Assignment 1
Blinking Lights
Due Friday, 29 January 2016

1 Summary

The first assignment involves controlling the LEDs available on the LaunchPad using the buttons on the LaunchPad. You will actually develop several versions of the program, ranging from a simple ON/OFF based on polling the switches to a timer-based, interrupt-driven program controlling the color of the RGB LED where the color is chosen via the pushbuttons.

2 I/O Ports on the LaunchPad

The LaunchPad has 10 user-accessible ports. Each port has 8 distinct input/output lines. The LaunchPad documentation refers to these using the notation $\mathbf{p}.\ell$, with the number **p** designating the port and ℓ indicating the actual line. Many of these ports can be designated as discrete I/O, meaning they read or write a digital signal. Nominally, a logical 1 corresponds to a voltage on the line of +3.3V, and a logical 0 is ground.

Each port has a set of memory-mapped registers which control its function and setting and reading the state of the lines associated with that port. These ports are addressed as memory locations; the included header file contains macros to define these values so we can refer to them simply by the macros.

These ports serve a multitude of purposes- discrete I/O (used for digital switch input and switching external components off and on), serial communications (including UART mode, I^2C, and SPI), analog-to-digital input, and PWM output. Most of the lines can be devoted to multiple purposes (e.g. digital input, digital output, or serial communication), so we need to set appropriate values in the control registers to configure them.

2.1 Red LED

The red LED on the LaunchPad is connected to line 0 of port 1. We can turn the LED on by writing a value of 1 on this line. First, we must make sure the port is properly configured for this purpose. There are three registers which need to be properly set in order for this to happen.

The **direction** register is addressed via the macro **P1DIR**[1]. Each line on the port is referenced by a single bit in this register. To set the direction to output, we need to write the value 1 to the corresponding bit. Since we don't want to alter any of the other lines, we

[1]This macro is defined in line 765 of the header file **msp432p401r.h**; the definition uses the macro **HWREG8**, defined in line 189 of the same header. Basically, it tells the compiler that it should write to a specific address, which is the location of a volatile unsigned 8-bit integer. The keyword **volatile** simply means that it can change without being changed in code, which limits the types of optimization the compiler can perform.

use the line

```
P1DIR|=0x01;
```

which (effectively, if not actually) reads the register, takes the logical or with the hexadecimal value 01, and writing the result back to the register. This ensures that bit 0 of the register is set to 1. There are macros defined for the bits themselves; using the corresponding macro would give the line

```
P1DIR|=BIT0;
```

To select the function of the line, there are two function selection registers. In this case, they are **P1SEL0** and **P1SEL1**. We want the port to be configured as a standard digital I/O port. Referring to the documentation (Table 6-32 on page 101 of the MSP432P401X Data Sheet (slas826a.pdf)), we see that for a digital I/O port, both should have the corresponding bit to 0. While this is the default value, and so we really need to nothing, it is better to force the configuration. There is one issue with doing this, however [2]. When both values need to be set, since the two are not contiguous, they need to be set consecutively. This may result in an intermediate configuration which is undesirable. To alleviate this, there is a complement register, **P1SELC**, which changes both bits simultaneously. We thus read the two selection registers, and determine if one or both need to be changed. Then we write back the desired values, for example by doing:

```
unsigned char reg0 = P1SEL0;
unsigned char reg1 = P1SEL1;
if(P1SEL0 & BIT0)
{
  if(P1SEL1 & BIT0)
    P1SELC|=BIT0;
  else
    P1SEL0&=~BIT0;
}
else if (P1SEL1 & BIT0)
  P1SEL1&=~BIT0;
```

By writing the value 1 to bit 0 of the **P1OUT** register:

```
P1OUT|=BIT0;
```

we turn the red LED on. We turn it off by writing the value 0:

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

We can toggle the state of the LED (sequentially turn it on and off) by XORing the current value with 1:

```
P1OUT^=~BIT0;
```

[2] Cf. Sec. 10.2.6 on p. 483 of the MSP432P4XX User's Guide (slau356a.pdf)

2.2 A real program

We are now almost ready to finish our first program, however simple. We need to remember one thing about embedded systems: there is no operating system running, so if the program ends, the processor halts. Our program must, therefore, contain an infinite loop to prevent the program from ending. An empty for loop is a good way to do this, so our code now looks like[3]:

```
#include <msp.h>

void main(void)
{
  WDTCTL = WDTPW | WDTHOLD;                    // Stop watchdog timer
/* Initialize the I/O port */
  P1DIR|=BIT0;
  if(P1SEL0 & BIT0)
  {
    if(P1SEL1 & BIT0)
      P1SELC|=BIT0;
    else
      P1SEL0&=~BIT0;
  }
  else if (P1SEL1 & BIT0)
    P1SEL1&=~BIT0;
  P1OUT|=BIT0;    //turn the LED on
/* Now we enter the loop */
  for(;;)  //idiomatic infinite loop; while(1) is also usable here
  {
    P1OUT^=BIT0;
  }
}
```

The first line stops the watchdog timer. The watchdog timer is useful for low-power applications, as it allows the CPU to sleep most of the time and only wake when needed. However, at our level, it complicates life, so we just shut it off for now. Running this program likely results in a dimly-lit LED. The processor is running markedly faster than your eyes, so you cannot distinguish the on-and-off transitions. So you will see the LED lit at about half its normal intensity. We need to slow the loop frequency down. We do this by adding a null loop[4]:

```
int k=0;
for(k=0;k<20000;++k);
```

[3]Those of you with much C experience will realize that the signature of the **main** function is atypical. Hosted implementations should, according to the standard, be defines as returning **int**. Embedded systems, according to the C standard, have implementation-defined signatures for this function.

[4]The TI compiler seems unhappy about declaring loop variables in the for statement.

The problem with the above code is compiler optimization. The compiler can recognize that nothing happens, and exclude the loop altogether. Under normal conditions, this is useful and why we like optimizing compilers. When we are trying to use the loop to force a delay, however, it is disastrous. We can defeat this by lying to the compiler and telling it that the value of the loop variable can be changed by things not within the code, preventing optimization. We do this by adding the keyword **volatile**:

```
volatile int k=0;
for(k=0;k<20000;++k);
```

Our first program is thus:

```
#include <msp.h>

void main(void)
{
  WDTCTL = WDTPW | WDTHOLD;                 // Stop watchdog timer
/* Initialize the I/O port */
  P1DIR|=BIT0;
  if(P1SEL0 & BIT0)
  {
    if(P1SEL1 & BIT0)
      P1SELC|=BIT0;
    else
      P1SEL0&=~BIT0;
  }
  else if (P1SEL1 & BIT0)
    P1SEL1&=~BIT0;
  P1OUT|=BIT0;    //turn the LED on
/* Now we enter the loop */
  for(;;)  //idiomatic infinite loop; while(1) is also usable here
  {
    volatile int k=0;
    for(k=0;k<20000;++k);
    P1OUT^=BIT0;
  }
}
```

3 Using a pushbutton

We can get information from the LaunchPad by use of the red LED. Now we want to change the light based on external input- the pussing of a button. There are two usable pushbuttons on the LaunchPad (the third is a Reset button, which reinitializes the processor). These are both on port 1, lines 0 and 4. We can read the state of the device by reading the register **P1IN**.

Naıvely, then, we try to use this in our loop. We add configuration of P1.1 in the same way that we configured P1.0 for the LED. We need to make sure that bit 1 of the **P1DIR** register is set to 0, to indicate that P1.1 is an input. Also, the pin is tied to ground when the button is pressed, so bit 1 of the **P1IN** register will be 0 when the button is pressed.

While it doesn't apply here, reading some registers (particularly when we are dealing with interrupts) causes the state of the register to change. It is also easier to debug if we store the state of the input at the time we read it, so we use a variable to store the value of the **P1IN** register. The resulting code is:

```
#include <msp.h>

void main(void)
{
  WDTCTL = WDTPW | WDTHOLD;                 // Stop watchdog timer
/* Initialize the I/O port */
  P1DIR|=BIT0;
  P1DIR&=~BIT1;
  if(P1SEL0 & BIT0)
  {
    if(P1SEL1 & BIT0)
      P1SELC|=BIT0;
    else
      P1SEL0&=~BIT0;
  }
  else if (P1SEL1 & BIT0)
    P1SEL1&=~BIT0;
  if(P1SEL0 & BIT1)
  {
    if(P1SEL1 & BIT1)
      P1SELC|=BIT1;
    else
      P1SEL0&=~BIT1;
  }
  else if (P1SEL1 & BIT1)
    P1SEL1&=~BIT1;
  P1OUT|=BIT0;    //turn the LED on
/* Now we enter the loop */
  for(;;)  //idiomatic infinite loop; while(1) is also usable here
  {
    unsigned char portIn=P1IN;
    if(!(portIn & BIT1))
      P1OUT^=BIT0;
    volatile int k=0;
    for(k=0;k<20000;++k);
  }
```

}

Running this program gives an unexpected result. The LED lights at first, as expected. However, pressing the button results in the LED flashing for a few seconds and then staying lit- not switching off and on with successive presses, as we would expect.

The reason is electronic. When we release the pushbutton, it does not immediately reach +3.3V because it is not connected to anything. We need to use a pull-up arrangement to accomplish this. The MSP432 has internal pull-up resistors for this purpose; we enable or disable them via the **PxREN** registers. These resistors can be used in either pull-up (to +3.3V) or pull-down (to ground) configurations. In this case, the pushbutton goes to ground, so we need to pull it up to +3.3V. The choice of pull-up or pull-down is controlled by writing a value to the output register. Writing a 1 to the appropriate line selects pull-up mode, while writing a 0 selects pull-down. So we need to include the lines

```
P1REN|=BIT1;
P1OUT|=BIT1;
```

to our initialization code to force the pin high when the pushbutton is released.

Our program thus ends up as:

```
#include <msp.h>

void main(void)
{
  WDTCTL = WDTPW | WDTHOLD;              // Stop watchdog timer
/* Initialize the I/O port */
  P1DIR|=BIT0;
  P1DIR&=~BIT1;
  P1REN|=BIT1;
  P1OUT|=BIT1;
  if(P1SEL0 & BIT0)
  {
    if(P1SEL1 & BIT0)
      P1SELC|=BIT0;
    else
      P1SEL0&=~BIT0;
  }
  else if (P1SEL1 & BIT0)
    P1SEL1&=~BIT0;
  if(P1SEL0 & BIT1)
  {
    if(P1SEL1 & BIT1)
      P1SELC|=BIT1;
    else
      P1SEL0&=~BIT1;
  }
  else if (P1SEL1 & BIT1)
```

```
    P1SEL1&=~BIT1;
  P1OUT|=BIT0;    //turn the LED on
/* Now we enter the loop */
  for(;;)  //idiomatic infinite loop; while(1) is also usable here
  {
    unsigned char portIn=P1IN;
    if(!(portIn & BIT1))
      P1OUT^=BIT0;
    volatile int k=0;
    for(k=0;k<20000;++k);
  }
}
```

We need to keep the busy loop at the end or the program will loop through many times when the button is pressed, putting the LED into an essentially unpredictable state.

Of course, having all of that initialization code in **main** is ugly. Move it to separate functions, called from **main**. I prefer to initialize each thing separately, so my code becomes:

```
#include "msp.h"

void InitializeLED(void)
{
  P1DIR|=BIT0;
  if(P1SEL0 & BIT0)
  {
    if(P1SEL1 & BIT0)
      P1SELC|=BIT0;
    else
      P1SEL1&=~BIT0;
  }
  P1OUT|=BIT0;    //turn the LED on
}

void InitializePushButton(void)
{
  P1DIR&=~BIT1;
  P1REN|=BIT1;
  P1OUT|=BIT1;
  if(P1SEL0 & BIT1)
  {
    if(P1SEL1 & BIT1)
      P1SELC|=BIT1;
    else
      P1SEL0&=~BIT1;
  }
  else if (P1SEL1 & BIT1)
```

```
    P1SEL1&=~BIT1;
  else if (P1SEL1 & BIT1)
    P1SEL1&=~BIT1;
}

void main(void)
{
  WDTCTL = WDTPW | WDTHOLD;                 // Stop watchdog timer
/* Initialize the I/O port */
  InitializeLED();
  InitializePushButton();
/* Now we enter the loop */
  for(;;)  //idiomatic infinite loop; while(1) is also usable here
  {
    unsigned char portIn=P1IN;
    if(!(portIn & BIT1))
      P1OUT^=BIT0;
    volatile int k=0;
    for(k=0;k<20000;++k);
  }
}
```

In principle, we could move the repeated code for setting the function select registers into a separate function as well. That is left as an exercise for the reader.

4 Assignment 1

The RGB LED has three lines associated with it: the red LED is 2.0, the green LED is 2.1, and the blue LED is 2.2. Write a program that uses the pushbutton on 1.0 to select between constant cycle and manual cycle, and cycles through the 8 simple colors of this LED (Off, Red, Green, Yellow (Red+Green), Blue, Violet (Red+Blue), Teal (Blue+Green), and White (Red+Green+Blue))[5] When manually cycling, use the pushbutton on 1.4 to cycle through the colors.

[5]This is trivially mapped to a number from 0 to 7.