UNIVERSITY OF CENTRAL FLORIDA

Lab Manual for EEL 4742C Embedded Systems

Department of Electrical and Computer Engineering

Rev. November 2018

Contents

Preface

This lab manual was developed at UCF for the course of EEL 4742C (Embedded Systems). The teaching goal of this lab is to train the students in low-power microcontroller applications, to demonstrate the use of industry-class hardware and to write embedded software based on the recommended practices.

If you have feedback about this manual or if you believe that you found a mistake, please send your comments to Dr. Zakhia (Zak) Abichar at: zakhia17@ece.ucf.edu

Dr. Abichar
November 2018
UCF

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 we write 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 choice of 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 262K colors, 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
- Using advanced timer features (multiple channels, timer-driven input/output)
- Providing concurrency via interrupts
- Performing communication with the UART, I2C and SPI protocols
- Using the Analog-to-Digital converter
- Programming pixel-based LCD graphics

Lab Policy

- Lab attendance is required; up to two absences are allowed.
- 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.
- Having two or more reports not submitted may result in a failing grade in the class.
- The lab report is graded based on correctness, coding style and writing quality.
- In the summer session, the labs titled “LCD Display” and “Concurrency via Interrupts” are omitted.

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 simple programs that flash the LEDs.

1.1 Documentation

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

- MSP430FR6xx Family User's Guide `(slau367o)`
- FR6xx 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 a bit similar to a book. 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 diverted 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, which is needed to write the code. This file also contains the schematic of the LaunchPad board.

The documentation for the Booster Pack is the following:

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

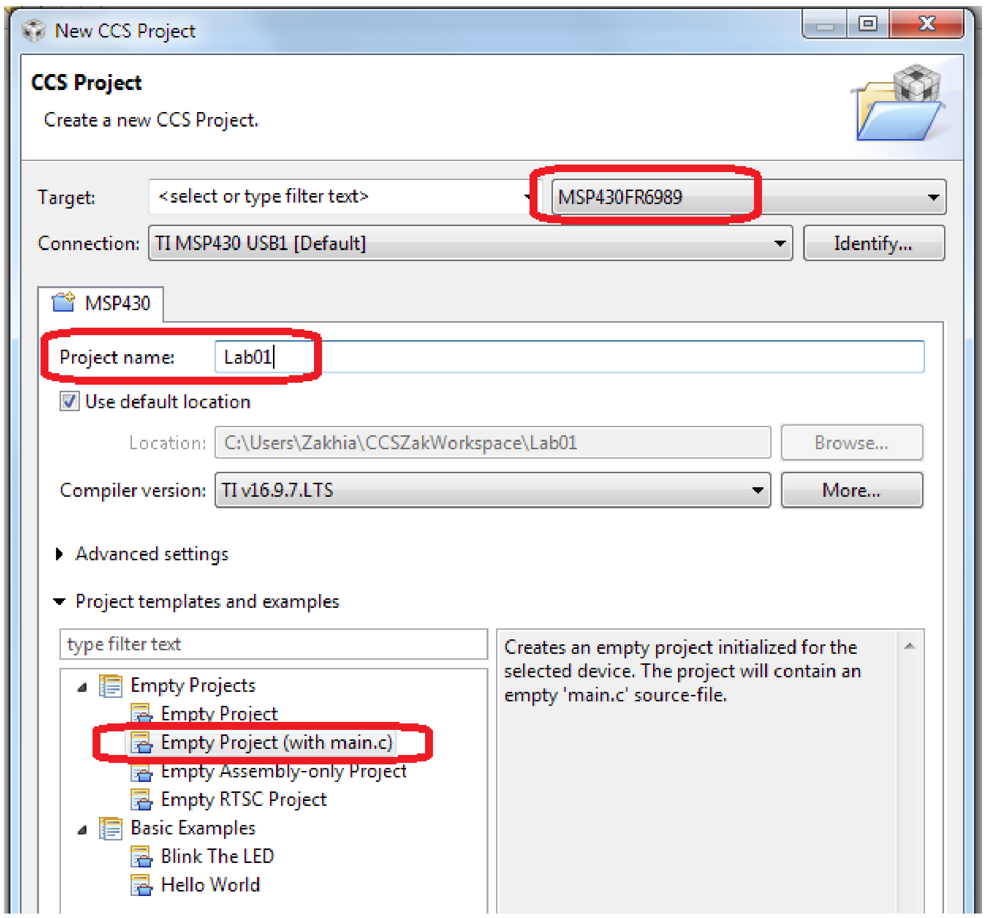
The first file shows the components that are connected to the Booster Pack such as the light and temperature sensors, the buzzer, the display, etc. This file has general information on how the components are connected together, but not on the inner working of these components. For each of the components, we would have to search for their specific data sheet from the respective manufacturer to see how they operate.

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.


New CCS Project

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

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.

The red LED is mapped to Port 1 Bit 0, in short P1.0. The header file (.h) included in the code defines a bunch of 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 the delay loop doesn't do useful computation and eliminate it altogether.

[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.

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.

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

The following line disables the WatchDog Timer (WDT) mechanism. This mechanism causes periodic resets unless it's periodically cleared explicitly by the code. 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.

```c
// 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 direct the pin as output and turn the LED off. Below, we can see the variables that control a port. Port 1 is eight bits. Each bit can be used as input or output. The second variable P1DIR (direction), directs each pin as either output (1) or input (0). For all the pins that are directed as output, the variable P1OUT is used to write to them (0 or 1). For all the pins that are directed as input, the variable P1IN is used to read from them (0 or 1).

```
Port 1:  _ _ _ _  _ _ _ _
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 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.

When we change a bit to a specific value, it is necessary to leave the other bits in the variable unchanged. 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. Bit 0 becomes 0 while all the other bits remain unchanged.

In your codes throughout all the experiments of this lab, you 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 en empty body. 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 allows programming the chip and enables debugging via CCS.

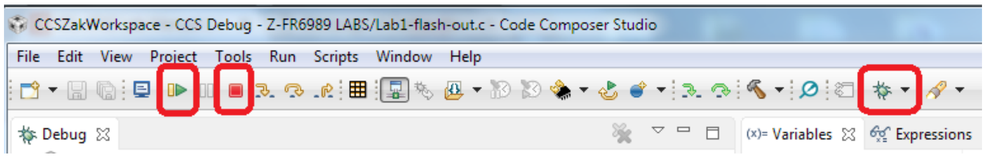
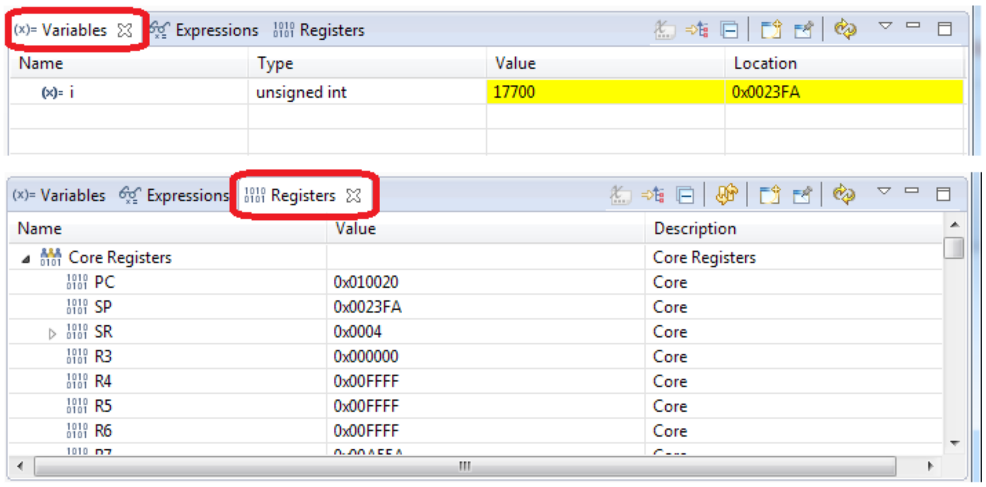

CCSZakWorkspace - CCS Debug - Z-FR6989 LABS/Lab1-flash-out.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
(x)= Variables
Expressions

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
(x)= 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 code resumes running. 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:

- Change the upperbound value of 'i' to make the LED flash slower or faster
- Run in debug mode; pause the code and check the variables and registers values
- Run in debug mode and test the reset button (the third button on the board); does it work?
- Run 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:

- Demo your code to the lab instructor.
- In your report, submit a snapshot of the registers and variables values obtained from the debug mode.

1.3 Setting a Long Period

In the previous code, the counter 'i' went from 0 to 20,000 to create a delay. To set a larger delay, we can incrase the value of 'i'. MSP430 is a 16-bit architecture. The registers and the ALU are 16-bit. An integer (type int) created in our C program is treated as a 16-bit variable by the compiler. An unsigned 16-bit variable can go from 0 up to $2^{16} - 1$, which is the range [0 - 65,535]. Accordingly, we can't raise 'i' to higher than 64K.

One interesting detail here is, on a desktop, 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 platforms upgrade it to 32-bit while the MSP430 environment keeps it as 16-bit.

Change the value of 'i' to 80,000 and run the code. Most likely the program will misbehave since the 16-bit variable can't go up to this value.

Modify the code by writing nested delay loops or back-to-back delay loops to obtain a large delay. This is a viable solution to obtain a large delay.

Even though MSP430 is a 16-bit architecture, it's possible to define 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 use this feature, 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.

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

Now, test the code by setting the upperbound of the delay loop to 120,000. This should work.

Deliverables:

- Demo your codes to the lab instructor.
- Submit your code with nested delay loop or back-to-back delay loop; show how you can obtain a large delay
- Submit your code using uint32_t data type; show how you can obtain a large delay

1.4 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 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 LaunchPad user's guide. Open this file `(slau267a)` on page 31. Find out where LED2 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 out of sync; when the green is on, the red is off and vice versa.

Perform the following actions:

- Start by defining a mask for the green LED: `#define greenLED BITx`
- Set the green LED pin to output by applying an OR operation at PxDIR
- Initialize the green LED to on (so that it's opposite of the red) by using an OR operation on PxOUT
- Toggle both LEDs inside the infinite loop by using XOR operations

In the above, it's up to you to determine the value of x in BITx, PxDIR, PxOUT.

Deliverables:

- Demo your code to the lab instructor
- Include a screenshot of the board user's guide schematic (p.31) that shows the LED connections
- Submit your code that flashes both LEDs out of sync
- This code can be easily modified to flash the LEDs in sync; insert one line in the code (comment it out) that shows how this can be done

1.5 Flashing Two LEDs at Different Rates

Write a code that flashes the red LED at twice the frequency of the green LED.

Deliverables:

- Demo your code to the lab instructor
- Submit your 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?

Lab 2

Using the Push Buttons

In this lab, we will learn using the push buttons of the LaunchPad board via a simple technique called polling.

2.1 Reading the Push Buttons

The LaunchPad board is equipped with three push buttons, listed below:

```
S1: Button connected to Port 1.1
S2: Button connected to Port 1.2
S3: Button connected to the reset (RST) pin
```

The buttons S1 and S2 are used for general-purpose and we'll use them in this lab. The button S3 is connected to the reset pin and causes a reset when pushed. The reset pin can be re-purposed as an interrupt pin and, therefore, can be used as a general-purpose pin to some extent.

The buttons S1 and S2 are connected in the active low configuration. That is, when they're pushed, they read as zero. Confirm this by looking at the schematic in the LaunchPad User's Guide `(slau627a)` on page 31 and include a snapshot of the button schematics in your report.

Port Configuration

The I/O ports are configured using the port configuration registers. These are presented in the FR6xx Family User's Guide `(slau367o)` in the chapter "Digital I/O" on page 363. We summarize the information that relates to this lab below in Table 2.1

Table 2.1: Configuration Registers for Port 1

P1DIR	Pin direction (0: input) (1: output)
P1IN	Reading input pin (0: input is low) (1: input is high)
P1REN	Built-in resistor enable (can be set as pull-up or pull-down) (0: disable resistor) (1: enable resistor)
P1OUT	Write to output (when direction is output) (0: write low) (1: write high) Configure resistor (when direction is input) (0: pull-down) (1: pull-up)

A port (e.g: Port 1) is 8-bit and, accordingly, each of the variables shown in the table is 8-bit, with a one-to-one correspondence to the pins. We have already used P1DIR, P1IN and P1OUT in the previous lab.

The configuration variable P1REN allows enabling a built-in resistor (inside the chip) for each I/O pin. The built-in resistor is optionally used only when the I/O pin is configured as input. We use P1REN to enable/disable the internal resistor. Furthermore, the resistor can be configured as pull-up (to Vcc) or pull-down (to ground). This is done via P1OUT. Note that when the I/O pin is configured as input, P1OUT is not used for data. Therefore, it's used to configure the resistor as pull-up or pull-down.

In this lab, the I/O pins where the buttons are connected will be configured as input, the built-in resistors will be enabled, and they will be set to pull-up. We can justify this by looking at the buttons' schematics. They're connected to ground when they're pushed. Therefore, we'll have them pulled-up to Vcc so they read high when they're released.

Masking Operation

To read the button's status, we need to inspect the button's corresponding bit inside P1IN. Below are the two masking AND operations that allow reading a bit inside a variable. In this example, we're reading BIT3 in the data.

```
  Data: ---- 0---                        Data: ---- 1---
  BIT3: 0000 1000   AND                  BIT3: 0000 1000   AND
Result: 0000 0000                      Result: 0000 1000
```

On the left side, the bit is 0 and the result of the AND operation is zero. On the right side, the bit is 1 and the result of the AND operations is nonzero. **Please note that the result is not equal to 1, since the bit 1 is not on the rightmost position.** For the case on the right side, we can check if the result is nonzero, or if the result is equal to BIT3 Accordingly, the piece of code below shows how we check the bit's value.

```
// Check if bit is zero
if( (Data & BIT3) == 0 ) ...

// Check if bit is one (two ways)
if( (Data & BIT3) != 0 ) ...
if( (Data & BIT3) == BIT3 ) ...
```

Turning on the LED with the Button

Let's write a code that turns on the red LED when button S1 is pushed. As long as the button is held down, the red LED should remain on. When the button is released, the red LED should turn off. Below is the code, fill the missing parts. Please remember that using hexadecimal masks is not allowed throughout all the experiments of this lab (we'll use the masks BIT0, etc or redefined names e.g. redLED).

```
// Turning on the red LED while button S1 is pushed

#include <msp430fr6989.h>
#define redLED BIT0              // Red LED at P1.0
#define greenLED BIT7            // Green LED at P9.7
#define BUT1 BIT1                // Button S1 at P1.1
```

```
void main(void) {
  WDTCTL = WDTPW | WDTHOLD;      // Stop the Watchdog timer
  PM5CTL0 &= ~LOCKLPM5;          // Enable the GPIO pins

  // Configure and initialize LEDs
  P1DIR |= redLED;               // Direct pin as output
  P9DIR |= greenLED;             // Direct pin as output
  P1OUT &= ~redLED;              // Turn LED Off
  P9OUT &= ~greenLED;            // Turn LED Off

  // Configure buttons
  P1DIR &= ~BUT1;                // Direct pin as input
  P1REN ...                      // Enable built-in resistor
  P1OUT ...                      // Set resistor as pull-up

  // Polling the button in an infinite loop
  for(;;) {
      // Fill the if-statement below...
      if ( ... button S1 is pushed ... )
          P1OUT |= redLED;      // Turn red LED on
      else P1OUT &= ~redLED;    // Turn red LED off
  }

}
```

This code polls the button infinitely and, therefore, keeps the CPU locked to this operation. A downside is the CPU can't do another task. Secondly, if the device is battery-operated, polling the button infinitely could drain the battery. A more advanced approach is to setup the button via interrupt and to put the microcontroller in low-power mode while waiting for the button push. We'll explore this approach in a later lab.

For this part, perform the following and include the answers in your report.

- Include snapshot of the buttons schematic from the LaunchPad User's Guide. Explain why the buttons are active low and why the pull-up resistor is used.
- Write the missing parts of the code and demo it to the TA.
- Submit the code in your report.

2.2 Using Two Push Buttons

Modify the code from the previous part to control two LEDs using the two push buttons. When button S1 is pushed, the red LED should turn on. When button S2 is pushed, the green LED should turn on. If both buttons are pushed simultaneously, both LEDs should turn on.

For this part, perform the following:

- Write the code and demo it to the TA.
- Test the case when both buttons are pushed.
- Submit the code in your report.

2.3 Using Two Buttons with Exclusive Access

Modify the code from the previous part so that the red LED is on while S1 is pushed and the green LED is on while S2 is pushed. However, the two LEDs should not be lit simultaneously. Imagine that they correspond to physical phenomena and a hazard occurs if both LEDs are lit at the same time. Your software should ensure this doesn't happen.

Accordingly, the button that is pushed first has precedence. Let's say S1 is held down and the red LED is on. If, in the meanwhile, S2 is pushed, the green LED remains off and the red LED continues to be on. This state persists until S1 is released. Now, S2 has the chance to turn on the green LED. And, likewise, if S2 is held down with the green LED on, S1 is ignored when pushed, until S2 is released.

For this part, perform the following:

- Write the code and demo it to the TA.
- Stress test the code; what happens if both buttons are pushed simultaneously?
- Submit the code in your report.

2.4 Your Own Design

In this part, combine elements from the previous parts and design an experiment that demonstrate the skills that you learned.

Perform the following:

- Demo your code to the TA.
- In your report, describe the experiment and submit your code.

Student Q&A

Submit the answers to the questions in your report.

1. When a pin is configured as input, P1IN is used for data, which leaves P1OUT available for other use? In such case, what is P1OUT used for?
2. A programmer wrote this line of code to check if bit 3 is equal to 1: `if((Data & BIT3)==1)`. Explain why this if-statement is incorrect.
3. Comment on the codes' power-efficiency if the device is battery operated. Is reading the button via polling power efficient?

Lab 3

Using the Timer

In this lab, we will program the microcontroller's timer module (Timer_A) to generate timing events. We will program two modes known as the continuous mode and the up mode. For both modes, we will use the polling technique, which is a simple approach based on reading a flag continuously.

3.1 The Continuous Mode

The MSP430FR6989 chip contains a timer module known as Timer_A. This module is versatile and has many features. In this lab, we will use the module to time durations.

The timer uses a clock signal of a known frequency as input and counts cycles to measure durations. Timer_A has a 16-bit register called TAR (Timer_A Register). This register counts up by one at each cycle of the clock signal. Since TAR is 16-bit, it can count up to 65,535 and usually rolls back to 0 and continues counting.

In the continuous mode, TAR counts from 0 to the largest value of 65,535, then rolls back to zero and continues counting, as shown below.

```
TAR:    0, 1, 2, ..., 65535, 0, 1, 2, ..., 65535, 0, 1, ...
                             ↑                    ↑
                         TAIFG set            TAIFG set
```

Each time TAR rolls back to zero **while counting**, the 1-bit flag TAIFG (Timer_A Interrupt Flag) is set automatically by the hardware, as shown above. If TAR is set to zero through a line of code (i.e. TAR did not roll back to zero while counting), TAIFG is not set. When the flag is set, we take the necessary action (e.g: toggle an LED) then clear the flag so it can be raised again by the hardware when the timer period elapses the next time.

Timer_A Configuration

Timer_A is started and configured using the TACTL (Timer_A Control) register, which format is shown in Table 3.1.

The field TASSEL is used to select the clock signal used by the timer. ACLK (Auxiliary Clock) is typically configured to a 32 KHz crystal that is soldered on the board. SMCLK (Sub Master Clock) is generated from an oscillator inside the chip and is set to 1.048576 MHz by default (this value is 2^{20}). Its frequency can be modified via software. Once a clock signal is selected, whether it's ACLK or SMCLK, it can be divided within the timer module by either 1, 2, 4 or 8, to slow down the frequency. The frequency division is done using the ID field.

The MC (Mode) field is used to select the mode. By default, it's 0 and the timer is stopped. We'll change it to 2 in this section to run the timer in the continuous mode.

The bit TACLR (Clear) is used to force TAR to go to zero. If TAR is counting and TACLR is asserted, TAR goes to zero immediately and resumes counting. We usually assert this bit at the start to ensure TAR starts at zero.

Finally, the bit TAIFG is raised to 1 by the hardware when TAR rolls back to zero. We'll monitor this bit to know when the timer period has elapsed.

Masking Operations

To simplify accessing the bit fields inside TACTL, the header file defines a bunch of masks that make our job easier. The mask TASSEL_1 has value of 1 at the position of TASSEL. Similarly, the mask TASSEL_2 has a value of 2 at the position of TASSEL. The masks ID_0, ID_1,

Table 3.1: Timer_A Control Register (TACTL)

TASSEL	Timer_A Source Select (selects the clock signal) (1: ACLK) (2: SMCLK)
ID	Input Divider (divides the input clock frequency inside the timer) (0: Div by 1) (1: Div by 2) (2: Div by 4) (3: Div by 8)
MC	Mode (0: Stop) (1: Up Mode) (2: Continuous Mode)
TACLR	Timer_A Clear (sets TAR to 0 when asserted)
TAIFG	Timer_A Interrupt Flag Raised when TAR rolls back to zero while counting

ID_2, ID_3 have the values 0, 1, 2, 3, respectively, at the position of ID. Similarly, there are two masks MC_1 and MC_2. Finally, there is mask TACLR that has 1 at the position of TACLR and a mask TAIFG that has 1 at the position of TAIFG.

As an example, to configure Timer_A to use SMCLK, divided by 4, continuous mode, and TAR starting at zero, we write the following line of code:

```
TACTL = TASSEL_2 | ID_2 | MC_2 | TACLR;
```

The line of code above writes 0 to TAIFG since the mask TAIFG is not selected. However, to be sure TAIFG is 0, we can clear it again using the line of code below.

```
TACTL &= ~TAIFG;    // AND with inverse of mask to clear the bit
```

In the two lines of code above, we used the variable name TACTL. The microconroller chip contains multiple independent Timer_A modules, called Timer0_A, Timer1_A, etc. We'll program the Timer0_A. Therefore, the variable name we'll use in the code it TA0CTL.

Configuring ACLK to the 32 KHz Crystal

In our code, we want to use the ACLK clock based on the 32 KHz crystal. By default, ACLK is configured to a built-in oscillator at a frequency of 5 MHz / 128 = 39 KHz. We will reroute ACLK to the 32 KHz crystal that's soldered on the LaunchPad. Looking at the LaunchPad user's guide (page 29), the schematic shows that the 32 KHz crystal is attached to pins that have dual functionality: LFXIN/PJ.4 and LFXOUT/PJ.5. The functionality we're looking for is LFXIN (Low-frequency crystal in) and LFXOUT (Low-frequency crystal out).

Looking at the chip's data sheet (page 123), we can find the settings to route the pins to the LFXIN/LFXOUT functionality. All that is required is setting PJSEL1 (Bit 4) to 0 and PJSEL0 (Bit 4) to 1.

Furthermore, when the crystal is started, we need to wait for it to settle. We'll do this using the local and global oscillator fault flags. When these flags are cleared and remain cleared, it means the crystal clock is ready for use. The whole configuration is shown in the function `config_ACLK_to_32KHz_crystal()` below. The function first diverts the pins for the crystal functionality then waits on the fault flags to remain cleared. We will use this function in future labs anytime we want to configure ACLK to the 32 KHz crystal.

```
//**********************************
// Configures ACLK to 32 KHz crystal
void config_ACLK_to_32KHz_crystal() {
    // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz

    // Reroute pins to LFXIN/LFXOUT functionality
    PJSEL1 &= ~BIT4;
    PJSEL0 |= BIT4;

    // Wait until the oscillator fault flags remain cleared
    CSCTL0 = CSKEY;            // Unlock CS registers
    do {
      CSCTL5 &= ~LFXTOFFG;     // Local fault flag
      SFRIFG1 &= ~OFIFG;       // Global fault flag
    } while((CSCTL5 & LFXTOFFG) != 0);

    CSCTL0_H = 0;              // Lock CS registers
    return;
}
```

Flashing the LED

Let's write a code that flashes the red LED based on a delay that's generated by the timer. Use the ACLK clock signal (configured to the 32 KHz crystal), use ACLK as is (do not divide it), run the timer in the continuous mode and clear TAR at the start. Below is the code. Fill the remaining parts.

```
// Flashing the LED with Timer_A, continuous mode, via polling
#include <msp430fr6989.h>
#define redLED BIT0               // Red LED at P1.0
#define greenLED BIT7             // Green LED at P9.7

void main(void) {
  WDTCTL = WDTPW | WDTHOLD;       // Stop the Watchdog timer
  PM5CTL0 &= ~LOCKLPM5;           // Enable the GPIO pins

  P1DIR |= redLED;                // Direct pin as output
  P9DIR |= greenLED;              // Direct pin as output

  P1OUT &= ~redLED;               // Turn LED Off
  P9OUT &= ~greenLED;             // Turn LED Off

  // Configure ACLK to the 32 KHz crystal (function call)
  ...

  // Configure Timer_A
  // Use ACLK, divide by 1, continuous mode, clear TAR
  TA0CTL = ...;

  // Ensure flag is cleared at the start
  TA0CTL &= ~TAIFG;

  // Infinite loop
  for(;;) {
      // Empty while loop; waits here until TAIFG is raised
      while( ... ) {}

      ...                       // Toggle the red LED
      ...                       // Clear the flag
  }

}
```

Perform the following and include the answers in your report.

- Complete the code and demo it to the TA.
- Write an analysis showing what delay you expect.
- Measure the observed delay with your phone's stopwatch for at least 20 seconds. The timing should match closely since the crystal is accurate. Does it match?
- Try dividing the clock by 2, 4 or 8 using the ID field. What delays do these correspond to? Do they match what you observe?

3.2 The Up Mode

In the up mode, we set the upperbound of TAR to any value that we choose. TAR counts from zero up to the value of register TACCR0. This is the register of Channel 0. The up mode is inherently linked to Channel 0. The timeline below shows how the timer runs. When TAR reaches TACCR0, it rolls back to zero and continues counting. Also, when TAR rolls back to zero (while counting), the TAIFG flag is set by the hardware.

```
TAR:    0, 1, 2, ..., TACCR0, 0, 1, 2, ..., TACCR0, 0, 1, ...
                              ↑                     ↑
                          TAIFG set             TAIFG set
```

The register TACCR0 is 16-bit. To have a period of 100 cycles, we set TACCR0=100-1=99. TAR then counts between 0 and 99, spending one cycle at each count.

Since our microcontroller chip has multiple Timer_A modules (called Timer0_A, Timer1_A, etc), we'll use the term TA0CCR0 in the code to indicate that we're using Timer0_A Channel 0 register.

The code for the up mode is very similar to the continuous mode. Below are the differences. First, we need to set TAR's upperbound using TA0CCR0. Set it so the timer's period is 1 second (flashing period is 2 seconds). We'll use ACLK (configured to the 32 KHz crystal) and divide it by 1.

```
// Flashing the red LED with Timer_A, up mode, via polling

// Set timer period
TA0CCR0 = ...

// Timer_A: ACLK, div by 1, up mode, clear TAR
TA0CTL = ...
```

Secondly, configure the timer using the TA0CTL register and select the up mode. The parts of the code not shown are similar to the code of the continuous mode.

Perform the following and include the answers in your report.

- Write the code and demo it to the TA.
- What value of TA0CCR0 did you use?
- Were you able to achieve a precise 1-second timer period? Compare it to your phone's stopwatch for at least 20 seconds, it should match closely.
- What value of TA0CCR0 achieves a delay of 0.1 seconds? Round up to the nearest integer and test this value.
- What value of TA0CCR0 achieves a delay of 0.01 seconds? Round up to the nearest integer and test this value. What do you observe?

3.3 Your Own Design

In this part, design two experiments similar to the above. One uses the continuous mode and another uses the up mode. You can add extra features to make the case more interesting. You can use the green and red LEDs.

Perform the following:

- Explain your design
- Demo your programs and submit your C codes

Student Q&A

Submit the answers to the questions in your report.

1. So far, we have seen two ways of generating delays: using a delay loop and using Timer_A. Which approach provides more control and accuracy over the delays? Explain.

2. Explain the polling technique and how it's used in this lab.

3. Is the polling technique a suitable choice when we care about saving battery power? Explain.

4. If we write 0 to TAR using a line code, does the TAIFG go to 1?

5. In this lab, we used TAIFG to time the duration. TAIFG is known as the Timer_A Interrupt Flag, which is an interrupt flag. Were we using interrupts in this lab? Explain.

6. From what we have seen in this lab, which mode gives us more control over the timing duration: the up mode or the continuous mode?

Lab 4

Interrupts & Low-Power Modes

In this lab, we will learn programming interrupts and using the low-power modes. We will program Timer_A and the push buttons with interrupts. We will engage the low-power modes so the microcontroller reduces its power consumption while waiting for the interrupt events to occur.

4.1 Timer's Continuous Mode with Interrupt

In this part, we will program Timer_A in the continuous mode to generate timing events. We'll set up an interrupt event which means we don't have to poll the timer's flag continuously. Instead, when the timer period elapses, the timer raises an interrupt and the hardware invokes a special piece of code to respond.

Interrupt Basics

Let's start by reviewing the basics of interrupts in the MSP430 architecture.

Global Interrupt Enable (GIE): The GIE bit is the master on/off switch for all the interrupts in the chip. It is located in the Status Register (SR), which is register R2. In the C code, the GIE bit is set or cleared using the intrinsic functions: `_enable_interrupts()` and `_disable_interrupts()`.

Interrupt Enable bit (xIE): Each interrupt event is enabled/disabled individually using its own interrupt enable bit. The term 'x' in 'xIE' designates the event, such as: TAIE (Timer_A Interrupt Enable bit), P1IE (Port 1 Interrupt Enable bits), etc. For an interrupt event to be enabled, the global interrupt bit (GIE) should be 1 and the event's enable bit should be 1.

One interrupt event in Timer_A is the rollback-to-zero event. When TAR rolls back to zero while counting, the TAIFG flag is raised. This flag raises an interrupt when its enable bit, TAIE, is set to 1. We'll run the timer in the continuous mode and setup this interrupt event.

Interrupt Flag bit (xIFG): This bit indicates that the interrupt event has actually occurred. It is raised by the hardware and throws an interrupt only when the xIE bit (and global bit GIE) is enabled.

For example, when TAR rolls back to zero while counting, the hardware raises TAIFG, as we saw in earlier labs. In the earlier labs, this did not raise an interrupt since we kept the enable bit (TAIE) equal to 0. However, in this lab, we'll set TAIE to 1 so that an interrupt is raised when TAR rolls back to zero.

Interrupt Service Routine (ISR): When an interrupt occurs, the responding code is in a special function, the ISR, rather than in the main code. Usually each interrupt event has a corresponding ISR function. However, multiple interrupt events sometimes share the same responding ISR function. When an interrupt occurs, the hardware is responsible for finding and launching the proper ISR. The ISRs are functions that are scattered around the memory. How does the hardware find the ISRs?

Vector Table: A vector is the start address of (pointer to) an ISR. The vector table contains the addresses of all the ISRs. The vector table's format and location are well-known to the hardware. Therefore, the hardware performs a lookup in the vector table to find the ISRs in the memory. The programmer is responsible for filling (linking) the vectors so the vector table contains the addresses of all the ISRs that are used by the code.

Table 4.1 summarizes the items that are used to configure the interrupt for the timer's rollback-to-zero event. First, we enable the event's interrupt bit (TAIE=1). Then, we ensure the interrupt flag is cleared initially (TAIFG=0). Then, we write the ISR and link its start address to the corresponding vector in the vector table. It turns out that the rollback-to-zero event is associated with the timer's A1 vector. We should also ensure that the ISR clears the flag (TAIFG) when an interrupt event is processed. Finally, we enable the global interrupt bit (GIE=1).

Table 4.1: Timer_A Rollback-to-Zero Interrupt Event

Event	**Enable Bit**	**Interrupt Flag**	**Vector**
TAR rollback-to-zero	TAIE	TAIFG	Timer's A1 vector

Code that Flashes the LEDs

Below, we'll write the code that runs Timer_A in the continuous mode using ACLK (based on the 32 KHz crystal) and flashes the LEDs when TAR rolls back to zero. Every time TAR rolls back to zero, an interrupt is raised and the responding ISR toggles the LEDs. To configure ACLK based on the 32 KHz crystal, use the function `config_ACLK_to_32KHz_crystal()` that we saw in an earlier lab.

As we saw in earlier labs, the microcontroller has multiple Timer_A modules and we'll use the timer module #0, called Timer0_A. Accordingly, the A1 vector is called **TIMER0_A1_VECTOR.** This is where we'll link the ISR.

As we also saw in earlier labs, the timer is configured using the variable TA0CTL. It contains the bit fields TASSEL (clock select), ID (clock divider), MC (mode), the interrupt flag (TAIFG) and the interrupt enable bit (TAIE).

```
// Timer_A continuous mode, with interrupt, flashes LEDs
#include <msp430fr6989.h>
#define redLED BIT0                // Red LED at P1.0
#define greenLED BIT7              // Green LED at P9.7

void main(void) {
  WDTCTL = WDTPW | WDTHOLD;        // Stop the Watchdog timer
  PM5CTL0 &= ~LOCKLPM5;            // Enable the GPIO pins

  P1DIR |= redLED;                 // Direct pin as output
  P9DIR |= greenLED;               // Direct pin as output
  P1OUT &= ~redLED;                // Turn LED Off
  P9OUT &= ~greenLED;              // Turn LED Off
  // Configure ACLK to the 32 KHz crystal
  config_ACLK_to_32KHz_crystal();

  // Timer_A configuration (fill the line below)
  // Use ACLK, divide by 1, continuous mode, TAR cleared, enable
     interrupt for rollback-to-zero event
  TA0CTL = ...

  // Ensure the flag is cleared at the start
  TA0CTL &= ~TAIFG;

  // Enable the global interrupt bit (call an intrinsic function)
  ...

  // Infinite loop... the code waits here between interrupts
  for(;;) {}
}

//******* Writing the ISR *******
#pragma vector = TIMER0_A1_VECTOR     // Link the ISR to the vector
__interrupt void T0A1_ISR() {
    // Toggle both LEDs
    ...
    // Clear the TAIFG flag
    ...
}
```

Complete the missing parts of the code. First, configure the timer similar to how we did in earlier labs. The only difference here is we need to enable the rollback-to-zero interrupt by setting the TAIE bit. Next, call the intrinsic function that sets the GIE bit.

The ISR is linked to the A1 vector using the pragma line of code. The ISR is marked with the keyword '__interrupt' to indicate so to the compiler. The ISR is now linked to the appropriate vector. Each time an interrupt occurs, the hardware calls this function. Fill the action by toggling the LEDs and clearing the flag (TAIFG).

The vector's name, TIMER0_A1_VECTOR can be looked up in the header (.h) file included at the top of the code. It should be spelled exactly as written in the header file. The function's name, T0A1_ISR, however, is not a keyword and can be replaced with any other name.

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Compute the delay that the timer should generate.
- Compare the timing to your phone's stopwatch for at least 20 seconds and ensure it matches closely (since the crystal is accurate).
- What happens if we don't clear the flag in the ISR? Explain.
- What is the CPU doing between interrupts?
- Who is calling the ISR? Is it the software? Explain.
- Submit the code in your report.

4.2 Timer's Up Mode with Interrupt

In this part, we'll write a code that runs the timer in the up mode and raises interrupts periodically. Before we get in the details, let's look at Table 4.2 which summarizes multiple interrupt events of Timer_A.

Table 4.2: Multiple Interrupt Events of Timer_A

Event	**Bits**	**Vector**	**MSP430 Policy**
Rollback-to-zero Channel 1 Channel 2	TAIE / TAIFG in TACTL CCIE / CCIFG in TACCTL1 CCIE / CCIFG in TACCTL2	A1	Programmer clears the flag (shared vector)
Channel 0	CCIE / CCIFG in TACCTL0	A0	Hardware clears the flag (non-shared vector)

First, we can see in the table the interrupt event we used in the last part (rollback-to-zero). The table shows the enable and flag bits (TAIE/TAIFG) which are located in TACTL, specifically TA0CTL since we're using the Timer_A module #0. Finally, the table shows the corresponding vector, A1.

Moreover, the table shows that the timer has three Channels (0, 1, 2) each of which can raise an interrupt. Their enable and flag bits are called CCIE/CCIFG but located in three different registers. For Timer0_A, these are TA0CCTL0, TA0CCTL1 and TA0CCTL2.

The Channel 0 interrupt event occurs when TAR=TACCR0. Let's recall how the up mode works by looking at the timeline below. TAR counts from 0 up to the value in TACCR0, then rolls back to zero and continues counting.

```
TAR:    0, 1, 2, ..., TACCR0, 0, 1, 2, ..., TACCR0, 0, 1, ...
                         ↑     ↑                   ↑     ↑
                      CCIFG  TAIFG              CCIFG  TAIFG
```

When TAR rolls back to zero, we know that TAIFG is raised and this raises an interrupt when enabled. Similarly, when TAR=TACCR0, the flag CCIFG is raised (as marked in the timeline) and raises an interrupt when enabled. In this part, we'll use the Channel 0 interrupt event (CCIFG event).

This is our plan. We'll set the Channel 0 enable bit (CCIE=1). Then, we'll ensure the flag is cleared initially (CCIFG=0). Then, we'll write the ISR and link its start address to the timer'a A0 vector in the vector table. Finally, we'll enable the global interrupt bit (GIE=1). Note that the bits CCIE and CCIFG are located in the register TA0CCTL0.

The last relevant piece of detail is shown in the rightmost column of the table. The rollback-to-zero event and Channels 1, 2 share the same vector and thus have the same ISR. In such case, the programmer is responsible for clearing the interrupt flag, as we did in the previous part of this lab. However, Channel 0 has its own non-shared vector, A0, and therefore, has its own non-shared ISR. In such case, the hardware clears the interrupt when the ISR is called; we don't have to write code to clear it.

Below is the code. Complete the missing parts.

```
// Timer_A up mode, with interrupt, flashes LEDs
#include <msp430fr6989.h>
#define redLED BIT0                // Red LED at P1.0
#define greenLED BIT7              // Green LED at P9.7

void main(void) {
  WDTCTL = WDTPW | WDTHOLD;        // Stop the Watchdog timer
  PM5CTL0 &= ~LOCKLPM5;            // Enable the GPIO pins

  P1DIR |= redLED;                 // Direct pin as output
  P9DIR |= greenLED;               // Direct pin as output
  P1OUT &= ~redLED;                // Turn LED Off
  P9OUT |= greenLED;               // Turn LED On (alternate flashing)
  // Configure ACLK to the 32 KHz crystal
  config_ACLK_to_32KHz_crystal();

  // Configure Channel 0 for up mode with interrupt
  TA0CCR0 = ...                  // Fill to get 1 second @ 32 KHz
  TA0CCTL0 ...                   // Enable Channel 0 CCIE bit
  TA0CCTL0 ...                   // Clear Channel 0 CCIFG bit

  // Timer_A: ACLK, div by 1, up mode, clear TAR (leaves TAIE=0)
  TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;

  // Enable the global interrupt bit (call an intrinsic function)
  ...

  for(;;) {}
}

//*******************************
#pragma vector = TIMER0_A0_VECTOR
__interrupt void T0A0_ISR() {
    // Toggle the LEDs
    ...
    // Hardware clears the flag (CCIFG in TA0CCTL0)
}
```

First, find the upperbound of TAR so the timer's period is one second. Remember that the clock frequency is 32,768 Hz. Then, enable the interrupt bit of Channel 0 and clear its interrupt flag. Then, called the intrinsic function that sets the GIE bit. Note how the ISR is linked to the A0 vector of Timer0_A. Finally, toggle the LEDs inside the ISR. There is no need to clear the flag as the hardware will do it.

Note in the line of code that starts the timer, we left TAIE=0 since the rollback-to-zero event is not generating interrupts. The interrupt is generated by Channel 0 (when TAR=TA0CCR0).

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Compare the timing to your phone's stopwatch for at least 20 seconds and make sure the timing matches closely (since the crystal is accurate).
- Modify the code so that the delay is 0.5 seconds (then try 0.1 seconds). Ensure the code

works for both cases.

4.3 Push Button with Interrupt

In this part, we will read the push buttons via interrupts. When button S1 is pushed, the red LED is toggled and when button S2 is pushed, the green LED is toggled.

As we have seen in earlier labs, the push buttons of the LaunchPad are wired in the active low configuration (they read 0 when pushed). When the button is pushed, there is a falling edge that is followed by a rising edge. The microcontroller can raise an interrupt on either edge. We'll setup the falling edge interrupt so the interrupt occurs as soon as the button is pushed (rather than when it's released).

The piece of code below shows how the buttons are configured with interrupts.

```
#define BUT1 BIT1        // Button S1 at Port 1.1
#define BUT2 BIT2        // Button S2 at Port 1.2
...
// Configuring buttons with interrupt
P1DIR &= ~(BUT1|BUT2);   // 0: input
P1REN |= (BUT1|BUT2);    // 1: enable built-in resistors
P1OUT ...                // 1: built-in resistor is pulled up to Vcc
P1IE ...                 // 1: enable interrupts
P1IES ...                // 1: interrupt on falling edge
P1IFG ...                // 0: clear the interrupt flags
```

Port 1 has eight pins each of which can raise an interrupt individually. Accordingly, each of Port 1's eight pins can enable its built-in resistor (P1REN is 8-bit), can enable its interrupt (P1IE is 8-bit) and has its own interrupt flag (P1IFG is 8-bit), etc.

Complete the code by writing 0 or 1 in each configuration variable according to the comments in the code. Then, enable the global interrupts bit (GIE bit) and write an empty infinite for-loop at the end of main function so the code waits there between interrupts.

The remaining task is to write the ISR and link it to its vector in the vector table. All the eight bits in Port 1 share the same vector and, therefore, have the same ISR. Accordingly, the two push buttons, which are both mapped to Port 1, are serviced by the same ISR.

What is the name of the Port 1 vector? We suspect it's something like P1_VECTOR or PORT_1_VECTOR. The vector names are defined in the header file, in our case, it is the file msp430fr6989.h. Try locating the file in the folder path shown below. Then open the file (in Code Composer Studio or Notepad), search for the term 'VECTOR' and find the Port 1's vector name so you can link the ISR in the pragma line of code.

```
C:\...\ccsv7\ccs_base\msp430\include_gcc\msp430fr6989.h
```

The Port 1 ISR is called if either of the buttons is pushed. Its outline is shown below. We can find out which button was pushed by inspecting P1IFG which contains eight flags. First, we perform an if-statement to check if button 1 was pushed (bit BUT1 in P1IFG is true). In such case, the red LED is toggled and the bit BUT1 in P1IFG is cleared. Remember the MSP430 policy on clearing the flags. Since the Port 1 ISR is shared among multiple interrupt pins, the programmer is responsible for clearing the flag (the hardware won't do it). Following, a similar if-statement checks for the second button and, if pushed, toggles the green LED and clears the flag. We should write two if-statements (rather than if-else) since both buttons may need to be processed simultaneously.

```
#pragma vector = ...              // Write the vector name
__interrupt void Port1_ISR() {
  // Detect button 1 (BUT1 in P1IFG is 1)
  if ( ... ) {
     // Toggle the red LED
     ...
     // Clear BUT1 in P1IFG
     ...
  }

  // Detect button 2 (BUT2 in P1IFG is 1)
  if ( ... ) {
     // Toggle the green LED
     ...
     // Clear BUT2 in P1IFG
     ...
  }
}
```

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.

- Is the code working flawlessly? Some buttons bounce when pushed (oscillate multiple times between low and high) and end up raising multiple interrupts in one push. Test each button by pushing it 20 or 30 times until you observe some cases of failure (i.e: you push the button and the LED doesn't toggle).
- Roughly, what is the success rate of this code?
- Submit the code in your report.

Note that the solution to deal with button bouncing is by applying a debouncing algorithm or by using debouncing hardware.

4.4 Low-Power Modes

Multiple low-power modes are supported in the MSP430 microcontroller. By shutting down more and more components that are not needed, more power can be saved. The most popular low-power modes are summarized in Table 4.3.

Table 4.3: The Popular Low-Power Modes in MSP430

Mode	**MCLK**	**SMCLK**	**ACLK**
Active mode	On	On	On
LPM0	x	On	On
LPM3	x	x	On
LPM4	x	x	x

In the active mode, all the clocks are on. MCLK (Master Clock) drives the CPU and SMCLK (SubMaster Clock) and ACLK (Auxiliary Clock) drive peripherals. Engaging LPM0 shuts down MCLK which means the CPU is off. It's possible for peripherals to continue running based on SMCLK and ACLK. LPM3 further shuts down SMCLK and peripherals can run using ACLK. Finally, engaging LPM4 disables all the clocks.

Note that MSP430FR6989 supports the overriding feature. For example, if we're in LPM4 and the timer requests ACLK, then ACLK turns on and stays on as long as the timer is using it. This prevents unintentional shutdown of peripherals due to a misconfiguration.

Interrupts and low-power modes go hand-in-hand. When a low-power mode is engaged, the CPU shuts down and possibly other clock signals do as well. The only way to reactivate the CPU is via an interrupt. Accordingly, anytime we engage a low-power mode, we enable interrupts and configure at least one interrupt event.

The typical flow of the program is to configure an interrupt event and to enter a low-power mode. When the interrupt occurs, the MCU engages the active mode automatically (activates the CPU and all the clocks), then launches the corresponding ISR, and finally returns to the same low-power mode that was engaged earlier. Comparing to the earlier three codes of this lab experiment, instead of waiting for interrupts in the empty for-loop at the end of main(), we'll now engage a low-power mode (to shut down the CPU and save power) and wait for the interrupts to occur.

The low-power modes are engaged via four bits in the Status Register (SR) called SCG1, SCG2, CPUOFF, OSCOFF. Instead of dealing with these bits directly, we use the intrinsic function below to engage a low-power mode, replacing 'x' with the actual number. Note that this function also enables the global interrupts (GIE bit) because interrupts are always used when a low-power mode is engaged. Accordingly, when this function is used, there is no need to call the intrinsic function that enables interrupts.

```
_low_power_mode_x();
```

Engaging Low-Power Modes

Revisit the three codes of this experiment and engage the appropriate low-power mode for each code. The goal is to minimize the power consumed, therefore, choose the lowest consuming power mode that keeps the code operational. Consult Table 4.3. Don't rely on the overriding feature, if a clock signal is needed, choose a low-power mode that keeps it active.

The only change needed for the earlier codes is replacing the for-loop at the end of the main() with the low-power mode intrinsic function. We can also erase the interrupt enable line since the low-power mode function automatically enables the global interrupt bit. By doing these changes, instead of having the CPU cycle in the infinite for-loop between interrupts, the CPU now is shut down while we're waiting for the interrupt events.

Perform the following and answer the questions in your report:

- Complete the codes and demo them to the TA.
- Which low-power mode did you choose for each of the three codes? Explain your choices.
- Test the three codes and ensure they remain operational.
- Submit only the few lines of code that you modified.

Student Q & A

1. Explain the difference between using a low-power mode and not. What would be the CPU doing between interrupts for each case?

2. We're using a module, e.g. the ADC converter, and we're not sure about the vector name. We expect it should be something like ADC_VECTOR. Where do we find the exact vector name?

3. A vector, therefore the ISR, is shared between multiple interrupt events. Who is responsible for clearing the interrupt flags?

4. A vector, and its corresponding ISR, is used by one interrupt event exclusively. Who is responsible for clearing the interrupt flag?

5. In the first code, the ISR's name is T0A1_ISR. Is it allowed we rename the function to any other name?

6. What happens if the ISR is supposed to clear the interrupt flag and it didn't?

Lab 5

LCD Display

In this lab, we will learn printing to the LCD display.

5.1 Printing Numbers on the LCD Display

The LCD screen is interfaced to the microcontroller via an LCD controller that is located inside the MCU. The LCD display requires more than just low/high signals to turn the segments on/off. The two terminals of an LCD segment should always oscillate; if the segment is provided with a continuous voltage, it burns out. Accordingly, the LCD controller is responsible for providing the alternating voltages on the segments' terminals and uses a clock signal for this purpose. The LCD controller on our microcontroller is the LCD_C module. It is covered in the FR6xx Family User's Guide `(slau367o)` in Chapter 36.

Another issue in interfacing LCD displays is that the number of segments can easily exceed the number of available pins on the microcontroller. For example, the LCD display on our LaunchPad board, the ADKOM model FH-1138P, has 108 segments. It is reasonable to assume that a microcontroller would have fewer than 108 pins available and, therefore, the LCD module is wired based on 4-way multiplexing. This means one pin from the microcontroller controls four segments on the display. Accordingly, there are (108/4) 27 lines used to control one end of the segments and four other lines used the control the other end of the segments, for a total of 31 lines. There are a few other lines, such as the supply voltage and the ground.

The "static drive" configuration is when a pin on the microcontroller controls one LCD segment (no multiplexing). Multiplexing usually ranges from 2-way to 8-way, where the latter uses the least number of pins on the MCU. The downside of multiplexing is that a pin controls multiple segments, one at a time (not simultaneously), but doing it fast enough that the user's eyes don't see the flickering. When the multiplexing scale is larger, e.g. 8-way multiplexing, the contrast control could become poorer.

Segment Memory Mapping & Shapes of Digits

The LCD display on the LaunchPad has six alphanumeric characters, which are 14-segment displays. They can display letters and numbers, as the font in Figure 5.1 shows. If only numbers are displayed, the digits can be slightly different, e.g, the dash can be omitted from the digit 0 since it won't be confused with the letter 'O'.

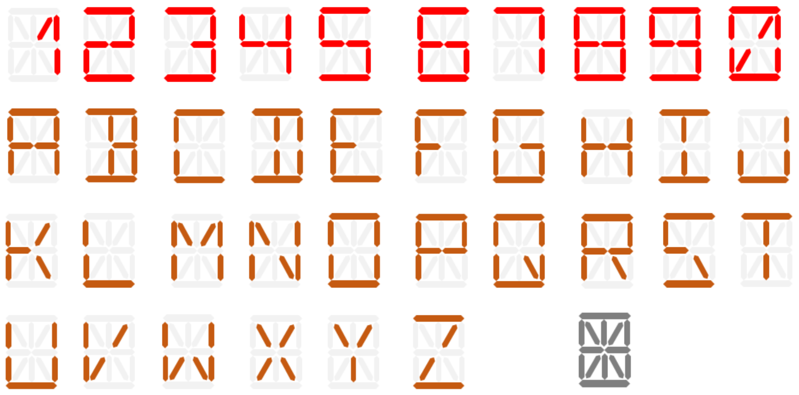
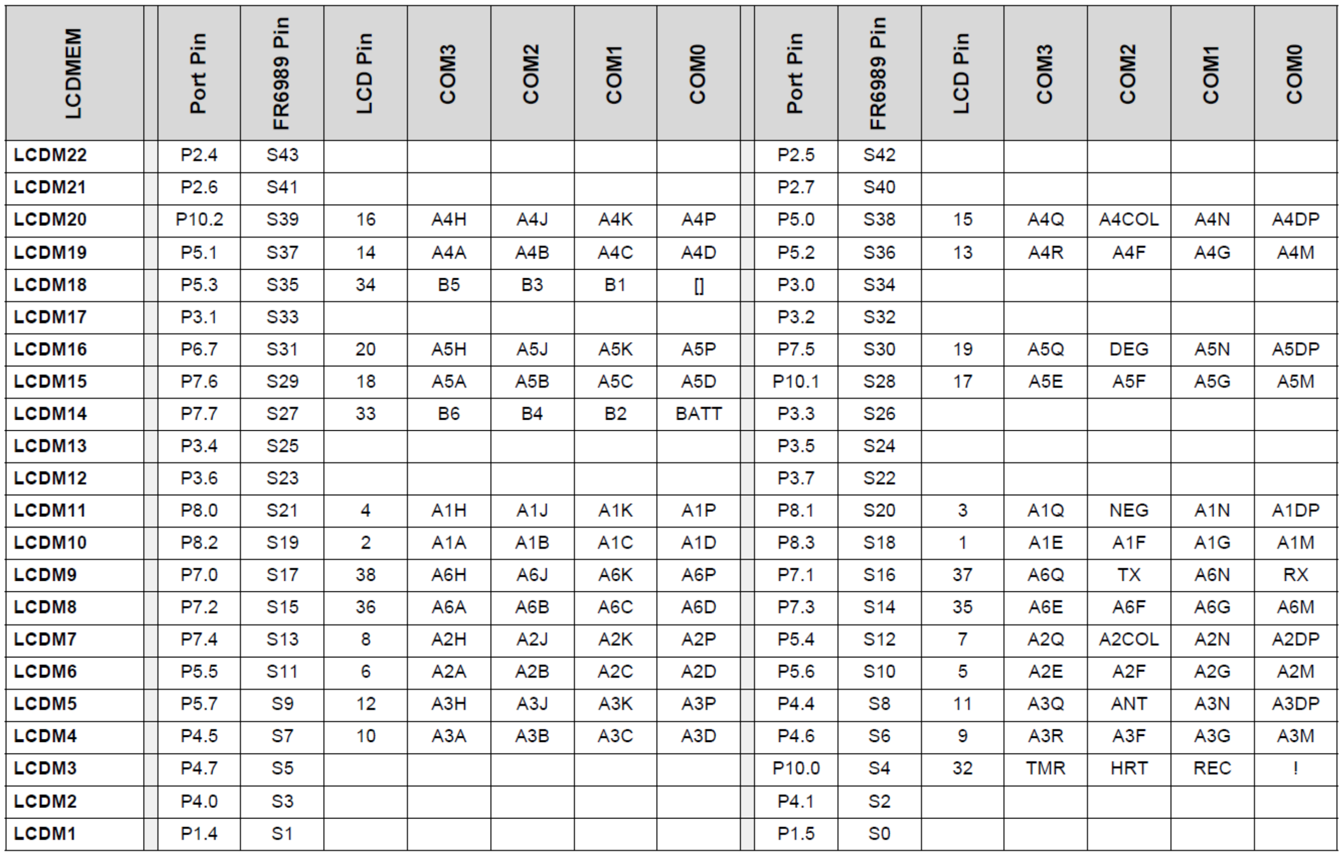


Figure 5.1: Alphanumeric font for a 14-segment display.

The segments of the LCD display are mapped to the memory variables of the LCD controller. The memory variables are called LCDM1, LCDM2, etc and are shown in Figure 5.2. Each variable is 8-bit and maps eight segments as shown in the figure.

Figure 5.3 shows the full layout of the LCD display and the names of the segments. We can zoom in on the leftmost character to see the segment names of the alphanumeric character.



LCDMEM	Port Pin	FR6989 Pin	LCD Pin	COM3	COM2	COM1	COM0	Port Pin	FR6989 Pin	LCD Pin	COM3	COM2	COM1	COM0
LCDM22	P2.4	S43						P2.5	S42					
LCDM21	P2.6	S41						P2.7	S40					
LCDM20	P10.2	S39	16	A4H	A4J	A4K	A4P	P5.0	S38	15	A4Q	A4COL	A4N	A4DP
LCDM19	P5.1	S37	14	A4A	A4B	A4C	A4D	P5.2	S36	13	A4R	A4F	A4G	A4M
LCDM18	P5.3	S35	34	B5	B3	B1	[]	P3.0	S34					
LCDM17	P3.1	S33						P3.2	S32					
LCDM16	P6.7	S31	20	A5H	A5J	A5K	A5P	P7.5	S30	19	A5Q	DEG	A5N	A5DP
LCDM15	P7.6	S29	18	A5A	A5B	A5C	A5D	P10.1	S28	17	A5E	A5F	A5G	A5M
LCDM14	P7.7	S27	33	B6	B4	B2	BATT	P3.3	S26					
LCDM13	P3.4	S25						P3.5	S24					
LCDM12	P3.6	S23						P3.7	S22					
LCDM11	P8.0	S21	4	A1H	A1J	A1K	A1P	P8.1	S20	3	A1Q	NEG	A1N	A1DP
LCDM10	P8.2	S19	2	A1A	A1B	A1C	A1D	P8.3	S18	1	A1E	A1F	A1G	A1M
LCDM9	P7.0	S17	38	A6H	A6J	A6K	A6P	P7.1	S16	37	A6Q	TX	A6N	RX
LCDM8	P7.2	S15	36	A6A	A6B	A6C	A6D	P7.3	S14	35	A6E	A6F	A6G	A6M
LCDM7	P7.4	S13	8	A2H	A2J	A2K	A2P	P5.4	S12	7	A2Q	A2COL	A2N	A2DP
LCDM6	P5.5	S11	6	A2A	A2B	A2C	A2D	P5.6	S10	5	A2E	A2F	A2G	A2M
LCDM5	P5.7	S9	12	A3H	A3J	A3K	A3P	P4.4	S8	11	A3Q	ANT	A3N	A3DP
LCDM4	P4.5	S7	10	A3A	A3B	A3C	A3D	P4.6	S6	9	A3R	A3F	A3G	A3M
LCDM3	P4.7	S5						P10.0	S4	32	TMR	HRT	REC	!
LCDM2	P4.0	S3						P4.1	S2					
LCDM1	P1.4	S1						P1.5	S0					

Figure 5.2: Segment mapping (LaunchPad User's Guide (`slau627a`) p. 13)

Accordingly, we can see that the segments A, B, C, D, E, F, G, M constitute the outer ring and the middle horizontal bars. These eight segments can be used to display the digits 0 to 9. Let's find the mapping of these segments to the memory variables LCDMx.

The six alphanumeric characters on the display are numbered 1 to 6, where 6 is the rightmost character. Let's look up the rightmost character in Figure 5.2. We can see that the variable LCDM8 corresponds to the segments A6A, A6B, A6C, A6D, A6E, A6F, A6G, A6M. Accordingly, LCDM8 is used to display a digit on the rightmost character.

Similarly, we can display digits on the second and third characters from the right, characters 5 and 4, using the variables LCDM15 and LCDM19, respectively.

To facilitate displaying digits (0 to 9) on the alphanumeric characters, it's a good idea to declare an array that stores the shapes of the digits. To display a digit, we look up its shape from the array and write it to the LCDMx variable. The declaration of the array, called LCD_Num, is shown below.

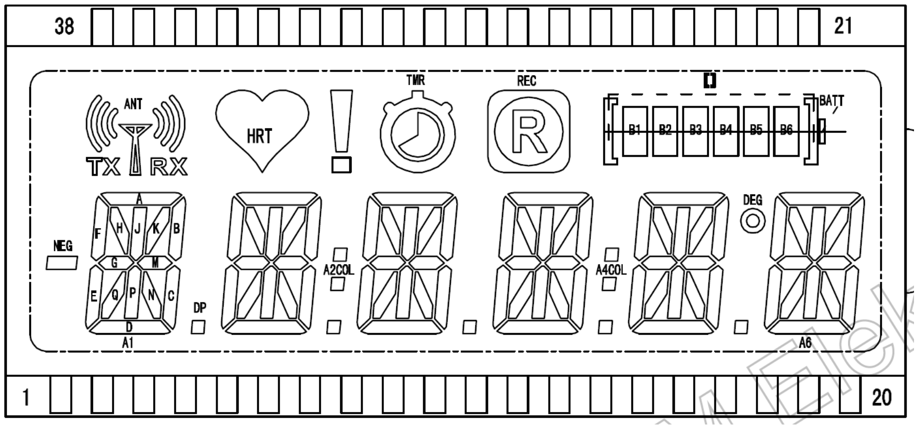


Figure 5.3: ADKOM FH-1138P LCD Monitor (leftmost character shows segment names)

Looking at Figure 5.3, we see that the shape of zero corresponds to the segments A, B ,C, D, E, F. Looking at the format of LCDM8 in Figure 5.2, we can see that all the segments should be turned on except the rightmost two. This corresponds to a binary value of 1111 1100, which is 0xFC. Hence, the array LCD_Num, stores the shape of zero (0xFC) at index zero. Similarly, to display 1 on the character, the segments B and C should be on. This corresponds to a binary value of 0110 0000, which is 0x60. Therefore, we'll store the shape of 1 (0x60) at index 1. Complete the array by storing the shape of 2 at index 2, etc for all the digits 0 to 9.

Note that all the memory variables LCDM8, LCDM15, LCDM19, etc have the same layout. Once the array LCD_Num is initialized, it can be used with all the six alphanumeric digits.

```
unsigned char LCD_Num[10] = {0xFC, 0x60, 0xDB, ...};
```

The code below prints the number 430 on on the rightmost three digits of the display. The only part missing from the code is the initialization of the LCD_Num array. Complete the code and run it.

```
// Sample code that prints 430 on the LCD monitor
#include <msp430fr6989.h>
#define redLED BIT0           // Red at P1.0
#define greenLED BIT7         // Green at P9.7
void Initialize_LCD();

// The array has the shapes of the digits (0 to 9)
// Complete this array...
```

```
const unsigned char LCD_Num[10] = {0xFC, 0x60, 0xDB, ...};

int main(void) {
    volatile unsigned int n;
    WDTCTL = WDTPW | WDTHOLD;       // Stop WDT
    PM5CTL0 &= ~LOCKLPM5;           // Enable GPIO pins

    P1DIR |= redLED;                // Pins as output
    P9DIR |= greenLED;
    P1OUT |= redLED;                // Red on
    P9OUT &= ~greenLED;             // Green off

    // Initializes the LCD_C module
    Initialize_LCD();

    // The line below can be used to clear all the segments
    //LCDCMEMCTL = LCDCLRM;         // Clears all the segments

    // Display 430 on the rightmost three digits
    LCDM19 = LCD_Num[4];
    LCDM15 = LCD_Num[3];
    LCDM8  = LCD_Num[0];

    // Flash the red and green LEDs
    for(;;) {
        for(n=0; n<=50000; n++) {} // Delay loop
        P1OUT ^= redLED;
        P9OUT ^= greenLED;
    }

    return 0;
}

//**********************************************************
// Initializes the LCD_C module
// *** Source: Function obtained from MSP430FR6989's Sample Code ***
void Initialize_LCD() {
    PJSEL0 = BIT4 | BIT5;       // For LFXT
    // Initialize LCD segments 0 - 21; 26 - 43
    LCDCPCTL0 = 0xFFFF;
    LCDCPCTL1 = 0xFC3F;
    LCDCPCTL2 = 0x0FFF;

    // Configure LFXT 32kHz crystal
    CSCTL0_H = CSKEY >> 8;        // Unlock CS registers
    CSCTL4 &= ~LFXTOFF;           // Enable LFXT
    do {
        CSCTL5 &= ~LFXTOFFG;      // Clear LFXT fault flag
        SFRIFG1 &= ~OFIFG;
    }while (SFRIFG1 & OFIFG);     // Test oscillator fault flag
    CSCTL0_H = 0;                 // Lock CS registers

    // Initialize LCD_C
    // ACLK, Divider = 1, Pre-divider = 16; 4-pin MUX
    LCDCCTL0 = LCDDIV__1 | LCDPRE__16 | LCD4MUX | LCDLP;

    // VLCD generated internally,
    // V2-V4 generated internally, v5 to ground
    // Set VLCD voltage to 2.60v
    // Enable charge pump and select internal reference for it
    LCDCVCTL = VLCD_1 | VLCDREF_0 | LCDCPEN;

    LCDCCPCTL = LCDCPCLKSYNC;     // Clock synchronization enabled

    LCDCMEMCTL = LCDCLRM;         // Clear LCD memory

    //Turn LCD on
    LCDCCTL0 |= LCDON;

    return;
}
```

Displaying a 16-bit Unsigned Number

Modify the code by writing a function that displays any 16-bit unsigned integer. The function's header is shown below. The unsigned integer is passed as a parameter. The 'unsigned int' on MSP430 is allocated as 16-bit. Accordingly, it ranged from 0 up to 65,535 and has at most five digits. Therefore, it can be printed on the six available characters. Ensure the unused characters on the display are turned off.

```
void display_num_lcd(unsigned int n);
```

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Test the function with small values (e.g: 0, 10, 12), with large values (e.g: 12345, 60000), and with the largest value (65535) and ensure it prints all of them correctly.
- Submit the code in your report.

5.2 Implementing a Stopwatch

In this part, modify the code of the previous part to implement a stopwatch. Use the Timer_A module with the 32 KHz crystal since it's a precise clock signal. Use the function from earlier labs that configures ACLK to the 32 KHz crystal. Use the timer in the up mode to generate a delay of one second. Start printing 0 on the display and, when a second elapses, the number counts up to 1, 2, etc. You don't have to implement hours/minutes/second. You can simply keep counting up: 59, 60, 61, ..., 65535, 0, 1, ...

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Compare the timing to your phone's stopwatch for at least 60 seconds and make sure the timing matches closely (the crystal is accurate).
- Submit the code in your report.

5.3 Stopwatch with Halt/Resume and Reset Functions

Modify the code of the previous part to implement the halt/resume and reset functions. When the button S1 is pushed, the counting should stop and the current number should remain on the display. Pushing S1 again should resume the counting from the currently displayed value. Pushing S2 should reset the count to zero. If the reset is performed during counting, the value should go to zero and the counting should continue. If the reset is performed when the counting is halted, the value should go to zero and the stopwatch should remain halted.

Indicate the counting/halted status using the LED lights. When the stopwatch is counting, the green LED should be on. When the stopwatch is halted, the red LED should be on.

Perform the following:

- Complete the code and demo it to the TA.
- Submit the code in your report.

Student Q & A

1. Explain whether this statement is true or false. If false, explain the correct operation. “an LCD segment is given 1 to turn it on and 0 to turn it off, just like the colored LEDs”. Explain whether it’
2. What is the name of the LCD controller the LaunchPad uses to interface the LCD display? Is the LCD controller located on the display module or in the microcontroller?
3. In what multiplexing configuration is the LCD module wired (2-way, 4-way, etc)? What does this mean regarding the number of pins used at the microcontroller?

Lab 6

Advanced Timer Features

In this lab, we will learn using advanced timer features. First, we'll use multiple channels of the timer module to generate independent periodic interrupts. Then, we'll use the timer's output patterns to generate a Pulse Width Modulation (PWM) signal that controls the brightness level of the LED.

6.1 Using the Timer with Two Channels

The Timer_A module has multiple channels that allow timing independent intervals. Usually, the module has three or five channels and is named Timer_A3 or Timer_A5, respectively. We have seen earlier that there could be multiple independent Timer_A modules in the MCU. For example, if there are two Timer_A modules with three channels each, they are called Timer0_A3 and Timer1_A3. The MSP430FR6989 is an advanced chip and has five Timer_A modules; three modules have three channels each and two modules have five channels each. This is information is presented in the chip's data sheet (`slas789c`) on p. 7 and is summarized on the first page of the data sheet.

To time independent intervals with multiple channels, the timer module is operated in the continuous mode. TAR counts from 0 up to 64K (65,535) then rolls back to zero. At the same time, the channels schedule their interrupts by looking ahead from the current value of TAR. This mechanism is shown in Figured 6.1.

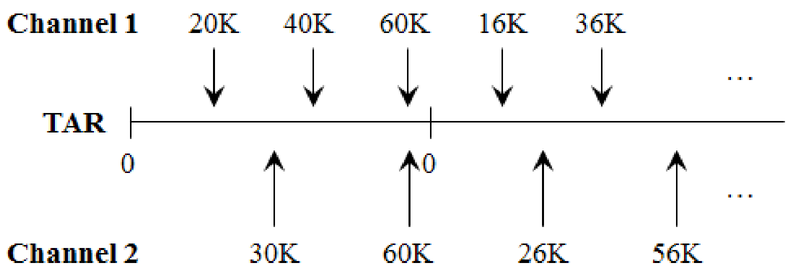


Figure 6.1: Using two channels of Timer_A

In the figure, Channel 1 is scheduling periodic interrupts every 20K cycles. Accordingly, the first interrupt is at 20K and the next two interrupts are at 40K and 60K. We can see that Channel 1's register increments by 20K. However, TAR and the channels' registers are 16-bit and can't go beyond 64K. For the next interval, Channel 1 adds 20K to its interval and the following math occurs:

```
60K + 20K = 80K (on 16-bit; leftmost bit worth 64K is dropped)
= 80K - 64K = 16K
```

Accordingly, the answer of 60K + 20K = 16K on 16-bit. It turns out, this results in the correct number of cycles for the next interrupt. To go from 60K to 16K, first TAR counts up to 64K (that's 4K cycles), then rolls back to zero and counts up to 16K for a total of 20K. Therefore, this overflow operation is harmless and we can simply keep adding 20K to Channel 1's register.

Let's validate the operation of Channel 2 in the figure. It's generating periodic interrupts every 30K cycles. The first two milestones are 30K and 60K. For the next interval, we get: 60K + 30K (on 16-bit) = 90K - 64K = 26K. Indeed, it takes TAR 30K cycles to count from 60K to 26K, passing by zero.

In this fashion, multiple channels can generate periodic interrupts while the timer is running in the continuous mode. Note that, in the continuous mode, Channel 0 is not a special channel and is used like Channels 1 and 2 to generate periodic interrupts. In contrast, Channel 0 is a special channel in the up mode since it designates the upperbound of TAR.

Interrupt Events of Timer_A

Multiple interrupt events of Timer_A are summarized in Table 6.1. The second column shows the trigger. The third column shows the enable and flag bits and the register in which these bits are located. Finally, the rightmost column shows the vector associated with each event. We can see that the rollback-to-zero event and Channels 1 and 2 share the A1 vector. On the other hand, Channel 0 exclusively uses the A0 vector. Remember the MSP430 policy regarding clearing the flags. Channel 0 has a non-shared vector; its interrupt flag is cleared by the hardware. The other three events share a vector and, therefore, their flags are cleared by the software.

Table 6.1: Multiple Interrupt Events of Timer_A

Event	**Trigger**	**Bits**	**Vector**
Rollback-to-zero	TAR=0	TAIE / TAIFG in TACTL	A1
Channel 0	TAR=TACCR0	CCIE / CCIFG in TACCTL0	A0
Channel 1	TAR=TACCR1	CCIE / CCIFG in TACCTL1	A1
Channel 2	TAR=TACCR2	CCIE / CCIFG in TACCTL2	

Flashing Two LEDs using Two Channels

We'll write a code that runs Timer_A using ACLK based on the 32 KHz crystal. Use the function from earlier labs that configures ACLK to the crystal. Channel 0 toggles the red LED every 0.1 seconds and Channel 1 toggles the green LED every 0.5 seconds. We'll use interrupts and engage a low-power mode to save power while waiting for the interrupts to occur. The code is shown below. Complete the missing parts.

```
// Using Timer_A with 2 channels
// Using ACLK @ 32 KHz (undivided)
// Channel 0 toggles the red LED every 0.1 seconds
// Channel 1 toggles the green LED every 0.5 seconds
#include <msp430fr6989.h>
#define redLED BIT0           // Red at P1.0
#define greenLED BIT7         // Green at P9.7

void main(void) {
    WDTCTL = WDTPW | WDTHOLD;        // Stop WDT
    PM5CTL0 &= ~LOCKLPM5;            // Enable GPIO pins

    P1DIR |= redLED;
    P9DIR |= greenLED;
    P1OUT &= ~redLED;
    P9OUT &= ~greenLED;

    // Configure Channel 0
    TA0CCR0 = 3277-1;              // @ 32 KHz --> 0.1 seconds
    TA0CCTL0 |= CCIE;
    TA0CCTL0 &= ~CCIFG;

    // Configure Channel 1 (write 3 lines similar to above)
    ...

    // Configure timer (ACLK) (divide by 1) (continuous mode)
    TA0CTL = ...

    // Engage a low-power mode
    ...

    return;
}

// ISR of Channel 0 (A0 vector)
#pragma vector = TIMER0_A0_VECTOR
__interrupt void T0A0_ISR() {
    P1OUT ^= redLED;               // Toggle the red LED
    TA0CCR0 += 3277;               // Schedule the next interrupt
    // Hardware clears Channel 0 flag (CCIFG in TA0CCTL0)
}

// ISR of Channel 1 (A1 vector) ... fill the vector name below
#pragma vector = ...
__interrupt void T0A1_ISR() {
    ...                        // Toggle the green LED
    ...                        // Schedule the next interrupt
    ...                        // Clear Channel 1 interrupt flag
}
```

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Do a visual comparison to your phone's stopwatch and check that the durations appear to be correct.
- Show how the number of cycles for each channel is derived.
- Submit the code in your report.

6.2 Using Three Channels

In this part, we will extend the previous code by using three channels. Channels 0 and 1 toggle the red LED and green LED every 0.1 seconds and 0.5 seconds, respectively, just like in the previous part. Channel 2 generates periodic interrupts every 4 seconds to halt and resume the flashing, as shown in Figure 6.2. For four seconds, the LEDs are flashing, each at its respective rate, then for the next four seconds, the LEDs must be off. This pattern repeats infinitely.

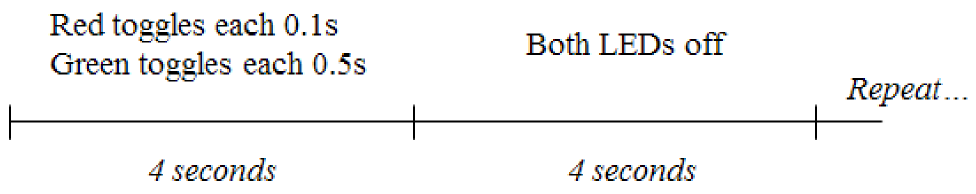


Figure 6.2: Using three channels of Timer_A

If we use ACLK @ 32 KHz, it may be hard to generate a four-second interval since this corresponds to 131,072 cycles and can't be stored in the 16-bit TA0CCR2 register. Typically, we slow down the frequency using the input divider.

For this part, divide ACLK so it becomes 8 KHz (8,192 Hz). Recompute the cycle durations for Channels 0 and 1 so they correspond to 0.1 seconds and 0.5 seconds, respectively.

The code is very similar to the previous part. Here are the differences. In the main, Channel 2 is configured for a four-second interval with the interrupt enabled. Secondly, the ISR of the A1 vector now services the interrupt events of Channels 1 and 2. Accordingly, this ISR needs to detect which interrupt(s) actually occurred. This is shown in the if-statements in the piece of code below.

```
// ISR of Channels 1 and 2 (A1 vector)
#pragma vector = ...
__interrupt void T0A1_ISR() {
    // Detect Channel 1 interrupt
    if((TA0CCTL1 & CCIFG)!=0) {
        ...
    }

    // Detect Channel 2 interrupt
    if((...)) {
        ...
    }
}
```

Finally, it may be useful to maintain a variable ‘status’ to keep track of whether the LEDs are currently flashing or not. This helps differentiating between the transitions from flashing to idle and vice versa. Such a variable can be declared as ‘static’ inside the ISR so that it keeps its value between calls.

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Do a visual comparison to your phone’s stopwatch and check that the durations appear to be correct.
- Show how the number of cycles for each channel is derived.
- Submit the code in your report.

6.3 Generating a PWM Signal with Timer_A

The Timer_A module can generate a Pulse-Width Modulation (PWM) signal on its own without any action from the CPU. The PWM signal is frequently used in interfaces. When the timer generates the PWM, the CPU may remain in low-power mode indefinitely, therefore, saving significant battery power.

A PWM signal is shown in Figure 6.3. The period is fixed and is marked by the vertical dashed lines (e.g: 1000 Hz). In a period, the high pulse’s duration is varied. The ratio of the high duration to the period is called the duty cycle. In the figure, the short pulse corresponds to a duty cycle of 25% and the long pulse corresponds to a duty cycle of 50%.

As an example, if the PWM signal is driving a motor, a higher duty cycle would correspond to higher motor speed. In this lab, we’ll direct the PWM signal to the LED. A higher duty cycle results in a higher brightness level. Note that it’s important to choose a suitable frequency. If the period is too long (e.g: 2 seconds), the user will notice the blinking of the LED. We’ll use a frequency of 1000 Hz, which corresponds to a period of 0.001 seconds. Doing a PWM with this

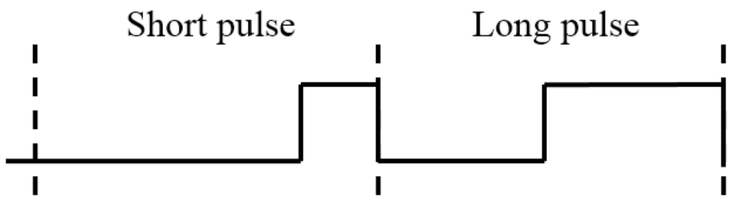


Figure 6.3: Pulse Width Modulation (PWM) Signal

period results in various brightness levels and the user won't see the blinking.

Directing the PWM Signal to the LED

The red and green LEDs on our LaunchPad are connected to P1.0 and P9.7, respectively. We can have a timer-driven PWM signal diverted to these LEDs only if these pins double as timer channel outputs. To find this out, we'll look at the chip pinout in the MSP430FR6989 Data Sheet `(slas789c)` on p. 9. Our chip is the 100-pin variety. A timer channel is marked on the pinout as TAx.y (Timer x, Channel y). The pinout shows that the pin of P1.0 doubles as TA0.1, which is Timer0_A Channel 1. Therefore, if we divert this pin to the TA0.1 functionality and drive a PWM signal on it, it reaches the LED and controls its brightness level.

Take a snapshot of the pinout and highlight the pins of P1.0 and P9.7. For each pin, answer whether it's possible to connect it to a timer-driven PWM signal.

The default functionality of a pin is the IO Port. Therefore, the pin P1.0/TA0.1 is configured as P1.0 at reset. That's why we didn't need to divert the pin in earlier labs when we controlled the LED via P1.0. To divert this pin to the TA0.1 functionality, we'll look in the data sheet `(slas789c)` on p. 96. The pin is diverted to TA0.1 by setting the values below:

```
P1DIR bit = 1          P1SEL1 bit = 0          P1SEL0 bit = 1
```

Note that P1SEL1 and P1SEL0 are 8-bit each. Along with P1DIR, we get 3 bits for each pin of Port 1. This makes it possible to divert a pin to one of eight different functionalities.

Output Patterns

The PWM signal is generated based on the output patterns supported by Timer_A. These are shown in the FR6xx User's Guide `(slau367o)` Chapter 25 on p. 649. We will choose the output mode Reset/Set, which is mode # 7. Its operation is shown in Figure 6.4.

Note that we're aiming to generate a PWM signal with a frequency of 1000 Hz. The period is 0.001 seconds and corresponds to 33 cycles based on a 32 KHz clock signal. Therefore, we'll run the timer in the up mode with a period of 33 cycles.

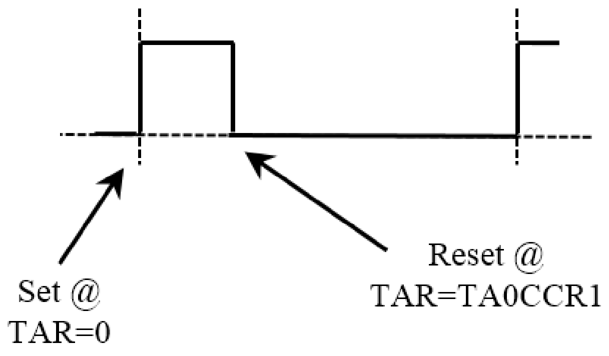


Figure 6.4: Reset/Set Output Pattern

The output mode we're using has two actions (action1/action2), i.e., Reset/Set. The way it works in the up mode is the first action occurs on the channel's event when TAR = TACCR1 (for Channel 1) and the second action occurs when TAR rolls back to zero. Accordingly, if we set TACCR1=10, we get a Set (second action) when TAR rolls back to zero and a Reset (first action) when TAR=10, as shown in the figure. This results in a duty cycle of (10/33) 30.3%. This is an intuitive setup since a higher value of TACCR1 results in higher brightness on the LED. In contrast, if we use the Set/Reset mode, a higher value of TACCR1 results in lower brightness.

The code is shown below. Read the code to understand how it works and complete the two missing lines.

```
// Generating a PWM on P1.0 (red LED)
// P1.0 coincides with TA0.1 (Timer0_A Channel 1)
// Divert P1.0 pin to TA0.1 ---> P1DIR=1, P1SEL1=0, P1SEL0=1
// PWM frequency: 1000 Hz -> 0.001 seconds
#include <msp430fr6989.h>
#define PWM_PIN BIT0

void main(void) {
    WDTCTL = WDTPW | WDTHOLD;                   // Stop WDT
    PM5CTL0 &= ~LOCKLPM5;

    // Divert pin to TA0.1 functionality (complete last two lines)
    P1DIR |= PWM_PIN;               // P1DIR bit = 1
    P1SEL1 ...                      // P1SEL1 bit = 0
    P1SEL0 ...                      // P1SEL0 bit = 1

    // Configure ACLK to the 32 KHz crystal (call function)
    ...

    // Starting the timer in the up mode; period = 0.001 seconds
    // (ACLK @ 32 KHz) (Divide by 1) (Up mode)
    TA0CCR0 = (33-1);    // @ 32 KHz --> 0.001 seconds (1000 Hz)
    TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;

    // Configuring Channel 1 for PWM
    TA0CCTL1 |= OUTMOD_7;        // Output pattern: Reset/set
    TA0CCR1 = 1;                 // Modify this value between 0 and
                                 // 32 to adjust the brightness level

    for(;;) {}
    return;
}
```

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Include the snapshot of the pinout and mention, for each of the LED pins (P1.0 and P9.7), whether it's possible to connect it to a timer-driven PWM signal.
- Vary the value of TA0CCR1 between 1 and 32 and ensure this results in various brightness levels.
- Do larger values of TA0CCR1 results in higher or lower brightness? Explain.
- Submit the code in your report.

6.4 Cycling through Brightness Levels

Modify the code of the previous part so that it demos multiple brightness levels. We have generated a PWM signal with a period of 33 cycles, and now we can vary Channel 1's register (TA0CCR1) between 0 and 32 to obtain various brightness levels.

In this part, write a code that cycles between six brightness levels which correspond to TA0CCR1 = 0, 5, 10, 15, 20, 25, 30. The code should stay for one second at each brightness level and cycles between them infinitely.

Since Timer0_A is used to generate the PWM signal and is running in the up mode with a frequency of 1000 Hz, it's a good idea to use another Timer_A module to generate the 1-second interval. Use Timer1_A for this purpose. Run this timer in the up mode and use interrupts. When the interrupt occurs, the ISR cycles TA0CCR1 between the values shown above. Also, in the ISR, toggle the green LED to indicate the activity. This helps us notice that the microcontroller is changing the brightness levels. Finally, engage a low-power mode.

To user Timer1_A, note that the configuration register is now TA1CTL and the Channel 0 register is TA1CCR0. Similarly, the A0 vector is called TIMER1_A0_VECTOR.

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Explain how the two timer modules are interacting with each other.
- Submit the code in your report.

Student Q&A

1. In the code with three channels, why did we divide ACLK so it becomes 8 KHz?
2. In the first part, we configured two periodic interrupts using two channels of the timer. Is this approach scalable? For example, using a Timer_A module with five channels, can we configure five periodic interrupts? Explain and mention in what mode the timer would run.
3. As an example, Channel 1's interrupt occurs every 40K cycles. The first interrupt is scheduled for when TAR=40K cycles. Explain how the next interrupt is scheduled? (hint: explain the overflow mechanism and why it results in a correct value)

Lab 7

Concurrency via Interrupts

In this lab, we will learn programming multiple interrupt events in a way where the interrupts interact with each other. This gives a sense of concurrency in the code where multiple events are processed in an overlapping way, rather than in a sequential way.

7.1 Long Pulse on the LED (non-renewing interval)

In this part, we will write a program that turns on the LED for 3 seconds when the button is pushed. The interval is non-renewing which means, if the button is pushed again during the 3-second interval, the timer doesn't renew. The timeline of events is shown in Figure 7.1. As the figure shows, when the button is pushed during the three-second interval, it is ignored.

In the program, read the button via interrupt. Time the three-second interval using Timer_A with interrupts. You can use any mode (up more or continuous mode) and channel that you prefer. Finally, engage a low-power mode while waiting for the button push and during the three-second interval.

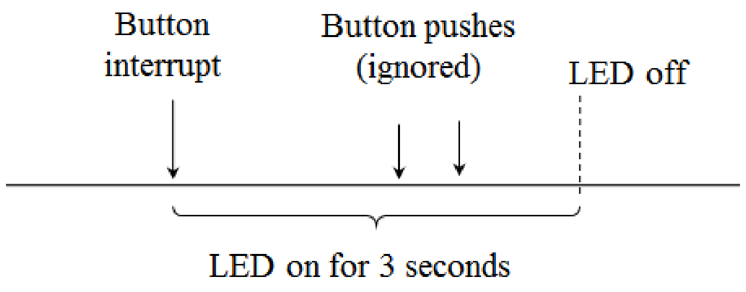


Figure 7.1: LED turns on for three seconds when the button is pushed

The video at the link below demonstrates the program's behavior:

https://youtu.be/v-q5EGTlHIU

The program has two ISRs: the button's and the timer's. The two interrupt events interact with one another. The button's ISR enables the timer's interrupt. Similarly, the timer's ISR, at the end of the three-second interval, re-enables the button's interrupt. In this program, interrupt events are enabled/disabled multiple times. The button's interrupt is disabled during the three-second interval so the button is ignored. Similarly, the timer's interrupt is disabled at the end of the three-second interval so the timer doesn't keep raising interrupts if the button is not pushed again.

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Looking at the figure, write a list of items to be done when the button raises an interrupt.
- Write a list of items to be done when the timer raises an interrupt at the end of the three-second interval.
- Explain your configuration: clock signal and divider, timer mode, channel used
- Which low-power mode did you use?
- Submit the code in your report.

7.2 Long Pulse on the LED (renewing interval)

Modify the code of the previous part so that the interval is renewing. If the button is pushed during the 3-second interval, the timer renews from the moment it is last pushed. As an example, let's consider the button is pushed at t=0, t=1s and t=2s. At t=0, the LED turns on and is scheduled to stay on until t=3s. At t=1s, the timer renews until t=4s. Finally, at t=2s, the timer renews again until t=5s. As a result, the LED stays lit from t=0 to t=5s. According to this procedure, if the user keeps pushing the button midway through the interval, the LED remains lit continuously.

Perform the following and submit the answers in your report:

- Write the code and demo it to the TA.
- Write a list of items to be done when the button raises an interrupt.
- Also, write a list of items to be done when the timer raises an interrupt at the end of the three-second interval.
- Submit the code in your report.

7.3 Button Debouncing

In this part, we will implement a push button debouncing algorithm. In an earlier lab, we wrote a code that toggles the LED when the button is pushed. The button was read via interrupt. We observed that the button doesn't work all the time because the bounces end up raising multiple interrupts and cancel out the toggle operation. In this part, we will debounce the button so that it works every time.

The debouncing algorithm we'll implement is shown in Figure 7.2. This algorithm takes two samples of the button separated by the maximum bounce duration. As the figure shows, when the button is pushed, the first falling edge (button is active low) raises an interrupt. At this point, we will start the timer for 20 ms, which represents the maximum bounce duration, so that we wait out all the bounces. During this interval, the button interrupt is disabled to avoid causing further interrupts. At the end of the 20 ms interval, we'll check the button status and, if it's still pushed, we'll interpret a button push and take the necessary action, e.g. toggle the LED.

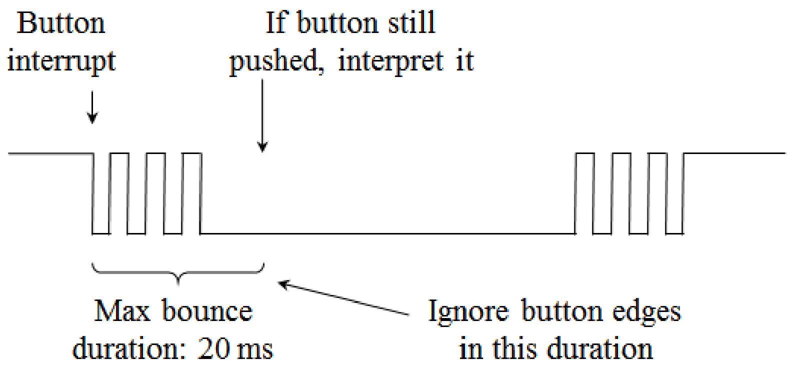

Button interrupt
If button still pushed, interpret it
Max bounce duration: 20 ms
Ignore button edges in this duration

Figure 7.2: Button debouncing algorithm

A relevant question is: what happens when the button is released? At release, the first falling edge triggers the same procedure. After waiting for the maximum bounce duration, however, the button will be found to be released and a button push is not interpreted.

The severity of bouncing usually depends on the quality of the button. Push buttons of higher quality have shorter bounce durations. Buttons with built-in debouncers have no bounces at all. The 20 ms duration in our algorithm represents the maximum bounce duration. This value can be found experimentally, especially when the button's data sheet is not available. If bounces continue to be observed, this duration can be increased. Otherwise, if no bounces are observed, this duration may be decreased as long as the button still works reliably. We note that our algorithm effectively introduces a delay of 20 ms to the response time. This delay is small and is not noticeable to the user.

Implement this algorithm so that the LED is toggled when the button is pushed. Read the button via interrupt. Time the interval using Timer_A with interrupt. Engage a low-power mode while waiting for the button push and during the timed interval. Experiment with the maximum bounce duration to find the smallest value such that the code works reliably.

Perform the following and submit the answers in your report:

- Write the program and test it. Push the button 30 or 40 times and make sure it works every time. Is this code more reliable than the code of the earlier lab?
- Demo your program to the TA
- Write a list of items to be done when the button raises an interrupt.
- Write a list of items to be done when the timer raises an interrupt.
- What value of the maximum bounce duration did you come up with?
- Explain your choices on the following: clock signal used by the timer, timer mode, channel used, low-power mode.
- Submit the code in your report.

Student Q&A

1. For the debouncing algorithm we implemented, is it possible the LED will be toggled when the button is released? Explain.

2. If two random pulses occur on the push button line due to noise and these pulses are separated by the maximum bounce duration, will our algorithm fail? Explain.

Lab 8

Universal Asynchronous Receiver and Transmitter (UART)

In this lab, we will learn using the UART interface to transmit data between the microcontroller and the PC.

8.1 Transmitting Bytes with UART

UART is a simple interface that allows transmitting bytes between two parties. UART is asynchronous in the sense that the transmitter and the receiver each has its own clock signal. The rising and falling edges are not guaranteed to coincide. UART uses one wire and transmits data in the same direction over the wire. Accordingly, it can be implemented with one wire to transmit in one direction (known as half-duplex) or with two wires to allow bidirectional simultaneous transmission (known as full-duplex).

The transmission pattern of UART is shown in Figure 8.1. The scheme is very simple. The line is idle at high. The line drops to low for one bit duration to signal the Start Bit. Typically, the receiver is always ready and listening for the Start Bit occurrence. After that, the data is transmitted bit by bit, usually starting with the least significant bit (LSB). Finally, the Stop Bit, which has a value of high, is transmitted to signal the end of the transmission.

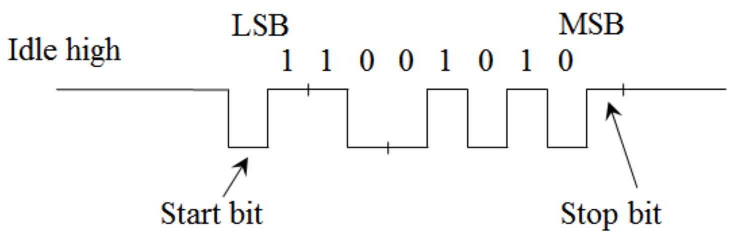


Figure 8.1: UART Transmission. Data byte: 01010011

The bit duration is defined by the baud rate (named after scientist Emile Baudot) which is simply the transmitter's clock rate. A popular rate is 9600 baud which corresponds to a clock frequency of 9600 Hz. Accordingly, a bit lasts 1/9600 seconds.

Table 8.1 shows the parameters of a UART configuration and highlights the most popular configuration. This is the configuration that we'll use.

Table 8.1: UART Parameters

Parameter	**Meaning**	**Popular Configuration**
Baud rate	Transmission speed	9600
Data size	Number of bits	8-bit
First bit	Either Most- or Least- Significant Bit	LSB
Parity	Bits to detect errors	None
Stop bit	Signals the end of the transmission	1-bit
Flow control	A mechanism to pace the transmission	None

The eUSCI Module

We will perform UART communication with the eUSCI (enhanced Universal Serial Communications Interface) module of the MSP430. This hardware module implements all the details of UART transmission and reception and our code interfaces with it using a few registers and flags. This keeps the programming simple.

The eUSCI module is organized in two channels. Channel A supports UART and SPI while Channel B supports I2C and SPI. The protocols SPI and I2C are two other transmission schemes.

Accordingly, to transmit with UART, we'll use Channel A.

An MSP430 chip may have multiple eUSCI modules to enable independent communication channels. These are usually called eUSCI0, eUSCI1, etc.

The LaunchPad Board Setup

MSP430 LaunchPad boards are set up with a backchannel UART over USB. This enables transmitting data between the LaunchPad and the PC. A virtual COM port is generated on the PC, therefore, any application that uses COM ports can communicate with the board. We'll use a terminal application (e.g: HyperTerminal or TeraTerm).

Let's start by looking at the LaunchPad User's Guide `(slau627a)` to see which eUSCI module is used by the backchannel UART. The document states (page 10) that the backchannel UART is connected to eUSCI_A1. This refers to eUSCI module #1 Channel A.

Next, we'll look at the pinout figure to see which I/O pins are shared with the UART signals. Looking in the same document (page 7), we find two cases:

```
Option 1:    P3.4/UCA1TXD    P3.5/UCA1RXD
Option 2:    P5.4/UCA1TXD    P5.5/UCA1RXD
```

The term UCA1 refers to eUSCI module #1 Channel A, the one we're interested in. The terms TXD and RXD refer to Transmit Data and Receive Data.

We need to find out whether the backchannel UART can be diverted to the pins of P3.4/P3.5 or to the pins of P5.4/P5.5. To find out which set to use, we'll look in the same document at the schematic (page 33). At the right side of the schematic, the jumpers layout show that the TXD/RXD at pins P3.4/P3.5 are linked to the emulation chip on the board which is connected to the USB.

Pins on the MSP430 chip are typically shared between multiple functionalities. The chip's data sheet `(slas789c)` shows how the pins can be diverted to the various functionalities. By default, our pins act as P3.4/P3.5. Let's look at the data sheet (page 102) to see how we can divert these pins to UCA1TXD/ UCA1RXD functionalities. We see that P3DIR is X (don't care) while P3SEL1 have 0 for both bits and P3SEL0 have 1 for both bits. The LCDS bits should be 0 (they are by default). Accordingly, we'll use the piece of code below to divert the pins to the backchannel UART.

```
// Divert pins to backchannel UART functionality
// (UCA1TXD same as P3.4) (UCA1RXD same as P3.5)
// (P3SEL1=00, P3SEL0=11) (P2DIR=xx)
P3SEL1 &= ~(BIT4|BIT5);
P3SEL0 |= (BIT4|BIT5);
```

Finally, it would be a good idea to ensure that the jumpers on the board close the connections between the MSP430 chip and the emulation chip as shown in the figure in `(slau627a)` page 8. The jumpers can be used to physically open/close the connections.

Generating the UART Clock Frequencies

There is a set of well-known baud rates that are typically used for UART. This set includes the following baud rates: 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400 and others. It is a challenge generating a clock signal at these frequencies since these are not typical frequencies found in the microcontroller. In a microcontroller, we may expect to have a 32,768 Hz crystal, but as it turns out, this is not a popular baud rate.

Let's look at the clock signal needs for UART transmission. To send data at 9600 baud, the transmitter uses a 9600 Hz clock signal. To receive data at 9600 baud, one approach is using a 9600 Hz clock signal. In such case, the receiver has one chance of sampling each data bit and this may not be reliable if there is noise on the channel. A second approach is having the receiver use a faster clock rate, at 16 times the transmitter's rate. The receiver then can sample each data bit multiple times and do a majority vote on the samples. This approach is known as oversampling. For example, to receive data at 9600 baud with oversampling, the receiver sets up a clock signal at (16 x 9600) 153600 Hz. The eUSCI module supports UART reception in both modes, with and without oversampling.

To recap, to communicate data at 9600 baud, our module generates a transmitter's clock at 9600 Hz and a receiver clock at either 9600 Hz (without oversampling) or at 16x9600 Hz (with oversampling). To generate these frequencies, the available clock signals are used (e.g. SMCLK at 1.048 MHz or ACLK at 32 KHz) and are divided and modulated.

The Family User's Guide `(slau367o)` (page 779) contains a table that shows the dividing and modulating parameters. Let's look up the parameters for our configuration. Starting with the default SMCLK at 1048576 Hz and aiming at a baud rate of 9600 and using oversampling (UCOS16=1), the dividing and modulating parameters are: UCBR=6, UCBRF=13 and UCBRS=0x22. We'll configure these parameters in our code.

eUSCI Module Configuration

At this point, we are ready to configure the eUSCI module. We'll use 9600 baud with oversampling. Let's start by noting that the eUSCI module is in reset state by default, i.e., it's not running. The configuration should be modified only while in reset state. Therefore, we'll do the configuration first, then we'll exit reset state to begin the transmission and reception.

The configuration registers for eUSCI (UART mode) are found in the Family User's Guide `(slau367o)` at the end of Chapter 30 (starting on page 783). Looking at these registers, we can see that most of the default values correspond to the most popular configuration shown in Table 8.1. Below is the code that configures the UART transmission.

```
// Configure UART to the popular configuration
// 9600 baud, 8-bit data, LSB first, no parity bits, 1 stop bit
// no flow control
// Initial clock: SMCLK @ 1.048 MHz with oversampling
void Initialize_UART(void){
    // Divert pins to UART functionality
    P3SEL1 &= ~(BIT4|BIT5);
    P3SEL0 |= (BIT4|BIT5);

    // Use SMCLK clock; leave other settings default
    UCA1CTLW0 |= UCSSEL_2;

   // Configure the clock dividers and modulators
   // UCBR=6, UCBRF=13, UCBRS=0x22, UCOS16=1 (oversampling)
    UCA1BRW = 6;
    UCA1MCTLW = UCBRS5|UCBRS1|UCBRF3|UCBRF2|UCBRF0|UCOS16;

    // Exit the reset state (so transmission/reception can begin)
    UCA1CTLW0 &= ~UCSWRST;
}
```

Programming Model

The eUSCI hardware handles the UART transmission and reception of data and interfaces with the code using a few registers and flags. These are listed at the end of the UART chapter in the family user's guide and summarized in Table 8.2.

To make our code easy to read, we'll rename the flags and registers with user-friendly names

Table 8.2: UART Registers and Flags

Flag (1-bit)	**Register**	**Use**
-	UCA1TXBUF	Transmit buffer contains the transmit byte
-	UCA1RXBUF	Receive buffer contains the received byte
UCTXIFG	UCA1IFG	0: transmission in progress; 1: ready to transmit
UCRXIFG	UCA1IFG	0: no new data; 1: new byte received

by defining these symbolic constants at the top of the code.

```
#define FLAGS       UCA1IFG  // Contains the transmit & receive flags
#define RXFLAG      UCRXIFG         // Receive flag
#define TXFLAG      UCTXIFG         // Transmit flag
#define TXBUFFER    UCA1TXBUF       // Transmit buffer
#define RXBUFFER    UCA1RXBUF       // Receive buffer
```

The transmit flag is 1 when the module is ready to transmit. To transmit a byte, the byte is copied to the transmit buffer. This causes the transmit flag to go to zero automatically and the transmission begins. When the transmission finishes, the transmit flag goes back to one. Accordingly, this is the function that transmits a byte over UART.

```
void uart_write_char(unsigned char ch){
  // Wait for any ongoing transmission to complete
  while ( (FLAGS & TXFLAG)==0 ) {}

  // Write the byte to the transmit buffer
  TXBUFFER = ch;
}
```

The receive flag is zero when there is no new data. When a byte is received, the receive flag becomes 1. As soon as we copy the byte from the receive buffer, the receive flag goes to zero automatically. This is the function that receives a byte over UART.

```
// The function returns the byte; if none received, returns NULL
unsigned char uart_read_char(void){
  unsigned char temp;
  // Return NULL if no byte received
  if( (FLAGS & RXFLAG) == 0)
    return NULL;

  // Otherwise, copy the received byte (clears the flag) and return it
  temp = RXBUFFER;

  return temp;
}
```

When we transmit bytes over UART to the terminal application on the PC, the latter interprets the bytes as ASCII characters. For example, if the MCU transmits a byte of value 65, the character ‘A’ appears on the terminal. And, vice versa, if we type ‘B’ on the terminal, a byte of value 66 is received by the MCU. Hence, we called our functions, `uart_write_char()` and `uart_read_char()`.

Testing the UART Transmission

In your code, combine the three functions shown above along with the mask definitions. In the main() function, write an infinite loop that transmits the characters 0 to 9 over the UART transmission. Write a delay loop that introduces a small delay between the characters. Also, toggle the red LED to indicate the ongoing activity.

Note that you don’t have to lookup ASCII numbers. Writing `(char ch='A';)` makes the variable equal to the ASCII of ‘A’. Accordingly, we can write this loop `for(ch='0'; ch<='9'; ch++)` to generate the ASCII numbers of the digits 0 to 9.

After every character, transmit a new line character ‘\n’ followed by the carriage return character ‘\r’. The new line character causes the cursor to go down one line and the carriage return character causes the cursor to go to the leftmost column of the line.

In your program, read the characters transmitted from the terminal application on the PC. If the user types 1 on the keyboard, turn on the green LED. Otherwise, if the use types 2 on the keyboard, turn off the green LED.

On the PC, open the terminal application and setup the popular configuration that we’re using, as shown in Table 8.1. To find out which COM port the MSP430 UART maps to on the PC, you can open the **Device Manager** in Windows and look at the COM ports. Some terminal applications, like TeraTerm, show the active COM ports within the application.

Perform the following:

- Complete the code and demo it to the TA.
- Your code should transmit the digits 0 to 9 in an infinite loop to the terminal application on the PC.
- When you type 1/2 on the keyboard, the green LED should turn on/off.
- Submit the code in your report.

8.2 Sending Unsigned 16-bit Integers over UART

To make use of the UART connection, we are interested in sending 16-bit unsigned numbers to the PC. This enables us to send measurements and inspect them or log them on the PC. This is the header of the function.

```
void uart_write_uint16(unsigned int n);
```

An 16-bit unsigned integer is in the range [0 - 65,535]. The function should disassemble the integer into digits and send the corresponding ASCII values, one by one, over UART. For example, if the integer is 123, the function should find the digits 1, 2, 3 separately. Then, it should send the ASCII of 1, followed by the ASCII of 2 and, finally, the ASCII of 3. Therefore, the text 123 appears on the terminal on the PC. Accordingly, we are converting from integer to ASCII text and then sending the ASCII text over UART.

Once this function works, modify the code from the previous part so that it sends incrementing numbers (0, 1, 2,..., 65535, 0, 1,...) to the terminal.

Perform the following:

- Complete the code and demo it to the TA.
- Test the function for small numbers.
- Test the function for larger numbers: 60000, 65000 and the largest value 65535.
- Submit the code in your report.

8.3 Sending an ASCII String over UART

In this part, write a function that transmits a string over the UART connection. This is the header of the function.

```
void uart_write_string(char * str);
```

This function should work for a string of any size and should call the function we wrote earlier, `uart_write_char()`, to send ASCII characters.

For example, if we declare the string below and call our function, it should transmit this string to the terminal application. Remember that strings in the C language are null terminated (they end with the NULL character). Therefore, the function should transmit all the characters until it reaches the NULL character.

```
char mystring[] = "UART Transmission Begins...";
```

Perform the following:

- Complete the code and demo it to the TA.
- Test the function for a few strings of different sizes, and containing spaces
- Submit the code in your report.

8.4 Changing the Configuration

In this part, we will modify the UART configuration by making two changes. First, we'll use ACLK based on the 32 KHz crystal as the clock source (rather than SMCLK). Secondly, we'll setup a baud rate of 4800 (rather than 9600). The other parameters remain the same as earlier.

In this configuration, it's not possible to do oversampling since we can't generate a receiver clock signal at 16 x 4800 Hz starting with a clock signal at 32768 Hz.

Two changes are needed to setup this configuration. First, use a suitable value of `UCSSEL` to select ACLK as the clock source. Look in the family user's guide at the end of the UART chapter. Secondly, find the new values of the dividers and modulators by looking in the family user's guide in the UART chapter. What are the values of: UCOS16, UCBR, UCBRF and UCBRS? Hint: it turns out UCBRF is don't care since it's only used with oversampling.

Do the changes in a new UART initialization function, `Initialize_UART_2()` and test it by transmitting incrementing numbers (0, 1, 2, ...) in an infinite loop. To configure ACLK to the 32 KHz crystal, use the function from earlier labs: `config_ACLK_to_32KHz_crystal()`. Finally, make sure to change the settings in the terminal application to a baud rate of 4800.

Perform the following:

- Complete the code and demo it to the TA.
- Submit the code in your report.

Student Q&A

1. What's the difference between UART and eUSCI?

2. What is the backchannel UART?
3. What's the function of the two lines of code that have P3SEL1 and P3SEL0?
4. The microcontroller has a clock of 1,000,000 Hz and we want to setup a UART connection at 9600 baud. How do we obtain a clock rate of 9600 Hz? Explain the approach at a high level.
5. A UART transmitter is transmitting data at at 1200 baud. What is receiver's clock frequency if oversampling is not used?
6. A UART transmitter is transmitting data at at 1200 baud. What is receiver's clock frequency if oversampling is used? What's the benefit of oversampling?

Lab 9

Inter-Integrated Circuit (I2C) Communication

In this lab, we will learn using the I2C interface. We will attach the Educational BoosterPack to the LaunchPad board and use the I2C protocol to read measurements from the light sensor.

9.1 I2C Transmission

The I2C communication protocol is based on a bus topology and has two wires, the Serial Data (SDA) and the Serial Clock (SCL). In the bus topology, all the devices plug to the same set of wires, as shown in Figure 9.1. This topology provides flexibility by allowing the addition of extra devices to the bus. Each device on the bus is assigned a 7-bit address so the devices can be distinguished from one another. One of the devices, usually the microcontroller, is designated as the master. All the transmissions are initiated by the master. The master can read from or write to the devices. The master is also responsible for driving the clock signal. The two wires of I2C are pulled-up to high via pull-up resistors. Therefore, they read high if no action is done.

Otherwise, devices must actively pull the wires to low to transmit a value of zero.

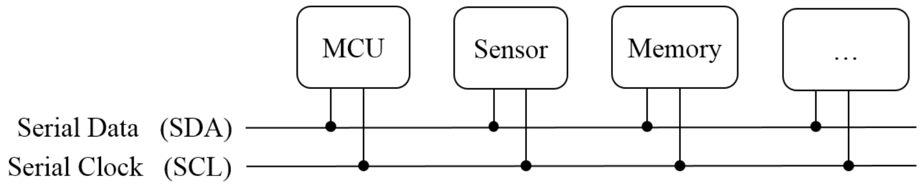


Figure 9.1: The I2C Bus

Figure 9.2 illustrates the read and write operations of the I2C protocol. The master starts and ends its transmission with the Start and Stop signals. These are unique signals that can't occur during the data bits and are used to delimit a transmission exchange.

In the read operation (top part of the figure), the master reads two bytes from the device at address 0x22. The master transmits the Start signal followed by a byte that contains the 7-bit address and the R/W (read/write) bit. The device acknowledges the request. The device then transmits the first byte and the master acknowledges (Ack) it. The device transmits the second byte. Since this is the last byte, the master doesn't acknowledge (Nack) it. Finally, the master transmits a Stop signal. The data received is 0x1234.

In the write operation (bottom part of the figure), the master writes 0xABCD to the device at address 0x33. The master transmits the Start signal followed by the address and the R/W bit. The device acknowledges the request. The master then transmits the data bytes which are acknowledged by the device. The master finally transmits the Stop signal.

```
Master                                                  Device
 ->         ->         <-     <-     ->     <-     ->     ->
Start / Address,R / Ack / Data / Ack / Data / Nack / Stop
          0x22              0x12         0x34


Master                                                  Device
 ->         ->         <-     ->     <-     ->     <-    ->
Start / Address,W / Ack / Data / Ack / Data / Ack / Stop
          0x33              0xAB         0xCD
```

Figure 9.2: I2C read (top) and write (bottom) operations

I2C Device with Internal Registers

A typical I2C device, e.g. a sensor, has multiple internal registers as the example in Figure 9.3 shows. The I2C address of the sensor is 0x22. The sensor has multiple internal registers that we can interact with. Each register has an 8-bit address and has a size of 16-bit. The register 0x50 contains the model number. This register always reads 0x1234 and can be used to test the I2C connection. The second register, 0x60, is used to write the configuration to the sensor since most sensors can operate in multiple modes. The configuration fields and format are found in the sensor's data sheet. Finally, the sensor's register 0x70 contains the result.

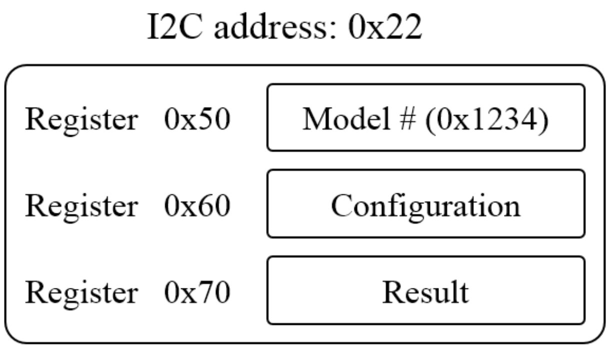


Figure 9.3: An I2C device with internal registers

Figure 9.4 shows the read and write operations based on our example. In the top part of the figure, the master reads the Model # register. This is register 0x50 on I2C address 0x22. The master starts a transmission with address 0x22 as a write operation because, first, the master needs to designate the register number. The following data byte contains the register number (0x50). Once the device acknowledges the byte, the master transmits a Repeated Start signal (a new Start without the Stop signal) to reverse the direction of data. The master repeats the I2C address but now requests a read operation. The device then provides the register's content, 0x1234.

Note that an I2C device usually remembers the internal register the master has read last. For example, to read the internal register 0x50 again, the master can start a read operation from I2C

```
Master                                                            Device
  >      >       <   >    <    >      >       <   <    >   <     >   >
Start/Address,W/Ack/Data/Ack/Start/Address,R/Ack/Data/Ack/Data/Nack/Stop
        0x22        0x50             0x22        0x12     0x34

Master                                       Device
  >      >       <   >    <   >    <   >    <   >
Start/Address,W/Ack/Data/Ack/Data/Ack/Data/Ack/Stop
        0x22        0x60     0x90     0x81
```

Figure 9.4: Read (top) and write (bottom) operations with the device's internal registers

address 0x22 without having to re-specify register 0x50. This results in a shorter exchange as in Figure 9.2. The device's data sheet confirms the availability of this feature.

The bottom part of Figure 9.4 shows the write operation to the configuration register. This is register 0x60 at address 0x22. In the figure, the master writes the configuration value 0x9081. The master starts by writing the device address (0x22), followed by the register address (0x60), and then followed by the configuration value (0x9081). All the operations are writes and, therefore, there is no need to reverse the direction of the data flow as in the previous transmission.

I2C Modes and the eUSCI Module

The I2C protocol supports multiple modes, each designating a maximum clock frequency. The modes as presented in the official I2C specs document[1] and summarized below.

```
Standard Mode         up to 100 KHz
Fast Mode             up to 400 KHz
Fast Mode Plus        up to 1 MHz
High Speed Mode       up to 3.4 MHz
Ultra Fast Mode       up to 5 MHz    (unidirectional)
```

To use a specific mode, both the microcontroller (the master) and the device should support the chosen mode. Most microcontrollers and I2C devices support up to 400 KHz but not all devices support the faster rates since they require advanced circuitry.

According to the FR6xx Family User's Guide `(slau367o)` on p. 819, the eUSCI module supports only the Standard Mode and the Fast Mode. Therefore, the master's clock should be limited to 400 KHz. This clock is setup in the eUSCI module by choosing either of SMCLK or ACLK and then applying a 16-bit divider to get the desired I2C clock frequency.

The Light Sensor

The light sensor on the BoosterPack is the Texas Instruments model OPT3001. The sensor's spectral response closely matches the human eye's since it's designed for user experience applications such as light automation, street lights, traffic lights, cameras and display backlight control. Download the sensor's data sheet, document `(sbos681b)`, since it's needed later in this experiment. You can search for the document's identifier in a search engine.

As presented in the data sheet, the light sensor supports the I2C Standard Mode (100 KHz), Fast Mode (400 KHz) and the High Speed Mode (up to 2.6 MHz). It turns out the sensor also

[1]Document: "UM10204: I2C-bus specification and user manual"

specifies a minimum frequency of 10 KHz. Accordingly, based on the eUSCI module and the sensor's requirements, we can use an I2C clock frequency between 10 KHz and 400 KHz.

The BoosterPack & LaunchPad Setup

In this lab, we'll have the Educational BoosterPack connected to the LaunchPad board. The BoosterPack has a multitude of sensors and I/O devices but doesn't have a microcontroller. Hence, it's used in conjunction with the LaunchPad. Download the BoosterPack User's Guide `(slau599a)`. On p. 5, we can see that the light sensor is connected to the pins of Jumper 1 that are shown below.

```
Serial clock:    I2C_SCL at J1.9
Serial data:     I2C_SDA at J1.10
```

Note the positions of J1.9 and J1.10 as the bottom two pins of the J1 jumper. Next, we'll look at the LaunchPad User's Guide `(slau627a)` on p. 17 for these two jumper pin locations. They correspond to the pins below. I2C is implemented in the Channel B of eUSCI.

```
UCB1SCL / P4.1  -->  eUSCI Module 1 Channel B clock / Port 4.1
UCB1SDA / P4.0  -->  eUSCI Module 1 Channel B data  / Port 4.0
```

By default, these pins act as P4.0/P4.1. To divert these pins to the I2C functionality, we'll look in the chip's data sheet `(slas789c)` on p. 103, where we find the following.

```
P4DIR=xx   P4SEL1=11   P4SEL0=00    LCDSz=00
```

The LCDSz bits are zero by default. The code below diverts the pins to the I2C functionality.

```
// Divert pins to I2C functionality
// (UCB1SDA same as P4.0) (UCB1SCL same as P4.1)
// (P4SEL1=11, P4SEL0=00) (P4DIR=xx)
P4SEL1 |= (BIT1|BIT0);
P4SEL0 &= ~(BIT1|BIT0);
```

eUSCI Module Configuration

At this point, let's configure the eUSCI module for I2C operation. The configuration registers in I2C mode are found in the Family User's Guide (`slau367o`), Chapter 32, starting on p. 841.

The function below performs the configuration. It starts by entering the reset state since the configuration should be modified only in the reset state. The pins are then diverted to the I2C functionality. The function then selects the I2C mode, master mode (the microcontroller is the master of the I2C bus) and SMCLK as the clock source. The default frequency of SMCLK is 1.048576 MHz. The clock is divided by 8, resulting in a frequency of 131.072 KHz. This I2C clock frequency is supported by both eUSCI and the light sensor. Finally, the function exits the reset state so the communication can begin.

```
// Configure eUSCI in I2C master mode
void Initialize_I2C(void) {
  // Enter reset state before the configuration starts...
  UCB1CTLW0 |= UCSWRST;

  // Divert pins to I2C functionality
  P4SEL1 |= (BIT1|BIT0);
  P4SEL0 &= ~(BIT1|BIT0);

  // Keep all the default values except the fields below...
  // (UCMode 3:I2C) (Master Mode) (UCSSEL 1:ACLK, 2,3:SMCLK)
  UCB1CTLW0 |= UCMODE_3 | UCMST | UCSSEL_3;

  // Clock divider = 8  (SMCLK @ 1.048 MHz / 8 = 131 KHz)
  UCB1BRW = 8;

  // Exit the reset mode
  UCB1CTLW0 &= ~UCSWRST;
}
```

The Programming Model

The programming model of I2C consists of initiating the Start and Stop signals and listening to the acknowledgements (Acks) from the device. The procedures are shown in the FR6xx Family User's Guide on p. 824-832. These procedures are implemented by the functions in Appendix A. The function headers are below. Both of these functions return zero when they return successfully.

```
int i2c_read_word(unsigned char i2c_address, unsigned char i2c_reg,
   unsigned int * data);

int i2c_write_word(unsigned char i2c_address, unsigned char i2c_reg,
   unsigned int data);
```

The first function reads two bytes from a register on the I2C device. The sensor's internal registers are 16-bit, hence this function always reads two bytes. The read data is in the third parameter and is passed by reference so the function can fill it up. As an example, the function call below reads two bytes from register 0x50 on I2C address 0x22. The variable 'data' is passed by reference and is filled up by the function.

```
// Reading two bytes from register 0x50 on I2C device 0x22
unsigned int data;
...
i2c_read_word(0x22, 0x50, &data);
```

The second function writes a 16-bit value to a register on the I2C device. As an example, the function call below writes the 16-bit value 0xABCD to register 0x60 on I2C device 0x22. We will use this function to write the configuration to the sensor.

```
// Writing 0xABCD to register 0x50 on I2C device 0x22
unsigned int data = 0xABCD;
...
i2c_write_word(0x22, 0x60, data);
```

Reading the Manufacturer ID and Device ID Registers

To test the I2C configuration, let's read the two internal registers on the sensor called Manufacturer ID and Device ID. These registers always return the same hardwired values and are useful for testing purposes. Browse the sensor's data sheet `(sbos681b)` and locate the table that summarizes all the internal registers, their addresses and their use. Find the addresses of the Manufacturer ID and Device ID registers. Also, find the value that each of these registers returns when read.

Now, we need to find out the I2C address of the light sensor. The light sensor can be configured to one of a few possible addresses using its address pin, therefore, the address depends on how the sensor is wired on the BoosterPack board. Find the sensor's address by looking in the BoosterPack User's Guide `(slau599a)` in the schematics. Also, find the value of the pull-up resistors that were used on the I2C lines. Include a screenshot of the schematics highlighting the I2C address and the resistor values.

Write a code that combines the I2C initialization function above and the read and write functions in Appendix A and perform I2C transmissions that read the Manufacturer ID and Device ID registers. Add the UART functions to your program and transmit the data received from the sensor to the terminal application on the PC so you can observe the values. Read the two registers from the sensor continuously in an infinite loop. Add a delay loop to slow down the readings to about one per second. Also, print an incrementing counter with every reading so you can see that the transmission is ongoing.

Perform the following and submit the answers in your report:

- Write the program and demo it to the TA.
- What is the address of the Manufacturer ID register? What value does this register return? What does this value mean?
- What is the address of the Device ID register? What value does this register return?
- What is the light sensor's I2C address?
- What is the value of the pull-up resistors on the I2C wires? Include a screenshot of the schematics highlighting the I2C address and the pull-up resistors.
- Submit the code in your report.

9.2 Reading Measurements from the Light Sensor

At reset, the light sensor starts in a low-power shutdown state. It was possible to read the Manufacturer ID and Device ID registers in this state. However, to read light measurements, the sensor should be configured first by writing to its configuration register. The sensor measures a lux value, which is the unit of measuring light intensity.

The light sensor returns a 16-bit result that consists of a 4-bit exponent (leftmost bits) and a 12-bit result (called mantissa). The 4-bit exponent specifies the value of the LSB bit. This setup is shown in the data sheet `(sbos681b)` in Table 9. We will set the exponent to 7, which makes the LSB worth 1.28. This means every time the (12-bit) result goes up by 1, the reading (lux value) goes up by 1.28. A 12-bit value varies between 0 and 4,095, which means the lux value varies from 0 to (4,095 x 1.28) 5,241 lux. This is the full range when the exponent is set to 7. This is a reasonable configuration since a well lit room registers a few hundred lux. If we shine a flash light close to the sensor, the lux value can reach the thousands.

As Table 9 shows, it's possible to configure the sensor so the full range is up to 83,865 lux. However, the LSB's worth, therefore the steps, become larger. Note that if the exponent is set to 12, the sensor automatically determines the most suitable exponent and returns it with the result.

Configure the sensor based on the setup below. Find the configuration register's address and layout in the data sheet. Based on the layout, derive the hex value that should be written to the configuration register.

```
RN=0111b=7          The LSB bit is worth 1.28
CT=0                Result produced in 100 ms
M=11b=3             Continuous readings
ME=1                Mask (hide) the Exponent from the result
```

The last field, (Mask Exponent), hides the 4-bit exponent from the result since we know it's equal to 7. We get a 12-bit value from the sensor that we can multiply by 1.28 to get the lux value. The leftmost 4 bits of the result will be zero.

Modify the previous code so that it reads the sensor repeatedly in an infinite loop. The delay loop should pace the readings to about one per second. Transmit the data over UART to the PC. Include an incrementing counter so you can see that the transmission is ongoing.

Interpret the data. A well lit room registers 100 to 200 lux. A desktop lit by a desk lamp registers around 400 lux. Cover the sensor with your finger and observe how the readings approach zero. Alternatively, shine your phone's flash light and observe how the lux value increases as you move the phone closer to the sensor. The reading should max out at 5,241 lux but should not exceed this value. Note that the sensor is more responsive when the light is directed at it straight rather than from the sides.

Perform the following and submit the answers in your report:

- Write the program and demo it to the TA.
- What is the address of the configuration register on the sensor?
- What configuration value (hex) did you write to the sensor? Show how this value is formatted into bit fields.
- Does the data make sense based on what you expected?
- Submit the code in your report.

Student Q&A

1. The light sensor has an address pin that allows customizing the I2C address. How many addresses are possible? What are they and how are they configured? Look in the sensor's data sheet.

2. According to the light sensor's data sheet, what should be the value of the pull-up resistors on the I2C wires? Did the BoosterPack use the same values?

Lab 10

Analog to Digital Converter (ADC)

In this lab, we will learn using the Analog-to-Digital Converter (ADC). The ADC type we'll use is the Successive Approximation Register (SAR) with Charge Redistribution. We'll use the ADC to read the two-dimensional joystick on the Educational BoosterPack.

10.1 Using the ADC SAR-Type

The ADC component converts an analog input signal to a binary number. The input signal, V_{in}, normally falls between the upper and lower reference voltages, V_{R+} and V_{R-}. The result is a binary number, e.g. 10-bit. The specific number of bits depends on the ADC module used and is known as the resolution. The ADC produces a result based on the following equation, where N is the full range value (e.g. for a 10-bit result, N=1,023):

$$Result = N \cdot \frac{V_{in} - V_{R-}}{V_{R+} - V_{R-}}$$

As V_{in} approaches the upper reference V_{R+}, the result approaches the full range value, N. On the other hand, as V_{in} approaches the lower reference V_{R-}, the result approaches zero. It is also typical that if V_{in} exceeds V_{R+}, the result should be the full range value and if V_{in} falls below V_{R-}, the result should be zero.

When V_{R-} is equal to zero (ground), the result equation becomes the following:

$$Result = N \cdot \frac{V_{in}}{V_{R+}}$$

The SAR ADC produces the n-bit result by doing multiple voltage comparisons. The result is produced bit by bit, starting from the Most Significant Bit (MSB). The procedure is shown in Figure 10.1 where the result is 3-bit. Therefore, the result space is 0 to 7.

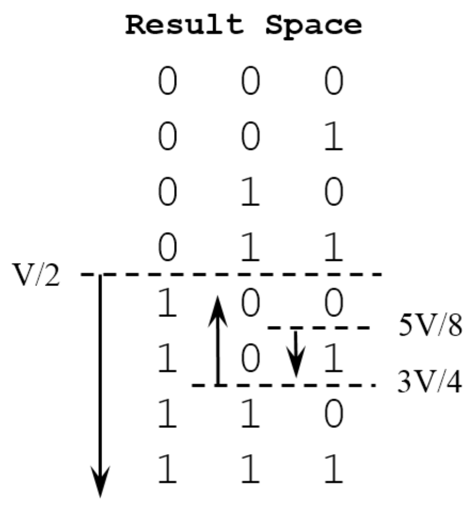


Figure 10.1: ADC conversion with a 3-bit result.

In the figure, the term V designates V_{R+} and we're assuming that $V_{R-} = 0$. To find the MSB of the result, the input voltage V_{in} is compared to the halfway point of the full range $[0, V]$. The halfway point is $\frac{V}{2}$. We find that $V_{in} > \frac{V}{2}$, therefore, the MSB is equal to 1.

For the next bit, we continue with the range $[\frac{V}{2}, V]$ and compare to the new halfway point, $\frac{3V}{4}$. We find that $V_{in} < \frac{3V}{4}$. Therefore, the next bit is 0 and we continue with the interval $[\frac{V}{2}, \frac{3V}{4}]$. The new halfway point is $\frac{5V}{8}$. We do the comparison and we find out that $V_{in} > \frac{5V}{8}$, therefore, the last bit is 1. The result is 101 binary, which is 5.

To do all the possible comparisons, the ADC should be able to generate all the voltage levels in increments of $\frac{V}{8}$ (which are $\frac{V}{8}$, $\frac{2V}{8}$, $\frac{3V}{8}$, ...,$\frac{8V}{8}$,). The voltage levels are generated using an array of capacitors of the values: C, C/2, C/4, C/8 and C/8. The last two capacitors are of the same value so the whole capacitance adds up to 2C. The comparisons are done by a comparator component inside the ADC. For more information on the ADC operation, refer to this document[1].

[1]T. Kugelstadt, "The operation of the SAR-ADC based on charge redistribution", TI doc (slyt176).

An ADC operation consists of the sample-and-hold period followed by the conversion. In the sample-and-hold time, the input signal V_{in} charges the capacitors. The higher V_{in} is, the more the capacitors will be charged, resulting in a larger binary result. The voltage in the capacitors follows the RC charging equation. For an accurate conversion (all bits including the LSB are correct), the capacitors should be charged to within the 1/2 LSB voltage. This way, even the LSB comparison results in the correct bit. The minimum sample-and-hold time is found with this equation:

$$t \geq (R_I + R_S) \,.\, (C_I + C_{pext}) \,.\, ln(2^{n+2}) \tag{10.1}$$

The terms R_I and C_I refer to the ADC's internal resistance and capacitance. These can be found in the ADC's data sheet (or in the microcontroller's data sheet when the ADC is built-in the microcontroller). The term R_S is the analog signal's resistance and the term C_{pext} is the analog signal's (external) parasitic capacitance. The term n is the number of bits in the result.

Computing the Sample-and-Hold Time

Let's compute the sample-and-hold time that corresponds to the joystick's analog signals. The ADC we'll use is the ADC12_B module that's built in the microcontroller and produces a 12-bit result. Find the values of R_I and C_I that correspond to the ADC by looking in the microcontroller's data sheet `(slas789c)`. If these values have a range, decide whether you should go with the lower or upper values. Explain your decision. For the joystick, use $R_S = 10\ k\Omega$ and $C_{pext} = 1\ pF$. Plug the numbers in Equation 10.1 and compute the minimum sample-and-hold time. You should receive a value that's around $3\ \mu s$. Show how you obtained your answer.

The ADC12_B module uses a clock signal to time the sample-and-hold time and to do the conversion. The ADC's clock frequency should be in the range of [0.45 to 5.4] MHz. The clock signal that's typically used with the ADC is MODOSC (Module Oscillator). It is a built-in RC clock that runs at a nominal frequency of 4.8 MHz and has an error variation in the range of [4 to 5.4] MHz. Inside the ADC, the clock signal can be divided by either of the dividers (1, 2, 3, 4, 5, 6, 7, 8). Starting with MODOSC and dividing it by the largest divider, 8, yields a clock frequency in the range [0.5 to 0.675] MHz, which still meets the requirements of the ADC. Therefore, we can freely use any of the dividers on MODOSC.

When multiple configurations are possible, a lower clock frequency reduces the ADC's power consumption but makes the conversion slower. The 12-bit conversion in the ADC12_B module always takes 14 cycles (not including the sample-and-hold time). On the other hand, using a higher frequency increases the power consumption but performs the conversion faster.

The sample-and-hold period is configured based on a number of cycles of the ADC's clock signal. The number of cycles is any number from the list below. Since the sample-and-hold time is a lowerbound, if we found that we need 100 cycles, we should go to the next value in the list, which is 128 cycles.

```
 Cycles:  4, 8, 16, 32, 64, 96, 128, 192, 256, 384, 512
```

Knowing the sample-and-hold time and the ADC's clock frequency range, we can determine the number of cycles for the sample-and-hold time. We should base the computation on the highest possible frequency since the sample-and-hold time is a minimum duration. For example, if the ADC's clock is in the range [4 to 5.4] MHz, we should base the computation on 5.4 MHz. If the frequency drifts to the lower range, the minimum sample-and-hold time is still met.

To sum up, compute the sample-and-hold time. Then, use MODOSC as the clock signal. If you're aiming to reduce the power consumption, divide by the largest divider, 8, and try finding a suitable number of cycles from the list above. If you're aiming at the fastest conversion possible, divide the clock by 1 and try finding a suitable number of cycles from the list. Indicate which approach you're using and show how you obtained your result.

The Joystick's Setup

Let's find the joystick's pin mapping. The 40-pin interface of the BoosterPack is divided into four rows (jumpers) with 10 pins each. Reading in the BoosterPack User's Guide `(slau599a)`, we find that the joystick's horizontal signal is mapped as below. Confirm the pin is correct and find the pin of the vertical axis since you'll need it in the next part.

```
HOR(X): J1.2 --> Horizontal direction:  Jumper 1 pin 2
```

Looking in the LaunchPad User's Guide `(slau627a)` on p. 17, we can find the microcontroller pin that corresponds to J1.2 as shown below. Confirm this is the correct pin and find the mapping for the vertical signal's pin.

```
Horizontal J1.2:  A10/P9.2 --> Analog input 10 / Port 9.2
```

The default functionality of the pins is GPIO. This means that the pin of A10/P9.2 acts as P9.2 by default. To divert the pin to the A10 functionality, we can look in the MSP430FR6989 data sheet `(slas789c)` (starting on p. 95). For the horizontal signal, we find the following. Confirm this is correct and find the setup for the vertical direction signal.

```
A10 functionality:  P9DIR=x, P9SEL1=1, P9SEL0=1
```

This is the code that diverts the pin of A10/P9.2 to the A10 functionality. Write the code that diverts the vertical signal's pin to the analog functionality since you'll need it in the next part.

```
// X-axis: A10/P9.2, for A10 (P9DIR=x, P9SEL1=1, P9SEL0=1)
P9SEL1 |= BIT2;
P9SEL0 |= BIT2;
```

The ADC12_B Module

At this point, we are ready to configure the ADC12_B module and read the joystick's horizontal direction. The configuration registers of the ADC12_B module are presented in the FR6xx Family User's Guide (`slau367o`) starting on p. 890. Browse these pages to make yourself familiar with the registers and their content. We are especially interested in the four registers ADC12CTL0, ADC12CTL1, ADC12CTL2 and ADC12CTL3. Notice that the tables show the default values that are loaded on reset.

The ADC12_B module can be setup so that it converts 32 analog inputs (called input channels) and drops the results in 32 registers called ADC12MEMx (x:0-31). Each of the result registers has its own configuration register called ADC12MCTLx (x:0-31). In this part, we'll configure ADC12MCTL0 for the horizontal axis and get the result in ADC12MEM0. In the next part, we'll also configure ADC12MCTL1 for the vertical axis and get the second result in ADC12MEM1.

Below is the ADC initialization function. Complete the missing parts. The code below shows the fields that require your attention. The fields that are not mentioned in the function should be left at default values. For the requested settings below, check each field's default value (starting on p. 890). If the requested value is the default, you can leave it unchanged.

```
void Initialize_ADC() {
  // Divert the pins to analog functionality
  // X-axis: A10/P9.2, for A10 (P9DIR=x, P9SEL1=1, P9SEL0=1)
  P9SEL1 |= BIT2;
  P9SEL0 |= BIT2;

  // Turn on the ADC module
  ADC12CTL0 |= ADC12ON;

  // Turn off ENC (Enable Conversion) bit while modifying the
     configuration
  ADC12CTL0 &= ~ADC12ENC;

  //*************** ADC12CTL0 ***************
```

```
    // Set ADC12SHT0 (select the number of cycles that you determined)
    ...

    //*************** ADC12CTL1 ***************
    // Set ADC12SHS (select ADC12SC bit as the trigger)
    // Set ADC12SHP bit
    // Set ADC12DIV (select the divider you determined)
    // Set ADC12SSEL (select MODOSC)
    ...

    //*************** ADC12CTL2 ***************
    // Set ADC12RES (select 12-bit resolution)
    // Set ADC12DF (select unsigned binary format)

    //*************** ADC12CTL3 ***************
    // Leave all fields at default values

    //*************** ADC12MCTL0 ***************
    // Set ADC12VRSEL (select VR+=AVCC, VR-=AVSS)
    // Set ADC12INCH (select channel A10)

    // Turn on ENC (Enable Conversion) bit at the end of the
       configuration
    ADC12CTL0 |= ADC12ENC;

    return;
}
```

In the main function, setup an infinite loop that performs a conversion and sends it to the PC via UART. The code in the loop should set the ADC12SC bit to start the conversion. The code should then wait for ADC12BUSY bit to clear, indicating the result is ready. The result is then read from the register ADC12MEM0 and transmitted via UART. Toggle the red LED at the end of the loop to indicate the ongoing activity. Finally, add a delay loop that paces the readings to about one every 0.5 seconds.

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- What are the values of the ADC's R_I and C_I? If these values have a range show the range.

Did you use the lower or upper range of these values? Justify your choice.

- What is the minimum sample-and-hold time? Show how you computed this duration.
- What divider of MODOSC did you use? How many cycles did you set the sample-and-hold time? Show your computation.
- Observe the values returned by the ADC when the joystick is in the center, all the way to the left and all the way to the right. Interpret the results and indicate if they are correct.
- Submit the code in your report.

10.2 Reading the X- and Y- Coordinates of the Joystick

Modify the code of the previous part so that the ADC converts the horizontal and vertical analog signals at every trigger. Incorporate the additions below to the ADC initialization function. Otherwise, the initializations used previously should remain the same.

```
void Initialize_ADC() {
  // Divert the vertical signal's pin to analog functionality
  ...

  //*************** ADC12CTL0 ***************
  // Set the bit ADC12MSC (Multiple Sample and Conversion)
  ...

  //*************** ADC12CTL1 ***************
  // Set ADC12CONSEQ (select sequence-of-channels)
  ...

  //*************** ADC12CTL3 ***************
  // Set ADC12CSTARTADD to 0 (first conversion in ADC12MEM0)
  ...

  //*************** ADC12MCTL1 ***************
  // Set ADC12VRSEL (select VR+=AVCC, VR-=AVSS)
  // Set ADC12INCH (select the analog channel that you found)
  // Set ADC12EOS (last conversion in ADC12MEM1)
  ...
}
```

The ADC module can perform and hold up to 32 results. We want to perform only two conversions at a time, therefore, we need to indicate the start and stop conversions, i.e., from ADC12MEM0 to ADC12MEM1. The field `ADC12CSTARTADD` should be set to zero to indicate the starting point. In `ADC12MCTL1`, we should set the bit `ADC12EOS` (end of sequence) to indicate the stop point. This way, the ADC performs two conversions at every trigger.

In the main function, setting the ADC12SC bit triggers both conversions. Also, waiting for ADC12BUSY to clear indicates that both conversions are ready. The vertical axis result can be found in the register ADC12MEM1. Transmit both results via UART so you can observe them on the PC. Toggle the red LED to indicate the activity and keep the delay loop.

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Your code should transmit the X- and Y-axes readings of the joystick.
- Turn the joystick in the four main dimensions (left, right, top, bottom) and combinations thereof. Interpret the results and indicate if they are correct.
- Submit the code in your report.

Student Q&A

1. How many cycles does it take the ADC to convert a 12-bit result? (look in the configuration register that contains ADC12RES).

2. The conversion time you found in the previous question does not include the sample-and-hold time. Find the total conversion time of your setup (sample-and-hold time and conversion time). Give the total cycles and the duration.

3. In this experiment, we set our reference voltages $V_{R+} = AVCC$ (Analog Vcc) and $V_{R-} = AVSS$ (Analog Vss). What voltage values do these signals have? Look in the MCU data sheet `(slas789c)` in Table 5.3. Assume that Vcc=3.3V and Vss=0.

4. It's possible for the ADC12_B module to use reference voltages that are generated by the module REF_A (Reference_A). What voltage levels does REF_A provide? (look in the FR6xx Family User's Guide on p. 859 and p. 869).

Lab 11

Serial Peripheral Interface (SPI) & LCD Pixel Display

In this lab, we will learn using the Serial Peripheral Interface (SPI) to communicate data. We will also learn using the software stack that enables printing to the LCD pixel display. The LCD pixel display on the Educational BoosterPack is interfaced via SPI. Therefore, the two skills learned in this lab are combined to print to the display.

11.1 Serial Peripheral Interface (SPI)

The SPI interface is a simple communication protocol where a master communicates with one or multiple devices. There are two data wires between the master and the device, hence, the protocol is full-duplex. The master generates the clock signal and transmits it to the device, therefore, the protocol is synchronous. The fourth wire of SPI is the chip select signal. The master activates this signal to enable the device to communicate. When this signal is disabled, the device should place high-impedance on its data out line. In total, the SPI interface uses four wires: Serial Data Out, Serial Data In, Serial Clock and Chip Select.

SPI is not officially considered a standard. Therefore, the terminology may differ from one manufacturer to another. The data transmitted by the master may be called: Serial Out, Serial Data Out, MOSI, SIMO, etc. The other pins are also known by a few name variants.

When the master communicates with multiple devices, one at a time, the chip select activates the selected device. The devices that are not selected place high-impedance on their data out lines so the signals don't clash. In the case where the master is interfaced with one device only, it's possible to activate the device's chip select signal all the time (e.g., by connecting it to low when if it's active low). This saves a pin at the master and is know as the 3-pin SPI.

SPI is implemented with two shift registers (e.g. 8-bit), one at the master and the other at the device. The communication is based on exchanging the contents of the two shift registers. The bit out of the master's shift register goes in the device's shift register and vice versa. When the registers are shifted eight times, the two shift registers' contents are exchanged. Before the registers are shifted, the MSB bit is typically latched into a D flip-flop so it remains stable during the shift operation. Therefore, transmitting a bit consist of latch/shift actions (also known as latch/communicate or update/communicate). These two actions are repeated eight times to exchange the two registers' contents.

Four modes of operation are defined for SPI based on how the clock signal is used (clock polarity) and based on what triggers the latch and communicate actions (clock phase). The possible setups are shown below. The four cases of (polarity/phase) equal to (0/0), (0/1), (1/0), (1/1) are known as SPI modes 0, 1, 2, 3, respectively. Mode 0 is the most popular mode.

```
Polarity 0:  Clock idle at low
Polarity 1:  Clock idle at high
Phase 0:     Latch at trailing edge, communicate at leading edge
Phase 1:     Latch at leading edge, communicate at trailing edge
```

The LCD Display

The LCD display on the Educational BoosterPack is the CrystalFontz CFAF128128B-0145T. The display is 1.45" diagonally, has 128x128 pixel resolution and supports 262K colors. It incorporates a built-in controller, the Sitronix ST7735S and an SPI interface. Printing to the display is done by sending data and command bytes via SPI to the on-board controller which implements the low-level control of the display. A few other pins are used in addition to the SPI interface.

The interface between the display, on one side, and the Educational BoosterPack and the microcontroller, on the other side, is shown in Figure 11.1. On the display's side, the first three pins are the SPI pins. The data out of the display is absent since the display doesn't transmit anything. This is a half-duplex implementation of SPI. The pin SPI3W/SPI4W is connected to Vcc on the BoosterPack, which means the SPI interface has four wires (it uses the Chip Select pin). The Data/Command (DC) pin indicates whether a data byte or a command byte is sent via SPI. The Reset pin is used to reset the display and is active low. It's set to high during normal operation. At start up, a pulse of certain duration should be driven on the Reset pin for the display to power on. Finally, the Backlight pin controls the display's brightness. It's possible to drive a PWM signal on this pin to setup multiple brightness levels.

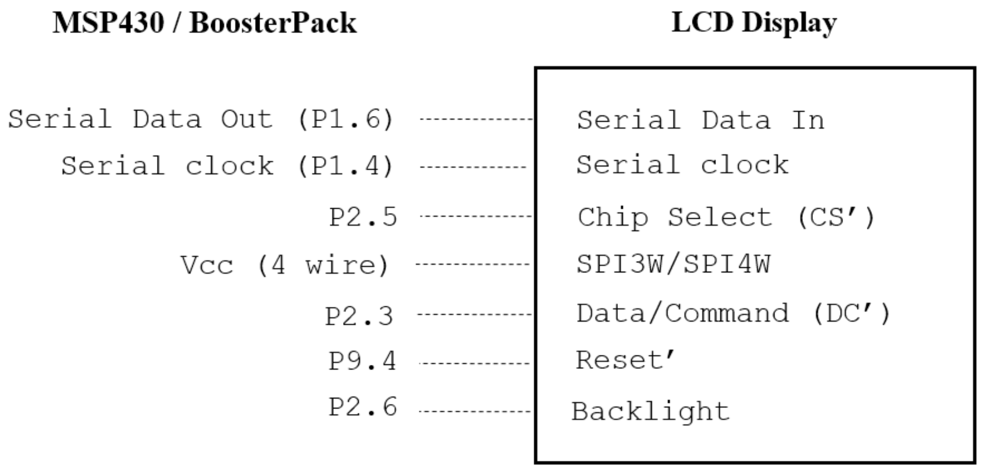


Figure 11.1: Educational BoosterPack Display Interface

The left side of the figure shows the microcontroller/BoosterPack side of the connections. The Serial Data Out and the Serial Clock are connected to the MCU's eUSCI_B0. This is eUSCI module #0 Channel B. Note that the Chip Select pin is connected to a general-purpose pin (P2.5) and not to eUSCI's Chip Select pin. This means the eUSCI module can't drive the chip select pin. To keep things simple, we'll set it to low all the time so the display's SPI interface is always enabled. The remaining three pins (DC, Reset, Backlight) are connected to GPIO pins at the MCU.

SPI Configuration

Download the Code Composer Studio project that is found at the link below. Also, download the graphics library's API documentation that you will use later in this lab. The code in this project prints a welcome message to the LCD display. The code has two functions that you need to write. They configure the pins and the eUSCI module for SPI communication.

http://www.ece.ucf.edu/~zakhia17/EEL4742

In the project, open the file `LcdDriver/middle_driver.c` and write the two functions that are listed below. The first function[1] configures the GPIO pins and the SPI pins. To divert the SPI pins to SPI functionality, look in the microcontroller's data sheet `(slas789c)` starting on p. 95. You should find the corresponding values of PxSEL, like we did in earlier labs. The regular GPIO pins are configured with the variables PxDIR. Note that for SPI, only two pins need to be configured: the pin that transmits data out of the master and the clock pin. The pin that brings data into the master is not used. The chip select pin is not used since the display's chip select is connected to a GPIO pin and we'll set it to low permanently.

```
// Divert and configure the pins
void HAL_LCD_PortInit(void);

// Configure eUSCI for SPI operation
void HAL_LCD_SpiInit(void);
```

The second function configures eUSCI Module #0 Channel B for SPI operation. The configuration registers of eUSCI in SPI mode are found in the FR6xx Family User's Guide `(slau367o)` at the end of Chapter 31. Note that SPI is implemented in both Channels A and B of eUSCI since it's a simple protocol. Since we're using Channel B, use the configuration registers UCBx at the end of the chapter (rather than UCAx).

The main function of the project has a few lines of code that configure SMCLK to 8 MHz. The SPI configuration will guide you to choose the SMCLK clock signal and divide it by 1. Therefore, the SPI clock frequency is 8 MHz.

The comments in the two functions guide you to the configuration that you should use. You can observe the default value of each field (in Chapter 31). If the requested value is the default value, you can leave it unchanged. The settings that are not mentioned in the function can be left unmodified.

Write the two functions and compile the code. The code should print a welcome message to the BoosterPack's LCD display.

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Write the pin configuration and the SPI configuration functions.

[1]The term HAL refers to Hardware Abstraction Layer which is a low-level software that is similar to a driver.

- Which SPI mode does the configuration correspond to?
- Submit the code (of the two functions) in your report.

11.2 Using the Graphics Library

In this part, we will use TI's Graphics Library (grlib) to draw shapes and text on the display. The graphics software stack consists of multiple layers. The lowest layer is the display driver that performs initializations such as driving a pulse on the reset pin to power up the display, setting the display's orientation and translating colors to the display's color format. The driver also provides primitive functions that draw to the display such as pixel draw. In our code, the driver software is implemented in the files `lcd_driver.c` and `middle_driver.c`. The first file performs the driver duties and the second file assists in transmitting the commands and data bytes via SPI.

The next layer in the software stack is the graphics library. It uses the services of the layer below, the driver, and provides services to the layer above, the application code. The graphics library works with any display once the driver is setup. The library provides higher-level functionalities such as drawing a circle, drawing a rectangle, drawing a line, drawing a string and drawing an image.

The library grlib is a small library and is easy to use. Read the main function of the project to see how the graphics context is initialized and how strings are drawn on the display. Also, browse the library's API documentation file to see what functions are available (you can also find the function headers in the file `grlib.h`).

Your goal is to write a demo that demonstrates the graphics library's capabilities. The demo at this video is an example.

`https://youtu.be/ZAwfe9gDpdE`

Your demo should meet the following requirements.

1. Set a new background color
2. Set a new foreground color
3. Draw at least one of each: an outline circle, a filled circle, an outline rectangle, a filled rectangle and a horizontal line.
4. Use at least three different colors on your screen (open the file `grlib.h` in the project to see the available colors)
5. Draw an image on the first screen (use the image in the file `logo.c`)
6. Setup an incrementing 8-bit counter on the second screen (it should continue counting

after rollback to zero)

7. Pushing the button transitions back and forth between the two screens
8. Set the GPIO pin P2.6 to high to engage the highest brightness level
9. Use two fonts on the screen (the project contains fonts in the folder `GrLib/fonts`)

Drawing an image is done using the function `Graphics_drawImage`. It takes as a parameter an object of type `Graphics_Image`, or its alias `tImage`. The file `logo.c` contains an object of this type. This file has been generated by grlib's Image Reformer Tool that converts an image file to a C file. This tool can be obtained by downloading the whole grlib package.

Ensure that your demo prints the incrementing counter correctly when the counter rolls back to zero. When single-digit numbers are printed after a rollback, there should be no remnants of the three-digit numbers. You can add spaces in the printout to take care of this.

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Include pictures of the two screens of your demo in your report.
- Submit the code (of your main.c file) in your report.

Student Q&A

1. Is the SPI implemented as half-duplex or full-duplex in this experiment?

2. What SPI clock frequency did we set up?

3. What is the maximum SPI clock frequency that is supported by the eUSCI module? Look in the microcontroller's data sheet in Table 5-18.

4. Is the display driver software specific to a display model or could it work with any display? Explain.

5. Is the graphics library (e.g. grlib) specific to a display model or could it work with any display? Explain.

Appendix A

I2C Functions

```
//////////////////////////////////////////////////////////////////////
/////////    Function Headers    ////////////////////////////////////
//////////////////////////////////////////////////////////////////////
int i2c_read_word(unsigned char, unsigned char, unsigned int*);   //
int i2c_write_word(unsigned char, unsigned char, unsigned int);   //
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
// Read a word (2 bytes) from I2C (address, register)
int i2c_read_word(unsigned char i2c_address, unsigned char i2c_reg,
   unsigned int * data) {
    unsigned char byte1, byte2;

    // Initialize the bytes to make sure data is received every time
    byte1 = 111;
    byte2 = 111;

    //********** Write Frame #1 ***************************
    UCB1I2CSA = i2c_address;    // Set I2C address
    UCB1IFG &= ~UCTXIFG0;
    UCB1CTLW0 |= UCTR;          // Master writes (R/W bit = Write)
    UCB1CTLW0 |= UCTXSTT;       // Initiate the Start Signal

    while ((UCB1IFG & UCTXIFG0) ==0) {}

    UCB1TXBUF = i2c_reg;        // Byte = register address

    while((UCB1CTLW0 & UCTXSTT)!=0) {}

    if(( UCB1IFG & UCNACKIFG )!=0) return -1;

    UCB1CTLW0 &= ~UCTR;         // Master reads (R/W bit = Read)
    UCB1CTLW0 |= UCTXSTT;       // Initiate a repeated Start Signal
    //****************************************************

    //********** Read Frame #1 ***************************
    while ( (UCB1IFG & UCRXIFG0) == 0) {}
    byte1 = UCB1RXBUF;
    //****************************************************

    //********** Read Frame #2 ***************************
    while((UCB1CTLW0 & UCTXSTT)!=0) {}
    UCB1CTLW0 |= UCTXSTP;       // Setup the Stop Signal

    while ( (UCB1IFG & UCRXIFG0) == 0) {}
    byte2 = UCB1RXBUF;

    while ( (UCB1CTLW0 & UCTXSTP) != 0) {}
    //****************************************************

    // Merge the two received bytes
    *data = (    (byte1 << 8) | (byte2 & 0xFF)    );
    return 0;
}
```

```
/////////////////////////////////////////////////////////////////////
// Write a word (2 bytes) to I2C (address, register)
int i2c_write_word(unsigned char i2c_address, unsigned char i2c_reg,
   unsigned int data) {
    unsigned char byte1, byte2;

    byte1 = (data >> 8) & 0xFF;  // MSByte
    byte2 = data & 0xFF;         // LSByte

    UCB1I2CSA = i2c_address;     // Set I2C address

    UCB1CTLW0 |= UCTR;           // Master writes (R/W bit = Write)
    UCB1CTLW0 |= UCTXSTT;        // Initiate the Start Signal

    while ((UCB1IFG & UCTXIFG0) ==0) {}

    UCB1TXBUF = i2c_reg;        // Byte = register address

    while((UCB1CTLW0 & UCTXSTT)!=0) {}

    //********** Write Byte #1 ***************************
    UCB1TXBUF = byte1;
    while ( (UCB1IFG & UCTXIFG0) == 0) {}

    //********** Write Byte #2 ***************************
    UCB1TXBUF = byte2;
    while ( (UCB1IFG & UCTXIFG0) == 0) {}

    UCB1CTLW0 |= UCTXSTP;
    while ( (UCB1CTLW0 & UCTXSTP) != 0) {}

    return 0;
}
```