UNIVERSITY OF CENTRAL FLORIDA

Lab Manual for EEL 4742C Embedded Systems

Department of Electrical and Computer Engineering

Rev. August 2022

Introduction

In this lab, we will learn designing embedded systems for low-power applications. Numerous embedded applications are low-power and run in the “bare metal” environment. This is an environment that doesn’t use an Operating System (OS) and where the software has full control over the hardware. The low-power embedded devices draw a very small current that can go down to micro-Amps and can operate on a small battery for years. Our platform is the Texas Instruments MSP430 architecture.

We will use the microcontroller board and the sensor board that are listed below. The sensor board has a variety of devices such as a light sensor, a temperature sensor, a 128x128 pixel LCD display with 18-bit color, a tri-color LED, a microphone, a buzzer and a 2D joystick.

- **Microcontroller Board:** Texas Instruments MSP430FR6989 LaunchPad
  Part number: MSP-EXP430FR6989
- **Sensor Board:** Texas Instruments Educational BoosterPack Mark II
  Part number: BOOSTXL-EDUMKII

Skills

The list below shows the skills that we will learn in this lab.

- Flashing the LEDs
- Using the push buttons
- Using the timer
- Programming interrupts
- Using the low-power modes
- Using the segmented LCD display
- Performing communication with the UART, I2C and SPI protocols

- Using the Analog-to-Digital converter
- Programming pixel-based LCD graphics
- Using advanced timer features (multiple channels, timer-driven input/output)
- Programming concurrent events with interrupts

Lab Policy

Below is the general policy of this lab. A more detailed policy on how the lab is graded is posted on WebCourses or as part of the syllabus.

- Lab attendance is required.
- All labs are weekly labs.
- The lab work should be demoed to the TA at the end of the session or by the start of the next session.
- If an experiment is not finished in its session, the student should come to the lab after hours to work on it and have it ready by the start of the next session.
- Lab access for after hours can be requested from the lab manager by following the instructions that are posted on the lab front door.
- Demoing the code to the lab instructor is required. The report will not be graded if the code was not demoed.
- A lab report that contains the requested deliverables should be submitted for each experiment. The lab report is due by the start of the next session; seven days after the current session. Use the report template included in this manual.
- The lab report is graded based on correctness, coding style and writing quality.

Coding Style

In this lab, we will practice writing code based on the recommended practices. The first practice is about modifying bits within a variable. Let's say that we are interested in modifying one bit of a variable. This implicitly means that all the other bits in this variable must remain unchanged. We should use AND, OR, XOR operations to clear, set or invert bits, respectively. The second practice is avoiding the use of hexadecimal masks since they are hard to read, especially when other people are reading our code. We should use symbolic masks instead.

As an example, let's set the rightmost bit of the variable `Data`. Out of the four cases below, the last one is the right way.

`Data = 0x01;` This statement is not correct since it changes all the other bits to zero. Secondly, hex masks should not be used.

`Data |= 0x01;` This is technically correct since only the rightmost bit is modified. However, hex masks should be avoided.

`Data = BIT0;` This statement uses the symbolic mask (BIT0 = 00000001 = 0x01); however, it's incorrect since it changes all the bits in the variable.

`Data |= BIT0;` This statement is the right way. Only the rightmost bit changes and the code is easily readable.

Note that, in some cases, we purposefully want to modify all the bits in the variable such as when we are configuring all the fields in a register. In this case, it is acceptable to use the assignment '=' operator (rather than using AND, OR, XOR). As an example, let's say we want to set the leftmost four bits of the variable `Control` and clear the rightmost four bits. Then, we can write the statement below.

```
Control = BIT7 | BIT6 | BIT5 | BIT4;
```

These recommended practices should be used in all the experiments of this lab.

Report Template

EEL 4742C: Embedded Systems

Name: ...

Lab number and title

Introduction

Write a paragraph that introduces all your work in this lab experiment.

Part 1: ...

Write a paragraph(s) that describes your approach to solving this part.

Include your code.

Answer the questions of this part.

Part 2: ...

For each part, do the same as Part 1 above

Student Q&A

Answer the questions.

Conclusion

Write a paragraph that has concluding notes on this lab experiment. Highlight what you learned and the significant parts of this lab.

Lab 1

Flashing the LEDs

In this lab, we will introduce the documentation files of the MSP430 boards and we will write programs that flash the LEDs.

1.1 Documentation

An essential part of programming microcontroller boards is being familiar with the documentation. For MSP430 boards, each board has three documents. Below are the ones that corresponds to our board, the MSP430FR6989 LaunchPad.

- MSP430FR6xx Family User's Guide `(slau367o)`
- MSP430FR6989 Chip Data Sheet `(slas789c)`
- LaunchPad Board User's Guide `(slau627a)`

The terms in parentheses are the file identifiers. We can obtain these files from TI's website or by searching for the file identifiers in a search engine. It's a good idea to download these files and keep them handy while doing the lab. Let's look at the content of these files.

The **family user's guide** explains how the microcontroller's components work. A microcontroller chip contains the CPU, memory and multiple peripherals such as timer, analog-to-digital converter, communication module, etc. The family user's guide has a chapter for each of these modules. This file explains how the things work, hence, it's similar to a textbook. The content of this file apply to a family of chips.

The **chip's data sheet** contains chip-specific information. It shows the pinout (what each pin is used for), how a multi-use pin can be configured to a specific function, the operating voltage levels, the accuracy of internal clocks, etc. We mostly use the data sheet to lookup information.

The **LaunchPad board user's guide** is specific to a LaunchPad board and explains how the board is wired and what external items are attached to the board. Typical attachments are LEDs and push buttons. This file shows to which pins the LEDs and buttons are attached and whether they are configured as active high or active low, information that is needed to write the code. This file also contains the schematic of the LaunchPad board.

The documentation for the Booster Pack board, which is our sensor board and will be used in later labs, is the following:

- Booster Pack User's Guide `(slau599a)`
- Various data sheets for each component on the Booster Pack

The first file shows the components that are attached to the Booster Pack such as the light and temperature sensors, the buzzer, the display, etc. This file has general information on how the components are connected together, but not on the inner working of these components. For each of the components, we would have to search for their specific data sheet from the respective manufacturer to see how they operate. For example, the Booster Pack has the light sensor from Texas Instruments model OPT3001. We would need to get the data sheet of this sensor in order to know how to operate it.

1.2 Flashing the LED

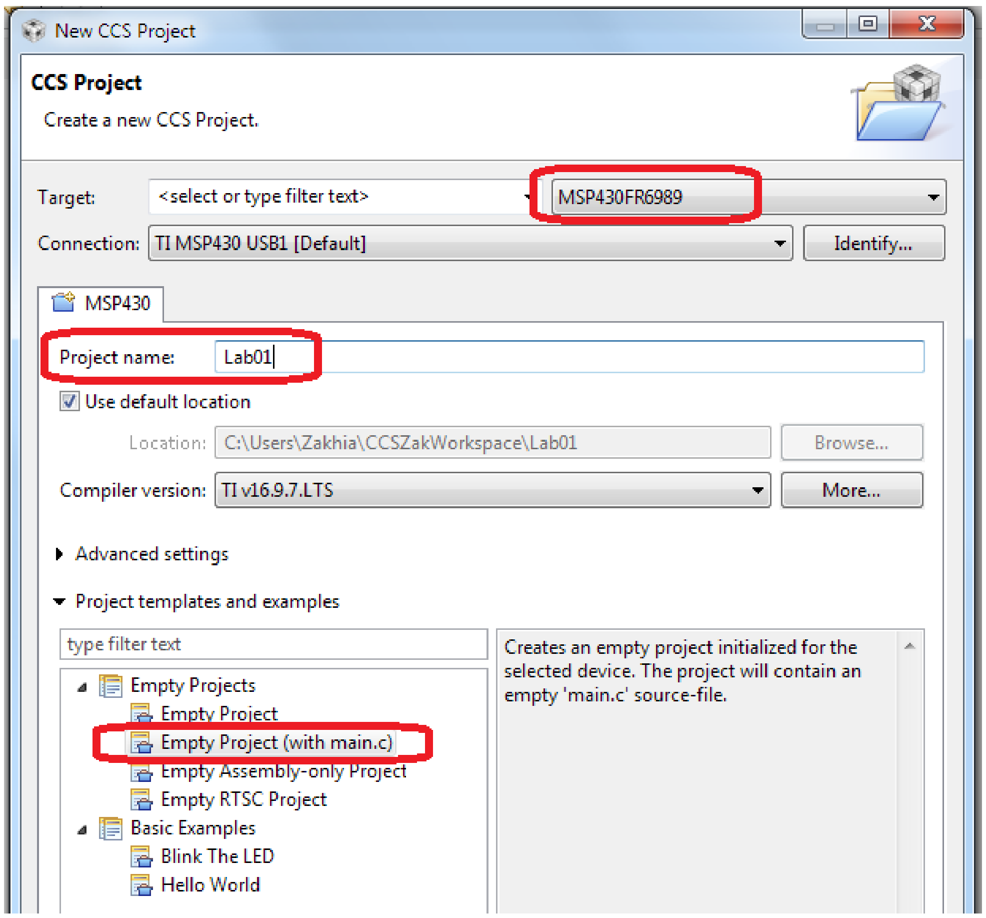
In this part, we will run a code that flashes the red LED on the board. Start Code Composer Studio (CCS) and click on the following menu items.

```
File -> New -> CCS Project
```

Create a project and select the chip number as shown in Figure 1.1.

Once the project is created, copy and paste the code below in the main file[1]. Let's go through the main concepts in the code.

[1]Note that when the code is copied to the editor, the characters ˜ (inverse) and ˆ (xor) may get corrupted and need to be retyped in the editor.



CCS Project

Create a new CCS Project.

Target: <select or type filter text> MSP430FR6989

Connection: TI MSP430 USB1 [Default]

Project name: Lab01

☑ Use default location

Location: C:\Users\Zakhia\CCSZakWorkspace\Lab01

Compiler version: TI v16.9.7.LTS

Creates an empty project initialized for the selected device. The project will contain an empty 'main.c' source-file.

Figure 1.1: Creating a project in Code Composer Studio

The red LED is mapped to Port 1 Bit 0, known as P1.0. The header file (.h) included in the code defines symbolic constants (masks) that facilitate accessing bits. These masks are (BIT0=00000001), (BIT1=00000010), ..., (BIT7=10000000). In general, BITn has all bits 0 except bit position n is equal to 1. Since the LED is mapped to P1.0, we'll redefine BIT0 as redLED so we can access the LED using this mask.

The variable 'i' is used in the delay loop. It counts from 0 to 20,000 to create a delay. It is declared as volatile so the compiler doesn't optimize away (eliminate) the delay loop. The compiler may assume that the delay loop doesn't do useful computation and eliminate it altogether. If you suspect that the compiler is eliminating the delay loop, go to the menu item below and reduce the level of optimization to level 0. Reducing the optimization level usually results in a larger executable size.

```
File -> Properties -> Build -> MSP430 Compiler -> Optimization
```

The following line disables the WatchDog Timer (WDT) mechanism. This mechanism causes periodic resets unless it's explicitly cleared by the code on a regular basis. It serves the purpose of protecting against software problems. We won't use the WDT in this code, thus, we disable it.

The following line clears the lock on the pins. Our board has nonvolatile memory. Upon startup, the memory is locked to keep the old data it has. We clear the lock so we can write new data to the memory and pins. Remember to include this line in every code you write, otherwise, the program most likely won't perform any action.

```
// Code that flashes the red LED
#include <msp430fr6989.h>
#define redLED BIT0              // Red LED at P1.0

void main(void) {
  volatile unsigned int i;
  WDTCTL = WDTPW | WDTHOLD;      // Stop the Watchdog timer
  PM5CTL0 &= ~LOCKLPM5;          // Disable GPIO power-on default high-
      impedance mode

  P1DIR |= redLED;               // Direct pin as output
  P1OUT &= ~redLED;              // Turn LED Off

  for(;;) {
    // Delay loop
    for(i=0; i<20000; i++) {}

    P1OUT ^= redLED;             // Toggle the LED
  }

}
```

The next two lines of code configure the pin as output and turn the LED off. Below, we can see the variables that control a port. Port 1 has eight bits, which are numbered 0 (rightmost) to 7 (leftmost). Each bit can be used as input or output. The second variable P1DIR (direction), configures each pin as either output (1) or input (0). For all the pins that are configured as output, the variable P1OUT is used to write to them (0 or 1). For all the pins that are configured as input, the variable P1IN is used to read from them (0 or 1).

```
Port 1:  _ _ _ _   _ _ _ _      The port has 8 pins
P1DIR:   _ _ _ _   _ _ _ 1      Sets a pin as input or output
P1OUT:   _ _ _ _   _ _ _ x      Write 0 or 1 to output pins
P1IN:    _ _ _ _   _ _ _ _      Reads (0 or 1) from input pins
```

Accordingly, the red LED is mapped to bit 0, the rightmost bit. Then, we write 1 (output) to P1DIR at bit position 0. To turn the LED on/off, we write to the variable P1OUT's rightmost bit. These bits are marked above by 1 and x, respectively.

When we change a bit to a specific value, it is necessary to leave the other bits in the variable unchanged so as not to affect other I/O (Input/Output) pins. Hence, the code performs an OR operation on P1DIR, as shown below. The terms a to h designate the eight bits of P1DIR. We OR P1DIR with the mask redLED. Bit 0 becomes 1 while all the other bits remain unchanged. This operation sets P1.0 to output so we can write to the LED.

```
 P1DIR: abcd efgh                         P1OUT: abcd efgh
redLED: 0000 0001   OR                  ~redLED: 1111 1110   AND
Result: abcd efg1                        Result: abcd efg0
```

To turn the LED off, we write 0 to its bit in P1OUT. This is an active high setting (1:on, 0:off). We AND P1OUT with the inverse of redLED as shown above (on the right side). Bit 0 becomes 0 while all the other bits remain unchanged.

Throughout all the experiments of this lab, we should use masking operations (AND, OR, XOR) as above to change individual bits. It's a bad practice to change all the bits when only one bit needs to be changed. Furthermore, hexadecimal masks should not be used because the code wouldn't be easily readable.

The remaining of the code starts an infinite loop. Inside, a delay loop counts up to 20,000 and has an empty body, therefore creating a delay as the CPU counts up and makes a comparison 20,000 times. When the delay elapses, the LED is toggled by applying the XOR operation. This is similar to the masking operations above. XORing a bit with 1 inverts it, while XORing the remaining bits with 0 leaves them unchanged. Therefore, this operation toggles the LED between on and off.

Running the Code

Let's build (compile) this program, flash it to the board and run it. Click on the green bug button shown in Figure 1.2 to start the build process. The code is compiled and flashed to the microcontroller. This is done via the JTAG (Joint Task Action Group) interface. JTAG is hardware on the board that works alongside software in CCS and allows programming the chip and enables debugging.

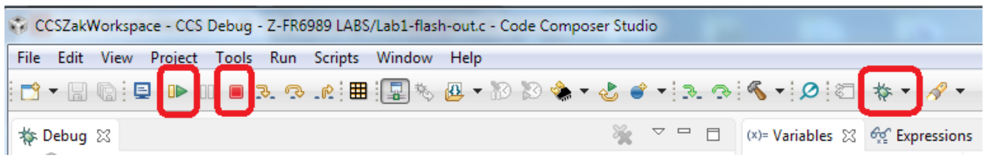
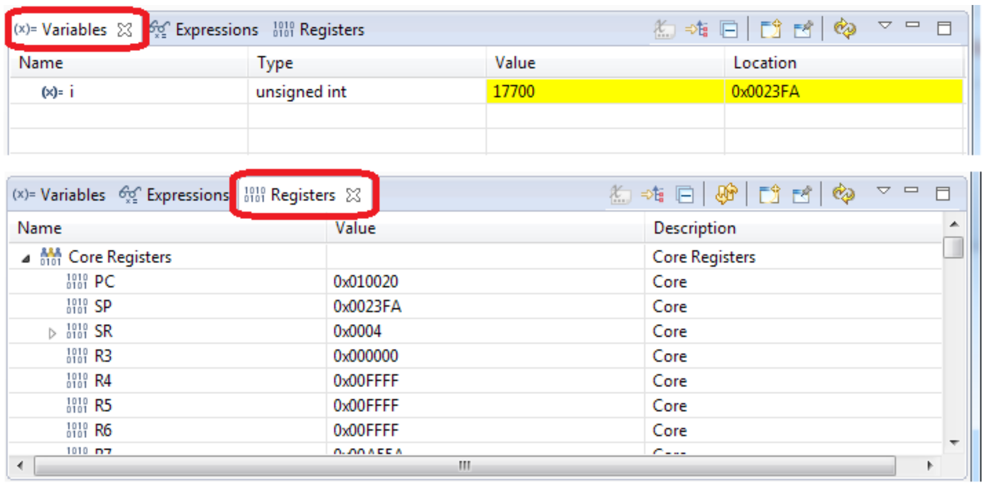

CCSZakWorkspace - CCS Debug - Z-FR6989 LABS/Lab1-flash-out.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
(x)= Variables
Expressions

Figure 1.2: Building (compiling) and launching a program.

Once the code is built successfully, the green play button and red stop button, shown in the figure, will appear. Click the green play button to start running the code in **debug mode**. In the debug mode, the code is running on the chip under the supervision of Code Composer Studio and the JTAG for the purpose of debugging. For example, we can click the pause button and inspect all the variables and the registers in the program, as shown in Figure 1.3. We can even change the value of variables and registers before resuming the execution.



Name	Type	Value	Location
i	unsigned int	17700	0x0023FA

Name	Value	Description
Core Registers		Core Registers
PC	0x010020	Core
SP	0x0023FA	Core
SR	0x0004	Core
R3	0x000000	Core
R4	0x00FFFF	Core
R5	0x00FFFF	Core
R6	0x00FFFF	Core

Figure 1.3: Checking the variables and registers values in debug mode.

It's also possible to create a breakpoint at a line of code by double clicking next to the line of code before building the program. In debug mode, the execution stops automatically when the break point is reached.

What happens if we click on the red stop button? We exit the debug mode and the code continues to run on the microcontroller in **normal mode**. While the debug mode and the normal mode generally exhibit the same behavior by the program, sometimes there is a difference in the program's behavior. For example, the reset button works in normal mode but not in the debug mode.

What happens if we unplug the board from the computer and plug it back? The board resets and the code restarts. It's running completely off the board and the code is not lost when the power is gone since it's stored in a non-volatile memory.

Perform the following actions:

- Experiment with the delay loop to make the LED flash slower or faster
- Run the code in debug mode; pause the code and check the variables and registers values (take a screenshot)
- Run the code in debug mode and test the reset button (the third button on the board); does it work?
- Run the code in normal mode and test the reset button; does it work?
- Disconnect the board from the computer and plug it back; does the code resume running?

Deliverables:

- In your report, submit a snapshot of the registers and variables values obtained from the debug mode.

1.3 Flashing two LEDs

In this part, we will modify the code so that it flashes both LEDs that are connected to the board. One is red and the other is green. To which port/bit is the green LED mapped?

One way to find out is to look at the board itself; it's written in small font next to the LED. Another way to find this out is by looking at the schematics in the LaunchPad user's guide. Open this file `(slau267a)` and check the schematics on pages 29-34. Find out where LED2 is connected. Take a screenshot of the schematic showing how the LED is connected. From the schematic, determine if this LED is active high (1:on, 0:off) or active low (0:on, 1:off).

Modify the code so that it flashes both LEDs. Like we did for the red LED, start by defining a mask called `greenLED` and configure the green LED pin as output.

Perform the following actions:

- Experiment with flashing the LEDs both in-sync and out-of-sync.

Deliverables:

- Demo your code to the lab instructor
- Include a screenshot of the board user's guide schematic that shows the LED connections
- Submit your code that flashes the LEDs in-sync (include the part that flashes the LEDs out-of-sync but comment it out)

1.4 Setting a Long Delay

In the code we have written, the delay loop upperbound (e.g. 20,000) specifies the delay we observe. Try an upperbound of 80,000 for the delay loop. Most likely this wouldn't work since the loop counter is declared as 'volatile unsigned int' and an 'int' type on the MSP430 is 16-bit. An unsigned 16-bit integer can go from 0 up to $2^{16} - 1$, which is the range [0 - 65,535]. Therefore, the largest delay we can set corresponds to 65,535.

One interesting detail here is, on a C compiler used for desktop development, an 'int' type is usually 32-bit, so how come it's 16-bit on the MSP430? The C standard specifies that an 'int' should be 16-bit **or more**. Hence, desktop compilers upgrade it to 32-bit while the MSP430 environment keeps it as 16-bit since it's a 16-bit CPU and has a 16-bit ALU.

Let's explore three solutions on how to set a larger delay. In the first solution, we use multiple delay loops back-to-back or nested to create a large delay. In the second solution, we will declare a 32-bit variable as the delay loop counter. Even though MSP430 is a 16-bit architecture, it's possible to use 32-bit variables. The compiler maps such variables to two registers and perform all the manipulation to synthesize 32-bit operations on the 16-bit CPU. To declare 32-bit variables, use the include line shown below. Declare the variable of the type shown below. This corresponds to 32-bit unsigned integer. This is standard C syntax and can be used on any C compiler.

```
#include <stdint.h>
...
volatile uint32_t i;    // unsigned int 32-bit type
```

In the third solution, we use the function shown below which creates a delay in terms of CPU cycles. This function is supported by the compiler. Its parameters is of type 'unsigned long' which is 32-bit on the MSP430 platform. Therefore, the maximum delay we can request is $2^{32} - 1$, which is more than 4 Billion cycles.

```
_delay_cycles(1000);     // Creates a delay of 1000 CPU cycles
```

Implement the second and third solutions by modifying one of the earlier codes. Note that the _delay_cycles() function counts CPU cycles and needs a larger number than a delay loop upperbound to accomplish a comparable delay. In each solution, try setting a delay of 3 seconds for the flashing; such a delay can't be accomplished by a simple delay loop with a 16-bit counter.

Deliverables:

- Demo your codes to the lab instructor
- Submit your code with the second and third solutions

1.5 Designing Firefighter Truck LED Patterns

You're working as an embedded engineer and your employer was contracted by the firefighter department to write software that produces LED flashing patterns that will be used on fire trucks. The flashing patterns are shown in the videos below.

Pattern #1: https://youtube.com/shorts/SlZDGJTgVAo

Pattern #2: https://youtube.com/shorts/SqBy7YAlmHA

Write two codes that produce the patterns shown. You have flexibility in the fine tuning of the patterns but they should largely resemble what is in the videos.

Deliverables:

- Demo your codes to the lab instructor
- Submit your codes

Energy Trace Feature

This part is for your own knowledge and no submission is needed for it. Our LaunchPad board implements the EnergyTrace feature which is capable of monitoring the current and power drawn from the chip in real-time. To generate a real-time power graph, build your code and click on the following menu option before starting the code:

```
Tools -> EnergyTrace
```

In the EnergyTrace window, click on the Power or Energy tab and start the code. You should get a real-time graph displaying the power drawn and energy consumption of the code.

Student Q & A

Submit the answers to the questions in your report.

1. In this lab, we used a delay loop to create a small delay; what is its effect on the battery life if the device is battery operated? Is it a good way of generating delays?
2. The MSP430 CPU runs on a clock that can be slowed down or sped up; what happens to the delay generated by the delay loop if the clock driving the CPU speeds up or slows down?
3. How does the code run in the debug mode? Is the microcontroller running as an independent computer?
4. How does the code run in the normal mode? Is the microcontroller running as an independent computer?
5. In which mode does the reset button work?
6. What is the data type uint16_t ? What about int16_t ? Are these standard C syntax?