

Physics 319 Spring 2017:

Introduction to the Programming and Use of Microprocessors

Sing Chow, Andrzej Kotlicki, Ryan Wicks, and Carl Michal

January 2017

This lab is going to introduce you to the world of microprocessors. As you probably know, most modern equipment, from streetlights, to cell phones, to cars and to planes are controlled by one or many microprocessors. In this lab you are going to learn how to build and program microprocessor based devices.

You will work with an evaluation board (EVB) with an MSP430 microprocessor mounted on it, powered by a USB port, which will also be used to communicate with a computer.

You will learn how to interface the MSP430 with sensors, actuators, displays, wireless links and other devices.

Remember these safety rules while working with the MSP430! Any mistake involving violation of these rules might damage the board. To continue, you will have to purchase another one!

- Static charge from your hands, wires or tools can damage the microprocessor; always touch some metal ground object like the BNC connector on the oscilloscope before touching the board.

- None of the EVB leads should ever be directly connected to voltages below ground or above 3.6V! If in your work in future, you are going to use +/- 15V supply, never connect it directly to the EVB. We will use 5V inputs, but only with large series resistors to protect the board.

- Turn off the power before working on the circuit.

- Do not connect any logic pins (any pins except power supply) directly to the power supply.

- Open inputs should be either grounded or pulled up (either internally or externally). They should never be left floating.

- Do not connect two or more logic outputs together.

- Discharge any electrolytic capacitors before inserting them into the circuit

Beginning in week 2, we will program the microprocessor from the desktop computers in the lab or using your own laptop. You are welcome to use the computers in the lab, but you will likely find it advantageous to use your own computer. Instructions for installing and using the software tools we'll need on computers running Windows, Mac OS X, and Linux are provided in separate documents on the course web page.

Lab Notes

As you work through the activities in the manual, you must keep notes on what you do. You may keep all your notes electronically. Your notes should contain things like: clear statements of what you are trying to do (your objective), circuit diagrams, notes of problems you encounter and how you solve them, file names of programs and data, links to documentation consulted, explanations to of things you've read about, explanations of help you've received from other students or course staff. You will need to have drawings of things like circuit diagrams that you may wish to draw

quickly by hand – these can be photographed or scanned and included in your lab notebook. This inclusion should be done immediately as otherwise it is easily forgotten. Your programs should be well commented so they are readable, and should include pointers to dates and locations in your lab notes where related circuit diagrams can be found. After each two-week block of labs, you will need to turn in your notes and programs to your TA.

As this is an upper level lab course, you are expected to show initiative. Performing the tasks recommended in the lab manual will not get you full marks for the labs, we expect you to go beyond the printed instructions and explore the equipment and software on your own.

Circuit Tidyness:

As your circuits become increasingly complicated, the tidyness of the circuits you build will become more important for two reasons: the first reason is that tidier circuits are much easier to debug when things go wrong. The second reason is that *tidier circuits actually work better*. With high speed logic circuits, tidier circuits tend to produce far less interference between nearby signal lines, improving the reliability of the circuit. Some general advice is to avoid using extra wire wherever practical – don't have huge loops of wire to connect nearby components. Lay out the components to minimize the distances between connections. Use the power rails on the breadboard for power supply and ground lines, but not for any sort of signal lines.

Lab 1

The Breadboard:

Begin by making a sketch of the solderless breadboard at the heart of your prototyping board, and explain clearly which holes are electrically connected to which other holes.

Manually setting a display

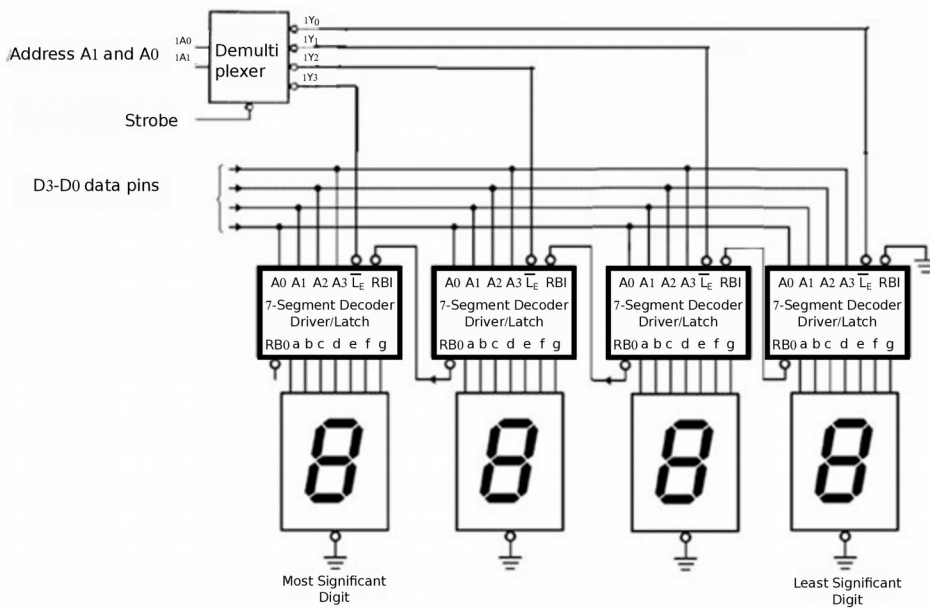


Figure 1: Schematic of lab breadboard display.

In Physics 219, you used the four-digit 7-segment display. On your board, you have a multiplexed version of this display, shown schematically in Fig. 1. You will display 4 different numbers on this display: initially by hand, and then with the microcontroller. The purpose of this exercise is to introduce you to the microprocessor at its most basic level by forcing you to write in assembly and learn a little about the MSP430's instruction set and architecture.

To begin, read the data sheet for the [DM9368](#) (7 segment decoder, driver and latch) and the [74HC139](#) (2 to 4 line decoder). Try to understand (and explain in your notes) the function of each of these chips, and how you use them to reduce the 16 wires needed to drive 4 displays down to 7 wires. Further, consult the [HC/HCT logic families' specifications](#). From these sources, explain V_{OH} min, V_{IH} min, V_{OL} max, and V_{IL} max. How much current flows into the inputs (I_I)? Also note the AC characteristics of the latch (how fast can the latch switch? How long does it take to respond to input changes?).

Now you will try to display the last 4 digits of your student number on the 7-segment displays, by hand. The 74HC139 is a CMOS (Complementary Metal Oxide Semiconductor) device, and while the DM9368 is a TTL (Transistor to Transistor Logic) device, it is designed to be driven directly by a CMOS device.¹ To be safe, however, to achieve a logical one, you will connect to 5V through a 1-10 k Ω resistor and connect directly to GND for a logical 0. The resistor acts as a current limit in the case of any misconnection or short circuit. **Best practice is that no pins should be connected directly to VCC except the one that powers the IC.** Use one of the outputs of the pulse push buttons for strobe. Remember the information appears on the output pins when the strobe (enable) input goes from high to low.

Write down the steps you followed to complete this task.

Show the display with the number to the TA

If you plan to use your own computer to program the MSP430, please install the assembler and flasher **before coming to the second lab session. Instructions can be found in the platform specific documents on the course website.**

Lab 2: Using the Microprocessor to set the display

The next task is to do the same thing with a microprocessor. Now, instead of changing the inputs manually, you connect them to your microprocessor, and write a program to set the value on each display. Before we get into the details of how the microprocessor executes the program, let's consider the connections we have to make.

Consult the electrical specification for the MSP430 and note the high and low states for inputs and outputs. In the PHYS 219 lab you were using TTL logic for gates, counters and inverters. Can you connect MSP430 outputs directly to TTL inputs and TTL outputs directly to MSP430 inputs? Explain in your lab notes.

Whenever connecting chips together, you should consider the input and output properties of the respective chips and ensure they are compatible before proceeding. Make the following connections:

¹ CMOS and TTL are names for the different technologies used in integrated circuits (IC's). Modern IC's use CMOS, since it is faster, smaller and uses less power. It is rare to find TTL IC's these days, except in legacy parts and a few niche applications. Be aware that different families of IC's are not always compatible, and may require extra circuitry to be connected together.

| | | | | | | | | |
|-----------|------|------|------|------|------|------|------|--------|
| LaunchPad | P1.7 | P1.6 | P1.5 | P1.4 | P1.3 | P1.2 | P1.1 | P1.0 |
| display | D3 | D2 | D1 | D0 | - | A1 | A0 | strobe |

Now you have to write software to control the outputs of the microcontroller. At the most basic level (and you can't get more basic than microcontroller programming), software is simply a list of instructions that the microprocessor steps through and executes. You write the software on a computer in assembly, or in a high level language like C, compile it into machine code, and then load it into a non-volatile section of memory on the microcontroller (normally Flash or EEPROM nowadays). On initial power up or on a reset, the MSP430 microcontroller reads the contents of a particular spot in memory, called the reset vector. This reset vector is loaded with the memory location of the start of your program. It takes this memory location, and loads it into a special memory register called the program counter (PC). This register contains the memory location of the next instruction to be executed by the microprocessor. It is incremented as each instruction is evaluated, and this process continues forever. Conditional branching and looping are achieved by changing the value of the program counter to point to an arbitrary spot in memory. We will get into conditional looping and branching in a later lab.

This is the function of any microprocessor, to execute a list of instructions. For the MSP430, there are a total of 27 different instructions available, plus a few derived instructions. You can find a listing of all of these instructions in Table 1 in the reference document. Your job is to choose the instructions that perform the same job you did by hand in part 1.

Before you attempt to write the program, take some time to understand how the microcontroller communicates with the outside world and its own peripherals by writing to special locations in memory.

Aside: General-purpose I/O registers address and overview:

The difference between a microprocessor and a microcontroller is that the microcontroller is a full system on a chip, containing a microprocessor and all the extra peripherals you need to run it, like memory, storage, clocks, counters and a variety of other modules that are useful in certain situations (like analog to digital converters, serial ports, PWM (pulse width modulators)). However, the microprocessor inside the microcontroller remains the same over an entire range of devices offered (just look at the number of MSP430 devices offered by TI). It is unnecessary to create a new microprocessor with new instructions for each iteration of the design. Therefore, the approach taken by all manufacturers is to make peripheral devices accessible by writing or reading information from special locations in memory (known as special function registers (SFR's)). Beyond just controlling peripherals, these specific memory locations also control the settings of the microcontroller. In this section, we will introduce you to the memory locations used to control the function of the external microcontroller general purpose input/output (GPIO) pins.

The digital inputs and outputs are organized into 8-bit ports called parallel-input-output ports (pio). The table below lists the 1 byte (8 bit) memory locations which control the input/output ports P_x , where x can be 1 or 2 (ie. there are two input/output ports). The hexadecimal numbers (indicated by the prefix 0x) are the addresses of these registers. When you write your code, you will import a file that contains all of these memory location names ->

memory location mappings, so that you can make your code more readable. If you consult the datasheet, you will see that it refers to these memory locations by name and not number. Long story short, don't bother remembering the hexadecimal codes, just remember the names.

| Px | PxIN | PxOUT 0:pd 1:pu | PxDIR 0: input 1:output | PxSEL 0:pio | PxIES 0:L->H 1:H->L | PxIE 0:irq disable 1:irq enable | PxIFG 0:irq clr 1:irq pending | PxREN 0:no pull 1:pu/pd |
|----|------|-----------------------|-------------------------------|----------------|---------------------------|---|--|-------------------------------|
| P1 | 0x20 | 0x21 | 0x22 | 0x23 | 0x24 | 0x25 | 0x26 | 0x27 |
| P2 | 0x28 | 0x29 | 0x2a | 0x2b | 0x2c | 0x2d | 0x2e | 0x2f |

Each of these registers controls the behaviour of each pin individually. When the bits of the register PxSEL (SEL = select) are set to 0 (which is the default, which means that they are set to 0 when the microprocessor is switched on) the port is set as an input/output port (as opposed to an ADC input, for instance). When PxSEL=0, the values of the bits in register PxDIR (DIR = direction) defines if the appropriate port pins are set as inputs (0) or outputs (1). If a bit in PxSEL = 1 then the pin is connected to some alternate function. Which function depends on the specific chip and pin – reading the datasheet is necessary.

An input pin should not be left open when the PxDIR=0 because electric noise can switch it up or down. One can prevent this by setting the bit corresponding to the open input in the register PxREN=1 (REN = resistor enable) to enable the pull down or up resistors. The pull up (bringing the input to high) or down (bringing the input to ground) is determined by the setting of register PxOUT. The pull up – pull down function is very useful when the pin is used as an input. For example if we pull the input pin up we could control its state with only one switch to ground (switch on will correspond to logical 0 and switch off to logical 1, as the pin will be pulled up). For input pins, PxIN can be examined to read in the state of signals applied to them.

Your program needs to deal with registers P1DIR and P1OUT, while the registers P1SEL and P1REN can be left as they are, because they are set to zero on power up. The use of the interrupt registers P1IES (IES = interrupt edge set), P1IE (IE = interrupt enable) and P1IFG (IFG = interrupt flag) will be demonstrated later.

Back to Lab 2:

Now that you understand how to change the state of the output pins and have connected up the circuit, you can start writing your code. There are some things you have to include in your code to make the chip work properly:

- initialize the RAM for stack operation (standard procedure, we'll explain what this means in a later lab)
- stop the watchdog timer (the watchdog timer is a feature that allows the cpu to detect and recover from some kinds of software bugs. We just want to disable it.)

Now you can begin with your program. Remember:

- Include the file defining the names of addresses, bytes and words (this file comes with the MSP430 assembler in the sub-directory include):
`.include "msp430g2553.inc"`
- Start with an `ORG 0xC000` that tells the assembler where to put the program in memory.
- Disable the watchdog (see the example program below)
- Configure 7 lines of Port 1 as outputs, all except P1.3.
- Look back at the steps you followed to write to the display by hand. Convert these steps to assembly to complete the task.
- After the body of your program, you have to do two things:
 - Either place a small loop at the end of the program or shut down the CPU so the program counter doesn't continue to step through memory, reading the random information stored past the end of your program.
 - Tell the assembler where the beginning of the program is, and to write this to the reset vector.

Here is what some commands look like in assembly:

```
.include "msp430g2553.inc"
org 0xc000
START:                                ;This is a label we'll use later
                                        ;to refer to this position

mov #0x0400, SP                        ;Initialize the stack pointer
mov.w #WDTPW|WDTHOLD, &WDTCTL ;Disable the watchdog, the symbol |
    ; indicates that values of WDTPW and WDTHOLD are bitwise OR'd
mov.b #11110111b, &P1DIR              ;Set all Port 1 pins, except 1.3 as output

;Your code here

bis.w #CPUOFF,SR                       ;Disable the CPU (end program)
org 0xffff                             ;Load the reset vector with the
dw START                               ;location of the program start
                                        ; after power up or reset.
```

Assembling and Loading the Code:

To assemble and disassemble the programs one can use any msp430 assembler for your operating system. For simplicity and small footprint we recommend the following one: naken430asm available from:

http://www.mikekohn.net/micro/naken430asm_msp430_assembler.php

(yes, the older naken430asm, not naken_asm)

Instructions for installing and using the assembler are found in the platform specific documents on the course website.

Connect the launchpad USB port to the computer.

Flash your program into the MSP430. If all is well the first digits of your student number will show on the 7-segment display.

Sources of information:

Launchpad info from TI: <http://e2e.ti.com/group/msp430launchpad/w/default.aspx>

MSP430X2XX family data sheet 693-page guide –
<http://www.ti.com/lit/ug/slau144j/slau144j.pdf>

Analyzing assembler programs

Assemble, load and try the following two programs. Notice what they do. Try pushing the two Launchpad buttons. Modify the programs as directed below each listing.

Explain what each command does and make appropriate comments on the program listing. Draw (eg with Libreoffice presenter or powerpoint etc) a flow chart for each program. Describe how it works.

The complete instruction set can be found in:

<http://www.ti.com/lit/ug/slau144j/slau144j.pdf> section 3.4

Information about interrupts is in the same manual in sections 2.2.2 to 2.2.4

The list of the interrupt vectors and flags for our microprocessor can be found in:

Microprocessor description msp430g2553 (slas735) page 11

The schematic showing how the LEDs and push buttons are connected can be found in:

LaunchPad Experimenter board user guide (slau318) page 15.

Program 1

```
.include "msp430g2553.inc"
org 0xc000
start:
    mov.w #WDTPW|WDTHOLD, &WDTCTL
    mov.b #0x41, &P1DIR
    mov.w #0x01, r8
repeat:
    mov.b r8, &P1OUT
    xor.b #0x41, r8
    mov.w #60000, r9
waiter:
    dec r9
    jnz waiter
    jmp repeat

org 0xffffe
```

```
dw start
```

A) Change the timing to make the blinking in Program 1 twice as fast. B) change the timing to make the blinking half as fast. If you have trouble copying and pasting this program out of the manual, you can download it from <http://www.phas.ubc.ca/~michal/P319/prog1.asm>

Program 2:

```
#include "msp430g2553.inc"
org 0x0C000
RESET:
    mov.w #0x400, sp
    mov.w #WDTPW|WDTHOLD, &WDTCTL
    mov.b #11110111b, &P1DIR
    mov.b #01001001b, &P1OUT
    mov.b #00001000b, &P1REN
    mov.b #00001000b, &P1IE
    mov.w #0x0049, R7
    mov.b R7, &P1OUT
    EINT
    bis.w #CPUOFF, SR
PUSH:
    xor.w #0000000001000001b, R7
    mov.b R7, &P1OUT
    bic.b #00001000b, &P1IFG
    reti

    org 0xffe4
    dw PUSH
    org 0xfffe
    dw RESET
```

Alter program 2 so that on pushes of the button, the LEDs will cycle through 1) both off → 2) red on/green off → 3) green on/red off → 4) both on → start over at 1 again. You can download this program at <http://www.phas.ubc.ca/~michal/P319/prog2.asm>

Labs 3&4 - Introduction to C programming of the microprocessor

There are several different environments available for compiling and running C programs on the MSP430. The most 'full-featured' is TI's Code Composer Studio (CCS). CCS is available for Linux, Mac, and Windows, though: the Linux and Mac versions **do not** support the MSP430G2553 LaunchPad board, and the Windows version doesn't work properly with LaunchPads connected to USB 3 ports (most modern computers). So, we will use an open source compiler tool chain: gcc and mspdebug (or msp430-gdbproxy). See the platform specific instructions on the course website for a guide to installing and using each of these environments.

You should do the installation **before** coming to the laboratory as the downloads may take quite a long time.

Below are the same 2 programs which you analyzed last week but this time written in C. Compile, load, and try them (follow the instructions in the Getting Started document for the platform you are using to compile and load them).

Look at the assembler code that your compiler produces and compare to the assembler versions from last week. Are they different? Make sure that you understand all the lines and comment them. Modify the C programs to obtain the same modifications you did in assembler. You can download this program at <http://www.phas.ubc.ca/~michal/P319/prog1.c>

Program 1 in C:

```
/*
 * PHYS319 Lab3 Timing example in C
 *
 * Written by Ryan Wicks
 * 16 January 2012
 *
 * This program is a C version of the assembly program that formed part of lab 2.
 * This is not the best way to implement timing, or to organize your code.
 * It is simply one way.
 *
 * This will almost certainly not give exactly the same timing as the assembly
 * program from lab 2, and the output assembly will also be very different, even
 * though the task is similar.
 */

#include <msp430.h>

void main(void) {
    volatile unsigned int count; //You must declare your variables in C
    // notice the label volatile. What happens if you remove this label?
    WDTCTL = WDTPW + WDTHOLD; //Stop WDT
    P1DIR = 0x41; //Set P1 output direction
    P1OUT = 0x01; //Set the output

    while (1){ //Loop forever
        count = 60000;
        while(count != 0) {
            count--; //decrement
        }
        P1OUT = P1OUT ^ 0x41; //bitwise xor the output with 0x41
    }
}
```

Program 2 in C (<http://www.phas.ubc.ca/~michal/P319/prog2.c>):

```
/*
 * PHYS319 Lab 3 Interrupt Example in C
 *
 * Written by Ryan Wicks
 * 16 Jan 2012
 *
 * This program is a C version of the assembly program that formed part of
 * lab 2.
 *
 */
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;           // Stop watchdog timer
    P1DIR = 0xF7;                       //C does not have a convenient way of
                                        //representing numbers in binary; use hex instead

    P1OUT = 0x49;
    P1REN = 0x08;                       //enable resistor
    P1IE = 0x08;                       //Enable input at P1.3 as an interrupt

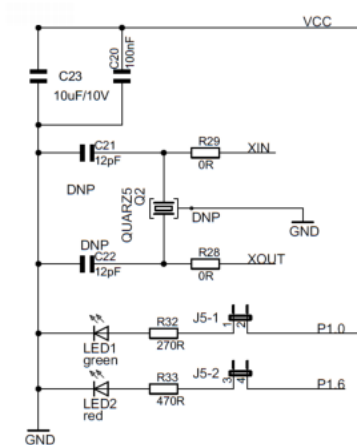
    _BIS_SR (LPM4_bits + GIE);         //Turn on interrupts and go into the lowest
                                        //power mode (the program stops here)
                                        //Notice the strange format of the function, it is an "intrinsic"
                                        //ie. not part of C; it is specific to this chipset
}

// Port 1 interrupt service routine
void __attribute__((interrupt(PORT1_VECTOR))) PORT1_ISR(void)
{
    //code goes here
    P1OUT ^= 0x41;                      // toggle the LEDS
    P1IFG &= ~0x08;                    // Clear P1.3 IFG. If you don't, it just happens again.
}

```

Analog to digital conversion (ADC) with Launchpad

The MSP430 microprocessor you are using has a 10 bit ADC that can sample the input of one of the IO pins. The example below will introduce you to its use. You are going to build a CMOS level tester. Your instrument will illuminate the red built-in LED when the voltage connected to the input pin level is High, the built-in green LED should indicate Low and an external yellow LED should light up when the voltage is in between acceptable levels. The diagram at the right shows the red and green LED's connected to pins P1.0 and P1.6 respectively. You'll need to look up the definitions of High and Low for 3.3V CMOS inputs.



The program below (by D. Dang from Texas Instruments) shows you how to set up the ADC. You will need to modify it to obtain the desired functionality.

For the 10-bit ADC full scale is $2^{10} - 1 = 1023 = 0x3FF$. The red LED should work with the example program, all you have to do is to get the green and yellow LEDs to indicate the respective logic levels. The test voltage should come from the potentiometer on your breadboard: connect one side to ground, the other side to 5V, and connect the wiper (center pin), **through a 1k Ω resistor**, to P1.1. Connect one of the other port 1 outputs to one of the yellow LED's on your breadboard. You can download the program from:

<http://www.phas.ubc.ca/~michal/P319/adc.c>

```
//*****
// MSP430G2x31 Demo - ADC10, Sample A1, AVcc Ref, Set P1.0 if > 0.75*AVcc
//
// Description: A single sample is made on A1 with reference to AVcc.
// Software sets ADC10SC to start sample and conversion - ADC10SC
// automatically cleared at EOC. ADC10 internal oscillator times sample (16x)
// and conversion.
//
//                MSP430G2x31
//                -----
//                /|\|                XIN|-
//                ||                |
//                --|RST                XOUT|
//                |                |
//                |                |
//                |                |
//                |                |
// input >---|P1.1/A1                P1.0|--> red Led onboard BIT0
//                |                |
//                |                P1.2|--> yellow Led
//                |                P1.6|--> green Led onboard BIT6
//
//
// D. Dang
// Texas Instruments Inc.
//*****
#include "msp430.h"

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;                // Stop WDT
    ADC10CTL0 = ADC10SHT_2 + ADC10ON;        // ADC10ON
    ADC10CTL1 = INCH_1;                       // input A1
    ADC10AE0 |= 0x02;                          // PA.1 ADC option select
    P1DIR |= 0x01 ;                            // Set P1.0 to output direction

    for (;;)
    {
        ADC10CTL0 |= ENC + ADC10SC;           // Sampling and conversion start
        while (ADC10CTL1 & ADC10BUSY);        // ADC10BUSY?
        if (ADC10MEM < 0x2FF)
```

```

    P1OUT &= ~0x01;           // Clear P1.0 LED off
else
    P1OUT |= 0x01;           // Set P1.0 LED on

unsigned i;
for (i = 0xFFFF; i > 0; i--); // Delay
}
}
//*****

```

Test the system and show the working instrument to your TA.

Pulse Width Modulation (PWM) waveform with Launchpad

PWM waveforms are frequently used to control power delivered to motors, lights, heaters and other devices without wasting energy in series resistors or voltage dividers. Our microcontroller has timer peripherals that allow creation of PWM waveforms without requiring the CPU to be continually setting output pins on and off – you configure the timer to do the work without CPU intervention. You are going to analyze a sample program producing a PWM waveform on output P1.2. You can download the program from <http://www.phas.ubc.ca/~michal/P319/pwm.c>

```

#include "msp430.h"

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Stop WDT

    P1DIR |= BIT2;           // P1.2 to output
    P1SEL |= BIT2;           // P1.2 to TA0.1

    CCR0 = 1000-1;           // PWM Period
    CCTL1 = OUTMOD_7;        // CCR1 reset/set
    CCR1 = 250;               // CCR1 PWM duty cycle
    TACTL = TASSEL_2 + MC_1; // SMCLK, up mode
    _BIS_SR(LPM0_bits);      // Enter Low Power Mode 0
}

```

Observe the resulting waveform on the oscilloscope. Connect the piezoelectric “beeper” to listen to the tone produced. Modify the program to create your favourite tone.

LED dimmer – combining the functionality of ADC and PMW.

Create a system which reads the voltage coming from your voltage source and sets the duty cycle of the PWM between 0 and full period. 0 voltage should correspond to 0 duty cycle and 3.3 V should correspond to 100% duty cycle. The period should be constant and of the order of 1 ms to avoid flickering of the LED. Connect the PMW output to the LED. You just created an LED dimmer!

Labs 5&6 - Data acquisition from Serial Port Interface and computer display

Your last task before beginning your individual project will be to use the MSP430 to read values from a sensor and report these values to the host computer where they are displayed and plotted. We'll begin by setting up some demonstration code that measures the temperature of the MSP430 from an internal sensor, and transmits this data to the host computer.

To get the data displayed on the computer you will need another program. We will use the python language on the host computer to talk to the Launchpad. See the platform specific instructions to install the needed components.

Links to download this example are provided with the platform specific instructions. You should install python and the other required libraries **before** coming to the lab. The msp430 program we'll use was originally supplied by an engineer at TI (and then simplified locally). The python program can be found in the same download.

Download the temperature_demo3 example code and follow the directions to get it working.

Newer Windows computers seem unable to properly configure the USB interface on the Launchpad board over a USB 3 port. If you have trouble getting this working, consider using the lab computers, or downloading the operating system image supplied to boot from a flash drive.

The C code for the Launchpad program is longer than the programs we've used so far, but you can get a lot of insight into Launchpad C programming by studying it. For now just download it, compile it, and load it to your microprocessor.

The program will start in an idle mode where it blinks the LED's. When the P1.3 button is pushed, it will begin taking temperature measurements and sending them to the USB port. Don't push the switch till you've got the python program started up.

Before starting the python program, you will have to check which serial port your Launchpad is connected to. See the platform specific instructions.

Once the python program is up and running, press the P1.3 (s2) button and the temperature will be updated on the window titlebar, and a graph of the temperature as a function of time should show on the screen (in Fahrenheit).

In this microprocessor program the ADC readings are taken from the built in temperature sensor.

The result is converted to the temperature in Fahrenheit:

```
TXByte = (unsigned char) ( ((tempAverage - 630) * 761) / 1024 );
```

and transmitted in RS232 data format via USB port from the void function `Transmit()`.

Inspect the compiled assembly code – note the length! One line of C may be converted into many lines of assembly.

Your task is to create a system that uses the MSP430 to make distance measurements using an SRF04 ultrasonic distance measurement sensor, and plot them in real time. You can find details of the sensor at:

<http://www.robot-electronics.co.uk/htm/srf04tech.htm> and here

<http://www.micropik.com/PDF/HCSR04.pdf>

You are free to use pieces of the demo programs. A complete report will include some demonstration of the accuracy of the device, including a graph of distances measured with a metre stick versus those measured with your sensor. Your lab notes should fully describe how the programs work, and all the relevant communications protocols.

You should read about the sensor, but the important bits of how to use it are:

- the microcontroller (ie your program) triggers a measurement with a $\sim 10 \mu\text{s}$ long pulse on the trigger pin.
- the sensor will initiate a measurement and then raise the echo pin
- when an echo returns (or the sensor gets tired of waiting, $\sim 36 \text{ ms}$), the sensor lowers the echo pin.
- wait 10 ms or more before repeating.

You need to measure how long the echo pin is held high by the sensor. That time, along with the speed of sound will give you a distance.

You will need to connect the trigger pin to a port P1.x pin that is configured as an output, and connect the echo pin, through a 4.7 or 10 k Ω resistor (its a 5 V device!) to a different port P1.x pin that is configured as an input.

Some advice: Start simple. Get distance measurements working, and communicate single bytes (for short distances) to the host computer. If you have time, revise the communications protocol to communicate multiple bytes to allow longer distances/higher precision measurements to be communicated. There are simple ways of measuring the time, and very elegant, high precision ways. Start simple. When you get a simple method working, save your code and if you have time, feel free to work on an elegant, higher precision version.