ECED3204 – Lab #3

STUDENT NAME(s): _____ .

STUDENT NUMBER(s): B00 _____ .

Pre-Lab Information

It is recommended that you read this entire lab ahead of time. Doing so will save you considerable time during the lab, as you will be required to write some simple C code during this lab!

Overall Objective

This lab has several main objectives:

- Learn about debouncing button presses
- Learn about multiplexing for writing multiple LEDs
- Learn about multiplexing for reading multiple buttons

Part #1: Debouncing Button Presses

Objective

- Familiarize yourself with button pressing

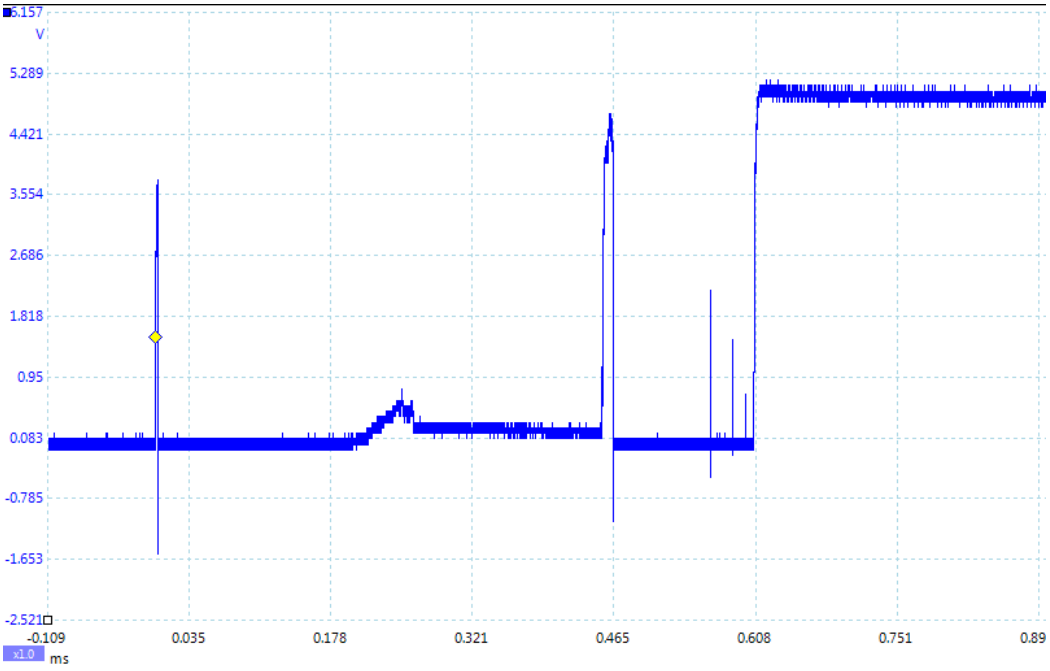
Required Materials

- Microprocessor Module with Programmer
- Breadboard
- USB Cable
- Power Supply
- Computer with Atmel Studio 6.2 and Programmer Utility installed
- Push Button

Background

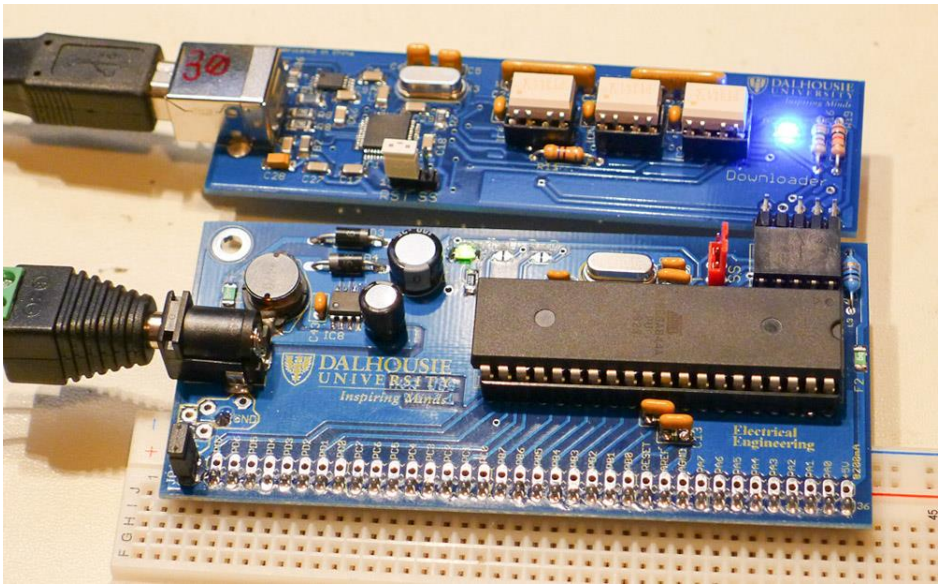
Pressing a button should be a simple task – but in reality it might not be! This lab will introduce some of the possible problems that happen when you enter the world of 'real hardware'. **This lab assumes you have already completed lab #1, as this lab will assume you already know how to build and download a file to the microprocessor module.**

In particular, the following shows a scope capture when pushing a button. The button is being solidly pressed, yet the scope trace has lots of small 'spikes'. Let's see what these spikes do to a digital circuit, which assumes a button goes cleanly from the '0' to the '5V' signal level.

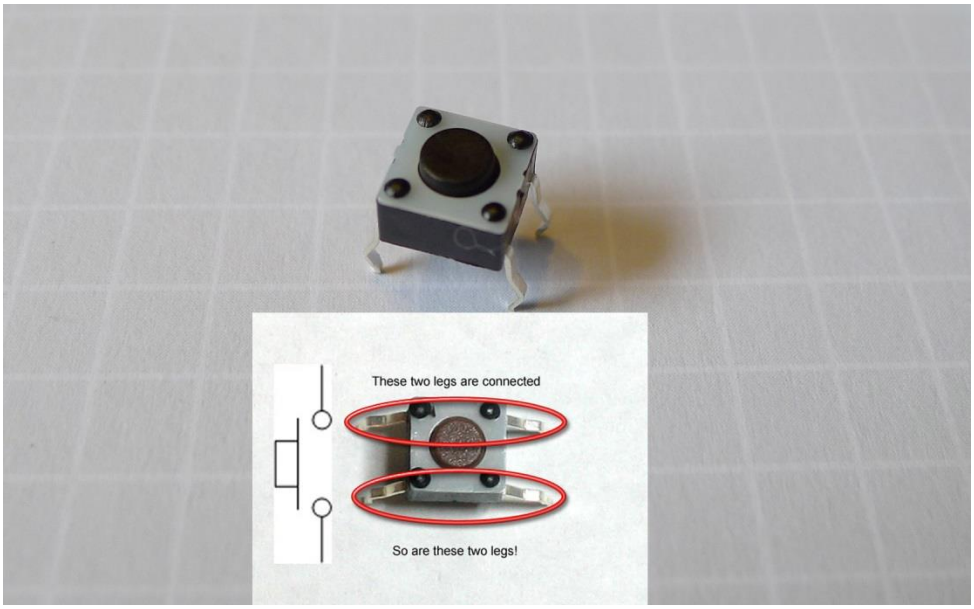


Procedure

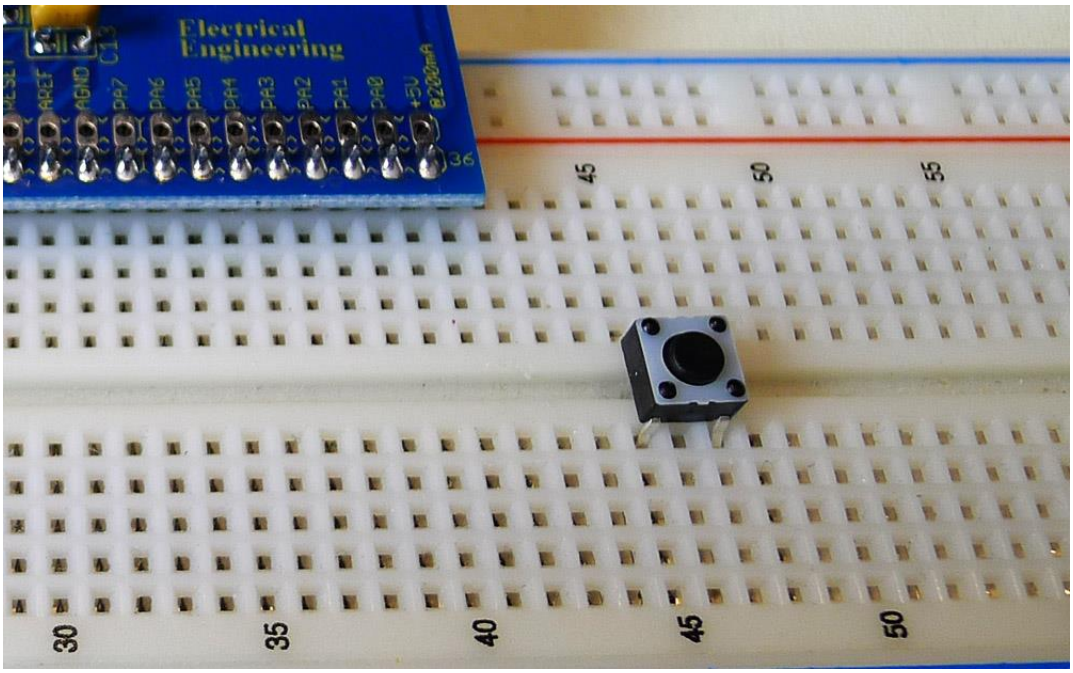
1. Plug the microprocessor board into the breadboard and apply power, see Lab #1 for details.



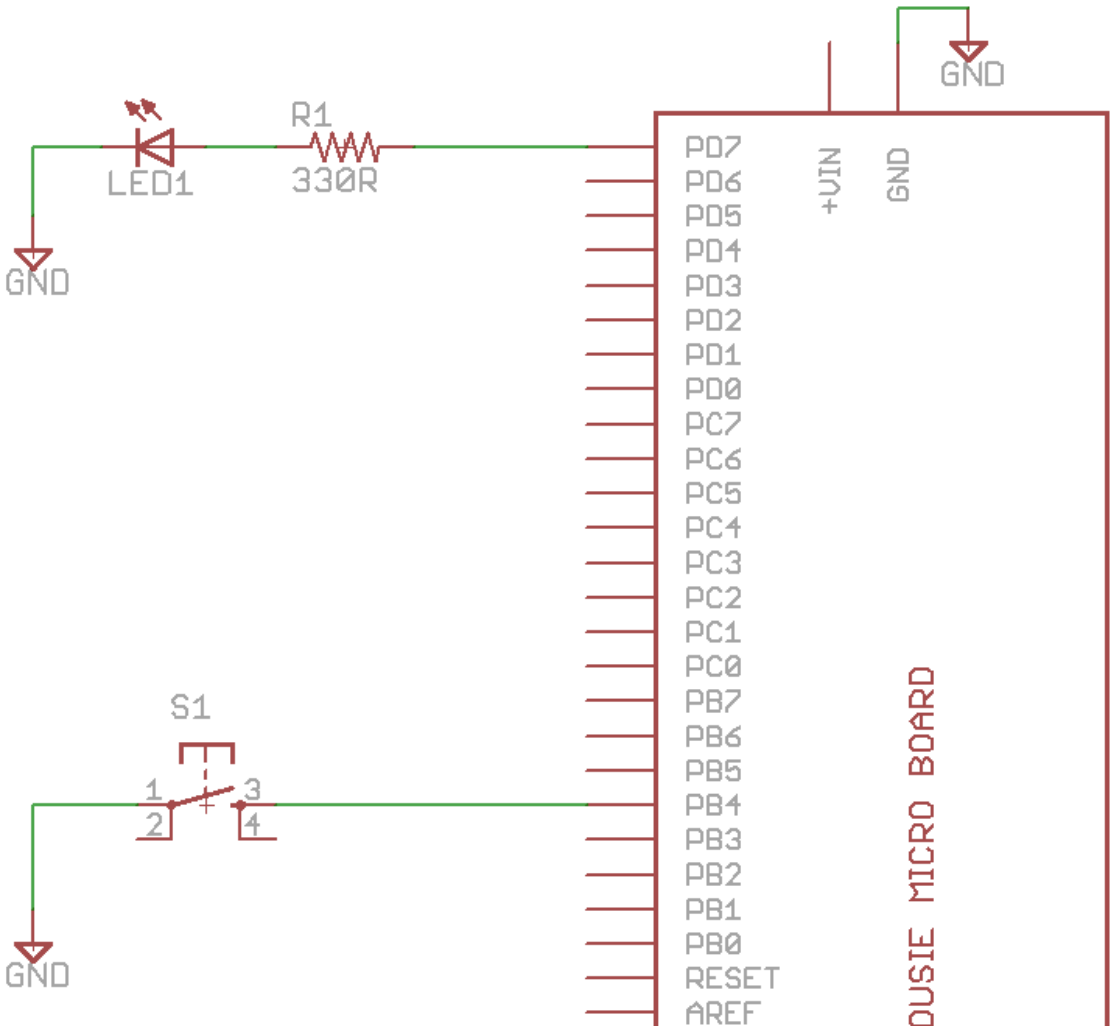
2. Select the ‘tactile push button’, note the pinout is as follows:



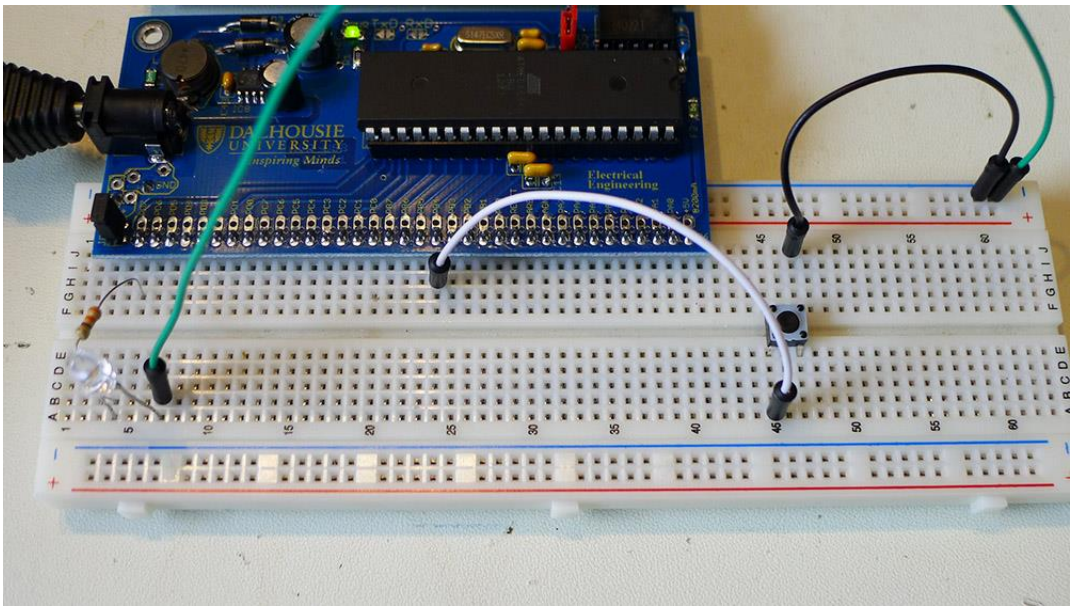
Insert the push-button into a section of the breadboard which **does not** connect to the microprocessor board:



Wire up the following schematic



Which might look as follows:



3. Start a new C/C++ project (see Lab #1 for details), and write a simple program which reads the push-button, and changes the state of the LED based on your pushing the button. This would have the following steps:
    a. Set PORTD.7 as an output
    b. Turns the pull-up on PORTD.4
    c. Reads the state of PINB.4, and sets the state of the PORTD.7 pin appropriately

    You can use the following as a base, but you'll have to fill in the sections marked with **????**:

```
#include <avr/io.h>

int main(void)
{
   //PORTD.7 is Output
   //PORTD.4 is Input
   PORTD = ?????;

   //Turn pull-up on PORTD.4
   PORTB = ?????;

   while(1){
          //Check state of push-button
          if(?????){
                 //Turn on PORTD.7
                 PORTD = ????;
          } else {
                 //Turn off PORTD.7
                 PORTD &= ~(1<<7);
          }
   }
}
```

    As a hint, consider the following useful pieces of information:

- `(1<<6)` would select the 6th bit
- `PIND & (1<<6)` would check the state of the 6th bit of PIND
- Pull-ups are enabled by setting the bit in the PORTD register when the pin is an input

Check the LED turns on or off when you hold the button down – note it's not important the LED 'matches' the button state. As written the LED will be ON when you are not pushing the button, and OFF when you press the button.

4. Next, we will be implementing a function which *toggles* the LED. This will allow us to see what happens when we don't use debouncing and read a switch directly. To do so, we implement the following C code:

```
#include <avr/io.h>

int main(void)
{
   DDRD = 1<<7;
   PORTB = 1<<4;

   unsigned char ledstate = 1;
   unsigned char buttonstate = 0;
   unsigned char lastbuttonstate = 0;

   while(1)
   {
          buttonstate = PINB & (1<<4);

          if ((buttonstate != lastbuttonstate) && (buttonstate == 0)){
                 ledstate ^= 1;
          }

          lastbuttonstate = buttonstate;

          if(ledstate) {
                 PORTD = 1<<7;
          } else {
                 PORTD = 0;
          }
   }
}
```

5. After downloading the above code, press the button. On every button press you should see the LED toggle.

   Note that if you press and hold the button the LED should toggle only once the button is first pressed. Try pressing and holding the button and then releasing – if things are working the LED should not toggle on the release.

   Try this a number of times and you might eventually catch a 'glitch', when the LED either doesn't toggle when you press the button, OR the LED toggles on both the press *and* release of the

button! You'll be asked in the question section why this might be, so give it some thought. You can also use an oscilloscope to look at the input signal on PORTB.4.

6. We will now fix this example to include 'debouncing'. Looking at the previous code, we could see the following chunk of code which is where we read the button state:

```
buttonstate = PINB & (1<<4);
```

We could replace this will a piece of code which checks the button state twice, with a delay in-between each read. If the state is the same, we can assume there is no 'bounce':

```
tempbuttonstate = PINB & (1<<4);
_delay_ms(20);

if ((PINB & (1<<4)) == tempbuttonstate) {
    buttonstate = tempbuttonstate;
}
```

This will require you to also define a new variable:

```
unsigned char tempbuttonstate;
```

And include the delay.h header file:

```
#include <util/delay.h>
```

You will have to add those two above lines to the appropriate place, in addition to changing the reading of the button.

7. Program the modified code into your board. What happens when you press the button now – are you still able to observe the 'invalid' states, where the LED appears to not change or change on the wrong edge?

Part #2: Multiplexing LEDs

Objective

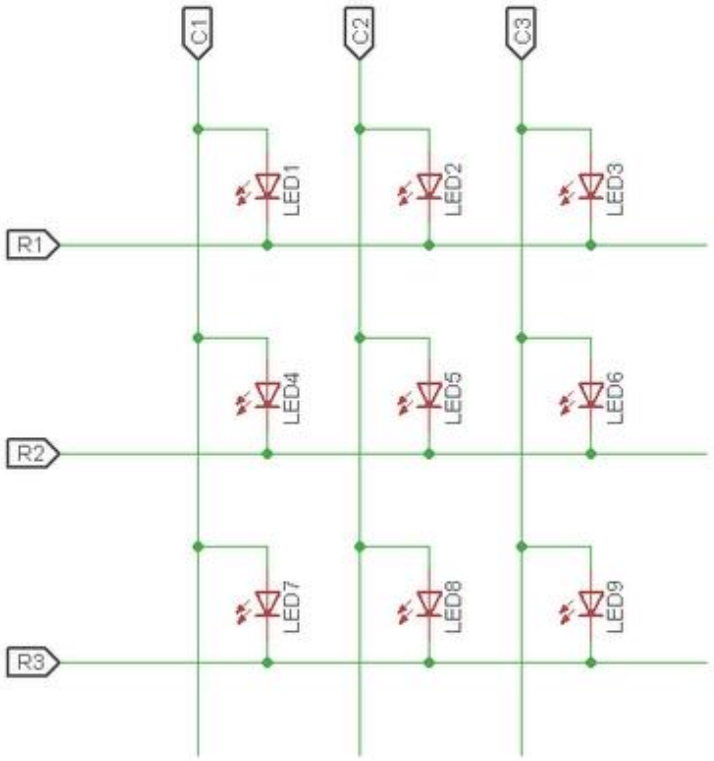
- Familiarize yourself with the idea of multiplexing LED outputs

Required Materials

- Lab setup from Part 1 (remove push-button)
- 4x LEDs
- 2x 330 ohm resistors

Background

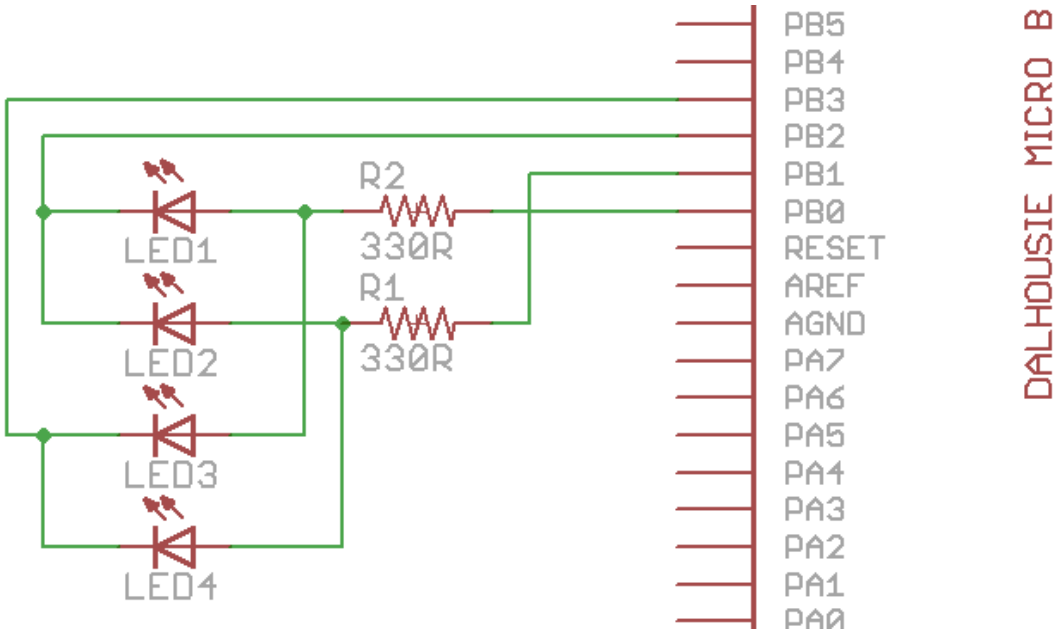
Multiplexing allows us to write or read to a larger number of components then you might otherwise be able to. This example will use a 'multiplexing' capability for only four LEDs. But consider the following, which shows a 3x3 array of LEDs, driven with only 6 I/O pins (instead of 9 required to drive each LED separately):



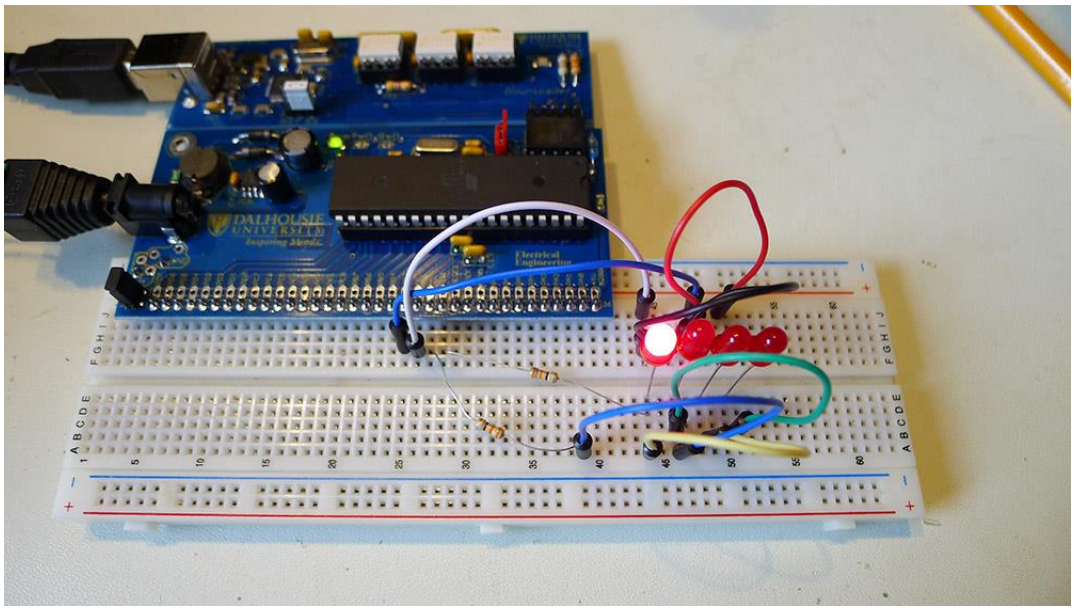
We will instead just be driving four LEDs. We don't actually realize any pin savings with just four LEDs, but you can expand this on your own time (what fun!).

Procedure

1. Plug the microprocessor board into the breadboard and apply power, see Lab #1 for details.
2. Wire up the following circuit on the breadboard:



Which might look something like this:



3. At this point, we need to determine how to turn on each LED. Using the schematic diagram, fill out the following table, which has a few entries given to you. The idea is to illuminate LED 0 for example, we need to set PORTB.0 high, and PORTB.2 low. That will drive current through LED0.

LED Number	PORTB Pin High	PORTB Pin Low
0	0	2
1		
2	0	

3		3

4. Code the following, where the appropriate PORTx pin has been set high or low based on the above table, and will cycle through each LED:

```
#include <avr/io.h>
#include <util/delay.h>

void set_led(unsigned char lednum)
{
   switch(lednum){
          case 0:
          DDRB = 1<<0 | 1<<2;
          PORTB = 1<<0;
          break;

          case 1:
          DDRB = 1<<1 | 1<<2;
          PORTB = 1<<1;
          break;

          case 2:
          DDRB = 1<<0 | 1<<3;
          PORTB = 1<<0;
          break;

          case 3:
          DDRB = 1<<1 | 1<<3;
          PORTB = 1<<1;
          break;
   }
}

int main(void)
{
   while(1){
          set_led(0);
          _delay_ms(2000);
          set_led(1);
          _delay_ms(2000);
          set_led(2);
          _delay_ms(2000);
          set_led(3);
          _delay_ms(2000);
   }
}
```

5. Download the above code, and confirm that your LEDs illuminate in sequence.

6. How would you illuminate multiple LEDs? The circuit itself stops you from independently illuminating LEDs, but consider what happens if you switch between two LEDs very quickly. Change the main loop to the following:

```
int main(void)
{
   while(1){
          set_led(0);
```

```
                set_led(2);
                set_led(3);
            }
        }
```

You can play with adding delays to better understand how this affects the physical LEDs.

7. Do not rip up the breadboard! You will build on this for the next part.

Part #3: Multiplexing Keypad

Objective

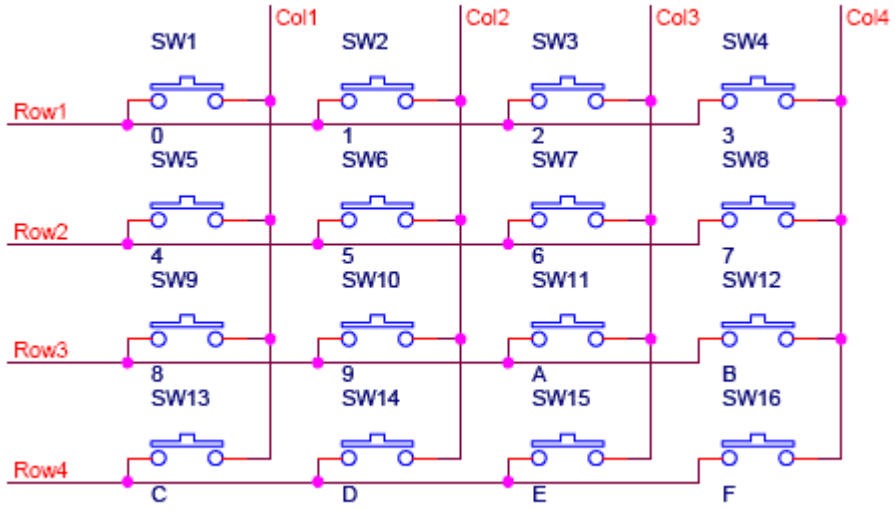
- Familiarize yourself with the idea of multiplexing switches on the inputs

Required Materials

- Lab setup from Part 2 (do not remove anything)
- 1x Keypad

Background

Just as we can save IO pins by multiplexing outputs, we can do the same things for inputs. In this case we will be using a keypad with 16 buttons, but using a 4x4 layout connecting the rows and columns as in the following schematic:

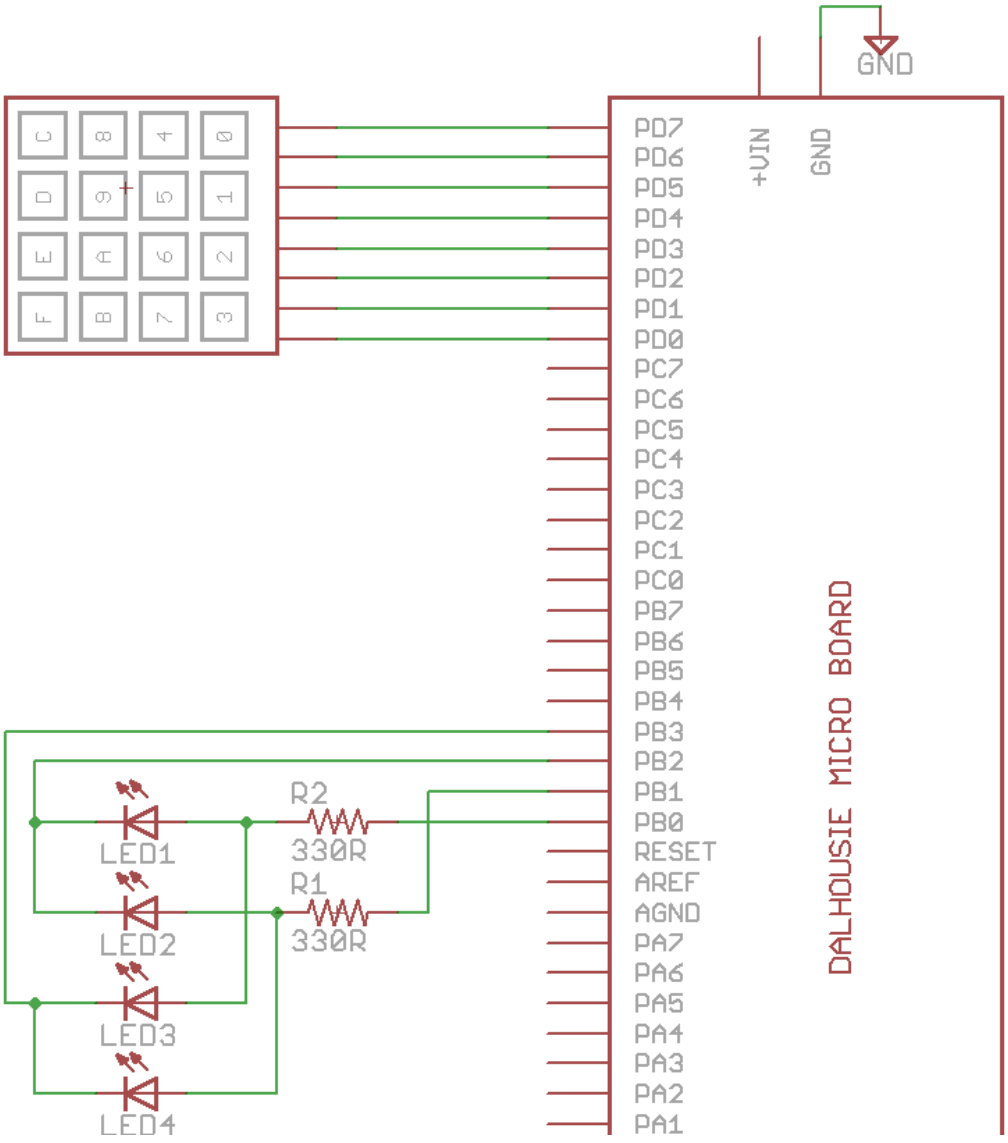


In order to read this we need a little more – for example how do you read SW3? This would mean setting the 'ROW1' pin high, and then seeing if 'COL3' was also pulled high.

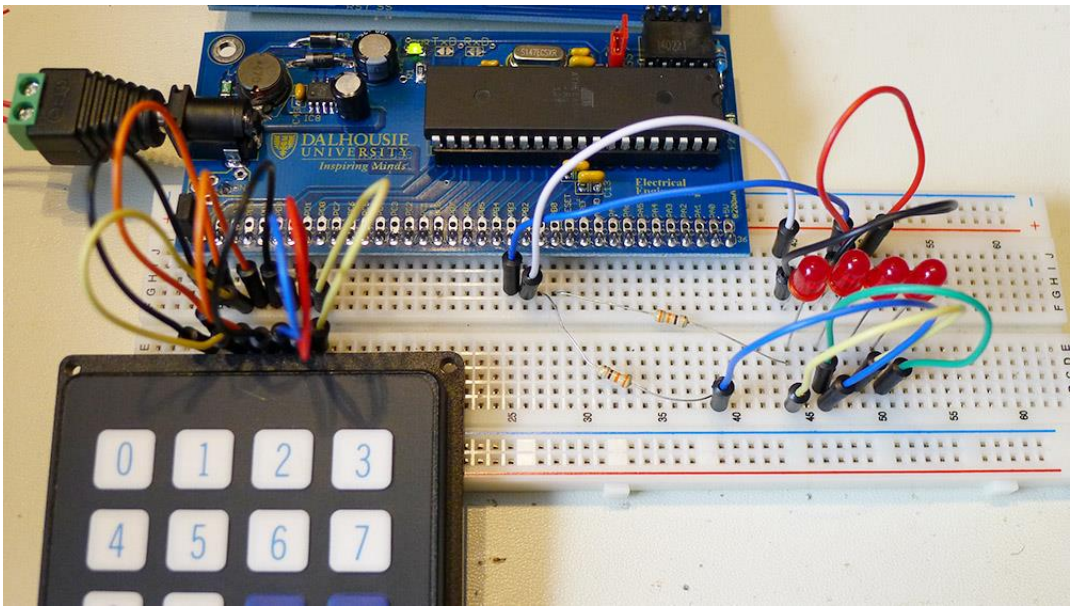
Procedure

1. Setup the breadboard as in Part 2 – the lab assumes you already completed Part 2

2. Wire in the 16-button keypad to PORTD. Looking at the keypad, note there are 8 pins on the bottom. You will wire them into the microcontroller:



Ensure they wire into PORTD0 to PORTD7, as the following image shows:



Note you might be able to plug the keypad directly into the pins without jumper wires.

3. You need to write a function which sets a certain row 'low', and then reads which of the column lines is low (if any). An example implementation is given below:

```
unsigned char read_button(unsigned char row, unsigned char col)
{
   unsigned char buttonstate = 0;

   //Turn pull-ups on
   PORTD = 0xF0;

   //Set row-col low
   DDRD |= 1<<row;

   //We need to wait for this to propagate before reading!
   _delay_us(10);

   if((PIND & (1<<(col+4))) == 0){
          _delay_ms(20);
          if((PIND & (1<<(col+4))) == 0){
                 buttonstate = 1;
          }
   }

   DDRD &= ~(1<<row);

   return buttonstate;
}
```

4. Using that function, you can modify your code from Part 2 to read the keypad, and then light a specific LED:

```
#include <avr/io.h>
#include <util/delay.h>
```

```
void set_led(unsigned char lednum)
{
   switch(lednum){
          case 0:
                 DDRB = 1<<0 | 1<<2;
                 PORTB = 1<<0;
                 break;

          case 1:
                 DDRB = 1<<1 | 1<<2;
                 PORTB = 1<<1;
                 break;

          case 2:
                 DDRB = 1<<0 | 1<<3;
                 PORTB = 1<<0;
                 break;

          case 3:
                 DDRB = 1<<1 | 1<<3;
                 PORTB = 1<<1;
                 break;
   }
}

unsigned char read_button(unsigned char row, unsigned char col)
{
   unsigned char buttonstate = 0;

   //Turn pull-ups on
   PORTD = 0xF0;

   //Set row-col low
   DDRD |= 1<<row;

   //We need to wait for this to propagate before reading!
   _delay_us(10);

   if((PIND & (1<<(col+4))) == 0){
          _delay_ms(20);
          if((PIND & (1<<(col+4))) == 0){
                 buttonstate = 1;
          }
   }

   DDRD &= ~(1<<row);

   return buttonstate;
}

int main(void)
{
   while(1){
          for(unsigned char row = 0; row < 4; row++){
                 if (read_button(row, 0)){
                        set_led(row);
                 }
          }
   }
}
```

Try pressing the buttons to see which ones are active. This code is only checking a single column, so we will need to expand this to scan all rows and columns.

5. Expand the previous code to scan over all rows. This will be done in a new function defined as follows:

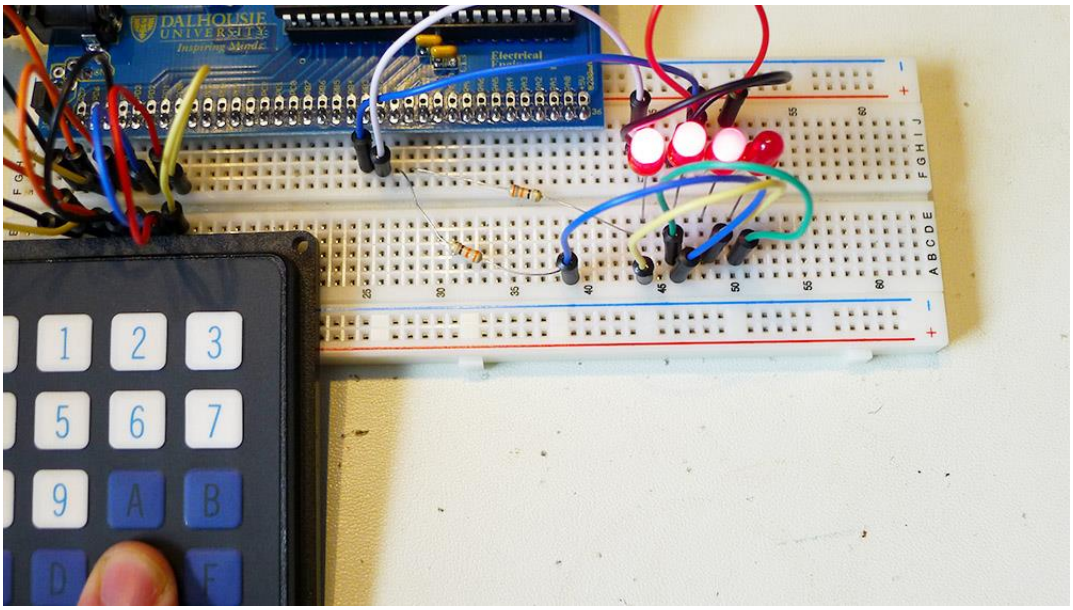
```
unsigned char decode_buttons(void)
{
   unsigned char row, col;
   unsigned char button = 0xff;

   for(row = 0; row < 4; row++){
          for(col = 0; col < 4; col++){
                 if(read_button(row, col)){
                        button = row | (col << 4);
                 }
          }
   }



   switch(button){
          case (0<<4) | 0: return 0x0F;
          case (0<<4) | 1: return 0x0E;
          case (0<<4) | 2: return 0x0D;
          case (0<<4) | 3: return 0x0C;
          case (1<<4) | 0: return 0x0B;
          case (1<<4) | 1: return 0x0A;
          case (1<<4) | 2: return 0x09;
          case (1<<4) | 3: return 0x08;
          case (2<<4) | 0: return 0x07;
          case (2<<4) | 1: return 0x06;
          case (2<<4) | 2: return 0x05;
          case (2<<4) | 3: return 0x04;
          case (3<<4) | 0: return 0x03;
          case (3<<4) | 1: return 0x02;
          case (3<<4) | 2: return 0x01;
          case (3<<4) | 3: return 0x00;
          default: return 0xFF;
   }
}
```

6. Scan through the button press, and display the button in the LEDs using binary code. For example this shows what happens if you press the ‘E’ button, which is 1110 in binary:



The following main function will get you started, but you will need to expand this, in particular not all statements were placed into the case statement! Add an example for each button and check they are working.

```
int main(void)
{
   unsigned char button, tempbutton;

   while(1){
          tempbutton = decode_buttons();

          if (tempbutton != 0xFF){
                button = tempbutton;
          } else {
                _delay_ms(30);
          }

          switch(button){
                case 0:
                      set_led(0xFF);
                      break;

                case 1:
                      set_led(3);
                      break;

                case 2:
                      set_led(2);
                      break;

                case 3:
                      set_led(3);
                      _delay_ms(50);
                      set_led(2);
                      break;

                case 4:
                      //FILL IN FROM 4 to 0x0D
```

```
                        break;

                    case 0x0D:
                        //FILL THIS IN
                        break;

                    case 0x0E:
                        set_led(0);
                        _delay_ms(50);
                        set_led(1);
                        _delay_ms(50);
                        set_led(2);
                        break;

                    case 0x0F:
                        //FILL THIS IN
                        break;

                    default:
                        break;
                }
            }
        }
```

Lab Questions

1. What causes problems when trying to read a switch, and how do we fix it?

2. The LED multiplexing could only illuminate certain LED patterns, as LEDs were linked together. How could we illuminate an arbitrary pattern (hint: our human eyes will blend together two quickly displayed patterns)?

3. Include code from each section in your report.