NAME 1: SID:

NAME 2: SID:

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 easily 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 toothbrushes, 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 strengths are low power dissipation and a regular instruction set.

Figure 1 shows the architecture of the MSP430 line (specifically the model MSP430F2012; we will be using the more complex MSP430F5438). 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 (18MHz 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.

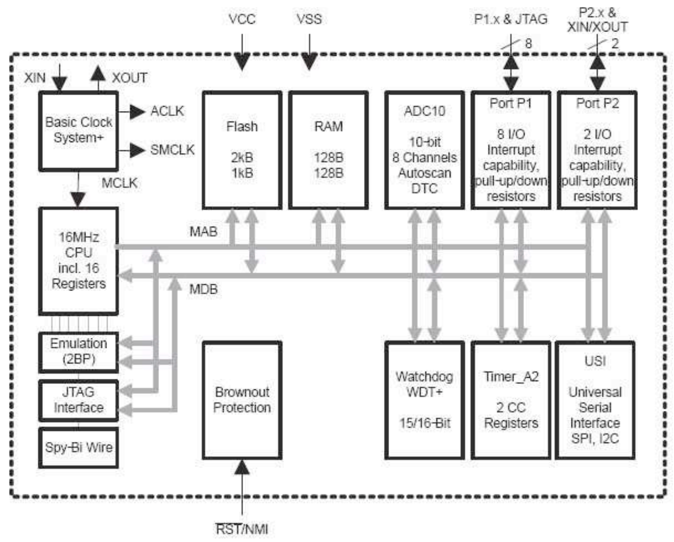

VCC
VSS
P1.x & JTAG
8
P2.x & XIN/XOUT
2
XIN
XOUT
Basic Clock System+
ACLK
SMCLK
MCLK
Flash 2kB 1kB
RAM 128B 128B
ADC10 10-bit 8 Channels Autoscan DTC
Port P1 8 I/O Interrupt capability, pull-up/down resistors
Port P2 2 I/O Interrupt capability, pull-up/down resistors
16MHz CPU incl. 16 Registers
MAB
MDB
Emulation (2BP)
JTAG Interface
Spy-Bi Wire
Brownout Protection
Watchdog WDT+ 15/16-Bit
Timer_A2 2 CC Registers
USI Universal Serial Interface SPI, I2C
RST/NMI

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 analog-to-digital converter (ADC).

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 kilobytes of flash and a few hundred bytes of RAM. Most laptops today have at least a gigabyte of RAM (a million kilobytes). You don't need this in an electric toothbrush. JTAG and Spy-Bi Wire are nifty interfaces for programming and debugging (the microcontroller has no keyboard or LCD display). We will use the JTAG interface to talk to the controller though USB 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/O buses 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. Another block we will use is the ADC, an 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!

In this laboratory we are using a fairly powerful MSP430, the MSP430F5438. Despite its large number of pins, there are no separate pins for the ADC. Instead, some 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.

The specialized board shown in Figure 2 makes working with the microcontroller hardware very simple. Many forms of I/O are provided, from simple buttons and LEDs to graphics and sound. Even serial communication with a computer is readily available.

In addition to the microcontroller, the board features a header along the top edge for interfacing with a ribbon cable to the USB interface, which will also supply power to the board. Another header below that one provides access to all eight bits of port P10. A third header strip on the bottom edge of the board provides various pins from many ports, each capable of digital I/O but some capable of ADC input or other functions.

The first two pins of port P1, P1.0 and P1.1, are permanently connected to on-board LEDs. These pins are not connected to any header, so there is no potential conflict of function, but the pins must be configured as digital outputs for the LEDs to work.

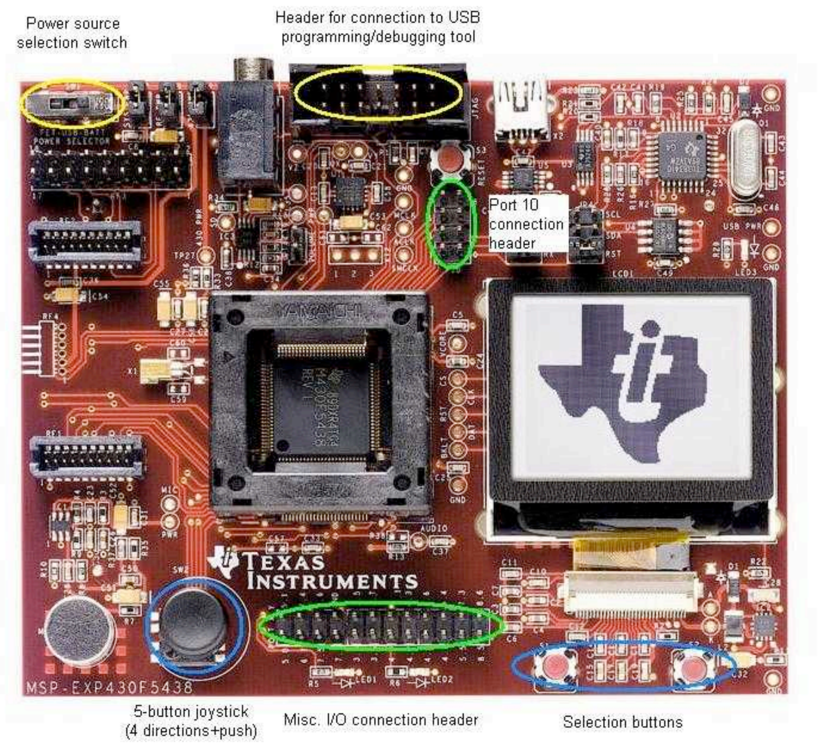

Power source selection switch
Header for connection to USB programming/debugging tool
Port 10 connection header
5-button joystick (4 directions+push)
Misc. I/O connection header
Selection buttons
MSP-EXP430F5438
TEXAS INSTRUMENTS

Figure 2 - Picture of the microcontroller prototyping board used in this laboratory. In addition to the MSP430, the board features two LEDs (connected to I/O port P1), a microphone, a headphone jack, an LCD display, and seven input buttons (connected to I/O port P2). We can do many things with the components on the board, and the MSP430 can communicate off-board using the pins on the I/O connection headers.

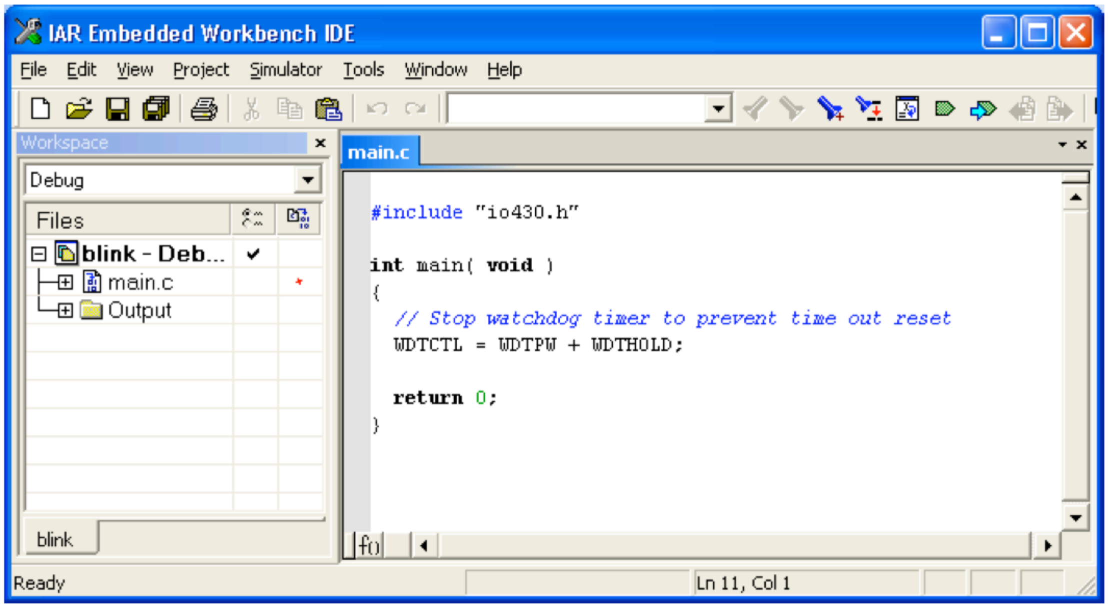
Digital Output (Flashing Light)

In this laboratory we will familiarize ourselves with the microcontroller board and the MSP430 development tool. We will start by writing the classic example of a first program, which for a microcontroller is a blinking light.

1. Connect the microcontroller PCB to the MSP430 USB-Debug-Interface (aka MSP-FET430UIF - the small gray box). 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 IAR website and install on your own computer.

2. 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.

3. Your screen looks as shown below, with a program already partially written.

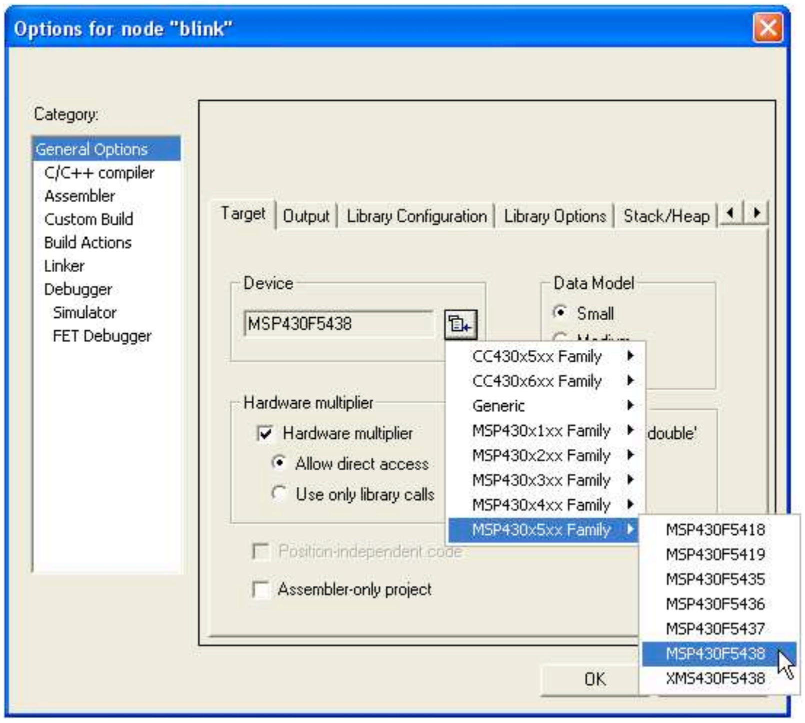

IAR Embedded Workbench IDE
File Edit View Project Simulator Tools Window Help
Workspace
Debug
Files
blink - Deb...
main.c
Output
blink
main.c
Ready
Ln 11, Col 1

```
#include "io430.h"

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

  return 0;
}
```

4. Before finishing our program we need to configure the tool for the MSP430F5438. 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.



Device: MSP430F5438

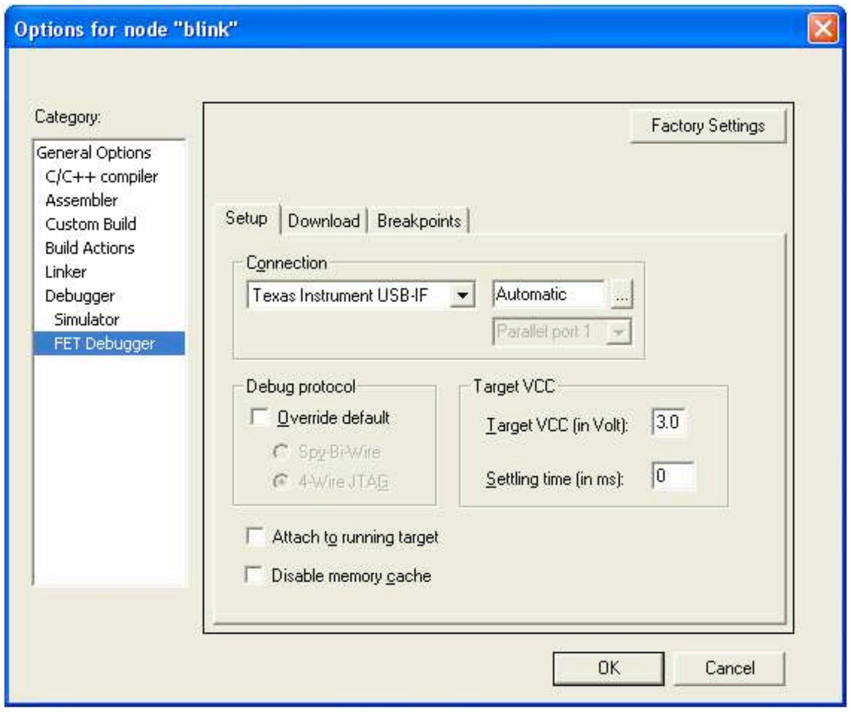
Data Model: [✓ Small]

Hardware multiplier: [✓ Hardware multiplier]; [✓ Allow direct access] Use only library calls

[ ] Position-independent code

[ ] Assembler-only project

5. Still in the options dialog, **verify under the "Debugger" tab that the driver is set to "FET Debugger"**. Also make sure that under the "FET Debugger" tab, "Connection" is set to "Texas Instrument USB-IF".


Options for node "blink"

Category:
- General Options
- C/C++ compiler
- Assembler
- Custom Build
- Build Actions
- Linker
- Debugger
- Simulator
- FET Debugger

Factory Settings

Setup | Download | Breakpoints

Connection: Texas Instrument USB-IF; Automatic; Parallel port 1

Debug protocol: ☐ Override default; Spy-Bi-Wire; 4-Wire JTAG

Target VCC: Target VCC (in Volt): 3.0; Settling time (in ms): 0

☐ Attach to running target

☐ Disable memory cache

OK Cancel

Click OK.

The LED we'll be using is already connected to the microcontroller pin by the circuit board; you don't have to add any parts. It's connected as shown in the schematic in Figure 3. All we need to do is write the program to turn P1.0 on and off.

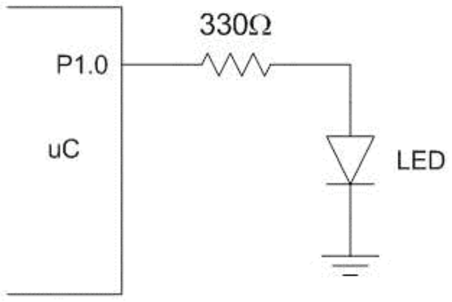


Figure 3 - Circuit diagram for connecting an LED to a microcontroller output port. The resistor is needed since LEDs behave like diodes with very large current flowing for voltages above a threshold (around 1 V). The microcontroller output changes between 0 V and the supply voltage (around 3 V) for logic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

In order to write the program, you'll need to know a bit about how ports like P1 work. Each port is a group of 8 pins; these pins' values can be set (if they are outputs) or read (if they are inputs) by the microcontroller. Each pin's value and input/output status can be individually set by manipulating 'registers' in the microcontroller; each register is a group of 8 bits within the microcontroller that may or may not directly affect pins.

For now, you need to know about the following registers. Ports other than P1 have similar registers that start with P2, etc:

`P1SEL`	Set function of each pin in P1. 0=basic digital I/O 1=special function
`P1DIR`	Set data flow direction of each pin in P1. 0=pin is input 1=pin is output
`P1OUT`	Set voltage on each output pin in P1. 0=pin low (GND) 1=pin high (*Vcc*)

Finally, you need to know how to change the values of the bits in a register. For convenience, we use the predefined values `BIT0` through `BIT7`. `BIT0` has bit 0 high and bits 1-7 low (i.e. 00000001), and so on. If we wanted to set pin 0 of port 1 (P1.0) high, we could simply write

```
P1OUT = BIT0;
```

but if any other pins in P1 had been high before, they would now be low. If we want to leave the other bits alone, we need to write

```
P1OUT = P1OUT | BIT0;
```

The vertical line (called a 'pipe') is a bitwise OR function - that is, it works on one bit at a time. It determines bit 0 of its output by ORing together bit 0 or P1OUT and bit 0 of BIT0, and so on. This will set bit 0 of P1OUT high, but leave the rest of P1OUT unchanged. We can shorten this line of code to

```
P1OUT |= BIT0;
```

This does the same thing, but uses a shorthand notation to save on typing. You can do the same thing with operators like `+`, `–`, `*`, and a few others. For example, to set P1.0 low, we can use either of the following lines:

```
P1OUT = P1OUT & ~BIT0;
P1OUT &= ~BIT0;
```

The ampersand (`&`) is a bitwise AND; it works on one bit at a time like the bitwise OR does, separately ANDing the eight bit pairs. The tilde (~) is a bitwise NOT; all the bits in its input are inverted, so BIT0 (00000001) becomes ~BIT0 (11111110). Can you figure out how to set multiple bits high with one line of code? How about setting multiple bits low? (Careful!)

Now, let's actually write the program to turns P1.0 on and off.

1. First we need to configure P1.0 as a digital output using the following instructions:

   ```
   P1OUT &= ~BIT0;  // P1.0 low (LED off)
   P1SEL &= ~BIT0;  // configure P1.0 for digital I/O
   P1DIR |= BIT0;  // configure P1.0 as output
   ```

   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 surprises. The next two statements configure P1.0 as digital I/O with direction set to output. (As you would expect, the same statements with BIT1 substituted for BIT0 would 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).

2. The following statements set the output low or high, or toggle its state:

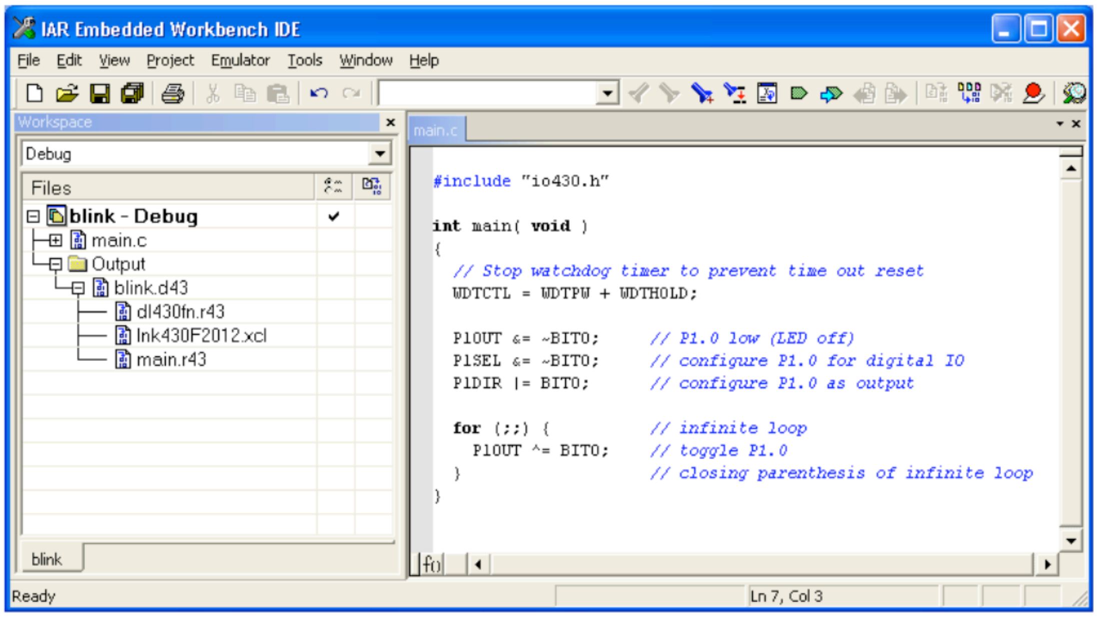
   ```
   P1OUT &= ~BIT0;  // P1.0 low
   P1OUT |= BIT0;  // P1.0 high
   P1OUT ^= BIT0;  // P1.0 toggle
   ```

   Text after // is treated as a comment and is 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.

3. 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
   ```

4. The complete program looks as follows:



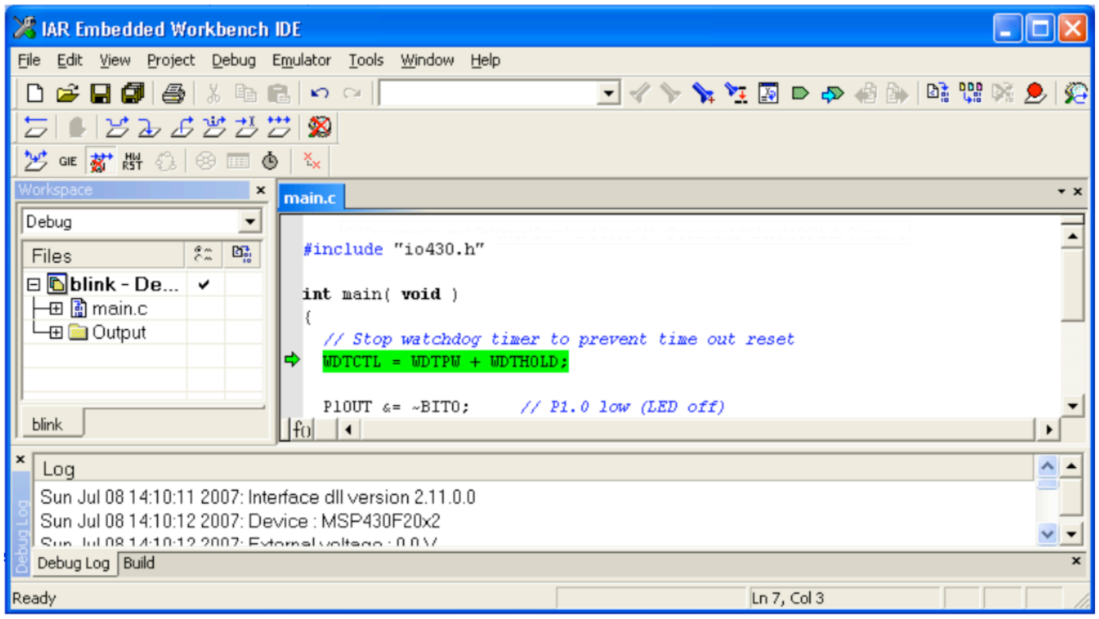
```
#include "io430.h"

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

  P1OUT &= ~BIT0;      // P1.0 low (LED off)
  P1SEL &= ~BIT0;      // configure P1.0 for digital IO
  P1DIR |= BIT0;       // configure P1.0 as output

  for (;;) {           // infinite loop
    P1OUT ^= BIT0;     // toggle P1.0
  }                    // closing parenthesis of infinite loop
}
```

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



```
#include "io430.h"

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

  P1OUT &= ~BIT0;      // P1.0 low (LED off)
```

```
Sun Jul 08 14:10:11 2007: Interface dll version 2.11.0.0
Sun Jul 08 14:10:12 2007: Device : MSP430F20x2
```

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

6. If everything went right the LED turns on but doesn't 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.

7. The simplest fix is to give the microcontroller a bit of extra work to slow it down. Microcontrollers cannot get bored 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 statement:

```
delay(30000);
```

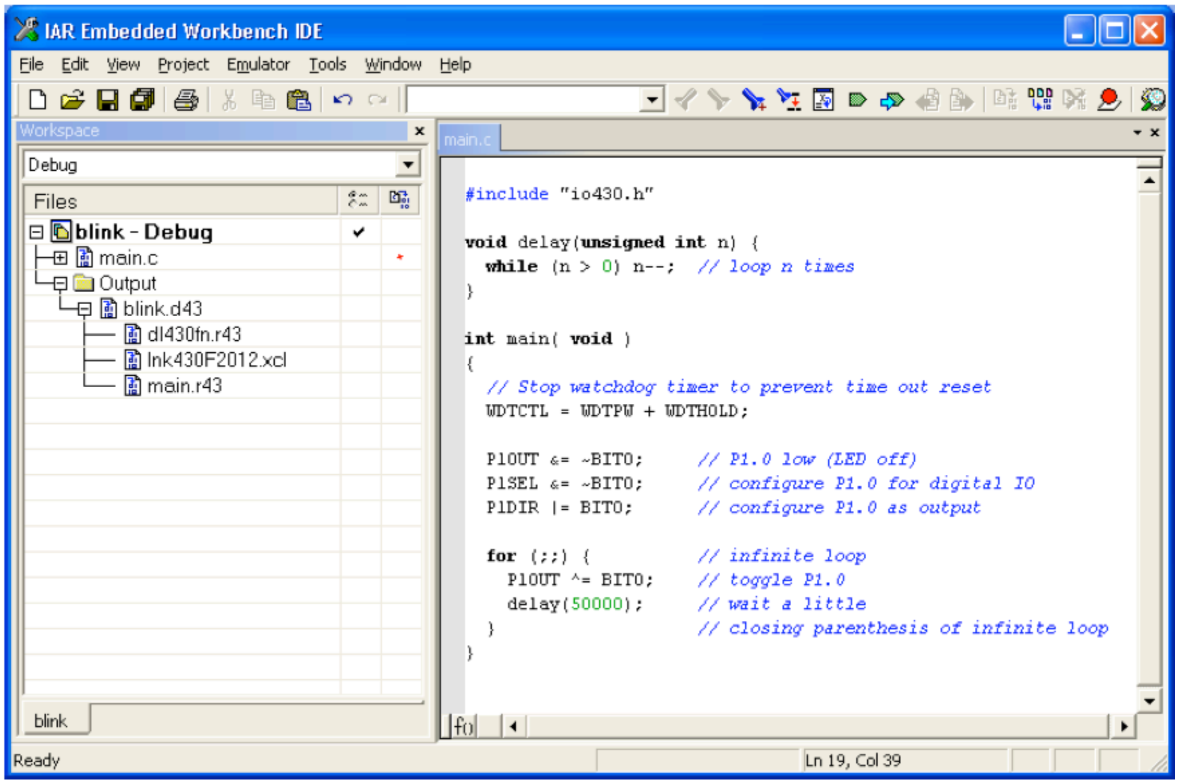
causes the microprocessor to spin 30,000 times in a loop, decrementing the variable n each time around.

A few words on subroutines:

1. They must be declared outside of all other routines, including the `main()` routine.
2. They must be declared earlier in the program than they are used.

For example, if we wanted to write a subroutine that called `delay()`, we would have to put our subroutine's code after the code for `delay()` but before it is used in `main()` or any other subroutines.

8. Let's try this in our program:



```
#include "io430.h"

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

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

  P1OUT &= ~BIT0;      // P1.0 low (LED off)
  P1SEL &= ~BIT0;      // configure P1.0 for digital IO
  P1DIR |= BIT0;       // configure P1.0 as output

  for (;;) {           // infinite loop
    P1OUT ^= BIT0;     // toggle P1.0
    delay(50000);      // wait a little
  }                    // closing parenthesis of infinite loop
}
```

9. 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, or write a subroutine that calls delay repeatedly. Can you get the LED to blink at a rate of 1Hz?

If you have 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*? 1 pt. 0

Show your light to the GSI. 1 pt. 0

Analog Output

So far we have used the microcontroller to turn an LED on or off. Frequently an “analog” output is required, i.e. a voltage that can assume any value between the supplied voltage *Vcc* and ground. Some microcontrollers 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 carefully selecting the times the output is high and low. For example, if the output is high (*Vcc*=3 *V*) during 3 clock cycles and then low (0 *V*) during 2 cycles, the average value of the output is (3/5)*Vcc* =1.8 *V*. Likewise, if the output stays high for 21 cycles and low for 77 cycles, the average output voltage is: 1 pt. 1

Write a subroutine function `dac(n)` that sets port P10.0 high during n cycles and low during 256-n cycles. (We cannot use port P1.0 as we have been because it is permanently connected to the LED, so convert all mentions of P1 into P10, e.g. `P1OUT` to `P10OUT`.) 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. 1

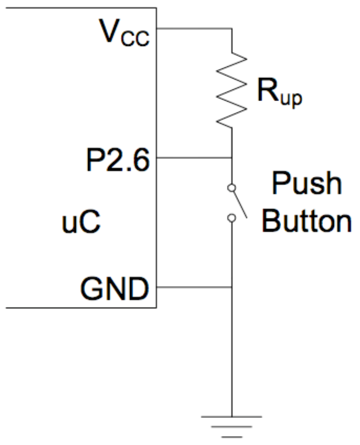
Test your “DAC” in the laboratory. First verify with the oscilloscope that for n=0 the output remains low (why is there a “glitch” - an undesired pulse - and how could you eliminate it?), and for n=256 the output remains high (except for a brief glitch). Then connect a first order RC filter with R = 10 *k*Ω and C = 4.7 *uF* to the microcontroller output and record the voltage v_0 across the capacitor on the oscilloscope. Adjust n such that v_0 = 1.3 *V* and ask the GSI to verify your result.

n = 1 pt. 1 v_0 = 1 pt. 1

One drawback of this DAC is that it keeps the microcontroller busy and unavailable to do other tasks. A better solution uses the microcontroller’s timer combined with a feature called interrupt to implement the delay. With this technique, the `dac` function can run at the same time as other functions. We will not explore this feature here; please check the manual and sample programs (see vendor website) if you want more information.

Digital Input

Now we will focus on the second half of I/O: inputs. In this section, we'll be using the leftmost button (S1, connected to P2.6) for the digital input to control the state of the LED.



the

Figure 4 - Circuit diagram for connecting a pushbutton to a microcontroller input pin. In the microcontroller we are using, R_{up} is built in already.

Figure 4 shows the circuit for connecting a button to the microcontroller. Normally P2.6 is pulled to Vcc (high) by the pull-up resistor R_{up}; pressing the button pulls P2.6 low. This entire circuit is mostly set up on the board already; the left pushbutton is connected between P2.6 and ground. R_{up}, on the other hand, is built into the microcontroller, and we must activate it before we can use the button.

Here is the code for enabling P2.6 as an input with the pull-up resistor enabled:

```
P2OUT = BIT6;
P2DIR = ~BIT6;
P2REN = BIT6;
```

The following statement stalls the program and waits for P2.6 to go low:

```
while (P2IN & BIT6); // wait for P2.6 to go low
```

We know about some new registers now:

P2REN	Enable or disable R_{up} on each pin in P2. 0=R_{up} disabled 1=R_{up} enabled
P2IN	Read voltage on each pin in P2. 0=pin low (GND) 1=pin high (Vcc)

We can also use the `while` instruction to repeat instructions, as done in `delay()`, rather than simply wait:

```
while (P2IN & BIT6) {
  ...do the things in the brackets repeatedly while P2.6 is high...
}
```

Combine this information with what you learned from the LED blinking program to write a program that turns the LED at P1.0 on when the button is pressed and turns it off when the button is released. Put your code in the box below:

2 pt.
1

Ask the GSI to verify the button's functionality.

1 pt.
1

Assuming P1.5 has been configured as an output, what is the code to set it to GND? 1 pt. 2

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 display which of several ranges a voltage falls within. ADCs compare an analog input v_i, e.g. 1.387 V, 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:

$$N = round\left(4095 \times \frac{v_i}{V_{ref}}\right)$$

Negative input voltages and inputs exceeding the reference voltage are clipped to zero and 4095, respectively.

The program skeleton below shows the code for using the ADC and configures P6.7 as its input. Conversion results are stored in the variable ADC12MEM0. (MEM-zero, not MEMO.) Be aware that the reference voltage V_{ref} is 2.5 V, not 3 V.

```
#include "msp430x54x.h"
int main( void ) {
   // Stop watchdog timer to prevent time out reset
   WDTCTL = WDTPW + WDTHOLD;
   ADC12CTL0 = ADC12REF2_5V  // reference is VCC (2.5 V)
               + ADC12REFON // turn on reference generator
               + ADC12SHT02 // sample rate
               + ADC12ON;   // enable ADC
   ADC12CTL1 = ADC12SHP;  // Use sampling timer
   ADC12MCTL0 = ADC12EOS  // End of sequence (since we are using 1 ADC)
               + ADC12SREF_1  // Define limit between Vref and AVss
               + ADC12INCH_7; // Read from ADC Channel A7

   P6SEL |= BIT7;   // Enable A/D channel A7
   P1OUT &= ~(BIT1 + BIT0);  // P1.0 & P1.1 low (LEDs off)
   P1SEL &= ~(BIT1 + BIT0); // configure P1.0 & P1.1 for digital IO
   P1DIR |= 0x03;   // P1.0 output
   for (int i=0; i<0x30; i++); // Delay for reference start-up

   for (;;) { // infinite loop
      ADC12CTL0 |= ADC12ENC + ADC12SC; // enable and start conversion
      while (ADC12CTL1 & ADC12BUSY); // wait for conversion to complete
      if (ADC12MEM0 > ...) P1OUT = ...; // display result on LEDs ...
   }
}
```

Notice that a new instruction has been added: `if`. It causes a section of code to execute (or not) based on a condition. An `else` section can be added if necessary. For example, the following statement turns on P1.0 if and only if n is even.

```
if (n & BIT0) { // check the ones bit of n
  P1OUT &= ~BIT0; // executed only if the ones bit is high (n is odd)
}
else { // if n is not odd, it must be even
  P1OUT |= BIT0; // executed only if the ones bit is low (n is even)
}
```

Any non-zero argument to an `if` statement will satisfy it, so be careful when checking for a bit with a low value.

Complete the program such that the on-board LEDs represent graphically the input voltage of the ADC for $V_{ref} = 2.5\ V$. Write your program such that no LEDs are on for $v_i < 500\ mV$, only LED #0 is on for $500\ mV < v_i < 1000\ mV$, only LED #1 is on for $1000\ mV < v_i < 2000\ mV$, and LEDs #0 and #1 are on for $v_i > 2000\ mV$. Write the infinite loop portion of the program, which will include the section you designed (i.e. don't copy out all the setup code given above) in the space below:

2 pt.
2

Connect a potentiometer between VCC and GND and connect the middle tap to P6.7. Verify proper circuit operation using a voltmeter and show the result to the GSI.

1 pt.
2

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

A final question: with V_{ref} set to 3 V, what value v_i corresponds to a reading N = 199? (There is a range of voltages that could produce this N; specify the middle of the range.)

1 pt.
3