

EE 2361 - Introduction to Microcontrollers

Laboratory # 3

Multiplexing



Table of Contents

[Background](#)

[Objectives and Sequence](#)

[Supplemental Resources](#)

[Required Components:](#)

[Pre-Lab Reading](#)

[Output Multiplexing](#)

[Input Multiplexing](#)

[Switch Bounce](#)

[Pre-Lab Procedure](#)

[Output Multiplexing](#)

[Input Multiplexing](#)

[Pre-Lab Checklist](#)

[Lab Procedure](#)

[Lab Procedure Checklist](#)

[Going Further](#)

[Lab Report](#)

[Appendix - Wiring Schematic \(thank you Master Yoda!\)](#)

[Acknowledgments](#)

Background

Pins are always in short supply in today's ICs. While technology has shrunk the size of the individual transistor on a chip, it has done little to decrease the physical size of the connections to the chips. The number of I/O pins is a critical contributor to the price of an IC. One of the methods to decrease the number of pins required to control external devices is called (time) multiplexing.

Time-based multiplexing utilizes high-speed microprocessors to drive outputs or sample inputs multiple times such that it appears simultaneous to the observing system (in our case the human brain.)

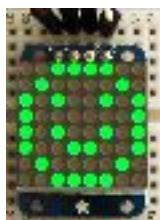
In this lab we will use the concept of multiplexing to lower the required pin count needed to implement a 16-button keypad and a 16-LED "dual seven-segment" display. Our PIC24FJ only has 28-pins (of which a maximum of 21 are usable I/O's,) thus we have some work to do!



Given a 16-button keypad, you could connect each button to one of 16 digital I/O pins and sense them individually. However, when arranged as a matrix of momentary (SPST) switches, organized into four columns and four rows, each switch will connect a unique row wire to a unique column wire. The PIC24 can then rapidly "scan each row" to determine which button is pressed. Note that, if you press multiple switches simultaneously, you cannot necessarily determine exactly which switches are closed.

This relationship, of the number of switches to the number of I/O pins in a matrix, is an inverse square, i.e., if you have a 4x4 switch matrix, you need 2x4 I/O pins, and if you have a 5x5 switch matrix, you need 2x5 pins. As the number of keys goes up, the advantage of multiplexing becomes overwhelming. For obvious reasons, 104-key keyboards use multiplexing to save a lot of pins!

Similarly when a display consists of many LEDs the persistence of vision can be used to make several lights appear lit with fewer outputs. Remember in Lab 1, when the frequency of the blinking LEDs increased, at some point they didn't appear to be blinking any longer even though they were off for half the time?



The persistence of vision technique is almost always used on LED matrix displays. For example, the [LED matrix display](#) from EE1301 has row and column drivers. The display cycles through each row lighting the appropriate columns, then moving on to the next row. It does this so fast that the entire display appears to be constantly glowing.

Objectives and Sequence

In this lab you will use multiplexing to overcome the inherent pin shortage on the PIC24 microcontroller. In doing so, you will again create libraries that can be added to future programming projects.

Lab 3 is the first of the “intermediate” labs. These labs will provide you with both the original device specifications and extended explanations of how to interpret them. They will no longer provide you with exact code examples, but will utilize pseudo code segments that describe the flow and content required, but you will be responsible for looking up the exact register names, syntax, and potentially other details of the implementation.

This lab is organized into two sections, output multiplexing and input multiplexing, each with a pre-lab. You will have two weeks to complete this lab. You should do both pre-labs the first week if you are behind on the previous labs.

Supplemental Resources

[Seven-Segment Display Datasheet](#)

[4x4 Matrix Keypad Datasheet](#)

[ASCII Character Table](#)

[Short Video on bit-masking](#)

Required Components:

Standard PIC24 requirements (caps, pullup, debug header, etc.)
4x4 Matrix Keypad
1 - Dual 7 Segment Display
2 - 2N3904 transistors
8 - 220 Ω resistors
2 - 1k Ω resistors

Pre-Lab Reading

In this laboratory you are going to build a nearly-trivial application which will accept characters from the keypad and display them, two at a time, on the display. To reduce the number of digital I/O pins required, the keypad and display will be driven in a multiplexed manner.

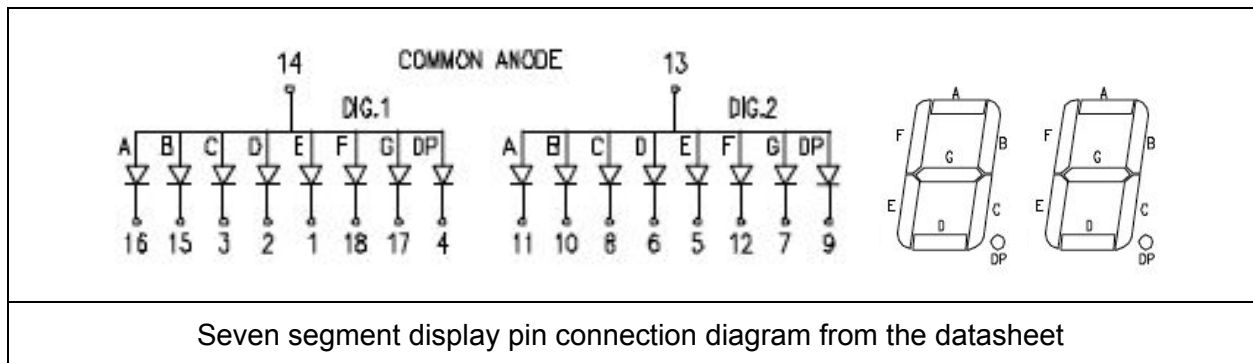
The pre-lab is split into two independent sections, [Output Multiplexing](#) and [Input Multiplexing](#). If you are behind on your labs, please complete both sections before coming to lab.

Output Multiplexing

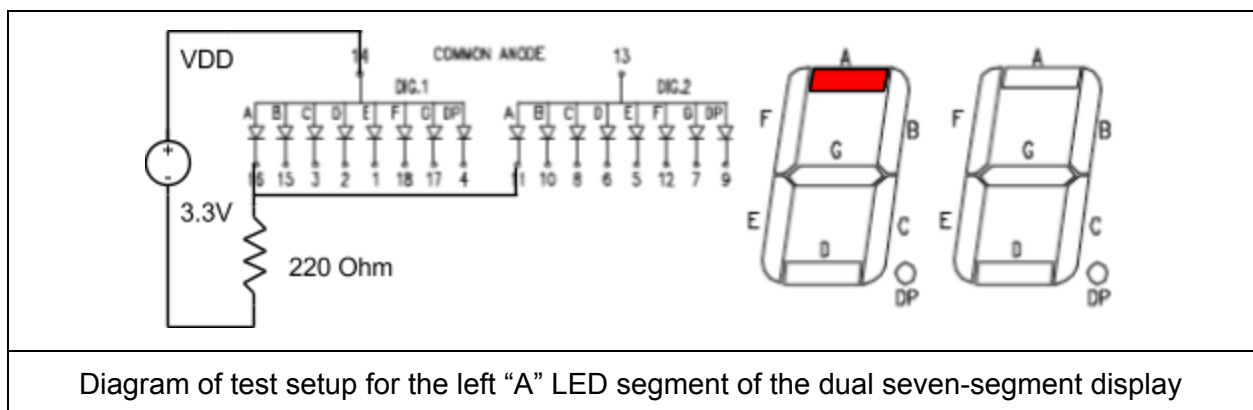
As mentioned in the lab background, we will utilize the persistence of vision effect to allow us to alternate displaying the left and right digit of the display, while the display appears to have both of these digits lit.



The 2-digit 7-segment display provided in your lab kit provides pins to access every cathode and one anode for each digit. The pin wiring is shown in the lower right corner of the [datasheet](#) (replicated below).



By pulling up the a shared “common” anode of an LED (for example, pin 14) to 3.3V and a single cathode (16) down to GND through a current limiting resistor, you can turn on an individual LED segment (Dig.1 “A”, the top bar). Even if the alternative cathode (11) is also pulled down to GND by a current limiting resistor, the segment will not light up (Dig.2 “A”) because the anode (13) is not pulled up. This allows pin 11 and pin 16 to share an MCU output without compromising the ability to individually light the segments (Dig.2 “A” and Dig.1 “A”).



This concept holds true across all 8 cathodes (the 7-segment bars, A-G, plus the decimal point, DP). Thus, we only need to control 2 anodes and 8 cathodes (10 pins) to individually address each of the 16 LED segments in the display.

Note: You can easily try this with just the seven-segment display, one 220 Ω resistor, and a single jumper wire. Manually stimulating your display can also be handy when debugging your initial wiring. Warning NEVER connect an MCU output directly to VDD or GND. Make sure the MCU outputs are not connected to the display when you debug this way.

Also note that if the common anode 14 is pulled up, then we can pull down more than one of the cathodes 4, 17, 18, ... and have multiple light segments light up, creating patterns such as the

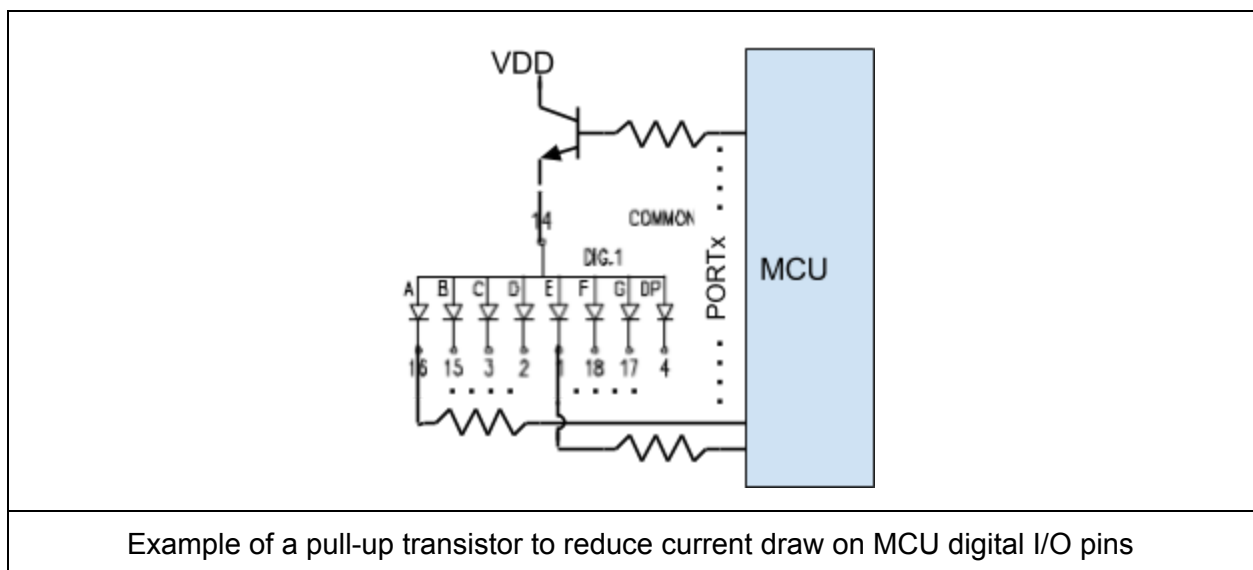
digit  and a cute, cuddly scorpion  (don't try this at home!).

When handling LEDs the user must always be careful not to exceed the current limits of the microprocessor outputs. The [PIC24FJ64GA004 datasheet](#) (page 3) lists a limit of 18mA per I/O pin. Also, using the [7-segment datasheet](#), it can be determined that the forward voltage drop of the LED is 1.7V (typical). Together with a 220 Ω resistor, it can be determined that each lit LED will consume about 7.3mA of current.

$$I_{LED} = \frac{3.3V - 1.7V}{220\Omega} = 7.3mA$$

Two LEDs lit by one pin would be safe (14.6mA). However, the common anode poses a problem as it is shared by 8 LEDs. Driving the common-anode HIGH (pin 14), and at the same time driving all 8 cathodes LOW via current limiting resistors (pins 16, 15, 3, 2, 1, 18, 17, 4) will result in an overcurrent condition (58mA) on the MCU pin (pin 14).

We can solve this problem by using a “pull up” transistor. The 2N3904 NPN transistor in your kit serves this purpose nicely. If you are curious, this transistor will reduce the current requirement of our MCU by a factor specified as “beta” or the current-gain of the transistor. A minimum beta of 30 is listed in the NPN [datasheet](#). Therefore, this fix is sufficient to drive roughly 70 LED segments. This should be more than sufficient for our purposes.



Once the physical connections have been determined (see the pre-lab procedure and lab appendix for the full circuit schematic,) the structure of the code needs to be set. The display

needs to be updated once every fraction of a second (usually faster than 60 fps → 16.6ms). You will want to structure your code in a loop. Each iteration through the loop the first digit will be displayed for half the time and the other digit will be display for the second half of the time. The following pseudo code demonstrates this:

```
void loop() {  
  ...  
  showChar7seg(char_myChar1, int_CharacterNum1);  
  delay(FramePeriod/2);  
  showChar7seg(char_myChar2, int_CharacterNum2);  
  delay(FramePeriod/2);  
  ...  
}
```

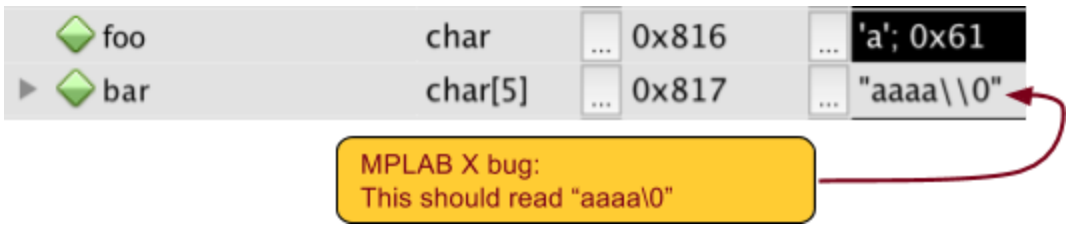
It is important that the code in the “...” above runs substantially quicker than the “delay()” commands. As an alternative to the loops structure you could use a Timer with interrupts to handle updating the display. (Timers are discussed in the following lab, but if the class has covered them already or you just want to work ahead, feel free to try to implement an interrupt based frame update.)

In this example the function `showChar7seg(char myChar, int CharacterNum)`, is a lookup table that converts from the ASCII character `myChar` to the LATx outputs needed to drive the cathodes of the LED-segments (a,b,c,d,e,f,g,dp).

It is customary in most C-libraries to pass character data as ASCII characters. A list of ASCII characters can be found [here](#). Luckily, C helps us greatly by making ASCII characters easy to use. The data type `char` understands character inputs described by single quotes (ie., 'a'). The data type `char[]` (a pointer to an array of char's), understands strings input by double quotes (ie., "aaaa").

The following code shows the setting of the variable “foo” to the ascii character ‘a’ (which is 0x61). And the variable “bar” to a pointer to the beginning of the string “aaaa” (four 0x61's terminated by a end of string symbol, NULL.)

```
char foo = 'a';  
char bar[] = "aaaa";
```



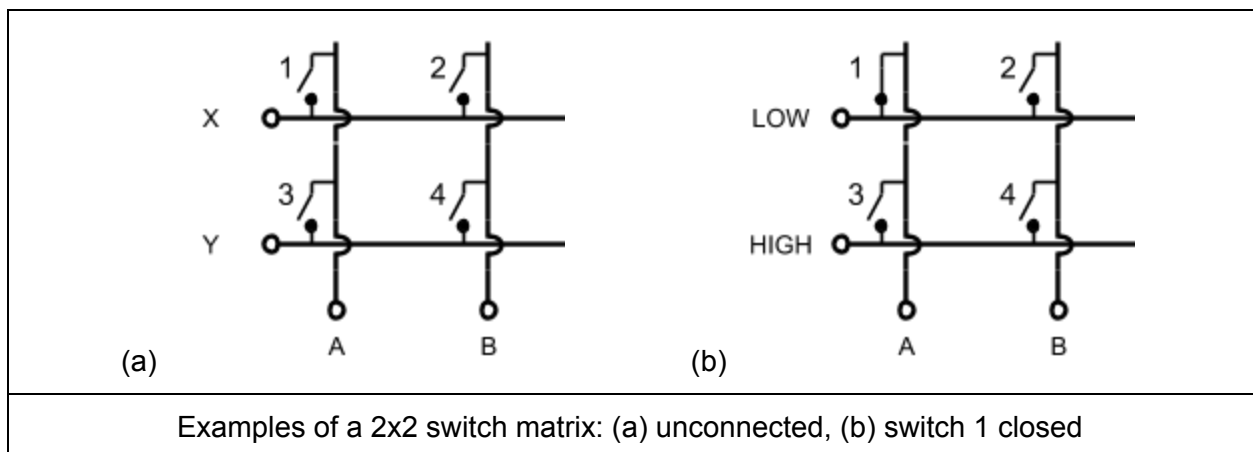
The screenshot shows two variables: `foo` of type `char` at memory address `0x816` containing the value `'a'; 0x61`, and `bar` of type `char[5]` at memory address `0x817` containing the value `"aaaa\0"`. A yellow callout box with a red arrow pointing to the `bar` variable contains the text: "MPLAB X bug: This should read 'aaaa\0'".

Example of setting characters and strings

When we create our library we will need to accept characters as input to our display interface functions.

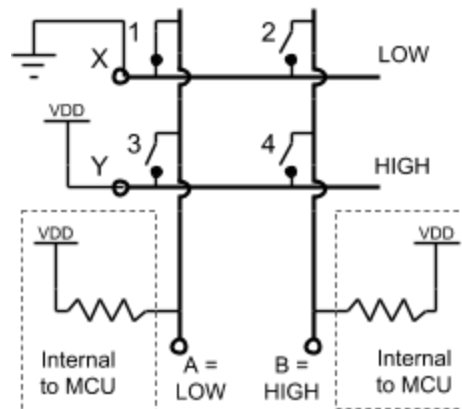
Input Multiplexing

This lab will utilize a switch matrix to implement input multiplexing. A switch matrix is an NxM matrix of switches that connects vertical and horizontal bus wires, as shown in the diagram below.



A basic 2x2 matrix switch is shown in diagram (a) above. We then connect up the matrix to an MCU such that pinX, pinY are outputs and pinA, pinB are inputs and press switch 1 (closed). It is possible to detect which switch is closed by cycling through output states (for X and Y). Specifically, in the example shown above in diagram (b), pinX is pulled low and as a result pinA is driven low through Switch 1.

There is a problem however, pinB is floating and could easily be read as a high or low. The use of the built-in MCU pull-up resistor on the input pins will cause pinB to have a high voltage by default, unless connected to ground by an external device, such as a switch. Note that input pinA is also pulled up through the on-chip pull-up resistor but will still read LOW as the LOW signal on the digital output pinX overrides the weak pull up. This configuration is shown on the diagram below:

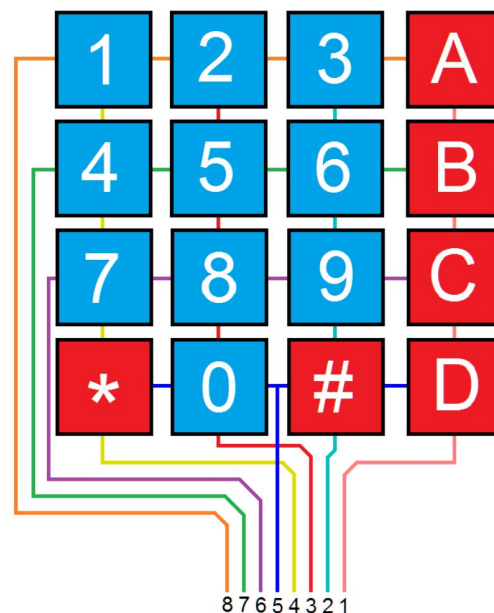


Example of 2x2 switch matrix utilizing the input pull up resistors in the MCU

If the programmer wishes to check if any of the buttons are pressed on row Y, the MCU would drive pinX=HIGH and pinY=LOW, then sense again on pinA and pinB.

IMPORTANT QUESTION: Can we also use the opposite approach and send pinX=LOW and pinY=HIGH to detect if any switches on row Y are pressed? Be prepared to explain your position to your TA.

The 4x4 keypad, included in your lab kit, is describe by its [datasheet](#). Please review the first two pages of this document now. The wiring diagram for this matrix switch is shown below.



Provided 4x4 keypad wiring diagram

The 4x4 keypad will require 4-inputs (with pull-up resistors turned-on) and 4 outputs from the MCU. You will need to sequentially set a single row's output LOW, then read all four inputs. Repeat this procedure for row and whichever input is low, uniquely identifies a pressed key. A pseudo code example is shown below demonstrating the detection of the "6" key.

```
Outputs<8:5> = 0b0111; // Set Row 1 output Low
delay(1us);           // this delay could be lower, probably 2-3 cycles
RowOne = Inputs<4:1>; // returns 0b1111 (no buttons pressed in row 1)

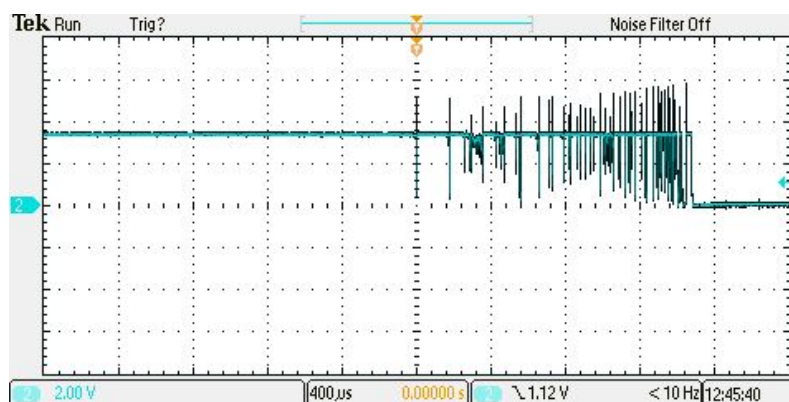
Outputs<8:5> = 0b1011; // Set Row 2 output Low
delay(1us);
RowTwo = Inputs<4:1>; // returns 0b1101 (the "6" key was pressed!)
...
Outputs<8:5> = 0b1110; // Set Row 4 output Low
delay(1us);
RowFour = Inputs<4:1>; // returns 0b1111 (no buttons pressed in row 4)
```

Pseudo code showing the detection of the "6" key.

It is important to note, that using a matrix keypad and a "scanning" architecture, it is not possible to always uniquely determine which of multiple keys are pressed, so in this lab we are going to assume that Dr. Doofenshmirtz is confined to a locked cell and no one is going to conspire to press more than one key at a time during this lab (way to go Perry the Platypus).

Switch Bounce

As previously mentioned in EE1301, real-world switches suffer from a phenomenon called "bounce". This is a physical effect where the metal contacts of a switch rapidly open and close for a short period of time after the switch has been depressed. Here's a picture of the typical square "tactile" microswitch.



Scope output of a switch bouncing

The time scale is very important for this discussion, notice how all the bouncing occurs in a $<2\text{ms}$ window. From the observational point of a view of a typical person, this switch is ideal. It would turn on and off an LED with no noticeable bounce.

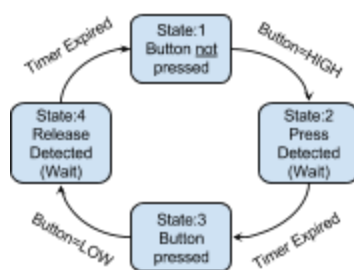
However, for a microcontroller making observations at a rate of one every 62.5ns , it is completely unclear what the state of this switch is for 1.2ms (or about 20 cycles.): is Superwoman sending a secret message in Morse code? Or a regular person is trying to turn a switch on? It should also be made clear, these “tactile” microswitches are some of the best and fastest switches available.

During the EE1301 lab, we were able to safely hide behind the fact that the Photon (in Automatic Mode) waits at least 1ms between iterations of `loop()`. From the user code point of view the switch state was never unknown.

For the PIC24FJ64GA002 however this is not the case. We must develop some type of filter to determine the state of our buttons during the “bounce” period. There are three methods usually used to debounce switches:

- Existing program delay
- State-based debouncing
- Hardware RC filter

The simplest of these filters is to utilizing existing program delay to sample the switch state once during an extended period of time. The Photon from EE1301, used this method, `loop()`'s max update rate was 1ms in Automatic mode. Generally speaking, if you are interfacing with human beings, then it is likely some other part of your code also updates at a relatively slow rate. In the case of this lab the display frame rate is also in the millisecond range. This method has the disadvantage that all user input will be delayed by the sample rate of the switch.



State-based debouncing utilizes interrupts (or a fast polling loop) to run an internal state machine. The state machine effectively “locks out” further changes in value of the pin after it has initially transitioned. This allows for the input to be observed within about one clock cycle. The disadvantage of this method, is it requires the use of semi-limited resources (interrupts and timers). This method is often used in simple hardware (like FPGA's, where timers and state machines are plentiful). A basic form of the state

machine is shown in the [EE1301 Quick Lesson - Debouncing Input Switches](#).

Finally, a hardware RC filter can be used to filter out the rapid changes in the output of the switch. This method requires that the RC filter be as slow as the last “bounce” and thus will lead to a delay in detection of button presses. Additionally, the RC filter method adds physical components to the system which is anathema to modern electrical system design practices

(whose goal to reduce every product down to a single IC chip with no external components.) This method is usually employed when dealing with initial prototypes or refitting an existing product where changing the code can be difficult.

Pre-Lab Procedure

Output Multiplexing

In this section you will wire up and test basic functionality of your 2-digit 7-segment display.

- 1.) Connect VDD to an anode and a single LED segment cathode through a 220 Ω resistor to GND (light it up). Verify that you understand the pinout and light up any LED segment manually.

The full schematic for this lab is available in the [appendix](#) as a full page diagram. Please review that now.

- 2.) Connect the dual seven-segment display, 220 Ω resistors, and pull-up transistors as shown in the schematic and described in the pre-lab reading. Leave the 220 Ω and 1000 Ω resistors *not* connected to the MCU.
- 3.) Test each individual led segment with two jumper wires, make sure they operate as expected.

The pins RB<9:2> will drive the cathodes of the seven-segment display. The pins RB<11:10> will drive the right and left anodes respectively. These choices are rather arbitrary, but please respect them for the sake of the sanity of your TA.

- 4.) Connect RB<11:10> to the anodes via 1k Ω and pull-up transistors (see schematic)
- 5.) Connect RB<9:2> to the cathodes via current limiting resistors (220 Ω) (see schematic)

You should now be setup to begin developing code for your display. It is suggested that you start small, by creating code that simply drives RB<11> HIGH, while driving RB<9> LOW, to verify your wiring.

- 6.) Create an initialization function for your display:

```
void init7seg(void);
```

This function should configure the appropriate values for AD1PCFG and TRISx, as well as safe initial values for the display such that it is blank. Create it in such a manner that it can coexist with a future Input Multiplexing initialization function.

- 7.) Develop a low level library function that drives a single character on the display at one time. Create the function specified below:

```
void showChar7seg(char myChar, enum DIGIT myDigit);
```

This function should set the outputs RB<11:10> and RB<9:2> to the appropriate values to show a single character on the display. Obviously, your code does not need to cater to complex characters such as 'M'. For now you can limit your code to handle 0..9, and A, b, C, d, E, F.

The use of `enum DIGIT myDigit`, is an implementation detail that you can choose to implement how you like, just make your code easily readable. You can read more about enums [here](#).

- 8.) Create a test function that displays the digits 1,2,3,4,5,6,7,8,9,0,a,b,c,d and two custom digits of your choice. You will show this test function in a program to your TA at the beginning of lab.

Your test function should be something like this:

```
void orserTest7seg(void) {  
    showChar7seg('0', LSB);  
    delay(250);  
    showChar7seg('1', MSB);  
    delay(250);  
    ...  
}
```

Example Test function to exercise the seven segment display

Input Multiplexing

In this section you will wire up and test the basic functionality of your 4x4 matrix keypad.

- 1.) Connect RA<3:0> to the columns of your keypad (pins<4:1>) and RB<15:12> to the rows of your keypad (pins<8:5>). See the schematic in the [appendix](#).
- 2.) Create an initialization function for your matrix:

```
void initKeyPad(void);
```

This function should configure the appropriate values for AD1PCFG, TRISx, and CNPUx. Create it in such a manner that it doesn't clobber the Output Multiplexing initialization function.

Note: The pullup resistors are provided for nearly all pins on the PIC24FJ64GA002 microprocessors. This feature is associated with the (C)hange (N)otification capability for input pins. However, the CNPUx register has a different pin number scheme. The CNx pins *are* notated on the provided [pinout](#) they simply have a different number than our standard RAx, RBx, or RPx pins. For example, to utilize the pull-up resistor on the pin RA0, the user must set the CN2PUE bit associated with pin CN2 in the CNPU1 register (ugh, I know...).



- 3.) Create a function that determines the raw button that is pressed on your keypad. For example:

```
unsigned int readKeyPadRAW(void);
```

The output type and encoding of this function is up to you.

- 4.) Use the “Debug Main Project” feature of MPLAB X to verify, in hardware, that you can detect any key press reliably. Be prepared to replicate this for your TA if you are having issues in the main lab procedure.

Pre-Lab Checklist

- ☐ Complete the Pre-Lab Reading
 - ☐ Output Multiplexing
 - ☐ Input Multiplexing
- ☐ Take pre-lab quiz
- ☐ Complete the following functions:
 - ☐ `void init7seg(void);`
 - ☐ `void initKeyPad(void);`
 - ☐ `int readKeyPadRAW(void);`
 - ☐ `void showChar7seg(char myChar, enum DIGIT myDigit);`
- ☐ Demo seven-segment display test program to your TA

Lab Procedure

Take what you built in the pre-lab and extend it. You will need to refine the functions of your pre-lab procedures such that they can work together. Create a mapping function between the output of the `readKeyPadRAW()` and the `showChar7seq()` functions. Make this function easily re-mappable to support custom overlays on the keypad matrix (for example arrow keys). Wrap all the functions together into a complete program that does the following:

- Start with all segments off
- Every <30ms, scan the keypad for input.
- (advanced, and optional) Better yet, if you have covered interrupts in class, utilize the Change Notification interrupt (CN), so that you only perform keyboard scanning when a key is pressed. For that to happen, you need to set all row sensitization signals to 0 by default so that ANY key press results in one of the columns to be pulled down, and for the CN interrupt to get activated. Then you have to send one-hot 0 sensitization signals to each row individually to detect which key was pressed.
- For every key press, shift the rightmost displayed value to the left seven-segment digit (most significant), then display the key pressed on the rightmost digit(least significant).
- The display should be multiplexed and show both digits simultaneously (at least to the human eye).

Additionally the following features are required:

- The program should function continuously, always displaying the last two digits pressed
- There should be no noticeable flicker, even when shaking the display
- All display and keypad functions need to be self contained in a separate library (.h and .c files)
- Externally facing library functions must take programmer readable inputs for characters

The following features are for you to define:

- Create custom character “fonts” for the * and # buttons

Demonstrate your program to your TA.

Lab Procedure Checklist

- ☐ An included library for use of the matrix keypad and dual seven-segment display
- ☐ Check that your display doesn't flicker when shaken

Going Further

- Build a mappable array of characters that can be easily redefined to support custom overlays on the 4x4 matrix. Use this array in the conversion function between the raw keypad read and the input to the display function.

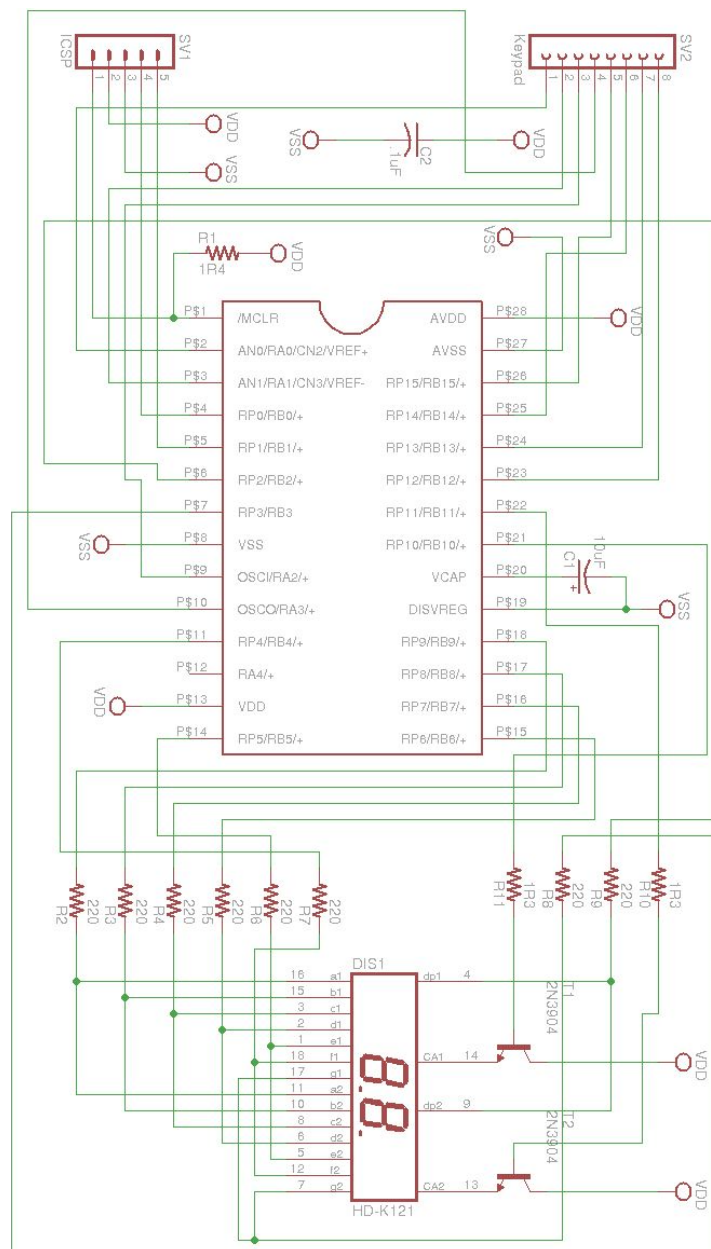
Lab Report

Put together a very brief report on the work in this lab. It should include the following:

- A paragraph on your experience with each of the devices (keypad, display) that you connected.
 - Did it work the first time you connected it and programmed it? What mistakes did you make? How did you resolve the issue. If you didn't have any issues, just say so.
 - Include the relevant code you used to get it to work. When “describing code”, it is important to break the code into sections and describe how and why you chose to method of implementation you used.
- The Lab TA must see a demo of your circuits. Make sure s/he checks off your work before you leave the lab (or, setup a time outside the lab to demo if you don't have time to finish).



Appendix - Wiring Schematic (thank you Master Yoda!)



Acknowledgments

Large parts of this lab were procured from the original EE2361 Lab 5, by Charles Rennolet. Thank you Charles for creating a fun, interesting, and challenging lab.