Multiplexing Input and Output

Matthew Beckler
beck0778@umn.edu
EE2361 Lab Section 007

February 27, 2006

Abstract

In this lab, a sixteen button keypad and two, seven segment LED displays are interfaced with a PIC microcontroller using a multiplexing scheme to reduce the total number of I/O lines required.

1 Introduction

When working with a microcontroller processor, the number of input and output lines is often a very limiting factor when designing hardware and software for the microcontroller. Many elaborate systems and schemes have been devised to reduce the number of I/O pins required, and one of the simplest and most elegant is a multiplexing system. For this project, the two, seven-segment LED digits are multiplexed, meaning that only one digit is lit at one time. This produces a very rapid flickering in the digits, because the active digit is rapidly switched back and forth. Since the human eye can only see flickering around 20ms, the digits both appear to be solid.

The 16 button keypad is also multiplexed. Normally, 16 buttons would require 16 I/O pins each, but with a multiplexing scheme, it is possible to use only 8 I/O lines. To use only 8 lines, you must arrange the keys in a 4x4 grid. Let's say that each button has two inputs, A and B. Connect all the A-inputs of each row together, they are four of the I/O lines. Connect all the B-inputs of each column together, they are the other four I/O lines. A graphical representation is shown in the circuit schematic attached at the end of this document.

2 Observations

For the scope of the main program, we have created a number of helper functions. One of these, the `void display(void)`, handles updating the LED displays. It is called very often, including inside spin loops and the debounce routine. It has a set of tabular data containing which segments are lit for each number. There is also a global variable `values[2]`, that holds the current digits to display. They are in the global scope so that any function may update their value, and the `display` function will have access to them. There is also the typical debounce function, to eliminate noise from the electro-mechanical

switches. It is different from the debounce routine from last lab, in that it calls the `display()` function every iteration to keep the digits updating. The `int input(void)` function returns the value of the key currently pressed, or `-1` if no key is pressed. This allows spin loops to be constructed quite easily to wait for a key-press or key-release.

Here is a pseudo-code implimentation of the main loop:

```
main
{
    wait until a button is pushed,
        updating in each loop

    update the digits' values

    wait a specified time for the button to debounce

    wait until all buttons have been released
        updating in each loop

    wait a specified time for the button to debounce

    repeat
}
```

To impliment at 48-key keyboard, could normally be implimented with a 6x8 grid, for a total of 48 pins. The 104 keyboard can be implimented with a 8x13 grid, for a total of 21 pins. The 104-key keyboard can be created with a grid of 8x13, or 21 pins. These numbers might seem acceptable, but since I didn't take EE2301 for nothing, I think we can do better. Assuming that only one row will be activated at one time, and only one key will be pressed at one time, we can use encoders and decoders to reduce the pin count. Since encoding and decoding is a translation between $n \Leftrightarrow 2^n$, we can adjust the number of rows and columns to optimize the pin count. We want to try to reduce one of the numbers (either rows or columns) to the next lowest power of two, without pushing the other number past its next power of two. For the 48-key keyboard, one of the numbers is already a power of two, so we can't reduce the total numbers at all. We can encode the 8 pins down to 3 pins, and the 6 pins also reduces to 3 pins. We can cut the 14 pins down to 6 pins total. For the 104-key keyboard, we leave the pin assignments as they are, as the 8 is already a power of two. It can be implimented with $3 + 4 = 7$ pins. For the 120-key keyboard, we adjust the grid from 10x12 to 8x15, which helps to reduce the pin count. We are left with $3 + 4 = 7$ pins.

3 Conclusion

If no multiplexing was used in this project, a total of 32 data I/O pins would be required. Through the use of multiplexing of both the keypad inputs and LED outputs, this number can be reduced down to 18 I/O pins, a savings of 14 pins. Another benefit of multiplexing the displays is in the area of power consumption. When each digit is only lit for half the time, approximately half the power is

required. This can be very advantageous for portable, battery-powered systems. A possible problem with the multiplexed keypad inputs is multiple key-presses. If the operator presses more than one key at the same time, a circuit can be created that would have ambigous results, i.e. the controller would not be able to properly determine which key is being pressed. This can be alleviated through the use of mechanical restrictions that restrict the user to pressing one button at a time.

4 Circuit Schematic

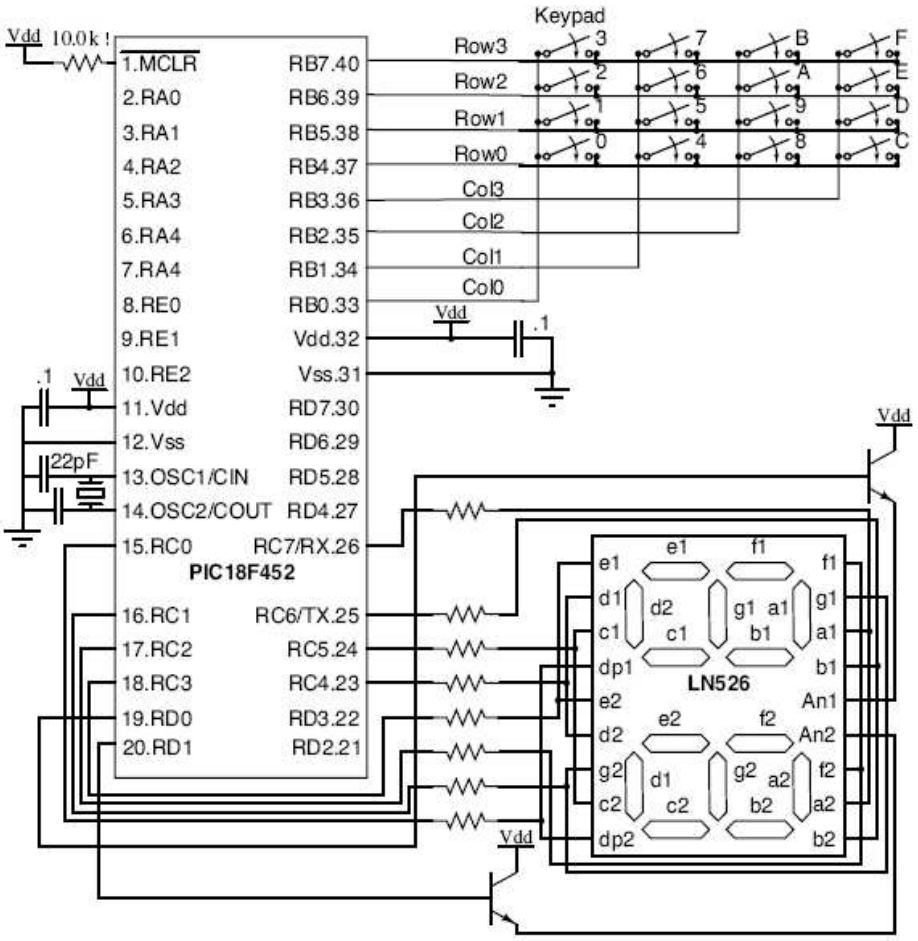

Figure 1: Circuit Schematic - Courtesy Professor Rennolet

5 C Language Program

```
#include <p18f452.h>

#pragma config OSC=HSPLL, WDT=OFF, BOR=OFF, PWRT=ON

static unsigned char values[] = {16, 16};
static signed int temp;

void display(void)
{
        static unsigned char numbers[] =
        {
                0b00000011,
                0b10011111,
                0b00100101,
                0b00001101,
                0b10011001,
                0b01001001,
                0b01000001,
                0b00011111,
                0b00000001,
                0b00001001,
                0b00010001,
                0b11000001,
                0b01100011,
                0b10000101,
                0b01100001,
                0b01110001,
                0b11111111
        };

        static unsigned char which = 0;

        if (!which)
        {
                PORTDbits.RD0 = 0;
                PORTC = numbers[values[0]];
                PORTDbits.RD1 = 1;
        }
        else
        {
                PORTDbits.RD1 = 0;
                PORTC = numbers[values[1]];
                PORTDbits.RD0 = 1;
        }

        which = !which;
        return;
}
```

```
void debounce(void)
{
        unsigned int count = 0;
        while (count < 0x2FFF)
        {
                display();
                count++;
        }
}

int input(void)
{
        /* Returns the button pressed
           or -1 if none are pressed */
        PORTBbits.RB6 = 1;
        PORTBbits.RB5 = 1;
        PORTBbits.RB4 = 1;

        PORTBbits.RB7 = 0;
        if( PORTBbits.RB0 == 0 )
                return 3;
        if( PORTBbits.RB1 == 0 )
                return 7;
        if( PORTBbits.RB2 == 0 )
                return 11;
        if( PORTBbits.RB3 == 0 )
                return 15;
        PORTBbits.RB7 = 1;

        PORTBbits.RB6 = 0;
        if( PORTBbits.RB0 == 0 )
                return 2;
        if( PORTBbits.RB1 == 0 )
                return 6;
        if( PORTBbits.RB2 == 0 )
                return 10;
        if( PORTBbits.RB3 == 0 )
                return 14;
        PORTBbits.RB6 = 1;

        PORTBbits.RB5 = 0;
        if( PORTBbits.RB0 == 0 )
                return 1;
        if( PORTBbits.RB1 == 0 )
                return 5;
        if( PORTBbits.RB2 == 0 )
                return 9;
        if( PORTBbits.RB3 == 0 )
                return 13;
```

```
        PORTBbits.RB5 = 1;

        PORTBbits.RB4 = 0;
        if( PORTBbits.RB0 == 0 )
                return 0;
        if( PORTBbits.RB1 == 0 )
                return 4;
        if( PORTBbits.RB2 == 0 )
                return 8;
        if( PORTBbits.RB3 == 0 )
                return 12;
        PORTBbits.RB4 = 1;

        return -1;
}

void main(void)
{
        INTCON2bits.RBPU = 0;   /* turn on PORTB's pull up resistors */
        TRISB = 0x0F;           /* bits: 0-3 = inputs, 4-7 = outputs */
        TRISC = 0;              /* 7-segment outputs */
        TRISD = 0;              /* segment-selector bits */
        PORTB = 0;
        PORTC = 0;
        PORTD = 0;

        while(1)
        {
                while ((temp = input()) == -1)
                {
                        /* waiting until a button is pressed */
                        display();
                }

                /* update digits */
                values[1] = values[0];
                values[0] = temp;

                debounce();

                display();

                while (input() != -1)
                {
                        /* wait until button is released */
                        display();
                }
                debounce();
        }
}
```