Lab Assignment # 6: PWM, LCD, and Matrix Keypad Decoding

Due Date: Week of October 19st, 2020,
half an hour before your lab session starts
on myMason (Blackboard)

Objectives:

- To learn how to interface with a 4x4 matrix keypad.
- To become familiar with pulse width modulation (PWM).
- To become familiar interfacing with the on-board liquid crystal display (LCD).
- To become more familiar with ISR functionality.

Parts:

Please make sure that you have the following parts before you start the lab.

- MSP-EXP430FR6989 LaunchPad
- Blue LED
- Red LED
- Yellow LED
- Green LED
- 4 current limiting resistors for the LEDS
- 4x4 Matrix Keypad
- Pin headers
- Rainbow ribbon cable with headers

Introduction:

In this lab you will set the brightness of LEDs through a PWM signal. You will be able to adjust the duty cycle of the PWM signal, and with this the brightness of the LEDs, through entering values on the Keypad. The on-boad liquid crystal display (LCD) will show the values you entered.

In this lab, we will focus on:

- Scanning a matrix keypad in order to determine which button was pressed.
- Setting boundaries for the user's input, thus making sense of the button presses.
- Displaying the user's input on the on-board LCD.
- Using the input to set the duty cycle of the external pulse-width modulated LEDs.

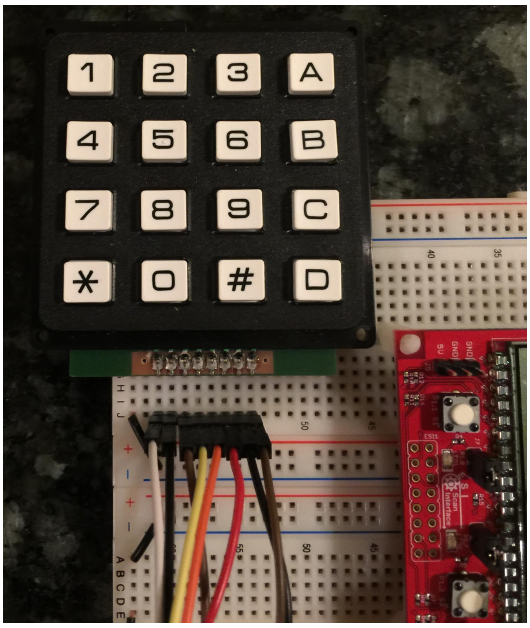

Figure 1: 4x4 matrix keypad connected to a breadboard.

Background:

4x4 Matrix Keypad

Now that you have mastered receiving a user's input via a standard push button, let's move onto something a little more interesting. For this lab, you will be given the task of identifying which button of the sixteen-button keypad was pressed and use that value to set specific registers within your MSP430 microcontroller (MCU).

The M (row) x N (col) matrix keypad is an input device that typically possesses M + N pins. Each pin corresponds to either a specific row or column and we will leave it up to you to figure out which pins map to where. An easy way to check which pins correspond to which column or row is to perform a continuity test with a multimeter while pressing buttons.

As learned in previous I/O labs, humans interact with computers at a rate that is much slower than a computer-to-computer interaction. Therefore, one method that can be used to identify keypad inputs is called keypad scanning. Keypad scanning is a technique that tactfully sets or

clears voltages on the column pins and reads the voltages that are present at the row pins (or vice versa).

In the class we discussed three different methods for scanning the keypad. In this lab you can choose your preferred method.

One method worked as follows. All rows had pull-up resistors. By setting all but one column pins, alternating through which column is de-asserted, and reading the values at each row pin, you will be able to determine which of the sixteen buttons has been pressed. As a hint, assuming only one button is pressed at any point in time, only one row pin will be equal to zero. This change in value will prompt you to check which column was de-asserted at that specific instant and voila! Once you determine which button has been pressed, and do so correctly, a value can be assigned to it. This means that the button with the symbol "5" can be set to equal the integer 5, and so on. Therefore, by accepting sequences of button presses, elaborate I/O schemes can be implemented to host all kinds of interesting MCU applications!

Pulse Width Modulation

In addition to dealing with the matrix keypad, this lab assignment also includes the topic of pulse-width modulation. Pulse-width modulation is a modulation technique used to encode a message into a pulsing signal. In Figure 2 below, there are two different voltage waveforms. The clear distinction between the two waveforms is that they both spend different amounts of time ON (when the waveform's voltage is not equal to zero). In fact, each waveform can be characterized by the percentage of time the waveform is not equal to zero per one period of time. This is more formally known as a duty cycle, which can be defined by the equation:

$$DutyCycle(\%) = \frac{t_{ON}}{T} \cdot 100$$

where T is the waveform's period and t_{ON} represents the amount of time, during one period, that the waveform is non-zero.

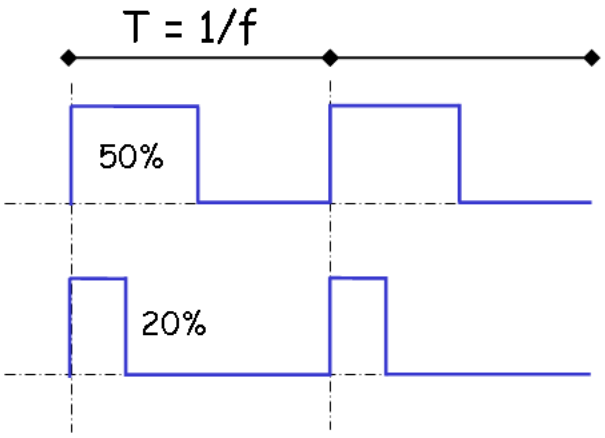

Figure 2: Two waveforms with different duty cycles.

On-board Liquid Crystal Display

Display devices are one of the most common types of interface used in embedded systems. LCDs, in particular, are a popular choice among this category as they are low power consumers

and relatively compact. There are numerous kinds of LCDs varying from the simpler numeric or alphanumeric to the more complex graphical panels. LCDs don't actually produce light – instead they control the intensity of reflected or transmitted light. Often times on graphical LCDs, there will be an LED backlight producing light for the display. The on-board LCD falls under the segmented category. These displays include the familiar seven-segment numerical displays found in watches, meters, and many other applications. Further segments can be added to each character to allow alphanumeric display. There are also special symbols that allow for additional functionality, these can be seen in Fig. 3 below. Each segment has an associated memory location in the MSP that allows us to control it. In order to toggle a segment on we must write a "1" to the memory location for that segment. The tables containing the memory locations can be found in the programmers user guide and on page 12 and 13 of SLAU627A.

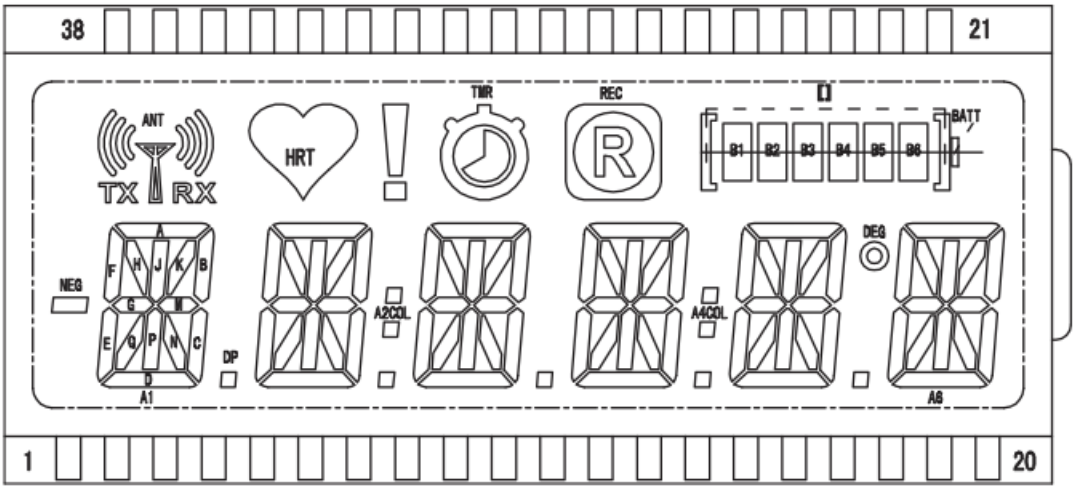

Figure 3: Segments of the on-board LCD

Further reading can be found in Section 8.6 of Introduction to Embedded Systems and Section 7.7 of MSP430 Microcontroller Basics.

Code Walk Through:

Three files are provided for this lab. The file "lcd.c" and "lcd.h" contain definitions and functions to work with the built-in LED. The file "lab6.c" contains a PWM sample code to get you started. It shows how to make use of the MSP430's timer modules to produce and output a modulated waveform that possesses pulses that are of varying widths.

lab6.c

Each timer module (Timer A0, Timer B0, etc.) contains a primary 16-bit counter (TA0R, TB0R, etc.) and multiple capture/compare (CC) registers (TA0CCR0, TB0CCR1, etc.). Each one of these CC registers is controlled by their own control register (TA0CCTL0, TB0CCTL1, etc.) and provides useful functions.

For this lab, we will make use of Output Modes, which are specific to CC registers. Each CC register is connected, via a special function pin, to a typical GPIO pin. By loading certain 16-bit

values into these CC registers, you will be able to determine the time at which each CC register's dedicated pin is HIGH or LOW. In the code below, the CC registers are being configured and connected to their specific I/O pins.

C code	Explanations
`P2DIR |= BIT6 | BIT7;`	P2.6 and P2.7 set as output
`P2SEL1 &= ~(BIT6 | BIT7);`	connect P2.6 and P2.7
`P2SEL0 |= BIT6 | BIT7;`	to TB0.5 and TB0.6
`P3DIR |= BIT6 | BIT7;`	P3.6 and P3.7 set as output
`P3SEL1 |= BIT6 | BIT7;`	connect P3.6 and P3.7
`P3SEL0 &= ~(BIT6 | BIT7);`	to TB0.2 and TB0.3

Output Modes, or OUTMODS, allow CC registers to determine when specific pins (P2.6, P2.7, P3.6, and P3.7 in this case) are HIGH or LOW. Additionally, the values put into the CC registers must be relative to some minimum and maximum counter value. These 16-bit counters are capable of taking on any unsigned value from 0 to $2^{16}-1$. Another way to think of it is that these counters can take on 2^{16} or 65,536 different values. If the counters are being driven by a clock that is $2^{15} \approx 32kHz$, it will take 2 seconds for the counter to count from 0 to its maximum value.

$$2^{16}ticks \cdot \left(\frac{1tick}{2^{15}Hz}\right) = 2$$

In the provided code, we will use a driving clock that is 8kHz and, instead of counting from 0 to 65,536, we will count from 0 to 100. This is shown in the two code segments below.

C code	Explanations
`TB0CCR0 = 100;`	(32kHz/4) / 100 ticks = 81.92 Hz

Before getting into the significance of TB0CCR0, it is important to pay some attention to how the provided code sets Timer B0's control register.

C code	Explanations
`TB0CTL = TBSSEL_ACLK | ID__4 | MC__UP | TBCLR;`	See device family guide (slau367f) page 465

TB0CTL is the control register that determines the functionality of the Timer B0 module. The most important portion of the above assignment of value to TB0CTL is "MC__UP", which tells the TB0R 16-bit counter to count from 0x0000 to TB0CCR0 instead of 0x0000 to 0xFFFF. Therefore, the value in TB0CCR0 now determines the period of time it takes for TB0R to count from its minimum to maximum value.

With CCR0 (CC Register 0) set to 100, TB0R counts from 0 to 100 at a rate of 8kHz. Coming back to OUTMODS, the above code segments now allow us to use other CCRs to control I/O pins based on a value that is relative to that which is in CCR0.

C code	Explanations
`TB0CCR2 = 20;`	20 ticks out of 100 (in TB0CCR0) ticks = 20% ON
`TB0CCR3 = 40;`	40 ticks out of 100 ticks = 40% ON
`TB0CCR5 = 60;`	60 ticks out of 100 ticks = 60% ON
`TB0CCR6 = 80;`	80 ticks out of 100 ticks = 80% ON

These assignments will now set the duty cycles of the outputs from pins P2.4 to P2.7. However, there is one more detail that determines the OUTMOD scheme. We know that the TB0R (Timer B0's counter) will now only count from 0 to 100, but when do the I/O pins get assigned values? This is where page 459 of the user's guide (slau367f) comes to the rescue. Each CCR's control register has a particular bit-field that defines the OUTMOD that should be used for that specific CCR. In the code below, it is clear that all of our CCR's should be using OUTMOD_7, which in the documentation represents the Reset/Set option.

C code	Explanations
`TB0CCTL2 = TB0CCTL3 = TB0CCTL5 = TB0CCTL6 = OUTMOD_7;`	Setting OUTMOD

What this means is that, when TB0R reaches the value that is equal to any TB0CCRx (x equals 3, 4, 5, or 6 in this case), reset (assign 0) the I/O pin that corresponds to the TB0CCRx that matches in value with TB0R. Additionally, once TB0R reaches the value in TB0CCR0, set (assign 1) the I/O pin that corresponds to whichever TB0CCRx that is operating in OUTMOD_7. Therefore, in short, the scheme Reset/Set could be thought of as (Pin assignment when TB0CCRx is reached)/(Pin assignment when TB0CCR0 is reached).

lcd.c

The file "lcd.c" contains the function `lcd_init()` which is responsible for initializing and configuring the LCD. This requires us to first select the low frequency crystal XT1 (LFXT) by setting `BIT4` and `BIT5` of `PJSEL0`. Next, we have to initialize the `LCD_C` port control registers by setting the bits corresponding to the LCD segments that we will be using. To do this correctly the first time, we need to cross reference Table 4 on page 13 of SLAU627A (included in the programmers reference guide) with Sections 27.3.6-8 on pages 746-7 of the SLAU367F (also included in the programmers reference guide). In addition, something to note when reading these sections is the `LCDON` bit must be "0" before configuring the control register.

C code	Explanations
`PJSEL0 = BIT4 | BIT5;`	Select the low frequency, 32kHz crystal oscillator XT1.
`LCDCPCTL0 = 0xFFD0;`	Initialize LCD segments 4, 6–15.
`LCDCPCTL1 = 0xF83F;`	Initialize LCD segments 16–21, 27–31.
`LCDCPCTL2 = 0x00F*;`	Initialize LCD segments 35–39.

Next, we have to activate ports by disabling the low power mode lock, as we did in Lab #1. We now have to unlock the clock system (CS) registers by writing the password. If this is done incorrectly write access will be disabled. We now have to enable the `LFXT` by clearing the `LXFTOFF` bit. We then clear the `LXFT` fault flag and corresponding oscillator fault interrupt flag and then check to make sure the flags are cleared. Once we have set up the CS, we write the incorrect password to the CS control 0 register to disable writing to the CS registers.

C code	Explanations
`PM5CTL0 &= ~LOCKLPM5;`	Unlock ports from power manager.
`CSCTL0_H = CSKEY >> 8;`	Unlock CS registers
`CSCTL4 &= ~LFXTOFF;`	Enable LFXT
`do {`	
`  CSCTL5 &= ~LFXTOFFG;`	Clear LFXT fault flag
`  SFRIFG1 &= ~OFIFG;`	Clear oscillator fault interrupt flag
`} while (SFRIFG1 & OFIFG);`	Test oscillator fault flag
`CSCTL0_H = 0;`	Lock CS registers

Once we have the CS configured, we now have to configure the LCD control register by selecting our clock source, frequency divider, frequency prescaler, MUX rate, and low power waveform. More information about these settings can be found on page 739 of the SLAU367F user guide. We now need to configure the LCD voltage control register. Selecting the `LCDCPEN` bit enables the charge pump and allows us to set the LCD voltage equal to 2.17 x V(REF) by setting the `VLCD_1` bit. Lastly, setting the `VLCDREF_0` bit sets charge pump reference to the internal reference voltage. The next step is to synchronize the LCD charge pump clock with the respective clock source. Now that we have the LCD control and voltage registers configured, we can clear the LCD memory and turn the module on.

C code	Explanations
`LCDCCTL0 = LCDDIV__1 | LCDPRE__16 |`	Initialize LCD_C: ACLK, Divider = 1,
`  LCD4MUX | LCDLP;`	Pre-divider = 16, 4-pin MUX
`LCDCVCTL = VLCD_1 | VLCDREF_0 | LCDCPEN;`	VLCD generated internally,
	V2-V4 generated internally, V5 to ground
`LCDCCPCTL = LCDCPCLKSYNC;`	Enable clock synchronization
`LCDCMEMCTL = LCDCLRM;`	Clear LCD memory
`LCDCCTL0 |= LCDON;`	Turn LCD on

We have now configured the LCD and can move on to the other functions.

The function `lcd_clear()` simply clears the LCD memory.

When we look at Fig. 3 above, we see the left most alphanumeric number maps to `A1`. In Table 4 on page 13 of SLAU627A, we see the `A1` segment A falls under LCD memory 10 or `LCDM10`. This memory is one byte wide and corresponds to segments A, B, C, D, E, F, G, and M in Fig. 4 below, so if we were to write `0xFF` to `LCDM10`, all of these segments would toggle on. The following memory location handles the segments H, J , K, P, Q, some symbol, N, and Decimal Point of the same digit, with exception of the symbol.

In order to display a digit on the LCD, we need a mapping from the number to the LCD segments that should be tuned on. The number '0' can be shown as '$\emptyset$' in order to be able to distinguish it from the capital letter 'O'. This requires us to switch on segments A, B, C, D, E, F, K, and Q. As this requires two 8-bit constants, a table for 10 digits with 2 bytes each is defined as shown below.

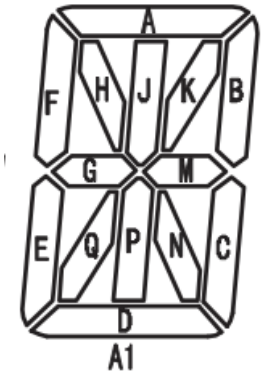

Figure 4: Segments of the on-board LCD

C code	Explanations
`const char digit[10][2] =`	10 digits with 2 bytes each
`{`	
`{0xFC, 0x28},`	LCD segments a+b+c+d+e+f+k+q
`{0x60, 0x20},`	LCD segments b+c+k
`{0xDB, 0x00},`	LCD segments a+b+d+e+g+m
`...`	and so on
`}`	

The function `void displayNum(unsigned int num, unsigned int digits)` calls the function `showDigit` for as many digits of the number `num` as specified in `digits`. The function does use the costly modulo '%' and division '/' operators. Modulo and division functions are costly as the CPU of the MSP430 does not have a multiplier or division unit. The C-compiler will automatically add the assembly code needed to perform these functions.

The left most digit on the LCD is called A1 followed by A2 and so on. The right most digit is A6. Unfortunately, they are not mapped to consecutive addresses in LCD memory. In order to be able to address one digit after another the LCD memory locations are defined in "lcd.h" and then put in order into an array called `digitpos[6]` in "lcd.c". The function `void showDigit(char c, unsigned int position)` makes use of this to place a number into the correct digit position.

In-lab Exercise (week of October 12th):

1. Add the red, yellow, blue, and green LEDs and 4 1 kΩ resistors to your breadboard. Connect the anode of each LED via a 1kΩ resistor to Vcc and the cathode to ground. You can determine which wire is the cathode by holding the LED against a light source and looking for the anvil. Connect the Vcc of your beard board to the 3.3 V from your launchpad and the ground from your bread board to GND on your launchpad.

2. You will notice that the LEDs will light up with different level of brightness. Replace the 1 kΩ resistor of the dimmest LED with the next smallest value from your lab kit. Continue this until it light up as bright as the brightest LED. Repeat these steps for the other LEDs until they are equally bright.

3. Now remove the connections from the resistors to Vcc and connect the resistor of the blue LED to P3.7 (CCR3), the resistor of the red LED to P3.6 (CCR2), the resistor of the yellow LED to P2.6 (CCR5), and the resistor of the green LED to P2.7 (CCR6).

4. Create a new "Empty Project" in Code Composer Studio and add the provided "lab-6.c" to it.

5. Run the project on the board. If everything is working, you should see that the brightness of the LEDs is different. If you have a logic analyzer, hook it up to P3.7, P3.6, P2.6, and P2.7 and observe the wave forms. Does the duty cycle of the LEDs and the brightness of the LEDs match the settings in the source code?

6. Demonstrate to the GTA (using your web camera) that the LEDs are lit up with different levels of brightness.

7. The last step of the in-lab exercises is to figure out the pinout of the keypad, i.e., which pin corresponds to which row or column. An easy way to check is performing a continuity test with a multimeter while pressing buttons. If you don't have a multimeter, you can also use an LED with current limiting resistor.

8. Discuss with the GTA your findings for the pinout of the Keypad. Pin 1 is the left most pin when looking at the buttons of the keypad.

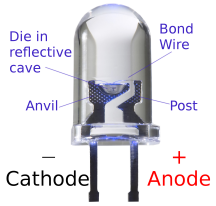

Figure 5: The anvil is the cathode of an LED

Exercises:

1. Create a new "Empty Project" in Code Composer Studio and add the provided "lab6.c" file to it.
2. Include the provided "lcd.h".
3. Add code to display a number with three digits on the LCD using the `displayNum` function.
4. Connect the keypad to your board. Header pins must be soldered on the keypad. Wiring must be neat (Note: points will be lost if wiring is not neat). Please make sure you follow the below port/pin assignments for the keypad.
   - Columns: P8.4, P8.5, P8.6, P8.7
   - Rows: P2.1, P2.2, P2.3, P2.4
5. Write a function to decode the keypad. It should return the value of the key (10 for 'A', 11 for 'B', etc).
6. Display the number or letter corresponding to the most recent button press on the LCD.
7. Use a port ISR to detect if a key was pressed and debounce the keypad by using a timer to provide the debouncing delay. The debounce delay should be 25 ms.
8. Modify your code so that multiple digits are accepted and the correct integer displayed on the LCD e.g., pressing '1' followed by '2' displays the number 12 on the LCD. The number should be stored internally in a variable.
9. Now make a copy of your program and modify it to have the following functionality.
   (a) The default value displayed on the LED should be '0'.
   (b) A user should be able to enter a number between '0' and '100'. As each digit is keyed in, it should be appended to the number on the right, thus making the entire number shift left by one digit.
   (c) Pressing '*' should remove the last typed in digit and the number should shift to the right by one digit.
   (d) Pressing '#' should reset the LCD to the '0' default.
   (e) If the user presses a letter, the currently displayed number should become the duty cycle of the corresponding LED. Letter 'A' maps to the blue LED, letter 'B' to the red LED, letter 'C' to the yellow LED, and letter 'D' to the green LED.
   (f) This duty cycle assignment should be acknowledged with the red LED on the launchpad turning on for 1/2 second.
   (g) After this the program should return to step 9a.
10. Write all code in a modular fashion so that each peripheral has its supporting code in its own file e.g. you'd have a keypad.c file that contains everything that you need for the keypad including port initialization routines, etc.

11. Modify your program so that it is in low power mode whenever possible. Keep in mind what functionality is turned off in each low power mode. You will have to justify why you used the particular low power mode.

Questions to be answered in the report:

- Document which timer you used for the debounce delay and how you set it up. Show using equations how you determined how many timer ticks are needed for the 25 ms debounce delay.
- You may want to use a state machine with states such as START, ENTRY, SETPWM, and so on to have a simpler to understand and simpler to debug code.
- Document which timer you used to turn the red LED on the launchpad on for 1/2 second. Show using equations how you determined how many timer ticks are needed for this.
- Explain why you used a particular low power mode.
- As usual, include the schematic of the hardware, the flowchart and state machine diagram of your software.

Points to be covered in the demo:

- Show that the programs starts with all LEDs off and a '0' on the LCD.
- Show that you can only type in numbers between 0 and 100.
- Demonstrate that the buttons are debounced.
- Demonstrate that the '*' button removed the last typed digit and that the '#' button sets the display back to '0'.
- Show that you can assign different duty cycles to the LEDs.
- Show that the red LED on the launchpad acknowledges the duty cycle selection.