

Enter the names and SIDs for you and your lab partner into the boxes below.

Name 1	<div style="border: 1px solid black; width: 350px; height: 25px;"></div>	SID 1	<div style="border: 1px solid black; width: 80px; height: 25px;"></div>
Name 2	<div style="border: 1px solid black; width: 350px; height: 25px;"></div>	SID 2	<div style="border: 1px solid black; width: 80px; height: 25px;"></div>

Be sure to come *very well prepared* to this laboratory to complete in three hours (or finish the following week).
Type in your code ahead of time (store on the server or a USB key).

Microcontrollers

Microcontrollers are very much slimmed down computers. No disks, no virtual memory, no operating system. Think of them just like other circuit components with the added benefit of being configurable with a program. Because of this microcontrollers can be coaxed to do all sorts of things simply that otherwise would require a large number of parts. Simple microcontrollers cost less than a dollar and hence can be used in almost any project. Indeed they can be found in toys, electric tooth brushes, appliances, cars, phones, electronic keys, you name it.

Being programmable also means that they must be programmed. In this lab we concentrate on the electrical interface of microcontrollers and their use as electronic components. The programs we use are very simple and consist to a large part of pasting snippets of code together. In fact, much like checking the application notes of electronic components for circuits that do what we need, it's always a good idea to search the web for code that performs the job we need or is at least a good starting point. Most of the code snippets shown in these lab guides are copies of code from the manufacturer's website. Feel free to improve on the example programs.

Microcontrollers are available from many manufacturers, all with their own advantages (and quirks). In this course we use the MSP430 from Texas Instruments whose strength are low power dissipation and a regular instruction set.

Figure 1 shows the architecture of the MSP430 (specifically the model MSP430F2012). The CPU block is the part that actually performs computations (e.g. additions). Note that microcontrollers usually lack hardware for multiplication or division. These operations can be emulated in software, albeit at the price of low execution speed. The clock system sets the operating speed (16MHz maximum for the controller we are using, compare this to 2GHz or so for present day laptops). A 555-timer like clock is built right into the chip; alternatively an external oscillator can be used if higher precision is required.

Flash is a nonvolatile memory for storing programs and configuration data. RAM is where temporary variables go. Note again the contrast to full blown computers: microcontroller memory is typically a few kBytes flash and a few hundred Bytes RAM. Most laptops today have at least a GByte RAM, a million kBytes. You don't need this in an electrical tooth brush. JTAG and Spy-Bi Wire are nifty interfaces for programming and debugging (the microcontroller has no keyboard or LCD display). We will use this interface to talk to the controller though USB

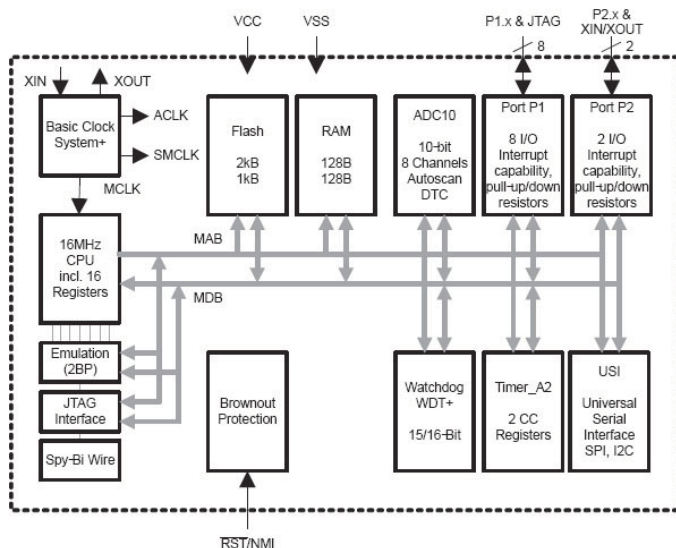


Figure 1 Block diagram of the MSP430 microcontroller. In addition to the processing unit (CPU), clock, and programming interface (Spy-Bi Wire), the chip includes program (Flash) and data (RAM) memory, digital inputs and outputs (Ports P1 and P2), Timers, and an analog-to-digital converter (ADC).

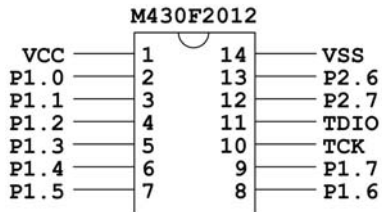


Figure 2 M430F2012 pin diagram. The MSP430 comes in many versions that include different amounts of Flash and RAM memory and combinations of input/output ports and other peripherals. Here we use a very small version with only 14 pin that costs less than a dollar in quantity.

from a desktop computer and program the Flash memory. The Spy-Bi Wire interface is used only for development, once completed the controller works standalone from the program stored in the nonvolatile Flash memory.

The really interesting parts are the peripherals. Ports P1 and P2 are digital I/Os that can be configured as inputs or outputs. They can be used for simple I/O with switches or LEDs; in later labs we will see much more sophisticated uses of this simple interface. The other block we will use is the ADC10, a 10-bit analog-to-digital converter that serves as a bridge between the usually analog “real” world and the microcontroller. For example we can use it to interface the strain gage circuit designed in an earlier lab to the microcontroller and make a full blown (simple) balance with display out of the combination!

Figure 2 shows the pinout of the particular MSP430 we are using in this laboratory, the MSP430F2012. It has only 14 pins for power (VCC) and ground (GND), the debug interface (TDIO and TCK) and ports P1 and P2. Looking carefully you will observe that there are eight pins for P1 but only two for P2. The other ones don’t fit with a 14 pin package. There also are no separate pins for the ADC. Instead, digital IO pins can be reprogrammed as ADC inputs as needed. Several dozen MSP430 microcontrollers are available with their main difference being the number of pins and the amount of memory. This permits you to start with a small version, and as the project grows move to versions with additional memory or pins without having to change the programs developed for the smaller parts.

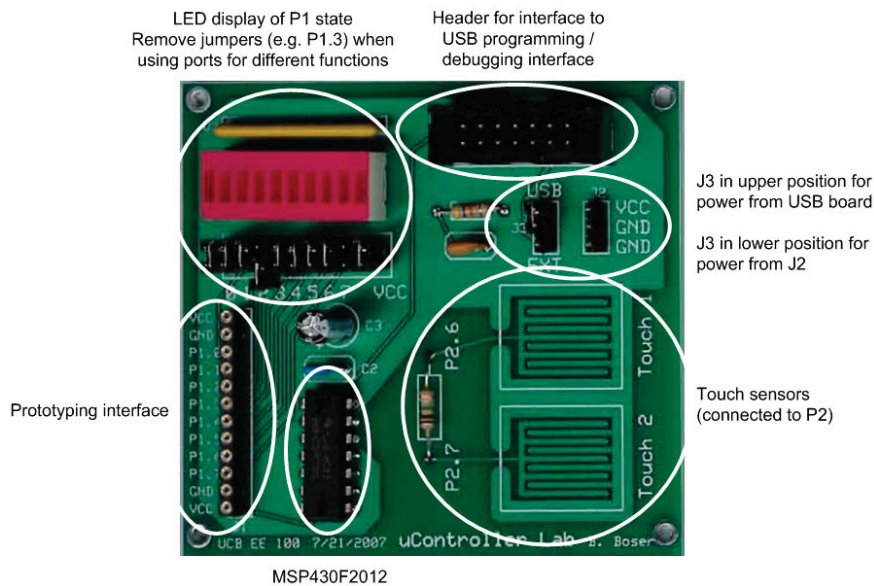


Figure 3 Picture of the microcontroller prototyping board used in this laboratory. In addition to the MSP430, the board features 8 LEDs and resistors with jumpers that can optionally be connected to IO port P1. The printed circuit board (PCB) traces labeled “touch sensor”, together with the adjacent resistor and a simple program allow the processor to react to finger touch input.

With only 14 pins we certainly could wire up the microcontroller on a protoboard. The custom board shown in Figure 3 makes wiring even simpler. It also features LEDs for debugging and a touch interface.

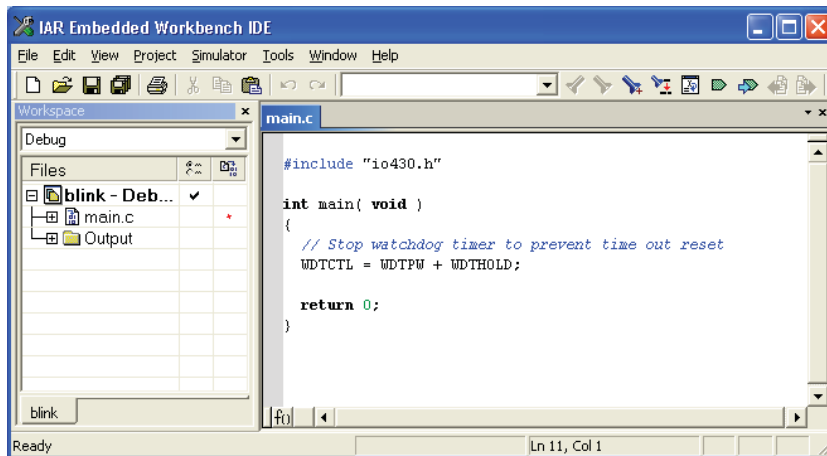
In addition to the microcontroller, the board features a header for interfacing with a ribbon cable to the USB interface which is also supplying power to the board. Three capacitors and a resistor are used for filtering power and resetting the device after power is applied. These devices are specified by datasheet of the device. A header strip on the left side of the board exposes P1 and P2 and power for prototyping.

Alternatively P1 also can be connected to on-board LEDs through jumpers. This of course makes sense only for pins that are configured as digital outputs. Removing a jumper (or setting it on a single pin as shown for P1.3 for not losing it) makes that port available for other functions such as digital input or the ADC. The rightmost position (VCC) is for monitoring the power supply and always on when power is supplied to the board and the jumper is inserted. Jumper the upper two terminals of J3 as shown to enable power from the USB interface.

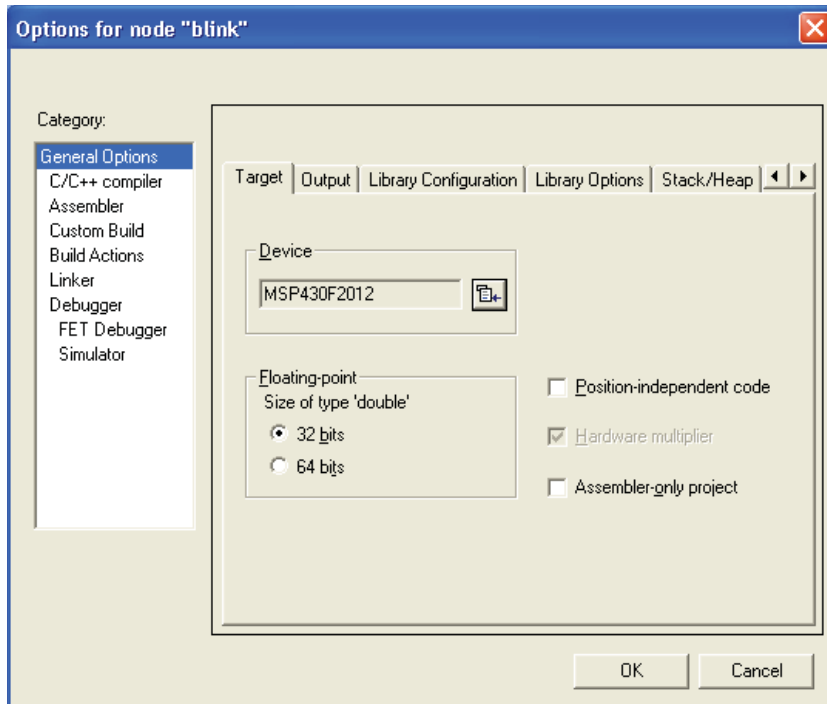
Flashing Light

In this laboratory we will familiarize ourselves with the μ Controller board and the MSP430 development tool. We will start with writing the notorious “Hello World” example, which for a μ Controller is a blinking light.

- Connect the μ Controller PCB to the MSP430 USB-Debug-Interface (MSP-FET430UIF). Use a standard USB cable to connect the debug interface to a computer with the “IAR Embedded Workbench IDE” software. This software is installed on the computers in the laboratory. Alternatively you can download it from the TI website and install on your own computer.
- Start the IAR Embedded Workbench IDE. Choose “Create new project in current workspace”. A dialog with options appears. We will write our program in the C language. Expand that choice, click on “main” and then click “OK”. The program asks you to name your project. Call it “blink”. Hit enter.
- Your screen looks as shown below, with a program already partially written.¹



- Before finishing our program we need to configure the tool for the MSP430F2012. Choose the menu “Project→Options ...” and click on the “General Options” tab. Set “Device” as shown in the screen below by clicking on the button to the right of the field and navigating the choices.



¹The newest version of the software no longer generates this code and the file `io430.h` no longer exists. Please enter the missing statements yourself and replace `io430.h` by `msp430x20x2.h`. You need to do this in all your programs.

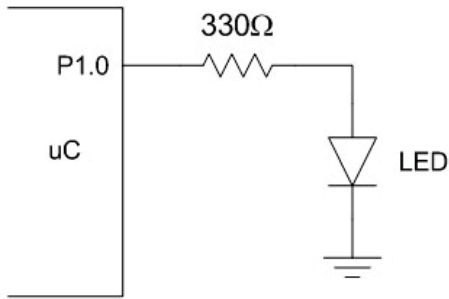
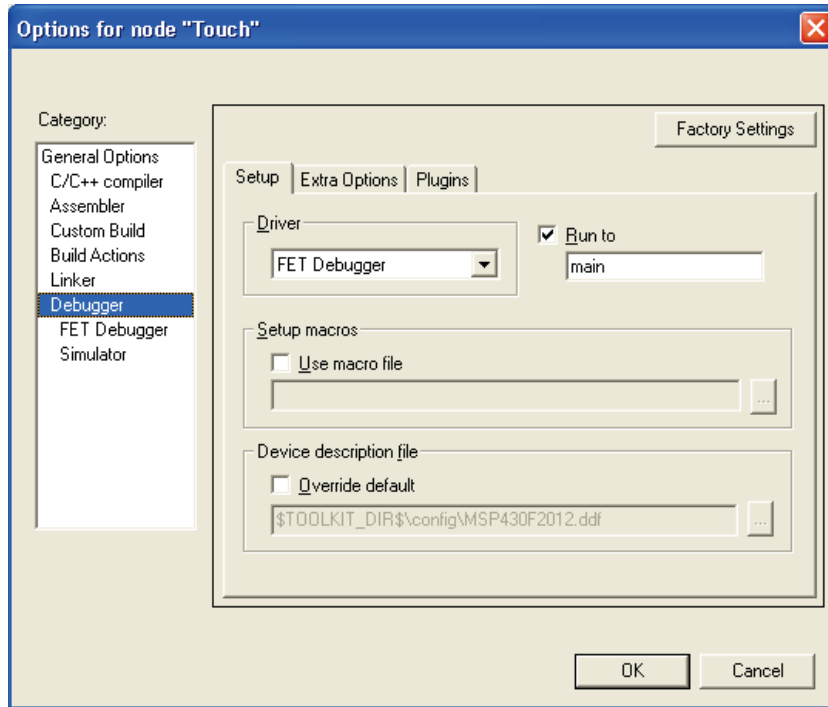


Figure 4 Circuit diagram for connecting an LED to a μ Controller output port. The resistor is needed since LEDs behave like diodes with very large current flowing for voltages above a threshold (around 1 V). The microcontroller output changes between 0 V and the supply voltage (around 3 V) for logic low and high, respectively. Without the resistor a large current flows. This can result in either the μ Controller port or the LED to burn out.

- e) Still in the options dialog, verify under the “Debugger” tab that driver is set to “FET Debugger”. Also make sure that under the “FET Debugger” tab “Connection” is set to “TI USB-IF”.



Click ok.

Put a jumper into position 0 of J5 to connect the microcontroller port P1.0 to the leftmost LED. Figure 4 shows the circuit diagram for the on-board LEDs.

Place a jumper also in the VCC position of J5 to verify that power is supplied to the board when you start debugging.

Now we write the program that turns P1.0 on and off.

- a) First we to configure P1.0 as a digital output using the following instructions (see online manual for more information):

```
P1OUT &= ~BIT0;
P1SEL &= ~BIT0;
P1DIR |= BIT0;
```

The first statement sets the output to zero. It is not strictly needed here since we do not care if the LED is on or off when the microcontroller starts. However it is a good idea to always first set an output to a known state before enabling it, to avoid potentially disastrous failures.

The next two statements configure P1.0 as digital IO with direction set to output. As you would expect, the same statements with BIT1 substituted for BIT0 enable P1.1 as output.

Put these statements just after the code that disables the watchdog timer (don't worry about that timer; just make sure the code is there to turn it off).

b) The following statements set the output low or high, or toggle its state:

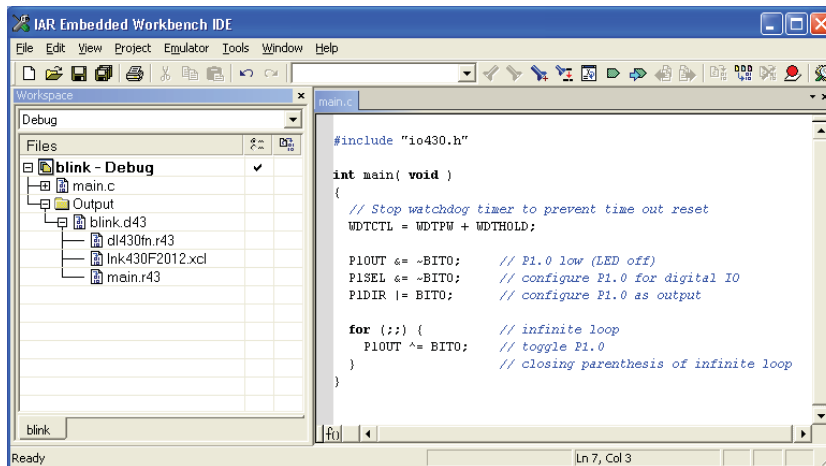
```
P1OUT &= ~BIT0; // P1.0 low
P1OUT |=  BIT0; // P1.0 high
P1OUT ^=  BIT0; // toggle P1.0
```


Text after // is treated as comment and there only for documentation. If you have ever taken a programming course you know that we are supposed to document our code but few of us actually do it. Join the few.

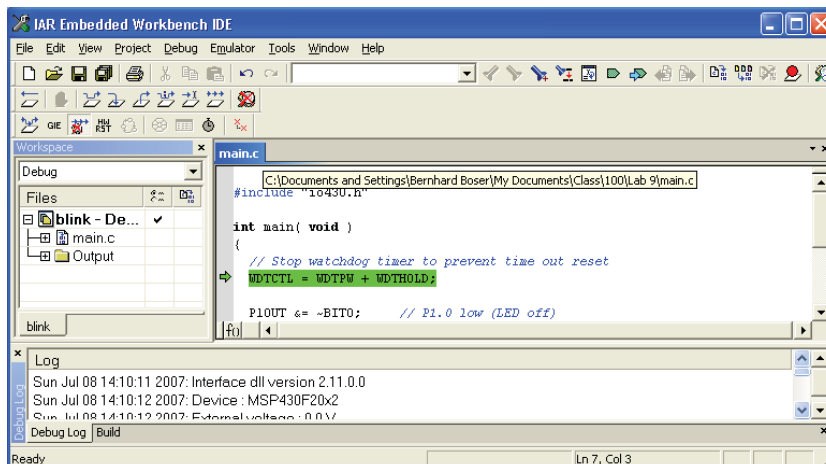
c) The toggle version of the above statements is appropriate for blinking a light. Since we want to do this repeatedly we enclose the statement in a loop:





```
for (;;) { // c-parlance for infinite loop
    P1OUT ^=  BIT0; // toggle P1.0
} // closing bracket for infinite loop
```

d) The complete program looks as follows:



e) Compile it by clicking on the right most button  in the toolbar. Carefully examine possible error messages (or ignore them and waste lots of time in frustration). A new window pops up:



Click the  button (2nd row in the toolbars). When done you can stop the program by clicking  or modify the code and click  to recompile and restart with .

f) If everything went right the light turns on but does not blink. At best it is a little dimmer than the power LED. The reason is that the microcontroller turns the LED on and off a few million times per second, too fast for our eyes to follow.

- g) The simplest fix is to give the microcontroller a bit of extra work to slow it down. Microcontrollers are not students and hence do not mind. Since we will often have need for such delays, we package this function in a subroutine that we can easily reuse in other programs. Here is the code:

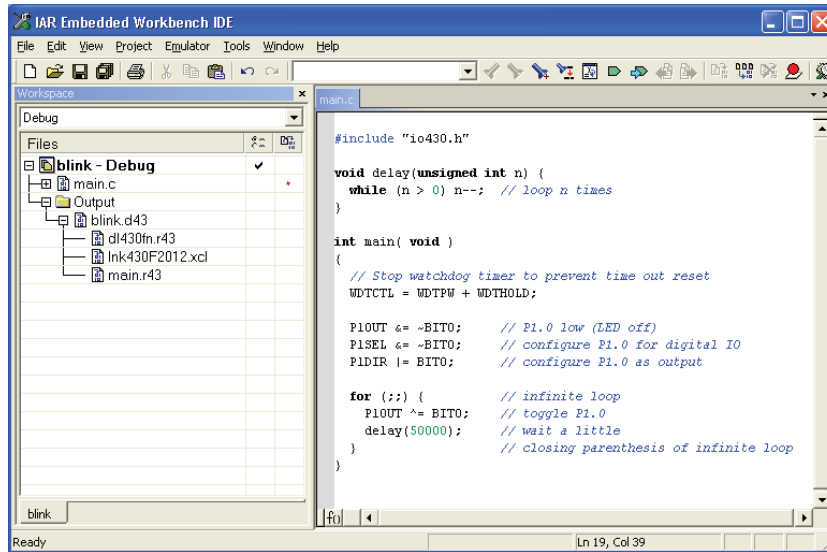
```
void delay(unsigned int n) {
    while (n > 0) n--;
}
```

calling this code with the following statement

```
delay(30000);
```

causes the microprocessor to spin 30'000 times in a loop decrementing the variable n.

- h) Let's try this in our program:



- i) You can change the rate of the blinking by varying the argument to delay. Beware: the maximum permitted value is a little over 65'000 (you'll get a warning if you exceed the maximum which you are free to ignore if you need a few hours of frustration with stuff that does not work). For longer delays you can e.g. use several delay statements. Can you get the LED to blink at a rate of 1Hz?

If you paid attention the answer to this question is trivial. Assuming P1.0 has been configured as an output; what is the code to set it to VCC? Omit extra blanks from your answer.

Analog Output

So far we have used the μ Controller to turn an LED on or off. Frequently an "analog" output is required, i.e. a voltage can assume any value between the supplies. Some μ Controller contain special peripherals called digital-to-analog converters (DACs) for this purpose. The model we are using lacks this feature. However, we can emulate an analog output with a digital output by rapidly switching its value between the supply voltage and ground and adjusting the relative times the output is high and low. For example, if the output is high ($V_{CC}=3\text{ V}$) during 3 clock cycles and then low (0 V) during 2 cycles, the average value of the output is $3/5V_{CC}=1.8\text{ V}$. Likewise, if the output stays high for 21 cycles and low for 77, the average output voltage is

Write a subroutine function `dac(n)` that sets the output high during n cycles and low during 256-n cycles. Here is a start:

```
void dac(unsigned int n) {
    ... set output high ...
}
```



```

delay(n)
... set output low ...
delay(256-n)
}

```

Call this function from your main program in an endless loop. Write your completed code into the box below:

1 pt.
4

Test your “DAC” in the laboratory. First verify with the oscilloscope that for $n=0$ the output remains low (why is there a “glitch” and how could you eliminate it?), and for $n=256$ the output remains high (except for a brief glitch). Then connect a first order RC filter with $R = 10\text{ k}\Omega$ and $C = 10\text{ nF}$ to the $\mu\text{Controller}$ output and record the voltage v_o across the resistor on the oscilloscope. Adjust n such that $v_o = 1.3\text{ V}$ and ask the GSI to verify your result.

One drawback of this DAC is that it keeps the $\mu\text{Controller}$ busy and unavailable to do other tasks. A better solution uses the $\mu\text{Controller}$ ’s timer combined with a feature called interrupt to implement the delay. With this technique, the dac function can run concurrently with other functions. We will not explore this feature here; please check the manual and sample programs (see vendor website) for more information.

Bar Graph

Now we will program a function void bar(int n) that turns on LEDs 0 ... n-1. E.g. the call bar(3) turns on LEDs 0, 1, and 2. Here is a start:

```

void bar(unsigned int n) {
    if (n >= 1) P1OUT |= BIT0;   else P1OUT &= ~BIT0;
    if (n >= 2) P1OUT |= BIT1;   else P1OUT &= ~BIT1;
    ... complete code for LEDs 2 ... 8
}

```

Write your completed code into the box below:

1 pt.
6

At the start of main, declare a variable i as follows and configure all bits of P1 as outputs (leave the code for the watchdog timer alone) and write code that spins through the bar graph:

```

int main( void )
{
    int i = 0;

    // Stop watchdog timer to prevent time out reset

```

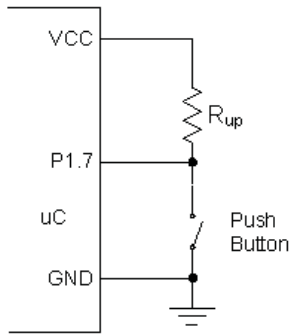


Figure 5 Circuit for digital push-button input.

```

WDTCCTL = WDTPW + WDTOLD;

P1OUT = 0;      // set P1.0 ... 7 low
P1SEL = 0;      // configure P1.0 ... 7 for digital io
P1DIR = 0xff;    // configure P1.0 ... 7 as outputs

for (;;) {
    bar(i);      // display
    i = i + 1;    // increase bar
    if (i > 8) i = 0; // maximum length of bar is 8 LEDs on
    delay(50000); // wait for humans to see result
}
}

```

Make sure all LED jumpers 0 ... 7 are inserted and test the program (optional).

Digital Input

Now will modify our code from the last section such that the bar graph advances one position each time a button is pressed. We will use P1.7 for the button input. Remove the jumper for LED 7.

Figure 5 shows the circuit for connecting a button to the microcontroller. Normally P1.7 is pulled to VCC (high) by the pull-up resistor R_{up} . Pressing the button pulls P1.7 low. The circuit is even simpler than this since the microcontroller has a built-in R_{up} , we just need to enable it. So all you need to do is connect a button between P1.7 and ground. Here is the code for enabling P1.0 ... 6 as outputs and P1.7 as an input with the pull-up resistor enabled:

```

P1OUT = BIT7;
P1SEL = 0;
P1DIR = 0x7f;
P1REN = BIT7;

```

The following statement checks for P1.7 to go low:

```

while (P1IN & BIT7);    // wait for P1.7 to go low

```

Put this into the infinite loop to control the bar graph display.

Start your program and verify first with the voltmeter that P1.7 is normally high and turns low when the button is pressed. When trying the program you will notice that things do not work as expected: when pressing the button the bar graph does not advance a single position as it should but a random number of positions. This has two reasons:

- Just like human eyes, human hands are pretty slow on a microcontroller timescale. Once the microcontroller has found that P1.7 is low and advanced the bar graph, it returns to the top of the loop, and checks again. If it finds the button still pressed, it advances the bar graph again.
- Most mechanical switches and buttons have a flaw called prelling: rather than just turning on and off, the button turns on and off rapidly in succession until it reaches the final value. The reason is mechanical resonance in the switch.

The first problem can be avoided by inserting a statement that checks that the button has gone back high into the loop:

```
while (!(P1IN & BIT7)) ;    // wait for P1.7 to go high
```

Depending on where exactly you insert the statement the bar graph advances immediately when the button is pushed or only when the button is released.

The solution for the second problem is inserting a delay into the loop. Fortunately we wrote that delay function! Code for bar graph with button:

3 pts.
7

Assuming P1.5 has been configured as an output; what is the code to set it to GND?

Build the circuit and verify the operation (optional).

Analog Input

Microcontrollers (and computers in general, for that matter) operate with digital data. However, many “real world signals” such as temperature are analog in nature. An analog-to-digital converter is needed to input such signals into a microcontroller. Analog-to-digital converters, or ADCs for short, are available as standalone electronic components or built into more complex devices. Fortunately our microcontroller has an ADC built in. In this laboratory we will use this ADC to interface the weight scale designed in an earlier laboratory to the microcontroller and have the bar graph display the number of weights put on the scale.

ADCs compare an analog input v_i , e.g. 1.387V, to a reference voltage V_{ref} to produce a digital number representing the ratio of the analog input to the reference rounded to the nearest integer. For example, the ADC in our microcontroller converts analog voltages to digital numbers according to the following equation:

$$N = \text{round} \left(1023 \times \frac{v_i}{V_{\text{ref}}} \right) \quad (1)$$

For example, with $V_{\text{ref}} = 3\text{ V}$ and $v_i = 1.387\text{ V}$, $N = 473$. Negative input voltages and inputs exceeding the reference produce are clipped to zero and 1023, respectively.

The program skeleton below shows the code for using the ADC and configures P1.7 as its input. Conversion results are automatically stored in the variable ADC10MEM.

```
int main( void ) {
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;

    ADC10CTL1 = INCH_7;           // P1.7 is ADC input
    ADC10CTL0 = SREF_0            // reference is VCC
        + ADC10SHT_2             // sample rate
        + ADC10ON;               // enable ADC

    P1OUT = 0;                   // initialize
    P1SEL = BIT7;                // P1.0 ... 6 are digital I/Os, P1.7 is A7
    P1DIR = 0x7f;                // P1 direction is output
```

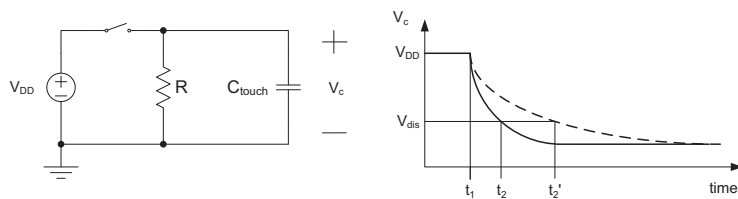


Figure 6 Touch sensor based on RC discharge time.

```
for (;;) {
    ADC10CTL0 |= ENC + ADC10SC;    // enable and start conversion
    while (ADC10CTL1 & ADC10BUSY); // wait for conversion to complete
    if (ADC10MEM > ...) bar(1);      // display result on bar graph
    ...
}
}
```

Complete the program such that the bar graph represents graphically the input voltage of the ADC for $V_{\text{ref}} = 3 \text{ V}$. I.e. no LEDs on for $v_i < 50 \text{ mV}$ and LEDs 0 to 6 on for $v_i > 2950 \text{ mV}$ with linear interpolation in-between. Write the complete program in the space below:

2 pts.
8

Connect a potentiometer between VCC and GND and connect the middle tap to P1.7. Verify proper circuit operation and show the result to the GSI².

With this introduction you are in a good position to tackle projects using μ Controllers for all sorts of applications including electronic thermometers, scales, touch sensor interfaces, light shows, etc.

With V_{ref} set to 3 V, what value v_i corresponds to a reading $N = 199$? Specify the middle of the range.

$v_i =$ ^{1 pt.}₃

Touch Sensor

We are now ready to design a real application with the microcontroller. The PCB contains two areas with closely spaced conductors that serve as touch sensors. We will now write a program to read them out.

How can a microcontroller possibly measure a capacitance? A particularly simple solution consists in precharging the capacitor to a known voltage and then measuring its discharge time through a resistor. Larger capacitance results in increased charge and hence increased discharge time.

Figure 6 illustrates the concept. The switch is initially closed and opens at time t_1 . The voltage $v_c(t)$ across the capacitor decreases as C_{touch} discharges through R . The discharge time is calculated as the difference between the time t_2 when $v_c(t)$ decays to some reference level V_{dis} and t_1 . Touching C_{touch} increases its value and hence the discharge time to t_2' . Figure 7 on the following page shows the connections to the microcontroller.

²You may need to replace `io430.h` with `msp430x20x2.h` to avoid compile time errors.

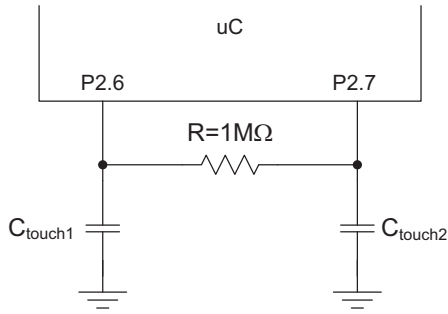


Figure 7 μ Controller touch sensor interface. The capacitor consists of copper traces on the controller PCB.

Measuring the discharge time of C_{touch1} involves the following steps:

- a) Set P2.7 low
- b) Configure P2.6 as output
- c) Set P2.6 high to charge C_{touch1} to V_{dd}
- d) Wait a brief period of time to ensure C_{touch1} is fully charged, e.g. `delay(10)`
- e) Change the direction of P2.6 to input
- f) Store the current time in variable t_1
- g) Wait for P2.6 to go low
- h) Store the current time in variable t_2
- i) The discharge time equals $t_2 - t_1$

The current time is available in a variable called TAR. For example, the statement `t1 = TAR` takes a snapshot of the current time and stores it in variable `t1`. The skeleton program listed below (also available from the website for download) sets up the microcontroller configuration and then calls the function `touch1` repeatedly to measure the discharge time of C_{touch1} .

```
#include "io430.h"

void delay(unsigned int n) {
    while (n > 0) n--;
}

// RC capacitive sensor on P2.6 discharging into P2.7
int touch1() {
    unsigned int t1 = 0;           // start of discharge interval
    unsigned int t2 = 0;           // end of discharge interval

    // Set P2.7 low
    ...
    // Set P2.6 high to charge Ctouch1 to VDD
    ...
    // Wait a brief period of time to ensure Ctouch1 is fully charged
    delay(10);
    // Change the direction of P2.6 to input
    ...
    // Store the current time in variable t1
    ...
    // Wait for P2.6 to go low
    ...
    // Store the current time in variable t2
    ...
}
```

```

    // The discharge time equals t2-t1
    return t2-t1;
}

int main(void){
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTCTL;
    // maximum clock speed
    BCSCTL3 = LFXT1S_2;          // frees up pins
    BCSCTL1 = CALBC1_16MHZ;
    DCOCTL = CALDCO_16MHZ;

    // turn on timer A free running
    TACTL = TASSEL_2 + MC_2;

    // P1 is feedback
    P1DIR = 0xff;                // P1 output
    P1SEL = 0;                   // digital out

    // configure P2 for capacitive sense
    P2OUT = 0;                   // P2 output low
    P2DIR = BIT6 + BIT7;         // P2 is output
    P2SEL = 0;                   // P2 is digital i/o
    P2REN = 0;                   // P2 disable pullup resistors

    for (;;) {
        unsigned int T = touch1();
        // display T on LED bar (T ~ 200, typ)
        ...
        // wait a little
        delay(100);
    }
}

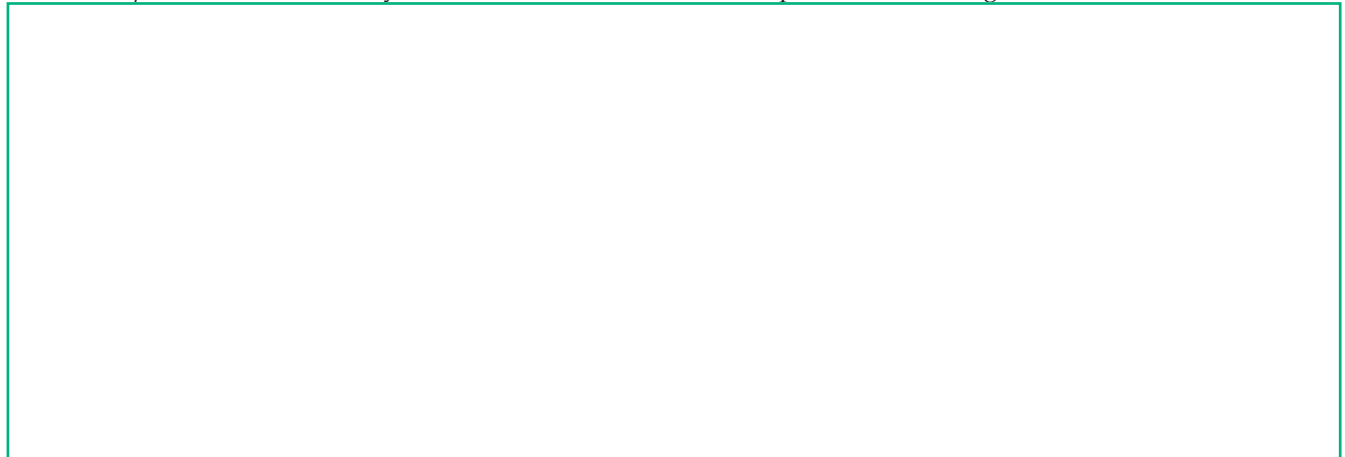
```

Complete the skeleton program to read touch sensor 1 before the start of the lab.



5 pts.
10

Run your program on the microcontroller. Attach a scope probe (set to 10x attenuation) to the resistor at port P2.6 of the μ Controller and verify that the waveform shows the exponential discharge.



1 pt.
11

Modify the code such that the length of the bar graph is an indication of the presence and size of a finger on the touch sensor. E.g. no or a single LED lit with no finger present, a few LEDs lit with a small finger or a finger at some distance from the sensor, and all LEDs lit with the entire touch sensor covered.

Have the GSI verify your result.



Password:

EE 42/100 Lab 6 Microcontroller I/O

Prelab Summary

Name: _____

SID: _____

Answers

0. _____

1. _____

2. _____

3. _____