UNIVERSITY OF CENTRAL FLORIDA

Lab Manual of **EEL 4742C Embedded Systems**

Department of Electrical and Computer Engineering

Rev. August 2024

Contents

Preface ii

Report Template v

1 Flashing the LEDs 1

2 Using the Push Buttons 10

3 Timer Module 15

4 Interrupts & Low-Power Modes 23

5 LCD Display 34

6 Universal Asynchronous Receiver and Transmitter (UART) 41

7 Inter-Integrated Circuit (I2C) Communication 50

8 Analog to Digital Converter (ADC) 60

9 Serial Peripheral Interface (SPI) &
LCD Pixel Display 69

10 Advanced Timer Features 75

11 Concurrent Event Handling 85

A I2C Functions 89

Preface

The goal of this lab manual is to train undergraduate students in low-power embedded systems. Such systems are based on low-power microcontrollers that draw power in the milliWatt range and are able to run on battery power. Developing embedded systems is a very exciting process since we can see how our code interacts with physical objects and transmit and receive data between multiple chips. This lab manual was developed for the introductory embedded systems course at UCF, EEL 4742C - Embedded Systems. After having taken this course and done the experiments in this manual, the student should be ready for more advanced courses in embedded systems or for applying for entry-level jobs in embedded systems.

Low-power embedded systems can be developed either by using a simple operating system (often called a Real-Time Operating System or RTOS), or in the bare-metal environment in which no OS is used. In this lab, we will write code in the bare-metal environment and this means the programmer is responsible for all the tasks that are happening in the system. The bare-metal approach is more low-level and enables us to gain a deep understanding of how embedded systems work. On the other hand, RTOS-based embedded systems are more suitable when the application becomes large, as it provides thread scheduling, and also helps with porability across different microcontroller platforms.

Hardware

This lab manual is based on the Texas Instruments MSP430 platform. We will use the microcontroller board and the sensor board that are listed below. The sensor board has a variety of components such as a light sensor, a temperature sensor, a pixel LCD display (128x128 pixels) with 18-bit color, a tri-color LED, a microphone, a buzzer, a 3-axis accelerometer and a 2D joystick.

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

Skills

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

- Flashing the LEDs
- Using the push buttons
- Using the timer
- Programming interrupts
- Using the low-power modes
- Using the segmented LCD display
- Performing communication with the UART, I2C and SPI protocols
- Using the Analog-to-Digital converter
- Programming pixel-based LCD graphics
- Using advanced timer features (multiple channels, timer-driven input/output)
- Programming concurrent events with interrupts

Lab Policy

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

- Lab attendance is required.
- All labs are weekly labs.
- The lab work should be demoed to the TA at the end of the session or by the start of the next session. Demoing the code is required for each lab.
- The instructor will provide more details about accessing the lab after hours or about borrowing the lab boards for the duration of the semester.
- A lab report that contains the requested deliverables should be submitted for each experiment. The lab report is due by the start of the next session; seven days after the current session. Use the report template included in this manual.
- The lab report is graded based on correctness, coding style and writing quality.

Coding Style

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

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

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

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

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

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

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

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

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

About the Author

This lab manual was written by Dr. Zakhia (Zak) Abichar who teaches in the computer engineering curriculum at UCF. Dr. Abichar is interested in embedded systems, wireless networks, operating systems and algorithms.

Feedback

If you have feedback on this lab manual, you are welcome to submit your feedback to Dr. Zak Abichar at: zakhia17@ece.ucf.edu

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.

List of Figures

1.1 Creating a project in Code Composer Studio 3
1.2 Building and launching a program. 5
1.3 Checking the variables and registers values in debug mode. 6

5.1 ADKOM FH-1138P LCD Monitor 35
5.2 Alphanumeric font for a 14-segment display. 35
5.3 Segment mapping (LaunchPad User's Guide `(slau627a)` p. 13) 36

6.1 UART Transmission. Data byte: 01010011 42

7.1 The I2C Bus 51
7.2 I2C read (top) and write (bottom) operations 51
7.3 An I2C device with internal registers 52
7.4 Read (top) and write (bottom) operations with the device's internal registers 52

8.1 ADC conversion with a 3-bit resolution. The refernce range is $[0, V]$ and $V_{in} = \frac{5.7}{8}V$. . . 61

9.1 BoosterPack's Pixel Display Interface 71

10.1 Using two channels of Timer_A 76
10.2 Using three channels of Timer_A 79
10.3 Pulse Width Modulation (PWM) Signal 80
10.4 Reset/Set Output Pattern 81

11.1 The compressor is toggled three seconds after the button is pushed 86
11.2 Button debouncing algorithm 87

List of Tables

2.1 Configuration Registers for Port 1 11

3.1 Timer_A Control Register (TACTL) 16

4.1 Timer_A Rollback-to-Zero Interrupt Event 24
4.2 Interrupt Events of Timer_A 27
4.3 Popular Low-Power Modes in MSP430 31

6.1 UART Parameters 42
6.2 UART Registers and Flags 45

10.1 Multiple Interrupt Events of Timer_A 77

Lab 1

Flashing the LEDs

In this lab, we will explore the development environment (Code Composer Studio), the documentation files of the MSP430 boards, and we will write codes that flash the LEDs.

1.1 Documentation

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

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

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

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

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

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

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

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

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

1.2 Flashing the LED

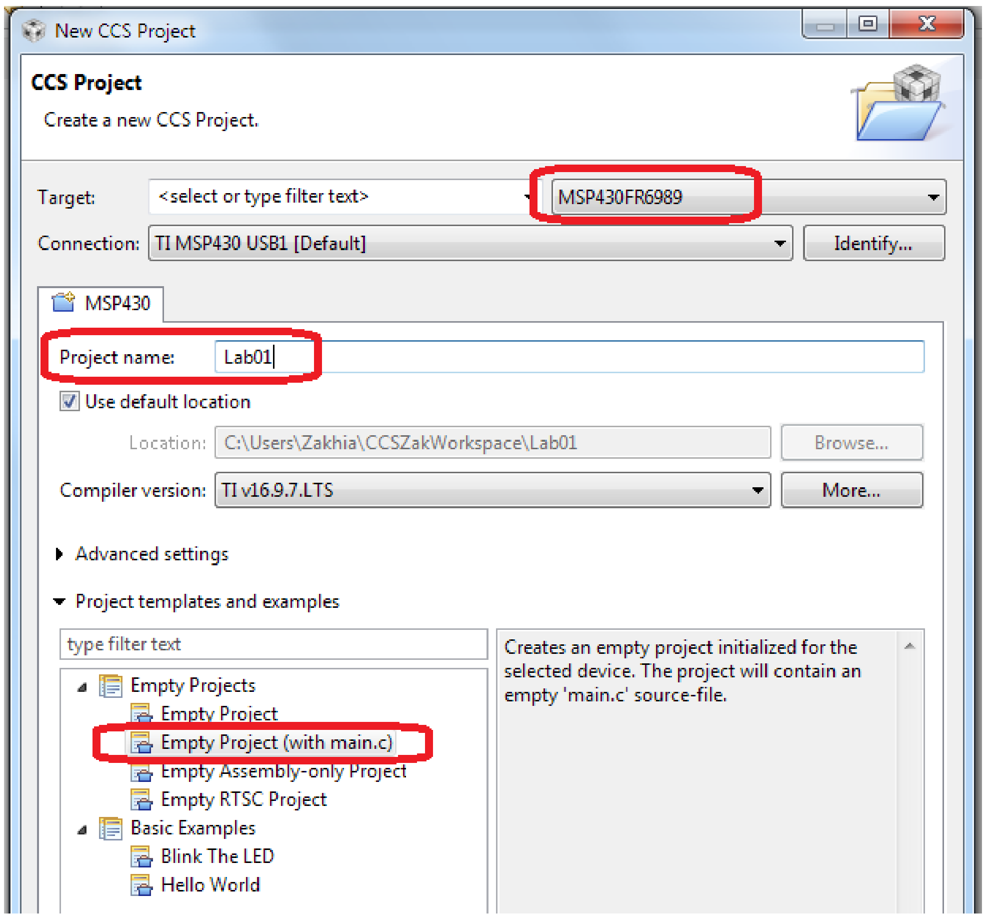
In this part, we will run a code that flashes the red LED on the board. Start Code Composer Studio (CCS) and click on the following menu items.

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

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

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

[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.


New CCS Project

CCS Project

Create a new CCS Project.

Target: <select or type filter text> MSP430FR6989

Connection: TI MSP430 USB1 [Default] Identify...

MSP430

Project name: Lab01

[✓] Use default location

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

Compiler version: TI v16.9.7.LTS More...

▸ Advanced settings

▾ Project templates and examples

type filter text

- Empty Projects
  - Empty Project
  - Empty Project (with main.c)
  - Empty Assembly-only Project
  - Empty RTSC Project
- Basic Examples
  - Blink The LED
  - Hello World

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

concepts in the code.

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

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

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

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

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

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

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

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

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

    P1OUT ^= redLED;              // Toggle the LED
  }

}
```

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

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

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

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

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

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

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

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

Running the Code

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

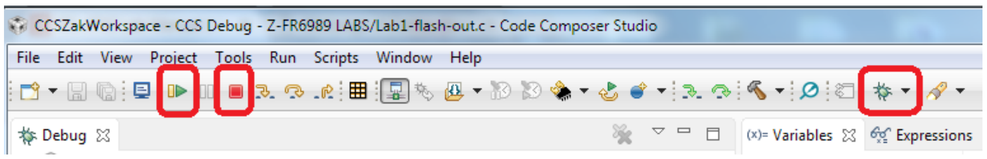
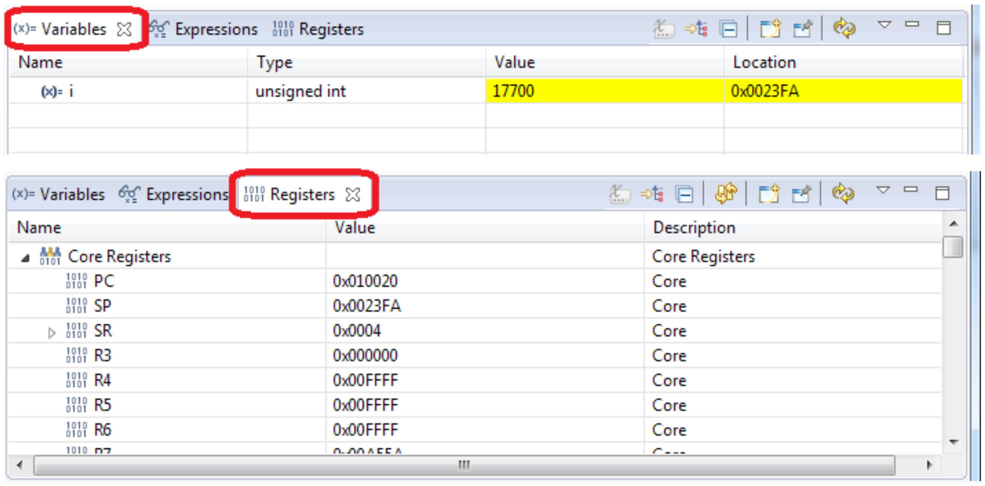

CCSZakWorkspace - CCS Debug - Z-FR6989 LABS/Lab1-flash-out.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
(x)= Variables
Expressions

Figure 1.2: Building 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 board resets and the code restarts. It's running completely off the board and the code is not lost when the power is gone since it's stored in a non-volatile memory.

Perform the following actions:

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

Deliverables:

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

1.3 Flashing two LEDs

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

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

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

Perform the following actions:

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

Deliverables:

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

1.4 Setting a Long Delay

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

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

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

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

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

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

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

Deliverables:

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

1.5 Designing Firefighter Truck LED Patterns

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

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

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

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

Deliverables:

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

Energy Trace Feature

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

```
Tools -> EnergyTrace
```

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

Student Q & A

Submit the answers to the questions in your report.

1. In this lab, we used a delay loop to create a small delay; what is its effect on the battery life if the device is battery operated? Is it a good way of setting delays?
2. The MSP430 CPU clock can be configured to multiple frequencies via software. What happens to the delay generated by the delay loop if the CPU clock frequency is changed?
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 basic 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 tasks and we'll use them in this lab. The button S3 is connected to the reset pin and holds the chip in reset state for as long as it's pushed. The reset state is a state in which the chip doesn't respond to any event. 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 wired in the active low configuration. That is, when they're pushed, they read as zero. Confirm this by looking at the schematics in the LaunchPad User's Guide `(slau627a)` on pages 29-34 and take a screenshot of the schematics portion showing the buttons.

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	Sets pin direction (0: input) (1: output)
P1IN	Reads input pins (0: input is low) (1: input is high)
P1REN	Enables built-in resistors (can be set as pull-up or pull-down) (0: disable resistor) (1: enable resistor)
P1OUT	Writes to output (for output pins) $\rightarrow$ (0: write low) (1: write high) Configures built-in resistors (for input pins) $\rightarrow$ (0: pull-down) (1: pull-up)

A port (e.g. Port 1) has 8 bits and, accordingly, each of the variables shown in the table is 8-bit. 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. The button's pin is connected to ground when the button is pushed. Therefore, we'll have the pin pulled-up to Vcc so it reads high when the button is released.

Masking Operation

To read a 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
```

The masking operation ANDs the data with the mask BIT3. 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 3 is zero
if( (Data & BIT3) == 0 ) ...

// Check if bit 3 is one (two ways)
if( (Data & BIT3) != 0 ) ...
if( (Data & BIT3) == BIT3 ) ...
```

We note that if we check two bits simultaneously in the masking operation, a nonzero result means one of three cases: 01, 10, or 11. However, when we test one bit in the masking operation (like the example above), a non-zero result means the bit is 1.

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.

```
// 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(;;) {
      // Rewrite the pseudocode below into C code
      if ( button S1 is pushed )
          Turn red LED on
          else Turn red LED off
  }

}
```

This code polls the button infinitely and, therefore, keeps the CPU locked to this operation. A downside is that 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 a button interrupt and to put the microcontroller in low-power mode while waiting for the button push. We'll explore this approach in a future lab.

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

- Include a screenshot of the schematics portion showing the buttons' wiring. 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 of the previous part by adding the same functionality on button S2 and the green LED. Therefore, the red LED should light up as long as S1 is held down and the green LED should light up as long as S2 is held down. The code should also work when both buttons are held down, and also when transitioning from one button to the other without lifting both of them.

For this part, perform the following:

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

2.3 Electric Generator Load Control (v1)

In this part, implement an industrial application in which two machine operators turn on their machines using the push buttons. Only one machine can operate at a time since the electric generator can't support the machines running simultaneously. It is your software's responsibility to ensure that the machines don't run simultaneously.

This is how the system works. Operator 1 requests to run his machine by holding button S1 down and, if his request is granted, the red LED turns on to indicate that the machine is running. Similarly, Operator 2 requests to run her machine by holding button S2 down and, if her request is granted, the green LED turns on to indicate that the machine is running. The operators hold their buttons down for as long as they wish to run their machines. The software should implement a first-come first-serve policy and an operator can run his machine for as long as he wishes.

Implement the following policy regarding simultaneous requests. If an operator is running his machine, it's okay for the other operator to make a request, but the current machine that is running continues to run. When the current machine stops, the other machine runs if its request is still pending. Basically, this policy means that the operators are allowed to make simultaneous requests and it's the software's responsibility to ensure that the machines don't run at the same time.

For this part, perform the following:

- Write the code and demo it to the TA.
- Stress test the code by checking all possible scenarios.
- Submit the code in your report.

2.4 Electric Generator Load Control (v2)

Modify the code of the previous part by implementing a stricter policy on simultaneous requests. The idea here is that the operators should never make simultaneous requests. If a machine is running and the other operator makes a request, the running machine stops immediately and no machine can turn on until both operators release their requests (i.e. both buttons are released).

Perform the following:

- Write the code and demo it to the TA.
- Stress test the code by checking all possible scenarios.
- Submit the code in your report.

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. Evaluate this lab's codes regarding power-efficiency if the device is battery operated. Is reading the button via polling power efficient?

Lab 3

Timer Module

In this lab, we will program the microcontroller's timer module (Timer_A) to setup time delays. We will use the timer in two modes known as the continuous mode and the up mode. Our codes 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. The timer module is versatile and supports 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 time 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 in the range [0 - 65,535].

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 (changed to 1) 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. The software acts based on the flag which can be done in two ways. The software can simply poll the flag (read it continuously), which is the approach that we will do in this lab. Alternatively, the software can set up an interrupt mechanism which relieves the burden of having to poll the flag (we'll do this in a future lab). Either way, when the flag is raised, the software takes action (e.g. toggle an LED), and clears the flag so it can be raised again. If the flag is not cleared, it remains high and wouldn't indicate anymore when the timer period has elapsed.

Timer_A Configuration

The configuration registers of any module are found in the respective chapter in the Family User's Guide (`slau367o`) at the end of the chapter. That is, the Family User's Guide has a chapter on Timer_A and the end of the chapter shows all the registers that configure the timer and the possible values they can have. Below, we show the main configuration register of the timer.

Timer_A is started and configured using the TACTL (Timer_A Control) register, which format is shown in Table 3.1. The TASSEL field 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 while SMCLK (Sub Master Clock) is generated from an RC oscillator that's inside the chip and is set to 1.000000 MHz by default. Its frequency can be modified via software. Once a clock signal is selected, it can be divided within the timer module by either 1, 2, 4 or 8, to slow down its frequency. The frequency division is done using the ID field.

Table 3.1: Timer_A Control Register (TACTL)

TASSEL	2-bit	Timer_A Source Select (selects the clock signal) (1: ACLK) (2: SMCLK)
ID	2-bit	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	2-bit	Mode (0: Stop) (1: Up Mode) (2: Continuous Mode)
TACLR	1-bit	Timer_A Clear (sets TAR to 0 when asserted)
TAIFG	1-bit	Timer_A Interrupt Flag Raised when TAR rolls back to zero while counting

The MC (Mode) field is used to select the mode. By default, it's 0 and the timer is stopped so it doesn't draw current. We'll change it to 2 in this section to run the timer in the continuous mode. If the timer is running and we wish to stop it, we can change MC back to 0.

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 set to 1 by the hardware when TAR rolls back to zero while counting. The software will 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 masks that make our job easier. The mask TASSEL_1 has a 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, 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.

When the mask is 1-bit, there's no underscore in the name of the mask. For example, a 1-bit field CAP will have a mask called CAP. But a 2-bit field RES will have masks called RES_0, RES_1, ... RES_3. These masks are defined in the .h file that we included in the code. One great benefit these masks provide is that we don't have to be aware of the bit position of the fields within the register when we are writing code.

As an example, let's configure Timer_A to use SMCLK, divided by 4, continuous mode, and TAR starting at zero. This is done with the following line of code:

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

The line of code above will set TAIFG to zero since it was not mentioned. Since flags are critical to correct operation, sometimes we elect to clear the flag again out of precaution, which is done as the following:

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

We note that when we do `TACTL=...` as in the first line, this intends to modify all the fields in TACTL. Any field not mentioned in the line of code will be set to zero. On the other hand, when we do `TACTL |=...` or `TACTL &=...`, only the fields mentioned will be affected and the other fields remain unchanged.

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. In this experiment, we'll program the Timer0_A, therefore, the variable name in our code will be TA0CTL.

Configuring ACLK to the 32 KHz Crystal

In this experiment, we want to use the ACLK clock based on the 32 KHz (32,768 Hz) crystal. By default, ACLK is configured to a built-in RC oscillator running at a frequency of 4.8 MHz / 128 = 37.5 KHz. We will reroute ACLK to the 32 KHz crystal that's soldered on the LaunchPad board. This is done by calling the function below. Here is how it works. 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 configure 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 configures the pins to the crystal functionality then waits on the fault flags to remain cleared.

```
//**********************************
// 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;
}
```

We note that if we don't call this function in the code, the default ACLK will be 37.5 KHz which can be easily confused with the crystal at 32 KHz.

Flashing the LED

Let's write a code that flashes the red LED based on a delay that's set by the timer. Use the ACLK clock signal and configure it to the 32 KHz crystal, do not apply a divider on the clock, run the timer in the continuous mode and clear TAR at the start. Below is the code template. Write the code.

```
// 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) {
  // Stop the Watchdog timer
  ...

  // Unlock the GPIO pins
  ...

  // Configure the LEDs as output
  ...

  // 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(;;) {
      // Wait in this empty loop for the flag to raise
      while( ... ) {}

      // Do the action here
      ...
  }

}
```

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 to observe and show how you computed the delay.
- Measure the observed delay with your phone's stopwatch for at least 20 seconds. The timing should match closely since the crystal is accurate. Report if it matches.
- Write an analysis showing what delays we expect to observe if the clock is divided by 2, 4 or 8. Test these cases and report if they match what you expected.

3.2 The Up Mode

In the up mode, the upperbound of TAR is set to a value that we choose. TAR counts from zero up to the value of register TACCR0. The timer has multiple channels and each channel has its own register and comparator. The registers of Channels 0, 1, 2 are TACCR0, TACCR1, TACCR2, respectively. When the timer runs in the up mode, the register of Channel 0, TACCR0, serves as the upperbound of TAR. Therefore, the up mode is inherently linked to Channel 0. The timeline below shows how TAR counts in the up mode. When TAR reaches TACCR0, it rolls back to zero and continues counting. Similar to the continuous mode, 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.

Let's write a code that runs the timer in the up mode and toggles the LED every 1 second. Configure the timer so that it uses ACLK based on the 32 KHz crystal, do not apply a divider, select the up mode and a number of cycles that corresponds to a period of 1 second. Note that the crystal's 32 KHz frequency is actually 32,768 Hz.

The lines below show the main differences from the code of the earlier part. We need to set the number of cycles in the up mode. It's a good idea to do this before starting the timer. The remaining parts of the code are similar to the code of the previous part. We should wait on the TAIFG flag to raise when TAR rolls back to zero and then take action.

```
// Set timer period
TA0CCR0 = ...

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

Perform the following and include the answers in your report.

- Write the code and demo it to the TA.
- Show how you computed the value of TA0CCR0.
- Compare the flashing time to your phone's stopwatch for at least 20 seconds and report if the timing matches.
- 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 Application: Signal Repeater

In this part, you will design a signal repeater that samples a pulse signal and replays it. The repeater can be used to combat signal fading when the transmission wire is too long or can be used to communicate between two circuits that cannot be interfaced directly (e.g. different voltage domains). We'll implement the approach of sample-then-replay, which means we're either sampling the input or we're replaying the signal, but not both at the same time.

The pulse signal arrives as input (emulated by the push button) and is transmitted as output on the red LED. The code should measure the duration of the pulse and replays the same duration on the red LED. One requirement is that the code should use the timer to measure the signal's duration and also to replay the signal (i.e. don't use delay loops). A signal duration of up to 65,535 cycles should be supported since we have a 16-bit timer. To save power, the timer should be turned off when it's not used, i.e., while we are waiting for the signal to arrive.

If the pulse duration exceeds 64K cycles, the green LED should turn on to indicate a state of error. The system stays in this state until button S2 is pushed. This would clear the green LED and the system is ready to sample a new signal.

Setup your code using ACLK based on the 32 KHz crystal and apply a divider of 1. Compute the largest delay that can be supported and test it.

The video at the link below shows a demo of the pulse repeater:

https://youtu.be/ZJux_cjxh30

Perform the following:

- Compute the maximum pulse delay that our code supports for all cases of the divider (1, 2, 4, 8) based on the 32 KHz crystal.
- Explain what the tradeoff is when we change between dividers.
- How can this code be modified to measure a pulse of any duration? (Hint: this can be done even when multiple rollbacks occur).

- Demo your programs and submit your C code

Student Q&A

Submit the answers to the questions in your report.

1. So far, we have seen two ways of timing 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 with the timer.
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 TAIFG go to 1?
5. 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?
6. In this lab, you were given a summary of the timer's main control register, TACTL, and the fields within. To practice reading the documentation, find this information in the Family User's Guide `(slau367o)` document and include a screenshot of TACTL's layout and the table describing the fields within it. This information can be found at the end of the Timer_A chapter in the Family User's Guide.

Lab 4

Interrupts & Low-Power Modes

In this lab, we will learn programming interrupts and using the low-power modes. We will apply these mechanisms to the timer and to the push buttons.

4.1 Timer's Continuous Mode with Interrupt

In this part, we will program Timer_A in the continuous mode so that it raises periodic interrupts. When we program interrupts, it's not necessary to poll the timer's flag continuously. Instead, when the timer's flag is raised, an interrupt event occurs and the hardware launches a special function that responds to the interrupt.

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 bits (xIE): The microcontroller has multiple events that can be enabled to raise interrupts individually with their corresponding xIE bit. For example, the timer's rollback-to-zero event is configured to raise an interrupt by setting TAIE to 1. Port 1 has eight pins that can be configured to raise interrupts individually with the 8-bit variable P1IE. We note that for an interrupt to be enabled, the Global Interrupt Enable (GIE) bit must be 1 and the event's own interrupt enable (xIE) bit must be 1 as well.

Interrupt Flag bits (xIFG): Each event that can raise an interrupt has an interrupt flag (xIFG) associated with it. For example, when the timer rolls back to zero, the flag TAIFG is set to 1, which then raises an interrupt if TAIE=1. For Port 1, the 8-bit variable P1IFG contains eight flags that are set to 1 when the corresponding pin experiences an edge (can be configured to rising edge or falling edge).

We note that, if the interrupt event occurs and its enable bit, xIE, is zero, the flag will raise but no interrupt event occurs. This is what we did in an earlier lab: when the timer rolled back to zero, the flag TAIFG became 1 and we detected this by polling the flag. There was no interrupt since TAIE was zero.

Interrupt Service Routine (ISR): When an interrupt event occurs, the hardware finds and launches a special function, the ISR, that responds to the interrupt. Usually each interrupt event has a corresponding ISR function. However, multiple interrupt events sometimes share the same ISR function. Here is a relevant question: how does the hardware find the ISR that is associated with an interrupt event?

Vector Table: A vector is the start address, or pointer, to an ISR. The vector table contains the addresses of all the ISRs. The vector table's format and location are fixed and provided in the chip's data sheet. Therefore, the hardware performs a lookup in the vector table in order to find the ISRs in the memory. The programmer is responsible for linking the ISRs to vectors so the vector table is filled correctly. This is done with a simple syntax that we'll see below. The idea of vector table allows the ISRs to be scattered around the memory and the hardware can find them quickly by doing a lookup in the vector table (it works like a small phone book that provides the addresses of ISRs).

Table 4.1 summarizes information that's related to the timer's rollback-to-zero interrupt event. In the code, we start by enabling the interrupt event (TAIE=1). We then clear the flag (TAIFG=0) initially. Note that TAIE and TAIFG are two bits located in the timer's configuration register TACTL. We then write the ISR and link its start address to the vector. This event's vector is the timer's A1 vector. We then ensure that the flag is cleared each time the interrupt is processed. If the flag is not cleared, the hardware will raise repetitive interrupts infinitely. We finally enable the global interrupt bit (GIE=1); it's better to do this at the end when everything has been configured.

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

Let's write a code that runs the timer in the continuous mode and configures an interrupt for the rollback-to-zero event. When the interrupt occurs the red LED is toggled. Use the ACLK clock signal and configure it to the 32 KHz crystal using the function we used 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. Therefore, the timer's A1 vector is called **TIMER0_A1_VECTOR** and we will link the ISR to it so the hardware can find the ISR. All the vector names can be found in the .h file (e.g. msp430fr6989.h).

In earlier labs, we used the timer's configuration variable TA0CTL which contains the fields: TASSEL (clock select), ID (clock divider), MC (mode), the interrupt flag (TAIFG) and the interrupt enable bit (TAIE). Therefore, the interrupt-related bits are in this variable.

```
// Timer_A continuous mode, with interrupts, 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;                // Configure pin as output
  P9DIR |= greenLED;              // Configure pin as output
  P1OUT &= ~redLED;               // Turn LED Off
  P9OUT &= ~greenLED;             // Turn LED Off

  // Configure ACLK to the 32 KHz crystal
  ...

  // Configure Timer_A
  // 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() {
  // Interrupt response goes here
  ...
}
```

Let's review the syntax of the ISR. The keyword pragma is used to add a functionality to the compiler. In this case, the start address of the ISR is linked to the vector TIMER0_A1_VECTOR. The vector name should be exactly as spelled in the .h file. The keyword __interrupt indicates to the compiler that this is an ISR. ISRs have a special return procedure in that they restore the PC and the low-power mode (regular functions only restore the PC). The function's name, T0A1_ISR, is not a keyword and can be changed to any name. It's not necessary to write a function header above the main function as this function won't be called from the software.

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Compute the period of interrupts that you should observe.
- Compare the timing to your phone's stopwatch for at least 20 seconds and ensure it matches closely (the crystal is accurate).
- What happens if we don't clear the flag each time an interrupt occurs? 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 run the timer in the up mode and raise periodic interrupts. Before we get in the details, let's look at Table 4.2 which summarizes multiple interrupt events of Timer_A.

The first line in the table corresponds to the rollback-to-zero interrupt event that we used in the earlier part. The following lines show that the timer has multiple channels and each one has an interrupt event called the compare event. For example, Channel 0 has a register called TACCR0. If we set this register to 200, when the counting register, TAR, passes by this value, Channel 0 raises its flag CCIFG (Capture/-Compare Interrupt Flag) and, if CCIE=1 (Capture/Compare Interrupt Enable), an interrupt is raised. Note that CCIE and CCIFG of Channel 0 are located in register TACCTL0, which is Channel 0's configuration register. Similarly, the two other channels have their own compare interrupt events.

Table 4.2: Interrupt Events of Timer_A

Event	Bits	Vector	MSP430 Policy
Rollback-to-zero Channel 1 (TAR=TACCR1) Channel 2 (TAR=TACCR2)	TAIE / TAIFG in TACTL CCIE / CCIFG in TACCTL1 CCIE / CCIFG in TACCTL2	A1	Programmer clears the flag (shared vector)
Channel 0 (TAR=TACCR0)	CCIE / CCIFG in TACCTL0	A0	Hardware clears the flag (non-shared vector)

The table also shows that the rollback-to-zero event and the compare events of Channel 1 and Channel 2 share the timer's A1 vector. It means that these three events are handled in the same ISR. One vector corresponds to one ISR. On the other hand, Channel 0 has its own dedicated vector, the A0 vector, and this means that it has its own ISR. In MSP430, when the vector is not shared the hardware clears the flag. That is, for Channel 0, when the ISR is launched, the hardware clears the flag CCIFG in TACCTL0. For the other events, the programmer is responsible for clearing the flag.

Let's recap on how the up mode works in order to select the interrupt event that we'll use. Below is how TAR counts in the up mode. The upperbound of the up mode is always set by Channel 0's register, TACCR0. At the end of the count, Channel 0's compare event triggers and CCIFG is raised. One cycle later, TAR rolls back to zero and TAIFG is raised. We can use either of these interrupt events. We'll use Channel 0's interrupt event.

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

Our plan is the following. We'll start the timer in the up mode (setting TACCR0), clear the interrupt flag of Channel 0 (CCIFG), enable the interrupt bit of Channel 0 (CCIE) and then enable the global interrupt bit (GIE=1). Finally, we'll write the ISR and link it to the A0 vector; the hardware clears the flag CCIFG each time this function is called since this ISR is not shared with other interrupt events.

Below is the code. Start the timer in the up mode and set a delay of 1 second. Configure Channel 0's interrupt. Use the ACLK clock signal based on the 32 KHz crystal using the function from earlier labs. Remember that 32 KHz is 32,768 Hz. 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
  ...

  // Configure Channel 0 for up mode with interrupts
  TA0CCR0 = ...                  // 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
  ...

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

  for(;;) {}
}

//*******************************
#pragma vector = TIMER0_A0_VECTOR
__interrupt void T0A0_ISR() {
    // Action goes here
    ...
}
```

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 (the crystal is accurate).
- Should we set TAIE to 1 in this code? Explain.
- Should the ISR clear the flag of Channel 0? Explain.
- Modify the code so that the delay is 0.5 seconds (then try 0.1 seconds).

4.3 Push Button with Interrupt

In this part, we will read the push buttons via interrupts and program the following: 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 on our board are in the active low configuration. Therefore, we'll configure a falling edge interrupt so the interrupt occurs as soon the button is pushed. The piece of code below shows how the buttons are configured for interrupts.

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

Since Port 1 is 8-bit and each of its bits can be configured individually, all of the variables above (P1DIR, P1IFG, P1IE...) are 8 bits. Therefore, we do AND and OR operations to configure the pins.

The code should look like the following. Complete the button configuration by following the directions in the comments. 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. There's no need to use the timer in this code.

Then, write the ISR of Port 1 and link it to the correct vector. All the eight interrupt events of Port 1 share the same vector/ISR. The vector name should be something like P1_VECTOR or PORT_1_VECTOR. To find the vector's exact name, look in the file `msp430fr6989.h` and search for the word VECTOR. You can locate this file by searching for it in the folder where CCS is installed, which should like the path below. An easier way is to hover the mouse pointer over the include line in CCS then click the Control button on the keyboard and click the left button the mouse and the file will open.

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

Let's take a closer look at how the ISR of Port 1 works. If any of Port 1 interrupt occurs, i.e. any bit in P1IFG goes to 1, the ISR is called. Since we enabled two interrupt events in Port 1, we should check which bit(s) in P1IFG is set and service the corresponding interrupt. This is done with the two if-statements in the code template below. Since multiple interrupt events share the ISR, it is the programmer's responsibility to clear the flags (bits in P1IFG). If the flags are not cleared, the hardware will raise repetitive interrupts infinitely.

```
#pragma vector = ...              // Write the vector name
__interrupt void Port1_ISR() {
  // Detect button 1 interrupt flag
  if ( ... ) {
     // Button 1 action
     ...
  }

  // Detect button 2 interrupt flag
  if ( ... ) {
     // Button 2 action
     ...
  }
}
```

We note that bouncing can be addressed by implementing a debouncing algorithms (that we'll see in future labs) or by using buttons with debouncing hardware built-in. A simple debouncing approach is to add a delay loop at the end of the ISR to wait out the bounces (do not use that when you report the failure rate of the code).

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.
- Roughly, what is the success rate of this code?
- Submit the code in your report.

4.4 Low-Power Modes

The MSP430 microcontroller supports multiple low-power modes. They work by shutting down more and more components that are not needed in order to save power. The most popular low-power modes are summarized in Table 4.3.

MCLK (Master Clock) is the clock that drives the CPU. SMCLK (SubMaster Clock) and ACLK (Auxiliary Clock) drive peripherals such as the timer. SMCLK is usually a high-frequency clock (e.g. 1 MHz) while ACLK is usually a low-frequency clock (e.g. 32 KHz). The active mode is the default mode and, in this mode, all the clocks are running. Engaging LPM0 shuts down MCLK and, therefore, the CPU is off. However, peripherals can run based on SMCLK or ACLK. LPM3 further shuts down SMCLK, but

Table 4.3: 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

peripherals can run using ACLK. Finally, engaging LPM4 disables all the clock signals.

Here is an interesting question: what happens if the programmer intends to use the timer but mistakenly engages LPM4 which shuts down all the clock signals? Our chip (MSP430FR6989) supports the overriding feature which means, if a low-power mode shuts down a clock but a peripheral requests that clock, that clock will turn on for as long as it's requested. Therefore, if the timer uses ACLK and we mistakenly engage LPM4, ACLK will turn on to run the timer. The overriding feature is enabled by default but can be disabled. Some basic MSP430 chips don't support the overriding feature.

Interrupts and low-power modes go hand-in-hand because, when a low-power mode shuts down the CPU and clock signals, the only way to reactive the CPU and the clock signals is via an interrupt. If we engage a low-power mode without configuring any interrupt event, the system will remain in low-power mode forever (or until a new software is flashed). Therefore, when we engage a low-power mode we configure at least one interrupt event.

The following is the flow of the program when a low-power mode is used. We start by doing initial configurations and configure at least one interrupt event then enter a low-power mode at the end of main. When an interrupt occurs, the hardware saves the state of the CPU, engages the active mode (CPU and all clocks reactivate), then launches the corresponding ISR. The hardware runs the ISR then restores the low-power mode that was active before the interrupt had occurred.

How does the hardware restore the low-power mode after the interrupt is processed? The low-power mode is indicated by four bits in the Status Register (SR). When an interrupt occurs, PC and SR are saved on the stack. The hardware runs the ISR, then restores PC and SR, therefore, restoring the low-power mode.

How does the programmer engage a specific low-power mode? 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. For example, we call `_low_power_mode_0();` to engage LPM0. Note that this function also enables the global interrupts (GIE bit) because interrupts are always used when a low-power mode is engaged. Therefore, there's no need to call `_enable_interrupts();` when this function is used.

```
_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.

4.5 Application: Crawler Guidance System

You are employed at Kennedy Space Center and you are asked to design a guidance system that guides the massive crawlers that transport rockets to launch pads. The guidance system will be operated by a spotter who has visual awareness of the crawler's position. The spotter gives guidance to the driver to move left, right or keep going straight.

The spotter uses two push buttons to request movement to the left or right. The driver sees two LEDs, the red LED indicating a movement to the left is needed and the green LED indicating a movement to the right is needed. The LEDs also indicate the extent to which correction is needed. The LED flashing pattern is the following. Both LEDs flashing at a slow pace indicates the vehicle is spot on and no correction is needed. The red LED may flash at one of three speeds indicating how much correction is needed to the left. Faster flashing indicates more correction is needed. The green LED works similarly by supporting three flashing speeds. In total, there are seven flashing patterns that are represented below where R and G designate Red and Green, respectively. A '+' sign indicates a faster flashing speed, that is, R++ flashes faster than R+ and, similarly, R+ flashes faster than RG. The transitions occur only between adjacent states.

```
R+++     R++     R+     RG     G+     G++     G+++
```

In this code, we are required to use the timer with interrupts to create the flashing. We are also required to use the buttons with interrupt. Polling the timer's flag, polling the buttons or using a delay loop to flash the LEDs is not permitted. One problem we may encounter is button bouncing, where one button push creates two or more interrupts. To combat this, we can add a delay loop at the end of the button ISR to wait out the bounces. A button's bounce duration may last up to about 20 ms. We can use the function `_delay_cycles();` to create a delay. The cycles we request are at 1 MHz, the default CPU clock. This is the only place in the code we're permitted to use a delay loop.

A demo of the application can be found at the video below. You have the flexibility to define the details but your design should largely resemble the demo.

`https://youtu.be/SLjYTANZmO0`

Perform the following:

- Describe your design in the report
- Demo your program to the TA & submit your code.

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 how to use the segmented LCD display.

5.1 Printing on the LCD Display

Our LaunchPad board is equipped with a segmented LCD display, which is the ADKOM model FH-1138P that has 108 segments and its layout is shown in Figure 5.1. Unlike simple colored LEDs, LCD segments can't be controlled by simply writing low/high to turn them off/on, respectively. Liquid crystal burns out if a continuous voltage is applied to it for a long time. Therefore, the LCD display is interfaced via a specialized controller that continually oscillates the signals that are applied to the segments. The controller can be either part of the display module or part of the MCU. In the case of our board, the LCD controller is a module on the MSP430 called the LCD_C module. It is described in the FR6xx Family User's Guide `(slau367o)` in Chapter 36.

When the LCD controller is part of the MCU, this introduces the problem of having too many wires between the LCD display and the microcontroller. Our display has 108 segments. Using a basic configuration called static drive, we would need 108 pins at the MCU to control the segments, which is an excessive number of pins. Therefore, the idea of multiplexing has been devised in which one pin of the

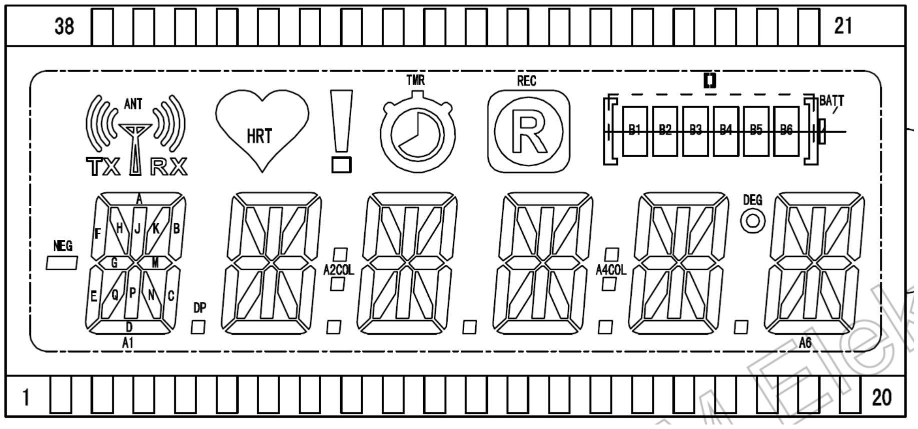


Figure 5.1: ADKOM FH-1138P LCD Monitor

microcontroller controls multiple segments on the display by switching fast between them. With two-way multiplexing, one pin on the MCU controls two segments on the display and, in addition, there are two lines called common backplanes which act as the ground lines. The display on our board is interfaced based on four-way multiplexing. Therefore, the number of pins needed on the MCU is (108/4) 27 and there are four common backplane lines. The LCD controller uses a clock signal in order to generate the waveforms that control the segments. Our configuration will use ACLK based on the 32 KHz crystal.

Segment Memory-Mapping

Our display has six digits and each one is 14-segment, as shown in Figure 5.1. Such 14-segment digits are alphanumeric and can show the whole alphabet. An example font is shown in Figure 5.2.

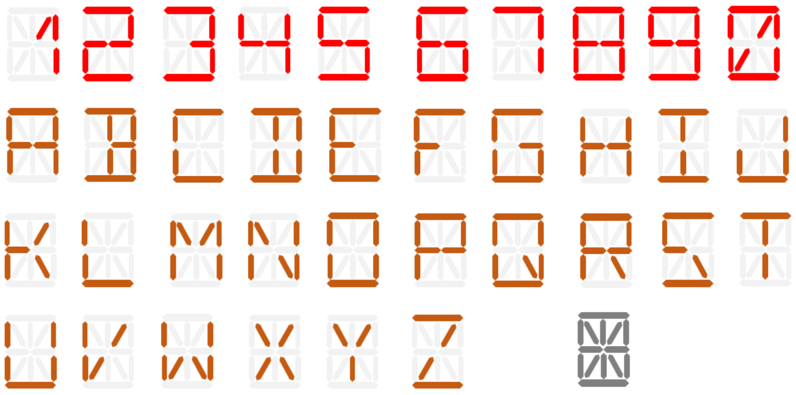
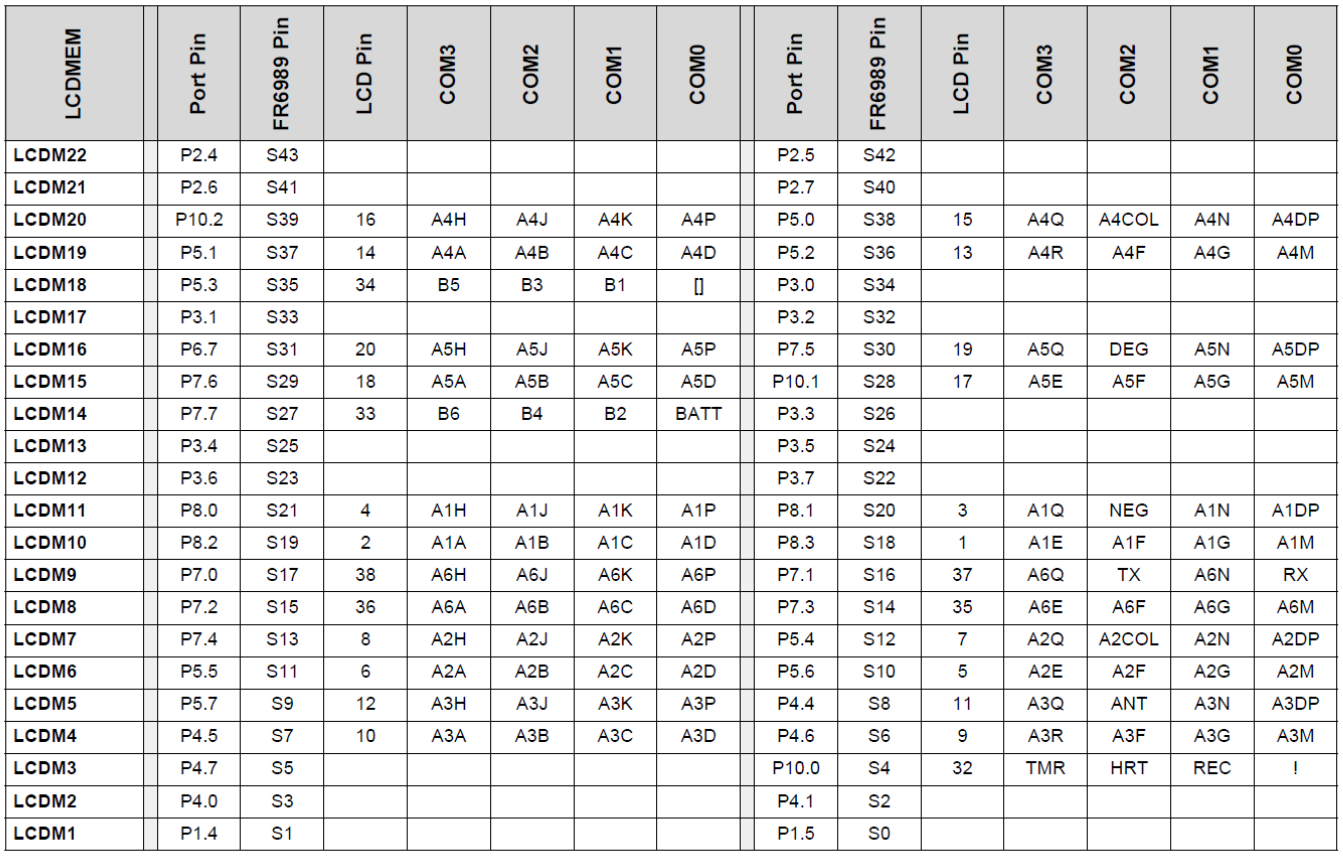


Figure 5.2: Alphanumeric font for a 14-segment display.

The leftmost digit in Figure 5.1 shows how the segments are labeled (A, B, C...). The outer ring and the two middle horizontal bars constitute an 8-segment display that resembles the classic 7-segment display (G and M would be one segment in the 7-segment display). Therefore, if we only want to display numbers, we can use these eight segments and keep the other segments turned off. The leftmost digit on



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

the display is labeled A1 and the rightmost is labeled A6, as shown in Figure 5.1.

In order to program the display, the segments are associated with memory-mapped variables called LCDMx (e.g. LCDM1, LCDM2...) that are shown in Figure 5.3. This figure was obtained from the LaunchPad User's Guide `(slau627a)` as it depends on the wiring between the display and the MCU. Each LCDMx variable is 8-bit and, therefore, maps eight segments on the display. Let's find the mapping of the rightmost digit on the display, A6. We can see that the variable LCDM8 maps the following segments of A6: A, B, C, D, E, F, G and M, in this order, starting from the most significant bit. Therefore, this variable acts like a 7-segment display (technically 8-segment) on the rightmost digit. We can also see that the variable LCDM9 maps the other segments of A6 which are: H, J, K, P, Q and N. It also maps the TX and RX symbols on the display. Accordingly, the table in Figure 5.3 enables us to find the variables LCDMx that map all the segments on the display.

The LCDMx variables are active high. Writing 1 to the bit position turns the corresponding segment on. For example, to display 8 on digit A6, we would need to turn on all the segments in LCDM8. This can be done in the software with: LCDM8 = 0xFF; although it's not a good practice to use hex numbers in the code because they are not easily readable. Instead, we'll store the shapes of the digits in an array.

To facilitate displaying digits (0 to 9) on the display, let's declare an array that stores the shapes of all the digits. The array is shown below:

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

In this array, we store the shape of 0 at index 0, the shape of 1 at index 1, etc. This makes it easy to retrieve the shapes and display them. Let's find the shape of 0. By looking at the layout of LCDM8 in Figure 5.3, we can see that segments A, B, C, D, E and F should be turned on while segments G and M should be turned off. Based on the format of LCDM8, this corresponds to the binary value 1111 1100 or 0xFC. Therefore, we store 0xFC (shape of 0) at index 0 in the array above. Similarly, we found the shapes of 1 and 2 to be 0x60 and 0xDB. Based on the array, we can show the digit 8 on A6 with the following line of code: LCDM8 = LCD_Shapes[8]; This is much more readable than using hex values.

The code below prints the number 430 on the right side of the display (on digits A4, A5, A6). The code contains an LCD initialization function that maps ACLK to the 32 KHz crystal so that it's used by the LCD controller and configures the latter based on 4-way multiplexing. For this code to work, complete the array that stores the shapes of the digits.

```
// 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_Shapes[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_Shapes[4];
    LCDM15 = LCD_Shapes[3];
    LCDM8  = LCD_Shapes[0];

    // Flash the red LED
    for(;;) {
        for(n=0; n<=60000; n++) {} // Delay loop
        P1OUT ^= redLED;
    }

    return 0;
}

//**********************************************************
// Initializes the LCD_C module
// *** Source: Function obtained from MSP430FR6989's Sample Code ***
void Initialize_LCD() {
    PJSEL0 = BIT4 | BIT5;                   // For LFXT

    LCDCPCTL0 = 0xFFD0;
    LCDCPCTL1 = 0xF83F;
    LCDCPCTL2 = 0x00F8;

    // 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 (do this at the end!)
    LCDCCTL0 |= LCDON;

    return;
}
```

Displaying a 16-bit Unsigned Number

Modify the code by writing a function that prints a 16-bit unsigned integer to the display. The function's header is shown below. The parameter is of type 'unsigned int' since this type is 16-bit in the MSP430 compiler. An unsigned 16-bit value goes up to 65,535, therefore at most five digits are needed and can be accommodated on our display.

```
void lcd_write_uint16 (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 and with large values; also test transitions from large values to small values to ensure unused digits are being cleared.
- Submit the code in your report.

5.2 Implementing a Counter

Write a code that shows an incrementing unsigned 16-bit counter on the display (0, 1, 2...). The counter goes up by one every second. Use the timer with interrupts based on the 32 KHz crystal. When button S1 is pushed, the counter goes to zero and continues counting. When button S2 is pushed, the counter jumps up by 1000 and continues counting. Interface the buttons with interrupts.

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 (the crystal is accurate).
- Submit the code in your report.

5.3 Application: Utility Chronometer

You are working as an embedded engineer at a sports equipment company and you are asked to design software that implements a utility chronometer that can be used as either a chronometer or a clock. The chronometer counts seconds, minutes and hours. The application has the following requirements. Pushing S1 starts/stops the timer. Long pressing S1 resets the time to zero and stops the counting. The chronometer can be used as a clock. Pushing S2 fast forwards the time value in order to set the current time. Holding S2 then pushing S1 rewinds the clock backwards (in case we went past the time that we're trying to set). Finally, when the timer is counting, the colon sign between hours and minutes should blink and the chronometer logo should be on. When the timer is stopped, the exclamation point sign on the display should be on. Use the timer with interrupts in order to advance the time. Interface the push buttons with interrupts. For some tasks that happen intermittently (e.g. long press, fast forward and fast rewind), it's acceptable to poll the buttons.

A demo of the utility chronometer can be found at the link below:

https://youtu.be/WflUzULpUeQ

Perform the following:

- Complete the code and demo it to the TA.
- Explain your design.
- What is the maximum duration your chronometer application supports? Explain.
- 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 works just like a colored LED. It's turned on/off by writing either digital high/low to it, respectively".
2. What is the name of the LCD controller that interfaces the LCD display of our board? 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

Universal Asynchronous Receiver and Transmitter (UART)

In this lab, we will learn using the UART interface and program the backchannel UART link that connects our board to the PC.

6.1 Transmitting Data over UART

UART is a simple interface that allows transmitting bytes between two parties in a point-to-point configuration. UART is usually implemented over two wires that carry data in both directions between the two devices. UART can be implemented over one wire but, in this case, the data always travels in the same direction. The former configuration is known as a full-duplex link and the latter is known as a simplex link. UART is considered asynchronous since the two parties are tuned to the same frequency but each one generates its own clock signal; they are not synchronized by the edges of the clock.

UART is a very simple transmission scheme and its frame is shown in Figure 6.1. The line is idle at high. The line drops to low for one bit duration to signal the Start Bit. Typically, the receiver is listening for the occurrence of the Start Bit. 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.

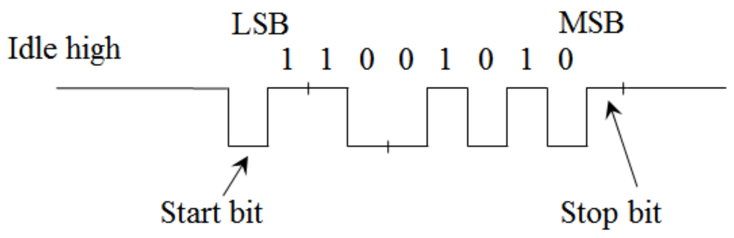


Figure 6.1: UART Transmission. Data byte: 01010011

The bit duration is defined by the baud rate (named after telegraph engineer Emile Baudot) which is simply the transmitter's clock rate. A common rate is 9600 baud which corresponds to a clock frequency of 9600 Hz. This means means that each bit lasts for 1/9600 seconds. Table 6.1 shows the parameters of a UART configuration and highlights the most popular configuration. This is the configuration that we'll use.

Table 6.1: 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

UART reception can be setup in two ways. In the basic approach, the receiver runs a clock at the baud rate (e.g. 9600 Hz) and samples each data bit once. This technique works but doesn't guarantee that the data bit is sampled in the middle of its duration. Therefore, it may not be reliable at high baud rates. A more advance technique is known as oversampling. In this case, the receiver runs a clock at 16x baud rate (e.g. 16x9600 Hz). Now each data bit lasts for 16 cycles at the receiver. The receiver lines up the start of the bit with cycle 1 and uses cycles 7, 8 and 9 to take three samples that are guaranteed to be in the middle of the bit duration. A majority vote is done on the three samples to determine the value of the sampled bit.

The eUSCI Module

Our microcontroller contains a communication module called eUSCI (enhanced Universal Serial Communications Interface) that implements UART and other transmission protocols such as SPI and I2C. The details of UART transmission and reception are handled by the module in the hardware and our code interfaces with the module using a few registers and bits, which makes the programming simple. We will use the eUSCI module in this lab. The module is described in the Family User's Guide `(slau367o)` (Chapter 30).

The eUSCI module is organized in two channels. Channel A supports UART and SPI while Channel B supports I2C and SPI, two communication protocols that we'll encounter in future labs. Therefore, we will use Channel A in this lab. An MSP430 chip may have multiple eUSCI modules to enable independent communication channels. These are usually called eUSCI0, eUSCI1, etc.

The LaunchPad Board Setup

On our LaunchPad board, one of the MCU's UART interfaces is routed to the USB connector so that it can reach the PC. This interface is known as the backchannel UART. When this interface is configured, it creates a COM port on Windows or a 'device' on Linux or MacOS. Any application on the PC that reads serial data can interact with the backchannel UART. We can use a serial application (e.g. TeraTerm or PuTTY), the console in CCS or a C or Python application. In this lab, we'll use a terminal application that shows incoming data and transmits data back to the MCU.

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, let's 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 options:

```
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 is the one on pins P3.4/P3.5 or pins P5.4/P5.5. To find this out, we 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 port.

Pins on the MSP430 chip are typically shared between multiple functionalities. The chip's data sheet `(slas789c)` shows how the pins can be configured 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 configure 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). Therefore, the piece of code below configures the pins to the backchannel UART.

```
// Configure pins to backchannel UART
// Pins: (UCA1TXD / P3.4) (UCA1RXD / 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 LaunchPad board (check the row of jumpers on the board) close the connections between the MSP430 chip and the emulation chip as shown in the figure in `(slau627a)` page 8. If the jumpers are missing, the backchannel UART won't work.

Configuring the UART Clock Frequencies

There is a set of commonly used baud rates which includes the following: 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400 and others. Such clock frequencies are configured in the eUSCI module by starting with a higher frequency and dividing an modulating to obtain the desired frequencies.

For example, let's use SMCLK at 1 MHz (1,000,000 Hz) to configure a UART clock of 9600 Hz. We need to configure the divider and modulator. The divider is simply (1,000,000/9600=104.16) 104. The modulator compensates for the .16 fraction. Modulation is the idea of mixing cycles from two frequencies F1 and F2 and to accomplish an average frequency that's in between. In this case, the hardware configures a clock frequency of 1MHz/104 = 9,615 Hz and another at 1MHz/105 = 9,523 Hz and mixes cycles of the two frequencies to accomplish an average frequency of 9,600 Hz. All this work is done in the eUSCI hardware. What the programmer needs to do is configuring the divider and modulators values in the eUSCI configuration registers.

The eUSCI module supports UART reception with and without oversampling. Therefore, if we're transmitting and receiving UART at 9600 baud, the transmitter clock is 9,600 Hz and the receiver clock is either 9,600 Hz (without oversampling) or 16x9,600 Hz (with oversampling).

The dividers and modulators are obtained from a table in the Family User's Guide `(slau367o)` (page 779). Let's look up the parameters for the configuration that we'll use. Starting with the default SMCLK at 1 MHz (1,000,000 Hz) and aiming at a baud rate of 9600 and using oversampling (UCOS16=1), the dividing and modulating parameters are: UCBR=6, UCBRF=8 and UCBRS=0x20. We'll plug these values in the eUSCI configuration registers in the UART initialization function.

eUSCI Module Configuration

Let's configure the eUSCI module for a baud rate of 9600 based on the popular configuration (shown in Table 6.1) with oversampling reception enabled and using SMCLK=1MHz (1,000,000 Hz). The configuration registers of eUSCI in UART mode are found in the Family User's Guide `(slau367o)` at the end of Chapter 30 (starting on page 783). The eUSCI module is in reset state by default, which means that it's not operational and is not drawing current. The configuration should be modified only when the module is in reset state. Therefore, the initialization function below configures the registers and, finally, exits the reset state.

```
// Configure UART to the popular configuration
// 9600 baud, 8-bit data, LSB first, no parity bits, 1 stop bit
// no flow control, oversampling reception
// Clock: SMCLK @ 1 MHz (1,000,000 Hz)
void Initialize_UART(void){
    // Configure pins to UART functionality
    P3SEL1 &= ~(BIT4|BIT5);
    P3SEL0 |= (BIT4|BIT5);

    // Main configuration register
    UCA1CTLW0 = UCSWRST;    // Engage reset; change all the fields to zero
    // Most fields in this register, when set to zero, correspond to the
    // popular configuration
    UCA1CTLW0 |= UCSSEL_2;  // Set clock to SMCLK

   // Configure the clock dividers and modulators (and enable oversampling)
    UCA1BRW = 6;                // divider
    // Modulators: UCBRF = 8 = 1000 --> UCBRF3 (bit #3)
    // UCBRS = 0x20 = 0010 0000 = UCBRS5 (bit #5)
    UCA1MCTLW = UCBRF3 | UCBRS5 | UCOS16;

    // Exit the reset state
    UCA1CTLW0 &= ~UCSWRST;
}
```

Programming Model

The eUSCI hardware handles UART transmission and reception 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 6.2.

Table 6.2: 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

To make our code more readable, we'll rename the flags and registers with user-friendly names 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. We should not write to the transmit buffer if the transmit flag is zero as this would disrupt the ongoing transmission. When a byte is copied to the transmit buffer, the transmit flag goes to zero and the transmission starts automatically. When the transmission finishes, the transmit flag goes back to one on its own. 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 ) {}

  // Copy the byte to the transmit buffer
  TXBUFFER = ch; // Tx flag goes to 0 and Tx begins!
  return;
}
```

The receive flag becomes 1 when a new data arrives. When the receive buffer is read, the receive flag is cleared automatically. The function below checks if a new byte was received and returns it. If no byte was received, it returns the null character (ASCII=0).

```
// The function returns the byte; if none received, returns null character
unsigned char uart_read_char(void){
  unsigned char temp;

  // Return null character (ASCII=0) if no byte was received
  if( (FLAGS & RXFLAG) == 0)
    return 0;

  // Otherwise, copy the received byte (this 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()`.

The two functions above are based on manually checking the flags. Alternatively, the UART transmit and receive flags can be setup to raise interrupts. The interrupt enable flags can be found in the configuration registers at the end of Chapter 30 in the Family User's Guide.

Testing the UART Transmission

Test the UART transmission by writing a code that transmits the numbers 0 to 9 repetitively over UART. Each number should appear on a new line. Introduce a small delay so the numbers don't go too fast and flash the red LED to indicate the ongoing activity. At the same time, read any incoming data over UART. If the user types 1 on the keyboard, turn the green LED on and if the user types 2, turn the green LED off. Incorporate the three functions above in your code along with the symbolic constants definitions.

A new line can be transmitted by transmitting the line feed (LF) character '\n' followed by the carriage return (CR) character '\r'. The first character brings the cursor one line down and the second character rewinds the cursor to the leftmost column on the screen.

On the PC, open the terminal application and setup the same configuration we did at the MCU as summarized in Table 6.1. On Windows, you can use Device Manager to find the COM port used by the backchannel UART. 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.
- Submit the code in your report.

6.2 Transmitting Integers & Strings over UART

Write two functions that transmit integers and strings over UART. The function headers are shown below. The first function transmits 16-bit unsigned integers. Since bytes transmitted over UART appear as ASCII characters on the PC terminal (e.g. 65 appears as A), the integer on the MCU needs to be parsed into digits. For example, to transmit the integer n=4567, it is parsed into the digits (4,5,6,7) and the ASCII of these digits are transmitted. Test this function for small values and for large values.

```
void uart_write_uint16 (unsigned int n);
void uart_write_string (char * str);
```

The second function is very simple as it transmits each character of the string over UART. The string is null-terminated. These two functions should use the functions that we implemented earlier.

Perform the following:

- Complete the code and demo it to the TA.
- Test the function with small values and large values.
- Submit the code in your report.

6.3 Modifying the UART Configuration

Modify the UART configuration by making the following changes. Use ACLK = 32 KHz instead of SMCLK and setup a baud rate of 4800 baud. Note that in this case we can't do oversampling reception since the 32 KHz frequency is not higher than 16x4800. Therefore, don't check the oversampling bit in the configuration.

This is how we should approach this modification. First, lookup the dividers and modulators from the Family User's Guide (Chapter 30) for the case of ACLK=32,768 Hz and 4800 baud. These are found in the large table and are called: UCBR, UCBRF, UCBRS, UCOS16. Note that UCBRF is not used since we're not doing oversampling reception. Secondly, look at the registers at the end of Chapter 30 and modify our UART initialization function and call it `void Initialize_UART_2()`. Finally, make sure that the settings in the terminal application are modified to match the settings at the MCU.

Perform the following:

- Complete the code and demo it to the TA.
- Test the new configuration by transmitting data to/from the PC (test in both directions).
- Submit the code in your report.

6.4 Application: Airport Runway Control

Many airports have intersecting runways which require careful coordination of takeoffs and landings to avoid simultaneous clearances. As an embedded engineer, you are in charge of designing a system that prevents simultaneous clearances to be granted on intersecting runways.

This is the flow of requests for an airplane to depart on runway 1 that intersects with runway 2. The airplane pilot contacts the tower operator and requests takeoff clearance for runway 1. Since this runway intersects other runways, the tower operator must get a "release" for that runway from the central operator before giving the airplane a clearance. Once the central operator grants the release, the tower operator clears the airplane to takeoff. Once the airplane has departed, the tower operator must contact the central operator and forfeit the release. This enables the central operator to grant another release on the same runway or on an intersecting runway. The main duty of safety falls with the central operator who never grants releases simultaneously on two intersecting runways.

In this application, the tower operators use the PC (keyboard and monitor) to make requests and monitor the current status. Such requests are transmitted over UART to the MCU board. The central operator uses the MCU board LEDs and buttons to grant releases and monitor the current status. Finally, the central operator should have a way of sending an inquiry message to the tower operator if they held to a release for a long time.

A demo of the system can be found at the link below:

https://youtu.be/tdRD_unS3xk

The following commands are helpful in implementing a dashboard on the terminal application. They control to cursor and allow printing on a specific screen location. These commands are transmitted as strings over UART.

```
COMMANDS TO MOVE THE CURSOR
\033[2J    // Clear the whole screen
\033[1;1H  // Move cursor to 1,1 position (top left) format is: line/column
\033[10B   // Move cursor down 10 lines
\033[5A    // Move cursor up 5 lines
```

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 at the frequency of 1,000,000 Hz and we're aiming 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 7

Inter-Integrated Circuit (I2C) Communication

In this lab, we will learn the I2C communication interface and apply it by communicating with the light sensor that's on the on Educational BoosterPack board.

7.1 I2C Transmission

I2C stands for Inter-Integrated Circuit and is designed to communicate between multiple chips over a short distance, e.g. chips that are on the same PCB. I2C has two wires, the Serial Data (SDA) and Serial Clock (SCL) and is based on a bus topology. The term bus means that multiple devices are connected to the same set of wires as shown in Figure 7.1. This topology is flexible since it's possible to add new devices without having to use extra pins at the MCU. Each device on the bus is assigned a 7-bit address so that the devices can be distinguished from one another. One of the devices, usually the MCU, is designated as the master and has two main responsibilities: the master generates the clock signal and shares it with the other devices; secondly, transmissions (read or write) can only be initiated by the master. The two wires of I2C are pulled-up to Vcc 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. I2C is considered to be synchronous since the clock signal is shared between the devices.

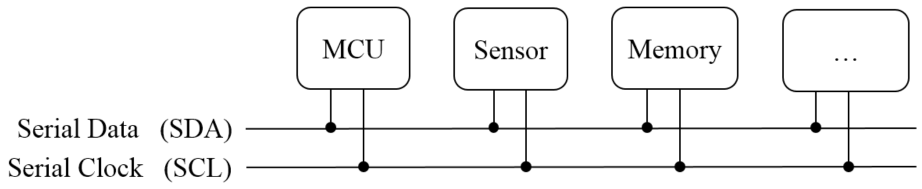


Figure 7.1: The I2C Bus

Figure 7.2 illustrates the read and write operations of the I2C protocol. A frame is delimited by the Start and Stop signals, which have unique patterns that can be distinguished from data bits. The top part of the figure illustrate a read operation in which the master reads two bytes from device address 0x22. The master starts by transmitting the Start signal, followed by a byte that contains the 7-bit address and the R/W' (read/write') bit. This bit is interpreted as 1:read and 0:write. Next, the device acknowledges receiving the request. The device then transmits the first byte (0x12) and the master acknowledges (Ack) it. The device then transmits the second byte (0x34). The master doesn't acknowledge the last byte (Nack = No Ack) so that the device stops transmitting. This enables the master to transmit the Stop signal. The data received is 0x1234 if we consider the first byte to be the most significant byte (MSB).

The bottom part of the figure illustrates a write operation in which the master writes two bytes to device address 0x33. The master starts by transmitting the Start signal, followed by the byte that contains the address and the R/W' bit. Next, the device acknowledges receiving the request. The master then transmits the data bytes which are acknowledged by the device. The master terminates the exchange by transmitting the Stop signal. The data received from the device is 0xABCD if we consider the first byte to be the most significant byte. Note that all the bytes are acknowledged by the device in a write operation.

```
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 7.2: I2C read (top) and write (bottom) operations

I2C Device with Internal Registers

An I2C device, e.g. a sensor, usually has multiple internal registers organized in a layout as shown in Figure 7.3. Let's assume that the device in the figure is a temperature sensor. The sensor's address is 0x22 and the sensor has multiple internal registers. Each register is 16-bit and has an 8-bit address. Register 0x50 contains the model number and always reads as 0x1234. Such a register is non-writable and is used for discovery or testing purposes. Register 0x60 is the configuration register, to which we write in order to configure the sensor. Register 0x70 contains the sensor's result and is read by the MCU. All such details about a sensor are outside the scope of I2C and are described in the sensor's data sheet.

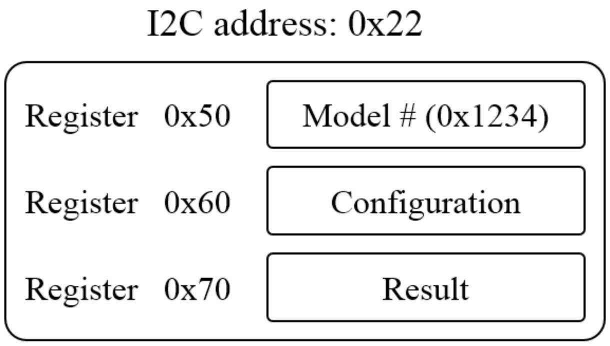


Figure 7.3: An I2C device with internal registers

The updated read and write operations that access the internal registers of an I2C device are shown in Figure 7.4. In the top part of the figure, the master reads the Model # register, which is register 0x50 on I2C address 0x22. The master starts with a write operation because, first, it needs to specify the internal address 0x50. Once the master transmits 0x50 and the device acknowledges it, the master puts a Repeated Start signal (there's no Stop Signal prior), then follows up with a read request to device 0x22. The master then reads two bytes and receives 0x1234.

I2C devices usually remember the last internal register that was read. This means if the master wants to read the Model # register again, it wouldn't have to transmit the value 0x50 again. It can immediately

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

start with a read request and read two bytes. The sensor will provide the content of register 0x50 since this was the last register that was read. If the device supports such a feature, it would be described in its data sheet.

In the bottom part of Figure 7.4, the master writes the value 0x9081 to the configuration register address. In this operation, the master writes three bytes back-to-back. The first byte is interpreted by the device as the internal register. Therefore, the first byte written is 0x60 (designates the configuration register) and it's followed by the two bytes 0x90 and 0x81.

I2C Modes and the eUSCI Module

The I2C protocol supports multiple modes, each designating a maximum clock frequency. The modes are described in the official I2C specs document[1] and are summarized below. In order to use a specific mode, both the master and the device(s) should support it. Most MCUs and I2C devices support the Standard Mode and the Fast Mode but not all devices support speeds beyond that since more sensitive filters in the circuitry would be required.

```
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)
```

The eUSCI module in our MCU supports only the Standard Mode and the Fast Mode as described in the FR6xx Family User's Guide `(slau367o)` on p. 819. Therefore, when using our MCU as I2C master, we should limit the I2C clock frequency to 400 KHz. The I2C clock can be configured by using either SMCLK (e.g. 1 MHz) or ACLK (e.g. 32 KHz) and then applying a divider to it. Modulators are not used with I2C since it's synchronous and, therefore, it's not essential to be very close to the target frequency as it wouldn't affect the correctness of communication. eUSCI supports 'device mode' in which the MCU oeprates as a regular device and responds to an address that we configure in the code.

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 control, traffic lights, cameras and display backlight control. Download the sensor's data sheet by searching for its identifier, `sbos681b`, in a search engine. You will need to use the data sheet later in this lab.

As described 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 requires a minimum

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

I2C frequency of 10 KHz. Therefore, based on joint requirements of the eUSCI module and the sensor, we can configure an I2C clock frequency in the range of 10-400 KHz.

The BoosterPack & LaunchPad Setup

In this lab, we will use the Educational BoosterPack sensor board which contains the TI OPT3001 light sensor and we'll attach it to the LaunchPad board using the 40-pin connector. Make sure the orientation is correct when you connect the two boards. Download the BoosterPack User's Guide `(slau599a)` document and look on p. 5 at the figure illustrating the 40-pin connector. The connector is organized into four rows, called jumpers J1 to J4, and they have a total of 40 pins labelled 1 to 40. The figure on p. 5 shows that the light sensor is connected to the pins of J1 that are shown below.

```
Serial clock (SCL):    I2C_SCL at J1.9
Serial data (SDA):     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 are configured as P4.0/P4.1. To configure 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 configures the pins to the I2C functionality.

```
// Configure 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, we are ready to configure the eUSCI module for operation as I2C master. 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 configuring the I2C pins then engages the reset state of the eUSCI module since the configuration should be modified only in the reset state. The function then selects the I2C mode, master mode and SMCLK as the clock source. The default frequency of SMCLK is 1 MHz (1,000,000 Hz). The clock is divided by 8, resulting in a frequency of 125 KHz, which is supported by both the MCU and the light sensor. The function finally exits the reset state so the communication can begin.

```c
void Initialize_I2C(void) {
    // Configure the MCU in Master mode

    // Configure 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);

    // Enter reset state and set all fields in this register to zero
    UCB1CTLW0 = UCSWRST;

    // Fields that should be nonzero are changed below
    // (Master Mode: UCMST) (I2C mode: UCMODE_3) (Synchronous mode: UCSYNC)
    // (UCSSEL 1:ACLK, 2,3:SMCLK)
    UCB1CTLW0 |= UCMST | UCMODE_3 | UCSYNC | UCSSEL_3;

    // Clock frequency: SMCLK/8 = 1 MHz/8 = 125 KHz
    UCB1BRW = 8;
    // Chip Data Sheet p. 53 (Should be 400 KHz max)

    // Exit the reset mode at the end of the configuration
    UCB1CTLW0 &= ~UCSWRST;
}
```

The Programming Model

The programming model of I2C consists of initiating the Start and Stop signals, listening to the acknowledgements (Acks) from the device and reading or writing bytes from/to the device. The detailed procedures are shown in the FR6xx Family User's Guide on p. 824-832 and are implemented by the functions in Appendix A. These functions assume the setup in Figure 7.3 where the I2C device has internal registers. Each internal register is 16-bit and has an 8-bit address, which is the setup used by the OPT3001 light sensor. Both of these functions return zero when they return successfully.

```c
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 an internal register on the I2C device. The third parameter is passed by reference and will have the data that was read from the device. The code below is an example that reads register 0x50 on I2C address 0x22. The parameter ‘data’ is passed by reference using the & sign.

```
// 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. The code in the example below writes 0xABCD to register 0x60 on I2C device 0x22.

```
// 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

We are now ready to communicate with the light sensor using I2C. You need to start by finding the sensor’s I2C address. As described in the sensor’s data sheet `(sbos681b)`, it’s possible to configure the sensor to multiple addresses using its address pin. In your report, describe what addresses are possible and how the configuration is done. The actual address of the sensor depends on the wiring of the BoosterPack board, therefore, look at the schematics in the BoosterPack User’s Guide `(slau599a)` to find out how the light sensor is wired. Find two pieces of information from the schematic: the I2C address and the value of the pull-up resistors used on the I2C lines. Take a screenshot of the schematics portion showing this information.

The light sensor contains two internal 16-bit registers called Manufacturer ID and Device ID. These registers always return the same values and are helpful for testing purposes. Look in the sensor’s data sheet `(sbos681b)` for a summary of the sensor’s internal registers. You should be able to find the internal (8-bit) addresses of these two registers and the values they’re supposed to return.

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 code 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 and add a delay loop to slow down the readings to about one per second.

Perform the following and submit the answers in your report:

- Write the program and demo it to the TA.
- What possible addresses can the sensor have and how is the address chosen?
- What is the address of the sensor as wired on the BoosterPack board?
- 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.
- What are the addresses of the Device ID and Manufacturer ID registers and what values are they supposed to return?
- Submit the code in your report.

7.2 Reading Measurements from the Light Sensor

At reset, the light sensor starts in a low-power shutdown state so that it draws a very small current. It was possible to read the Manufacturer ID and Device ID registers in this state. However, in order to read light measurements, the sensor should be configured 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
```

When the last field ME (Mask Exponent) is set to 1, the result register will always have zeros in the leftmost 4-bit field that would normally have the exponent field. We chose a fixed exponent of 7 in our configuration, therefore, we know that the 4-bit exponent field will be 7 in every reading. Having this field set to zero makes our job easier since we only need to multiply the result by 1.28 to get the lux value. Otherwise, we would have to shift out the exponent field from the result. Note that if a variable exponent is chosen (where the sensor chooses the exponent with every reading), then it would be necessary to inspect the exponent field in the result in order to determine the multiplier.

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 to check if the readings make sense. A well lit room should register 100 to 200 lux. A desktop lit by a desk lamp should register around 400 lux. Cover the sensor with your finger and check if the readings approach zero. Alternatively, shine your phone's flash light and check if the lux value increases as you move the phone closer to the sensor. The reading should max out at 5,241 lux. The sensor should be 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.
- Do the sensor's readings seem reasonable and consistent?
- Submit the code in your report.

7.3 Application: Lux Logger

As an embedded engineer, you are in charge of implementing a lux logger. A sample output of the application is shown below. The logger takes a sensor reading every minute and prints it to the terminal along with a timestamp. At the start, low/high limit values are established which are -+10 lux from the current reading. When the reading goes out of range, a message of Up or Down is printed and the low/high limits are updated then. Based on the sample below, the first reading is 501 lux, therefore, the low/high limits are set to 491 lux and 511 lux, respectively. When the reading reached 513 lux, the message Up was printed and the low/high limits were updated to 503 lux and 523 lux. Later on, the reading reached 503 lux and the message Down was printed. Finally, implement a feature that enables the end user to set the time. When button S2 is pressed, the user can enter the time as shown in the sample. The keystrokes don't show on the monitor but the MCU confirms the time that was set.

One requirement in this code is to maintain the time using the timer via interrupt and updating the time each second.

```
*** Lux Logger ***

12:00   501 lux   <Up>

Enter the time...(3 or 4 digits then hit Enter)
Time is set to 12:54

12:54   501 lux
12:55   509 lux
12:56   510 lux
12:57   513 lux   <Up>
12:58   513 lux
12:59   514 lux
1:00    514 lux
1:01    510 lux
1:02    506 lux
1:03    503 lux   <Down>
1:04    501 lux
1:05    501 lux
```

Perform the following and submit the answers in your report:

- Write the program and demo it to the TA.
- 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?
3. What I2C clock frequency do each of the eUSCI module and the sensor support?

Lab 8

Analog to Digital Converter (ADC)

In this lab, we will learn using the Analog-to-Digital Converter (ADC) and use it to interface the joystick that's on the Educational BoosterPack. The ADC type we'll use is the Successive Approximation Register (SAR) with Charge Redistribution.

8.1 Using the ADC SAR-Type

The ADC component converts an analog input to a binary number and has multiple applications, e.g. converting a sensor's reading to a digital value. To setup ADC conversions, lower and upper reference voltages are chosen, V_{R-} and V_{R+}, which are expected to be the limits of the input signal we're converting. The ADC converts the analog input, V_{in}, based on where it falls between the lower and upper reference voltages. The ADC's resolution is the number of bits in the result (e.g. 10-bit). The analog input is mapped linearly to the binary result space, as described by the equation below. The term N is the full range (for a 10-bit resolution, N=1,023):

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

Based on the equation, if V_{in} is equal to the lower reference V_{R-}, the result is zero and if V_{in} is equal to the upper reference V_{R+}, the result is N, the full range. Typically, if V_{in} exceeds V_{R+}, the result should max out at the full range value and if V_{in} falls below V_{R-}, the result should be zero.

When V_{R-} is set to ground, which is a popular setup, the equation becomes the following:

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

The SAR ADC produces the n-bit result by comparing the analog input, V_{in}, to multiple voltage levels. Each comparison produces one bit of the result, starting from the Most Significant Bit (MSB). The procedure is shown in Figure 8.1 for a 3-bit resolution, hence, the result space is 0 to 7. Real-life SAR ADCs usually have a resolution of 8-bit or more.

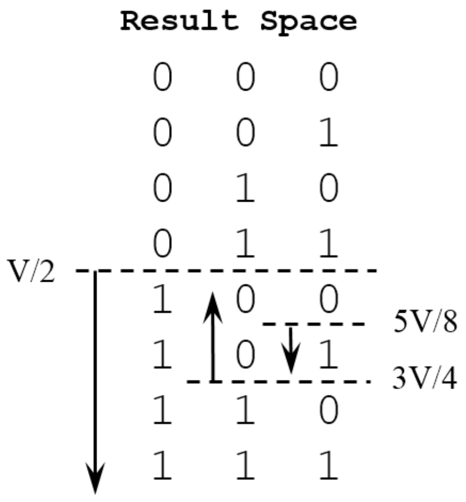


Figure 8.1: ADC conversion with a 3-bit resolution. The refernce range is $[0, V]$ and $V_{in} = \frac{5.7}{8}V$.

In the figure's example, the reference voltage range $[V_{R-}, V_{R+}]$ is represented by $[0, V]$ and the input voltage that we're converting is $\frac{5.7}{8}V$. The input voltage is first compared to the midpoint of the reference voltage range, which is $V/2$, in order to find the MSB. We find that $\frac{5.7}{8}V > V/2$, therefore the MSB is 1. Since the bit is 1, we continue with the upper half of the voltage reference range, which is $[V/2, V]$. We then compare V_{in} to the midpoint of this range and we find that $\frac{5.7}{8}V < \frac{3}{4}V$, therefore, the next bit is 0. Since the bit is 0, we continue with the lower half of the remaining range, which is $[V/2, \frac{3}{4}V]$ and compare V_{in} to the midpoint of this range. We find that $\frac{5.7}{8}V > \frac{5}{8}V$ and, therefore, the LSB is 1. This results in a binary value of 101, which is 5 in decimal.

Here is another example. Let's convert the input voltage $V_{in} = \frac{3.2}{8}V$ where the reference voltage range is $[0, V]$. The first comparison is $\frac{3.2}{8}V < V/2$, which results in the MSB being 0. The next comparison is $\frac{3.2}{8}V > V/4$ and results in the second bit of 1. The last comparison is $\frac{3.2}{8}V > \frac{3}{8}V$ and results in the LSB of 1. The result is 011, which is 3 in decimal.

With a 3-bit resolution, an ADC conversion requires three voltage comparisons. However, to cover all the possible cases, the ADC should be able to setup voltage levels in increments of $\frac{V}{8}$ (which are $\frac{V}{8}$,

$\frac{2V}{8}$, $\frac{3V}{8}$, ..., $\frac{8V}{8}$,). These voltage levels are setup using an array of capacitors with the values: C, C/2, C/4 and C/4. The last two capacitors are of the same value so that the whole capacitance adds up to 2C. The comparisons are done by a comparator component inside the ADC. More information on the operation of the SAR ADC can be found in this document[1].

An SAR ADC conversion consists of the sample-and-hold period followed by the conversion during which the bits of the result are found. During the sample-and-hold time (SHT), the analog input V_{in} charges the ADC's internal capacitors. A higher value of V_{in} charges the capacitors more and results in a larger binary value. During the sample-and-hold time, the voltage across the capacitors follows the RC charging equation. To get an accurate conversion, the error between V_{in} and the capacitors should be less than the 1/2 LSB voltage. For example, with a 3-bit resolution, the MSB is sensitive to an error of $\frac{V_{R+}-V_{R-}}{2}$, the next bit is sensitive to an error of $\frac{V_{R+}-V_{R-}}{4}$ and the LSB is sensitive to an error of $\frac{V_{R+}-V_{R-}}{8}$. Therefore, the error between V_{in} and the capacitors should be $\frac{V_{R+}-V_{R-}}{16}$ or less to ensure that all the bits in the result are correct. In general, for an n-bit resolution, the voltage error should be $\frac{V_{R+}-V_{R-}}{2^{n+1}}$ or less. The equation below provides the minimum sample-and-hold time that satisfies this condition. The SHT time is usually a small value in the microseconds range.

$$t \geq (R_I + R_E) \,.\, (C_I + C_E) \,.\, ln(2^{n+1}) \tag{8.1}$$

The terms R_I and C_I refer to the ADC's internal resistance and capacitance and can be found in the ADC's data sheet (or in the MCU's data sheet in case the ADC is a component in the MCU). The terms R_E and C_E are the external resistance and capacitance of the signal. We often place an RC filter between the analog signal and the ADC since the process of charging the ADC's capacitors causes a drop in the signal. More information on the RC filter design can be found in this document[2].

Computing the Sample-and-Hold Time (SHT)

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 MCU and has a 12-bit resolution. 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 $R_E = 10\ k\Omega$ and $C_E = 1\ pF$. Plug the numbers in Equation 8.1 and compute the minimum sample-and-hold time. You should receive a value that's around 3 μs. Show how you obtained your answer.

The ADC12_B module uses a clock signal to time the sample-and-hold duration and to do the conversion. A 12-bit result is produced in 14 clock cycles (not counting the SHT duration). The ADC works with a clock frequency in the range of [0.45 - 5.4] MHz and we often use the MODOSC (Module Oscillator) clock signal since it's designed to work with the ADC. MODOSC is an RC, built-in, non-configurable

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

[2]R. Pavlanin, "Cookbook for SAR ADC Measurements", Freescale Semiconductor, Application Note, Document AN4373, Apr. 2014.

clock signal with a nominal frequency of 4.8 MHz and drifts in the range [4 - 5.4] MHz due to temperature and voltage variations. Its frequency range fits the ADC clock requirement. The clock signal can be divided inside the ADC by any of these dividers (1, 2, 3, 4, 5, 6, 7, 8). When MODOSC is divided by 8, it corresponds to the range [0.5 - 0.675] MHz, which still fits the ADC. Therefore, any divider of 1 to 8 can be applied on MODOSC inside the ADC.

The SHT duration is set with a number of cycles from the list below based on the clock that's chosen for the ADC. Using MODOSC divided by 1, the clock frequency range is [4 - 5.4] MHz. The SHT duration is converted to a number of cycles based on the highest frequency value in the range, 5.4 MHz, since the SHT is a lowerbound. The ADC conversion will be accurate if the capacitors are charged beyond the minimum SHT but not if the charging duration falls below the derived SHT. By deriving the number of cycles based on the maximum frequency, the charging duration will be longer if the clock drifts to a lower frequency. Once the number of cycles is derived, that number or the next number up in the list is chosen. For example, if the SHT corresponds to 100 cycles, the value 128 is chosen from the list below and is configured in the ADC12_B module.

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

In some cases, it's necessary to apply a divider to the ADC's clock. For example, if the number of SHT cycles is found to be 600 (not supported in the list), a divider of 2 is applied to the clock and the SHT cycles become 300.

Here is a summary of the procedure to setup the joystick's SHT. Compute the sample-and-hold time that corresponds to the joystick using Equation 8.1. Convert the SHT duration to a number of cycles using the MODOSC clock signal. If the number of cycles exceeds 512, a clock divider must be applied. Finally, choose the smallest number from the list that's equal to or higher to the SHT cycles that you computed. The divider and SHT cycles will be entered in the ADC's configuration registers.

The Joystick's Setup

In this part, we will find the joystick's pin mapping and configure the corresponding pins at the MCU to analog inputs. 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 MCU 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 function of the pins is GPIO, which means that pin A10/P9.2 is configured to P9.2 by default. To configure this pin to the A10 function, 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 axis signal.

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

The code below configures pin A10/P9.2 to the A10 functionality. Write the code that configures the vertical axis 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 found 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 in these registers.

The ADC12_B component has one conversion core and 32 result registers. This means we can setup 32 analog inputs (called input channels) that will be converted sequentially on the single core and the results will be placed in 32 result registers, as illustrated below. The results registers are called ADC12MEMx (x: 0-31) and each result register is configured with a configuration register called ADC12MCTLx (x: 0-31). In the example below, the configurator ADC12MCTL0 specifies that the result of analog pin A12 will be placed in ADC12MEM0 (any analog pin can go in any result register). The configurator also specifies the values of V_{R-} and V_{R+} that will be used for this pin, among other things.

```
ANALOG INPUT         RESULT REGISTERS          CONFIGURATION REGISTERS
   Pin A12    -->       ADC12MEM0                    ADC12MCTL0
   Pin A8     -->       ADC12MEM1                    ADC12MCTL1
   ...                  ...                          ...
   Pin A6     -->       ADC12MEM31                   ADC12MCTL31
```

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() {
  // Configure 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 computed)
  ...

  //*************** 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 the result to the PC via UART. The code in the loop should set the ADC12SC bit to start the conversion and wait for ADC12BUSY bit to clear, which indicates that 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 and 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.
- 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 by indicating if they make sense and if there are any blind spots at the edges.
- Submit the code in your report.

8.2 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. Incorporate the additions below to the ADC initialization function. Otherwise, the initializations used previously should remain the same.

```
void Initialize_ADC() {
  // Configure 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)
  ...
}
```

In the function above, we are configuring a sequence of channels so that one trigger of ADC12SC performs two conversions (vertical and horizontal axes) and places the results in ADC12MEM0 and ADC12MEM1. The field `ADC12CSTARTADD` should be set to zero to indicate that the sequence starts at ADC12MEM0. In `ADC12MCTL1`, we should set the bit `ADC12EOS` (end of sequence) to indicate that the sequence ends at ADC12MEM1.

In the main function, setting the ADC12SC bit triggers both conversions and waiting for ADC12BUSY to clear indicates that both results are ready. The vertical axis result can be found in the register ADC12MEM1. Transmit both readings via UART so you can observe them on the PC. Place the code in an infinite loop that's paced by a 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 make sense.
- Submit the code in your report.

8.3 Application: Platform Balancing Control

As an embedded engineer, you are in charge of designing a platform balancing control interface that raises and lowers a platform using the joystick. The interface is shown on the terminal using UART. The platform can be raised and lowered with four motors that are mounted under the platform's four corners. The joystick is used to select one corner at a time and, once a corner is selected, to raise it or lower it. Each corner's position is represented by an 8-bit value that varies in the range 0-255 and corresponds to a unit of millimeter (mm). The default value is 127 mm and from there, each corner can be raised or lowered. Typically, the operator is targeting a requested height, e.g. raising the platform to level 180 mm. One requirement is that the difference between any two corners should never exceed 10 mm, otherwise, a dangerous situation occurs. On the terminal, show a value that indicates the maximum delta between any two corners, and if this value exceeds 10 mm, display a message of "Danger" and turn on the buzzer that's on the BoosterPack.

A video demonstrating the functionality of the application can be found at the link below:

`https://youtu.be/lqh7on4SMoU`

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- 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. 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.

Lab 9

Serial Peripheral Interface (SPI) & LCD Pixel Display

In this lab, we will learn Serial Peripheral Interface (SPI) data communication and use it to interface the pixel display that's on the BoosterPack board. We will also learn how the graphics software stack works and use it to draw on the pixel display.

9.1 Serial Peripheral Interface (SPI)

SPI is a synchronous protocol that transmits data in a full-duplex manner over two data wires. One device in SPI is designated as the master and is responsible for initiating communication and for driving the clock signal. In its basic form, SPI has four wires: MOSI (Master Out Station In), MISO (Master In Station Out), SCLK (Serial Clock) and CS (Chip Select). MOSI pins at the master and device are connected together, and so are MISO pins. The master's SCLK and CS pins are also connected to the same pins at the device.

Each of the master and device has an internal shift register (e.g. 8-bit) and these shift registers are connected into a loop. The output of the master's shift register is tied to the input of the device's shift register and vice versa. The communication starts when the master asserts the CS pin to activate the device (CS is usually active low). The master then drives eight clock cycles on the SCLK pin which exchanges the contents of the master's and the device's shift registers. At the end, the master deasserts the CS pin. In some configurations, the device's CS pin is permanently asserted (e.g. tied to ground since it's active low) and this is referred to as 3-pin SPI.

SPI was introduced by Motorola but is not officially considered to be a standard, therefore, the terminology differs across manufacturers. MISO and MOSI can be referred to as SOMI or SIMO, respectively, or they may simply be called SDO (Serial Data Out) and SDI (Serial Data In) or any other similar names. The CS pin may be referred to as CE (Chip Enable), STE (Station Enable) or other similar names.

In order to make the SPI operation reliable, the output of each shift register is first latched into a D flip-flop that's inside the device. At one clock edge (e.g. falling edge), the bit out of the shift register is latched in the flip-flop and, at the opposite clock edge (e.g. rising edge), the shift registers are shifted. This approach enables the shift registers to read stable data from the flip-flops. The communication between two 8-bit shift registers therefore consists of eight latch/shift operations, which are also referred to as latch/communicate operations.

Four modes of operation are defined in SPI based on whether the clock signal is high or low at idle (clock polarity) and based on what clock edges (rising or falling) trigger 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. The same mode should be used at the master and the device. In practice, 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
```

BoosterPack's Pixel Display

The display module that's on the BoosterPack is the CrystalFontz CFAF128128B-0145T. The display is square, has a diagonal size of 1.45", has 128x128 pixel resolution, and supports 16-bit color for a total of 262,000 colors. The display module has a built-in controller which is the Sitronix ST7735S. The controller incorporates an SPI interface that is used to communicate with the MCU.

The interface between the MCU and the display is shown in Figure 9.1. The display module's pins are shown on the right side and have the following functions. The first three pins are the SPI pins. There's no Serial Data Out (MISO) pin since the display doesn't transmit any data back to the MCU. The pin SPI3W/SPI4W is used to select between 3-pin SPI mode (in this mode, the CS pin is not used) and 4-pin SPI mode. On the BoosterPack, this pin is wired to Vcc which means that the CS pin is used. The Data/Command (DC) pin is used to label each byte that's received via SPI. The software driver sets the

DC pin to 0/1 to indicate that the byte being transmitted over SPI is either a command byte or a data byte, respectively. The Reset pin is used to start up and shut down the display's controller. At startup, the display's controller is not active until the reset pulse, which has specific duration requirements, is applied on the reset pin by the software driver. Finally, the Backlight pin is used to control the display's brightness. Writing low or high to this pin corresponds to the minimum and maximum brightness levels, respectively. Intermediate brightness levels can be implemented by driving a PWM on this pin.

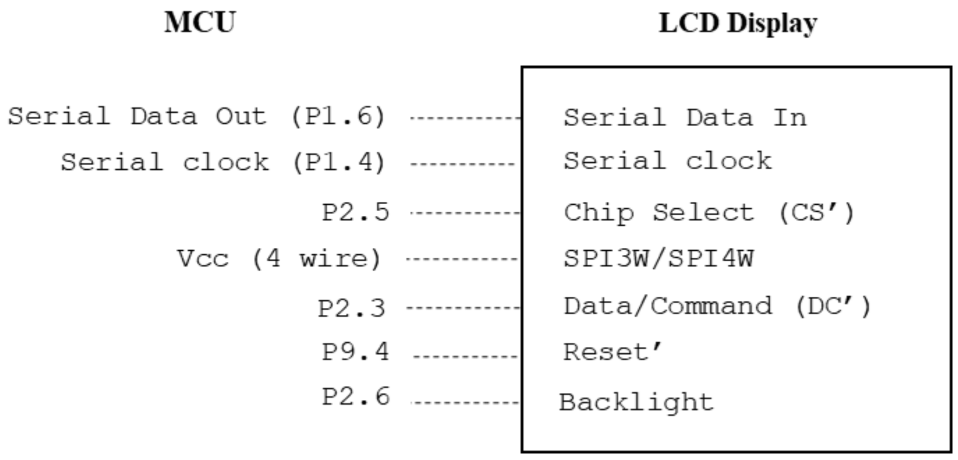


Figure 9.1: BoosterPack's Pixel Display Interface

The left side of the figure shows the MCU side of the interface. These mappings can be found by looking in the BoosterPack User's Guide `(slau599a)` Figure 3 and Table 7 and matching the pins to the LaunchPad User's Guide `(slau267a)` Figure 10 and Figure 3.

On the MCU side, the SPI data and clock lines are connected to eUSCI_B0, which is eUSCI module #0 Channel B. The display's Chip Select (CS) pin is wired to a GPIO on the MCU rather than eUSCI's Chip Select (CS) pin. This means that the eUSCI module cannot control the CS pin. To keep things simple, our code will set this pin to low permanently so that the display's SPI interface is active all the time. The remaining three pins (DC, Reset and Backlight) are connected to GPIO pins at the MCU.

SPI Configuration

We will configure the eUSCI module to setup an SPI link to the display. Download the Code Composer Studio (CCS) project and the documentation that are found at the link below. The CCS project contains the pixel display driver (in the folder `LcdDriver`) and the graphics library (in the folder `GrLib`). The main function prints a welcome message to the display. However, for this code to work, you need to write two functions in the lower driver. These functions configure the pins and setup the SPI interface.

http://www.ece.ucf.edu/~zakhia17/EEL4742C

Open the file `LcdDriver/lower_driver.c` and write the body of the two functions that are listed below. The first function[1] configures the GPIO pins and the SPI pins. By default, pins in MSP430 are configured as GPIOs. To change the functionality of the pins, look in the MCU's data sheet `(slas789c)` starting on p. 95. Find the corresponding values of PxSEL, like we did in earlier labs. 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 and the Chip Select (CS) pin is not used since the display's chip select is connected to a GPIO pin and we'll set it to low permanently.

```
// 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. We're using Channel B, therefore, we'll use the configuration registers UCBx at the end of the chapter (rather than UCAx).

The main file of the project contains a function that configures the clock system to the following. The DCO oscillator is configured to 16 MHz, MCLK is configured to fDCO/1 = 16 MHz (the maximum frequency supported by the CPU) and SMCLK is also configured at fDCO/1 = 16 MHz. It's a good idea to have the CPU running at maximum speed since the graphics library and the display driver may perform a lot of operations when the display is being refreshed frequently. In the SPI configuration function, your goal is to select the SMCLK clock and divide it by 2 so that SPI runs at a clock frequency of 8 MHz. The maximum SPI frequency supported by the display module is 10 MHz.

The comments in the two functions will guide you to doing the configuration. Write the two functions and compile the code. The code should print a welcome message to the display.

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Which SPI mode does the configuration correspond to?
- Submit the code in your report. Submit only the two functions that you wrote.

9.2 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 driver and is implemented in two source files in our code: `lower_driver.c` and `lcd_driver.c`. The first file represents the lower

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

level of the driver and has the task of establishing the physical link to the display, which is the SPI link in our case. The first file implements SPI write functions. The second file implements the general driver tasks such as driving the reset pulse to activate the display and drawing a pixel on any x-y coordinates on the display. The function that draws a pixel on the display accomplishes its task by calling the SPI write functions of the lower driver. In general, each layer in the software stack uses the services of lower layers and provides services to upper layers.

The next layer up in the software stack is the graphics library. The library uses the draw pixel function of the driver and provides intuitive functions that can be used in the application code. Such functions draw circles, rectangles, lines, text and images and enable changing the background and foreground colors. The grlib library that we're using is open source and is included in the project as source files. Browse the library's API documentation file to see what functions are available. You can also browse the file `grlib.h` in the project to see the functions' headers. This file also contains color definitions as 24-bit constants.

To draw a picture on the display, the picture file needs to be converted to a C file using a tool called Image Reformer Tool. This tool is available for download from TI. The resulting C file contains an object of type `Graphics_Image`, or its alias `tImage`. The picture is drawn on the display by passing the image object to the grlib function `Graphics_drawImage`. The project provided to you has a file called `logo.c` that contains an image object.

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 incrementing 8-bit counter that counts up to 255 and rolls back to zero and keeps counting
6. Draw an image on the first screen (you can use the image object in `logo.c` or make a new one)
7. Pushing the button transitions back and forth between the two screens
8. Use two different fonts on the screen (the project contains fonts in the folder `GrLib/fonts`)

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 in your report. Submit only the main.c file.

9.3 Application: Optical Counter

As an embedded engineer, you are in charge of designing an optical counter based on the BoosterPack's light sensor. The optical counter detects objects passing in close proximity to it, e.g. one inch or closer, and counts all such occurrences. Such a counter has many applications in industrial settings (e.g. counting items passing on a conveyor belt), automated environments and sport equipment. The counter should have a minimum accuracy of 90% and should be able to count objects passing at a rate of once per second or slower. The counter should also detect objects that are passing very slowly.

In the application, print the sensor's real-time value on the display and draw a progress bar that reflects the sensor's value. The progress bar should have a full-range of 1000 lux. For example, if the reading is 400 lux, the progress bar should be at 40% full. When the lux value goes beyond 1000 lux, the progress bar should max out at 100%.

The CCS project provided to you in this lab configures SMCLK to 16 MHz, therefore, you need to update your old I2C initialization function by modifying the I2C clock divider. Your old function was based on the default SMCLK frequency of 1 MHz. Modify the clock divider so that it results in an I2C clock frequency of 320 KHz.

A demo of the application can be found at the link below:

`https://youtu.be/7BKjwMoeFvo`

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Submit the code in your report. Submit the main.c file only.

Student Q&A

1. Is SPI implemented as simplex or full-duplex in this experiment?
2. What SPI clock frequency did we set up in this lab?
3. What I2C clock frequency did we set up in this lab?
4. 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.
5. Show how you computed the I2C clock divider in the last part.

Lab 10

Advanced Timer Features

In this lab, we will learn using multiple channels of the timer module that enable timing concurrent durations. We will also learn timer-based output and use it to implement a Pulse Width Modulation (PWM) signal that controls the brightness of the LED. Finally, we will learn using timer-based input to create timestamps on events.

10.1 Timer's Multiple Channels

The Timer_A module has multiple channels that allow timing recurring intervals simultaneously. For example, a Timer_A module with three channels, referred to as Timer_A3, is capable of timing three recurring durations simultaneously based on the one counting register (TAR). A Timer_A module with five channels is referred to as Timer_A5. MSP430 MCUs often have multiple instances of Timer_A with multiple channels each. For example, an MCU may have two modules with three channels each and are referred to as Timer0_A3 and Timer1_A3, one being module #0 and the other module #1. The features of our chip are summarized in the data sheet `(slas789c)` on p. 7. This page shows that our MCU has four Timer_A modules; one module has two channels, two modules have three channels each and one module has five channels.

A duration is timed by using Timer_A's compare events. As an example, let's use Channel 0 to get an interrupt 4000 cycles from now. Channel 0 has a 16-bit register called TACCR0. We write TACCR0=4000-1 and we start TAR at 0. When TAR reaches the value in TACCR0, the compare event of Channel 0 triggers an interrupt. Similarly, we can time simultaneous intervals on Channels 1 and 2 by using their respective registers TACCR1 and TACCR2.

To time simultaneous intervals with multiple channels, the timer module is operated in the continuous mode, in which TAR counts from 0 up to 65,535 (64K). The channels schedule their interrupts by looking ahead from the current value of TAR, as illustrated in Figure 10.1.

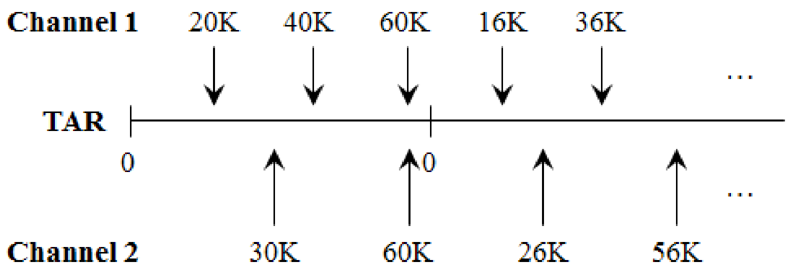


Figure 10.1: Using two channels of Timer_A

In the figure, Channel 1 is scheduling periodic interrupts every 20K cycles (K=1024). Each time an interrupt occurs, the register of Channel 1 is advanced by 20K cycles in order to schedule the next interrupt. What happens after 60K knowing that TAR and the channels' register are 16-bit? When the code adds 20K to Channel 1's register, the following math occurs:

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

The answer is 16K since the hardware is doing a 16-bit addition. It turns out that this answer will result in the correct duration for the next interval. To go from 60K to 16K, 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, which is setting up periodic interrupts every 30K cycles. After 60K, the next interrupt is schedule at: 60K + 30K (on 16-bit) = 90K - 64K = 26K. From 60K, TAR counts up to 64K (that's 4K cycles), then rolls back to zero and counts up to 26K, for a total of 30K cycles.

Based on this approach, the timer can support multiple channels that work simultaneously. Note that Channel 0 is a special channel in the up mode since it sets the upperbound of TAR. However, Channel 0 doesn't have any special responsibility in the continuous mode and can be used interchangeably with the other channels.

Interrupt Events of Timer_A

Timer_A has multiple interrupt events that are summarized in Table 10.1. The first column describes the event and 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. The rollback-to-zero event and Channels 1 and 2 share the A1 vector. Therefore, the interrupt flags of these events are cleared by the software. On the other hand, Channel 0 exclusively uses the A0 vector. Therefore, Channel 0's interrupt flag is cleared by the hardware.

Table 10.1: Multiple Interrupt Events of Timer_A

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

Flashing Two LEDs using Two Channels

Write a code that sets up two periodic interrupts and toggles two LEDs when the interrupts occur. Use ACLK based on the 32 KHz crystal and divide it by four. Set up Channel 0 so that it raises periodic interrupts every 0.1 seconds and toggles the red LED and set up Channel 1 so that it raises periodic interrupts every 0.5 seconds and toggles the green LED. Engage a suitable low-power mode. Complete the missing parts of the code.

```
#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 = ...                   // Find # cycles
    TA0CCTL0 |= CCIE;
    TA0CCTL0 &= ~CCIFG;
    // Configure Channel 1
    ...

    // Start the timer (divide ACLK by 4)
    ...

    // 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)
#pragma vector = ...                // fill the vector name
__interrupt void T0A1_ISR() {
    ...
}
```

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.

10.2 Using Three Channels

Modify the code of the previous part so that it implements the pattern shown in Figure 10.2. 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. In addition, Channel 2 sets up periodic interrupts every 4 seconds. These interrupts halt and resume the flashing as shown in the figure. For four seconds, the LEDs are flashing, each at its respective rate, then for the next four seconds, the LEDs are off.

In the code, the ISR of the A1 vector services the interrupt events of Channels 1 and 2. Therefore, the ISR needs to detect which of the two interrupts has occurred, as shown in the code excerpt below.

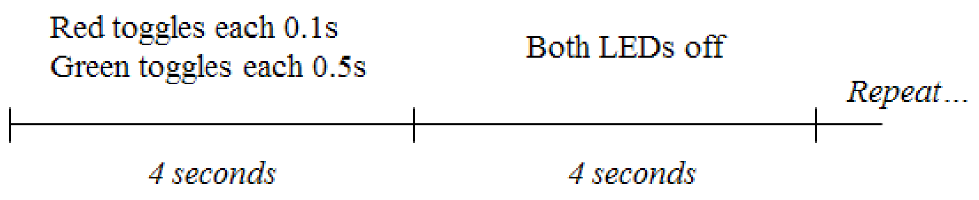


Figure 10.2: Using three channels of Timer_A

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

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

In this code, it is helpful to declare a status variable that keeps track whether the LEDs are flashing or are currently off. The status variable is used at the 4-second transition point.

One requirement in this code is that the interrupts of Channels 1 and 2 must be turned off during the off period. This means the xIE bits of Channels 1 and 2 are changed to zero during the off period and are re-enabled at the end of the four-second interval. One thing we would need to consider here is when is the best time to clear the flag: when an interrupt is disabled or when an interrupt is re-enabled? Since we want to ignore events (flags being raised) during the off period, the flags should be cleared when the interrupts are re-enabled.

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.
- Submit the code in your report.

10.3 Driving a PWM Signal on the Pin

The Timer_A module supports output patterns that can be driven on the pins and these patterns can be used to implement a PWM signal. Such output signals are driven directly by the timer without CPU intervention, therefore, the CPU can remain in low-power mode while the PWM is ongoing.

A PWM signal is shown in Figure 10.3. The signal's period is fixed and is marked by the vertical dashed lines. If the period is 0.001 second, then the PWM frequency is 1000 Hz. The relative duration of the high level within a period is described as the duty cycle. In the figure, the first pulse corresponds to a duty cycle of 25% and the second pulse corresponds to a duty cycle of 50%.

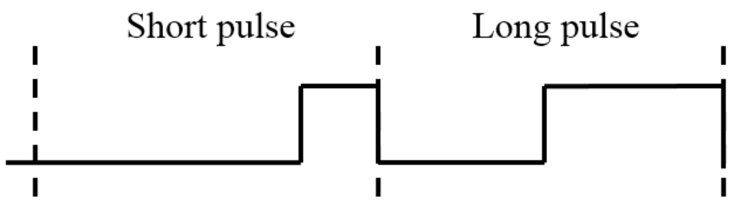


Figure 10.3: Pulse Width Modulation (PWM) Signal

A PWM signal can be used to drive a motor, in which case a higher duty cycle corresponds to a higher motor speed. In this lab, we'll drive a PWM signal on the pin to which the LED is connected. By modifying the duty cycle, multiple brightness levels appear on the LED. It's essential that the PWM frequency is high enough so that the user doesn't see the LED blinking. We'll set up a PWM frequency of 1000 Hz.

Configuring the Pin

Any pin on the MCU that is described as having the function TAx.y on the pinout diagram can be used to drive timer-based output. On our LaunchPad board, the red and green LEDs are connected to P1.0 and P9.7, respectively. Let's find out if these pins support timer-based output. Looking at the pinout diagram in the MCU's Data Sheet `(slas789c)` on p. 9, we find the pin of P1.0 described as the following.

```
P1.0/TA0.1/DMAE0/RTCCLK/A0/C0/VREF-/VeREF-
```

The TAx.y format we're looking for is shown above as TA0.1 This means that the pin can be used for output driven by Timer_A module #0 Channel 1. Since the red LED is connected to this pin, a PWM driven on this pin implements brightness levels on the red LED. Find out if P9.7 supports timer-based output.

The default functionality of all pins in MSP430 is GPIO. Therefore, the pin above is configured to P1.0 at reset. The pin can be reconfigured by looking at the data sheet tables `(slas789c)` on p. 96. This page shows the information below to configure the pin as TA0.1:

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

The values shown above should be set in bit 0 since the overlapping GPIO pin is P1.0.

Output Patterns

The output patterns that are supported by the timer are described in the FR6xx User's Guide `(slau367o)` Chapter 25 on p. 649. We will use Reset/Set mode, which is mode #7. Its operation is illustrated in Figure 10.4.

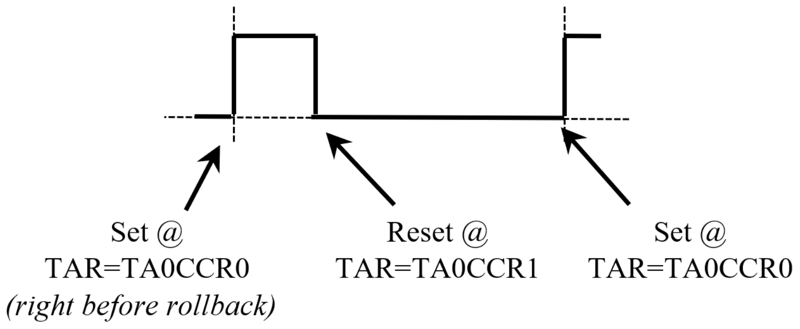


Figure 10.4: Reset/Set Output Pattern

We're aiming at setting up 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.

The output mode we're using has two actions (action1/action2), i.e., Reset/Set and is set up at Channel 1. The first action is triggered by Channel 1's compare event (i.e. when TAR=TACCR1) and the second action is triggered by Channel 0's compare event (i.e. when TAR=TACCR0=33)[1]. As shown in Figure 10.4, the vertical dashed lines mark the timer's period in which TAR counts from 0 to TACCR0=33. The figure's example corresponds to TACCR1=8. The Reset event is triggered by Channel 1's compare event when TAR=8 and the Set event is triggered by Channel 0's compare event when TAR=33, which happens right before TAR rolls back to zero. This corresponds to a duty cycle of 8/33 that's close to 25% as shown in the figure. The duty cycle can be increased by changing TACCR1 to a larger value, up to 33.

Activating an output mode (e.g. Reset/Set) at a channel is done by modifying the OUTMOD field in the channel's register (TACCTLx). This register's format can be found in the Family User's Guide `(slau367o)` in the Timer_A chapter (at the end of the chapter).

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

```
// Setting up a PWM on P1.0 (red LED)
// P1.0 coincides with TA0.1 (Timer0_A Channel 1)
// Configure 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
```

[1]When the continuous mode is used, the second action is triggered by the rollback-to-zero event.

```
    PM5CTL0 &= ~LOCKLPM5;

    // Configure 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
    config_ACLK_to_32KHz_crystal();

    // Configure Channel 1 for Reset/Set mode (Mode #7)
    ...                          // Modify OUTMOD field to 7
    TA0CCR1 = 1;                 // Modify this value between 0 and
                                 // 32 to adjust the brightness level

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

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

Once you finish the code, test for multiple values of TA0CCR1 and these should result in different brightness levels. Incorporate the joystick in your code so that moving the joystick left and right decreases and increases the LED's brightness, respectively. Moving the joystick up should set maximum brightness immediately and moving the joystick down should set minimum brightness immediately. A video demo can be found at the link below.

https://youtu.be/cUkwFgXNnlA

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Vary the value of TA0CCR1 between 1 and 32 and ensure this results in various brightness levels.
- Submit the code in your report.

10.4 Timer Input Capture

The timer supports input capture known as the Capture Mode. In this mode, the timer observes when a change occurs at a pin (a rising edge, a falling edge or either) and saves TAR into the register of a timer channel (e.g. TACCR1). The value of TAR serves as a timestamp that indicates when the change was observed at the pin.

Similar to timer-based output, timer-based input capture is available at pins that are described as having the functionality TAx.y in the pinout diagram. The button S1 on our board is connected to P1.1 which has the following description in the pinout diagram. Based on this description, the pin can be used for input capture with the functionality TA0.2, which corresponds to Timer_A module #0 Channel 2.

```
P1.1/TA0.2/TA1CLK/COUT/A1/C1/VREF+/VeREF+
```

Follow the steps below to write a code that performs input capture when the button is either pushed or released and transmits the timestamps to the PC over UART.

Step 1) Configure the pin to TA0.2 functionality by looking at the table in the chip's data sheet on p. 96. You're looking the set up the functionality TA0.CCI2A (note that it's input A rather than B as we'll need this information later on). Write down the values of P1DIR, P1SEL1 and P1SEL0 and set the corresponding bit values in your code.

Step 2) Configure Channel 2 of Timer #0 for the capture mode. This is done using this channel's register TA0CCTL2. The format of this register is in the Family User's Guide, Timer_A chapter, on p. 657. These are the fields you need to set: CM (capture on both edges), CCIS (capture input A), CAP (set the channel to capture mode), CCIE (to trigger an interrupt when a capture occurs).

Step 3) Start Timer #0 in the continuous mode based on the 32 KHz crystal. Therefore, the timestamps will be in the range 0-65,535.

Step 4) Write the ISR of Timer #0 Channel 2. In the ISR, read the timestamp, which will be in Channel 2's register (TA0CCR2), and transmit it over UART to the PC. You can use a delay loop to wait out the bounces, so they don't transmit a new timestamp. It's also a good idea to clear the flag after the delay loop when all the bounces have elapsed.

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- How long does a button push last? Give the answer as clock cycles and as a duration.
- Submit the code in your report.

Student Q&A

1. Copy the description of P9.7 from the pinout diagram and determine whether this pin supports timer-based output.

2. In the code with three channels, why was it necessary to divide ACLK?

3. 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.

4. 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? Explain the overflow mechanism and show why it results in a correct value.

Lab 11

Concurrent Event Handling

In this lab, we will learn programming concurrent events that are processed with interrupts. This lab will show us how to program multiple interrupt events that interact with one another and how to program interrupts that are enabled/disabled in various phases of the code.

11.1 HVAC Control (non-renewing delayed toggle)

In this part, we will write a program that takes user input and controls the compressor of an HVAC system. The compressor of an HVAC is damaged if it's cycled on/off at a high frequency (e.g. toggled every second). The program we'll write takes user input, represented by the button, and toggles the compressor three seconds after the button is pushed. The compressor is represented by the red LED. This mechanism is illustrated in Figure 11.1. As the figure shows, the button is not processed if it's pushed during the 3-second interval. Throughout the program, the green LED flashes every 0.5 second to indicate that the power is on.

Your program should satisfy the following requirements:

- Run the timer in the continuous mode and use the channels to time events.
- Use Channel 1 with interrupts to time the 0.5-second interval that flashes the green LED.

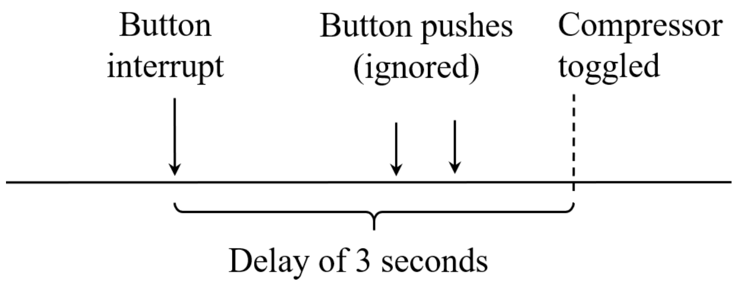


Figure 11.1: The compressor is toggled three seconds after the button is pushed

- Use Channel 2 with interrupts to time the 3-second interval when it's needed.
- When Channel 2 is not being used, its interrupt enable bit (xIE) must be set to zero.
- The button is interfaced with interrupts.
- During the 3-second interval, the button's enable bit (xIE) must be set to zero.
- Engage a low-power mode.

The requirements above are meant to practice enabling and disabling interrupt during various phases of the program.

The following observation is helpful when writing this code. In the pseudocode below, two interrupt events share an ISR and are processed with two if- statements. The first interrupt event is always enabled (xIE=1) in the program, therefore, it's sufficient to check that its flag is high before it's processed. On the other hand, the second event is enabled/disabled in various phases of the program. Therefore, it is serviced if it is currently enabled and its flag is currently high. If we don't check xIE, it's possible that the ISR is called due to the first event; if the second event is currently disabled (xIE=0) and its flag is high, it will end up being processed, which is not supposed to happen.

```
void Timer_A1_ISR() {
  // Detect Channel 1 (Channel 1 interrupt is always enabled)
  if(channel 1 xIFG =1)
    ...

  // Detect Channel 2 (Channel 2 interrupt is enabled/disabled in various
     phases)
  if(channel 2 xIE=1  AND  channel 2 xIFG =1)
    ...
}
```

It is helpful to think about this code by drawing a timeline similar to Figure 11.1 and writing under each action a list of events to be done. That is, write a list of actions that should be done when the button interrupt occurs, and another list of actions that is done when the timer interrupt occurs.

Perform the following:

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

11.2 HVAC Control (renewing delayed toggle)

Modify the code of the previous part so that the 3-second interval renews if the button is pushed. For example, if the user pushes the button at t=0s, the toggle is supposed to happen at t=3s. However, if the button is pushed again at t=2s, the toggle is delayed until t=5s. If the user keeps pushing the button within the 3-second interval, the toggle will keep getting delayed.

Perform the following:

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

11.3 Button Debouncing

In this part, we will implement a button debouncing algorithm that makes the buttons of our board more reliable. In an earlier lab, we wrote a code that toggles the LED when a button interrupt occurs and we observed that the button doesn't work all the time because the bounces raise additional interrupts that cancel out the toggle operation.

The debouncing algorithm is illustrated in Figure 11.2. This algorithm takes two samples of the button that are 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, if the button is still pushed, we'll interpret a button push and take the necessary action, e.g. toggle the LED.

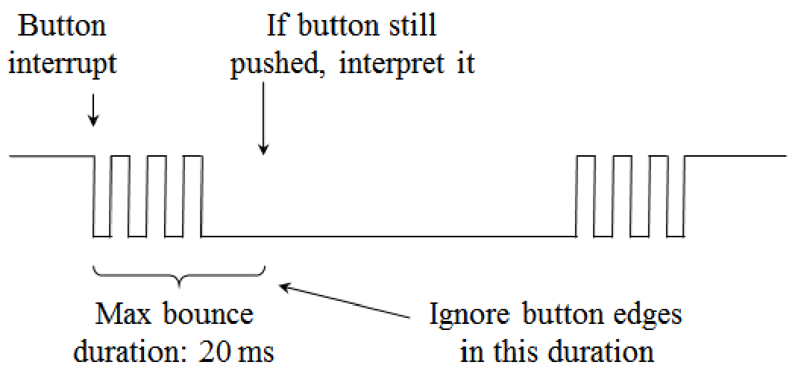

Button interrupt
If button still pushed, interpret it
Max bounce duration: 20 ms
Ignore button edges in this duration

Figure 11.2: 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.

We will start by assuming that the maximum bounce duration is 20 ms, however, we will find this value experimentally. High quality buttons have a short bounce duration and buttons with built-in debouncers exist although at a higher price point. Simple buttons like the ones on our board may not have a data sheet, therefore, we'll experiment with the maximum bounce duration in order to find the smallest such delay that makes the button reliable. If the button is not reliable when we set the maximum bounce duration to 20 ms, we should increase it. Otherwise, we can try decreasing this duration as long as the button remains reliable. It's possible that the two buttons on our board may have differing maximum bounce durations. Note that our algorithm effectively introduces a 20 ms response delay which should not be noticeable to the user.

Implement this algorithm so that the LED is toggled when the button is pushed. Use interrupts for the button and the timer. The interrupts of the button and the timer must be disabled (xIE=0) when they are not needed. Engage a low-power mode. Experiment with the maximum bounce duration to find the smallest duration with which the code works reliably.

It is helpful to think about this code by drawing a timeline similar to Figure 11.2 and writing under each action a list of events to be done. That is, write a list of actions that should be done when the button interrupt occurs, and another list of actions that is done when the timer interrupt occurs.

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.
- Demo your program to the TA
- What is the maximum bounce duration that is set in your code?
- Submit the code in your report.

Student Q&A

1. An interrupt uses a shared ISR and is always enabled in the program. What does the if-statement in the ISR check for before servicing this interrupt?
2. An interrupt uses a shared ISR and is enabled/disabled in various phases of the program. What does the if-statement in the ISR check for before servicing this interrupt?
3. For the debouncing algorithm that we implemented, is it possible that the LED will be toggled when the button is released? Explain.
4. 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 debouncing algorithm fail? 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=0, byte2=0; // Intialize to ensure successful reading

  UCB1I2CSA = i2c_address;  // Set address

  UCB1IFG &= ~UCTXIFG0;

  // Transmit a byte (the internal register address)
  UCB1CTLW0 |= UCTR;
  UCB1CTLW0 |= UCTXSTT;

  while((UCB1IFG & UCTXIFG0)==0) {}      // Wait for flag to raise
  UCB1TXBUF = i2c_reg;                   // Write in the TX buffer
  while((UCB1IFG & UCTXIFG0)==0) {}      // Buffer copied to shift register;
     Tx in progress; set Stop bit

  // Repeated Start
  UCB1CTLW0 &= ~UCTR;
  UCB1CTLW0 |= UCTXSTT;

  // Read the first byte
  while((UCB1IFG & UCRXIFG0)==0) {}       // Wait for flag to raise
  byte1 = UCB1RXBUF;

  // Assert the Stop signal bit before receiving the last byte
  UCB1CTLW0 |= UCTXSTP;

  // Read the second byte
  while((UCB1IFG & UCRXIFG0)==0) {}       // Wait for flag to raise
  byte2 = UCB1RXBUF;

  while((UCB1CTLW0 & UCTXSTP)!=0) {}
  while((UCB1STATW & UCBBUSY)!=0) {}

  *data = (byte1 << 8) | (byte2 & (unsigned int)0x00FF);

  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;

    UCB1I2CSA = i2c_address;    // Set I2C address

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

    UCB1IFG &= ~UCTXIFG0;

    // Write 3 bytes
    UCB1CTLW0 |= (UCTR | UCTXSTT);

    while( (UCB1IFG & UCTXIFG0) == 0) {}
    UCB1TXBUF = i2c_reg;
    while( (UCB1IFG & UCTXIFG0) == 0) {}
    UCB1TXBUF = byte1;

    while( (UCB1IFG & UCTXIFG0) == 0) {}
    UCB1TXBUF = byte2;

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

    UCB1CTLW0 |= UCTXSTP;
    while( (UCB1CTLW0 & UCTXSTP) != 0 ) {}
    while((UCB1STATW & UCBBUSY)!=0) {}

    return 0;
}
```