main()*. Use the timing method that we have used before (with the 32-bit timer). Print the times to the console in µs. Also print the function's starting address to ensure that it is either in SRAM or flash memory. Compute the speedup using the flash-memory function as the baseline.

The default system clock frequency is 3 MHz. Let's change it to 48 MHz to see how the program behaves. From the Project Explorer, open *system_msp432p401r.c*. Look for the following line in the file.

```
#define  __SYSTEM_CLOCK    3000000
```

Change the frequency from 3000000 (3 MHz) to 48000000 (48 MHz). Be sure that the number is changed correctly as invalid numbers may make the board inoperable when the program runs. Build and run the program so that we get a different set of data with a much higher frequency.

By default, the read buffering feature of the flash memory controller is enabled. To get a more accurate measurement of the flash memory access performance, we should disable the feature. Add the following two lines at the beginning of *main()*.

```
FlashCtl_disableReadBuffering(FLASH_BANK0, FLASH_INSTRUCTION_FETCH);
FlashCtl_disableReadBuffering(FLASH_BANK1, FLASH_INSTRUCTION_FETCH);
```

Lab Report Submission

1. Show the debug console outputs with two different clock frequencies.
2. Describe your observations from the data and provide an explanation.
3. Provide the 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.