Name _____

Name _____

(Most parts are team so maintain only 1 sheet per team)

ME430 Mechatronic Systems:

Lab 5: ADC, Interrupts, Steppers, and Servos

The lab team has demonstrated the following tasks:

_____ **Part (A) Using the Potentiometer and the ADC***

_____ **Part (B) LEDs and Stepper Motors with Interrupts***

_____ **Part (C) PIC on a Breadboard, with LEDs**

_____ **Part (D) Breadboard PIC Running a Stepper Motor**

_____ **Part (E) Simple Interrupts***

_____ **Part (F) Breadboard PIC Running a Servo Motor**

* Indicates that the part is done on the green board. All green board parts are individual (in all labs).

Part (A) Using the Potentiometer and Analog-to-Digital Conversions

In this part of the lab, we want to read an analog (not digital) input into the PIC, and then display that analog value on the LCD screen. We will need to have a bit of background before we can begin.

Overview of Analog to Digital Concepts:

In previous labs we have always used digital inputs. A digital input is either a 0 or a 1 (i.e. a Low or a High). The other type of input is an analog input. For example, if you use your multimeter to check whether you have a good battery, the (analog) reading might be 1.536 volts. We can also cause the PIC to read in these types of values.

For this lab we will need a way to create an analog input for the PIC to read. Fortunately, the green boards have two handy potentiometer circuits which are attached to pins RA0 and RA1 on the PIC. The potentiometers are little grey boxes near the power jack. When you rotate the potentiometer (turn the x-shaped slot), it creates a variable output voltage somewhere between 0 volts and 5 volts.

How do potentiometers work in this context? They have three leads (see Figure 1). One of the leads is connected to ground all the time, one of the leads is connected to power, and the third lead gives us the output voltage. The total resistance R_1+R_2 is always the same, but when we rotate the knob we are actually physically moving the "wiper" (the location of the V_{out} line) which adjusts how much of the total resistance is R_1, and how much is R_2. When the wiper is near the ground lead, the potential V_{out} will be close to 0 Volts—perhaps 0.121 Volts. When the wiper is near the power lead, the potential will be close to 5 Volts—maybe 4.934V. When the wiper is about halfway between ground and power, we might get 2.437V.

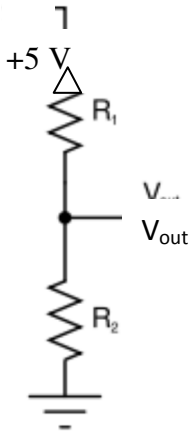


Figure 1. Basic Potentiometer Circuit.

The RA0 and RA1 pins on the PIC are connected to these V_{out}s. The PIC can read this value and store the analog value as an integer. But how can the PIC store 2.437 as an integer? The PIC18F4520 uses an internal analog-to-digital converter (ADC) to convert the analog V_{out} to an "int" and stores it that way. (Actually, it uses the bottom 10 bits of an unsigned int variable to store the result). When V_{out} is 0.000 volts the analog-to-digital converter (ADC) stores the value 0b0000 0000 0000 0000 (0 in decimal). When the value is 5.000 volts the ADC stores the value

0b0000 0011 1111 1111 (1023 in decimal). For 2.500 Volts the ADC stores the value 0b0000 0001 0000 0000 (512 in decimal). For 1.000 Volts the ADC would store the value 205 (1023*1/5 = 205).

For now, we will simply use the analog-to-digital conversion functions that come with the PIC libraries.

Your Overall Goal for Part A:

Have the PIC read the analog input from RA0, convert it to a digital number from 0 to 1023, and display the result on the LCD screen. This should happen continuously, so that if you rotate the potentiometer and change the value on RA0 while the program is running the screen should update itself. The LCD screen should say:
"The ADC is ####"
Here #### should be the ADC value from 0 up to 1023.

For instance, when the potentiometer knob is turned to the middle it should say:
"The ADC is 512"
When the potentiometer knob is turned completely clockwise the LCD should display:
"The ADC is 1023"
When the potentiometer knob is turned completely counter-clockwise the LCD should say:
"The ADC is 0"

Step-by-step:

1. First, it would be a good idea to get the A/D conversion working without worrying about the LCD display.

Create a new project from "template.c" and "Debug Project". It's just a template file though, and it doesn't do anything.

Go to the “C Library Files” pdf on the courseware page, and find section 2.2 “A/D Converter Functions”. This gives us the functions we can use with the A/D converter, and also tells us what library needs to be included to use the functions. We will need to use the following functions (in this order):

```
OpenADC
ConvertADC
BusyADC
ReadADC
```

Do NOT close the ADC—we’re running it continuously for our application.

The most complicated of these functions is the `OpenADC` command. Modify the last (ADCON1) entry in the OpenADC command in your file to read just a single analog input by changing `0x0B` to `0x0E`:

```
OpenADC(ADC_FOSC_8 & ADC_RIGHT_JUST & ADC_12_TAD,
  ADC_CH0 & ADC_INT_OFF & ADC_REF_VDD_VSS, 0x0E);
```

You can read about these parameter settings in the “C Library Files” pdf. (The voltage setting in the documentation is out-of-date; ADC_REF_VDD_VSS is correct.)

The other commands are simpler and so we want you to study the “C Library Files” pdf to figure out how they should be used. Be sure to check out the “Examples” section 2.2.2. Again, do not close the ADC for this application.

Program the PIC to simply read in the input from RA0 and store it in a variable. When you pause the program, you should be able to see the value in a Variables watch window.

2. Next we want to display this numerical value on the LCD screen, using the techniques you learned in Lab 4. Be sure to drop the leading zero on numbers like 0512! (In addition to the techniques you used to print to the LCD in Lab 4, you might want to look at the ``sprintf'' function which is also described in the “C Library Files” pdf. It is not essential, but it is fairly cool.)

3. Now add the words to the LCD display. Make sure that everything is running correctly and call your instructor over to check this part off on the front page.

Part (B) LEDs and Stepper Motors with Interrupts

The PIC on the green board is wonderful, but at some point we are going to want to hook up those PIC chips we ordered. We're going to learn how to hook them up on breadboards in this lab, and then later we will actually disconnect them from the laptop completely.

For this part, we're going to download and study one more program on the green board. Then we'll use exactly the same program with the chips we wire on the breadboards ourselves.

Go to the "labs" page of the course website and download the program "Stepper_Motor_using_interrupts.c". Create a new project and "Debug Project". It takes a moment to get started, but then the LEDs should flash in a regular (stepper motor step) pattern.

Let's start by looking at the code a bit. (It's long, so we haven't printed it here. You could print it out if that helps you—otherwise you can look at it on the screen as we go through the pieces.) First the opening comment block:

```
/*****************************************************************
* FileName:        Stepper_Motor_using_interrupts.c
* Processor:       PIC18F4520
* Compiler:        MPLAB C18 v.3.36
*
* This file uses the timer 0 to set an interrupt event.  When the
*  interrupt occurs, it changes the RC0:RC4 state.  You can modify
*  code within the high priority interrupt to change how often the
*  interrupt occurs.
*
*  H-Bridge connections for driving a stepper motor.
*       RC0             =       L293 Enable line
*       RC1             =       Phase A control line
*       RC2             =       Phase A control line
*       RC3             =       Phase B control line
*       RC4             =       Phase B control line
```

This code will eventually be used to cause the breadboarded PIC chip to control a stepper motor through an H-bridge chip.

A bit farther through the code, you will see this portion of the code:

```
// Run the clock at 500 kHz (I could've picked about anything)
OSCCONbits.IRCF2 = 0;
OSCCONbits.IRCF1 = 1;
OSCCONbits.IRCF0 = 1;
```

This sets the internal clock frequency to 500 kHz.

The next important thing that happens here is that we set up timer0 to let us know when to flash the lights (step the motor):

```
// Setup the timer with a 1:4 prescaler with 16 bits resolution
// Therefore the timer0 freq is 500 kHz / 4 / 4 = 31.25 kHz
OpenTimer0( TIMER_INT_ON & T0_16BIT &
                T0_SOURCE_INT & T0_PS_1_4 );
// Should take a little over 2 seconds to overflow the counter from TMR0 = 0
// If you write in a different starting value for TMR0 it'll overflow sooner
```

With a 500 kHz clock, and a 1:4 prescaler, timer0 is running at 31.25 kHz. This command also sets up timer0 to cause an interrupt whenever it "overflows", or gets to 0xFFFF. If the timer starts at zero (0x0000) it will take 65536 timer0 ticks, or just over 2 seconds, before the timer triggers an interrupt. It's "just over 2 seconds"—how much time is it exactly?

We calculate that it will take "exactly" _____ seconds for the timer to overflow. (Don't worry right now that the lights aren't going at this speed—we'll explain that shortly.)

When the timer overflows we want that event to trigger an interrupt. Although we set up the timer0 to cause interrupts, we also need to make sure and turn on Global Interrupts as well. That comes next in the code:

```
// Enable Global interrupts (I'm using Compatibility mode)
    INTCONbits.GIE = 1;          //  Enable Global interrupts
```

Next, we need to prepare the RC0 through RC4 pins for digital output:

```
// Setup the digital IO pins
ADCON1 = 0x0F;                  // Make sure they are digital not analog
TRISC = 0xE0;                   // Make the RC4:RC0 outputs
PORTC = 0x00;                   // Clear the bits to start with
```

Now look over the main loop:

```
        while (1) {
            // A blank while loop, think of all the things you could do here!
            // When you use an interrupt the main loop is free for something else
        }
```

Indeed your while loop does NOTHING in this program. It is free to be used for anything else if you needed to add to this program. That's the whole idea behind interrupts-- you can multitask!

Near the bottom we find the code for the interrupt service routine—this tells the PIC what to do when timer0 overflows:

```
/*****************************************************************
* Function:        void high_isr(void)
* Overview:  This interrupt changes the state of the RC4:RC0 pins when
*             the timer zero overflows (0xFFFF -> 0x0000) and triggers
*             this interrupt code to run
*****************************************************************/
#pragma interrupt high_isr
void high_isr(void) {
if(INTCONbits.TMR0IF) {
                INTCONbits.TMR0IF = 0;   // Clear interrupt flag for TIMER Zero
                switch (recentState) {
                        case STEP1:
                                recentState = STEP2;
                                break;
                        case STEP2:
                                recentState = STEP3;
                                break;
                        case STEP3:
                                recentState = STEP4;
                                break;
                        case STEP4:
                                recentState = STEP1;
                                break;
                        default:
                                recentState = STEP1;
                                break;
                }

                PORTC = recentState | ENABLE_PIN;
        }
}
```

With this portion of the code, we're checking to make sure it was the timer0 overflow that caused the interrupt. Unless something is very messed up, it was, because the timer0 interrupt is the only one we are using. However, in the future you might add other interrupts so it's good to have the framework ready for adding other interrupts.

Next, this piece of code uses a switch statement to set up the PORTC lines, which are (as you know) connected to the LEDs on the green board.

Even though we calculated that timer0 should overflow every 2 seconds or so, you may have noticed that the lights are changing faster than that. The reason for the shorter

delay is due to the fact that we don't restart timer0 at 0 each time. Take a look at the next chunk of code:

```
// The Timer0 frequency is 31.25 kHz
// Pick where to start the time to determine how fast it overflows
// Every overflow the stepper motor will take a single step
//WriteTimer0(3036);   // 1 step every 2 seconds
//WriteTimer0(18661);  // 1 step every 1.5 seconds
//WriteTimer0(34286);  // 1 step every 1 seconds
WriteTimer0(49911);  // 1 step every 0.5 seconds
//WriteTimer0(57723);  // 1 step every 0.25 seconds
//WriteTimer0(62411);  // 1 step every 0.1 seconds
//WriteTimer0(63973);  // 20 step every second
//WriteTimer0(64911);  // 50 step every second
//WriteTimer0(65224);  // 100 step every second
//WriteTimer0(65380);  // 200 step every second
```

If we set the timer to 0, it takes a bit more than 2 seconds to overflow. But, the larger we make the starting value the less time it takes to get to 65535 and overflow. Study this code, and comment out the 1 step every 0.5 seconds. Pick a different line and uncomment it—play with it until you think you understand what is happening.

Once you believe you understand this piece of code reasonably well, start the programming running on your green board and call your instructor over to check off this part on the front page.

Part (C) PIC on a Breadboard, with LEDs

In this part of the lab, our overall goal is to take a PIC chip that you got in the mail, hook it up properly on the breadboards, download the program from the last part, and run it. Needless to say, this could take a little bit of explaining.

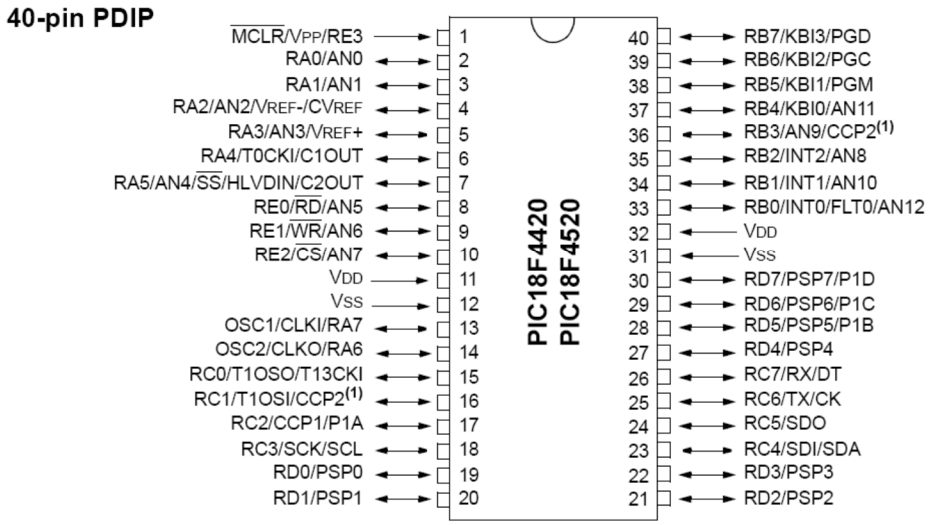
Background on connections between the PICkit3 and the PIC:

Find the little double-sided 6-pin male-to-male header in your lab kit and plug it into your red PICkit3. Of these 6 oins only 5 lines are really connected-- one line isn't used. The five lines that are actually used are:

PIN		
1	MCLR	Master Clear, connects to Pin 1 on the PIC
2	V_{DD}	Voltage at the drain (i.e. Power, 5 Volts)
3	V_{SS}	Voltage at the source (i.e. Ground, 0 Volts)
4	Programming data (PGD)	The line for the programming data
5	Programming clock (PGC)	The clock input connecting PICkit3 to PIC

Note that Pin 1 is closest to the white triangle on the PICkit3.

Next, we need to understand where these lines go into the PIC18F4520.

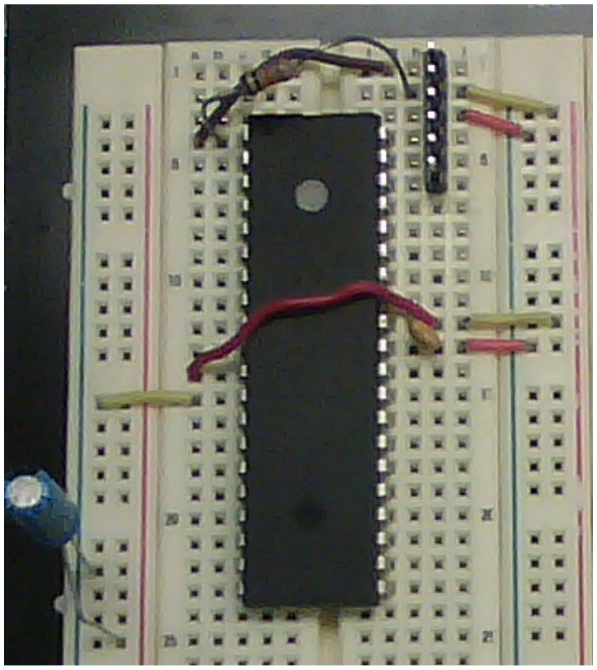


Recall that the MCLR line is Pin 1, the two VDD (power) lines are on Pins 11 and 32, and the two VSS (ground) lines are on Pins 12 and 31. The Programming Data (PGD) line is Pin 40, and the Programming Clock (PGC) line is Pin 39.

Hooking up the PIC on the Breadboard: We are going to use this set up for several labs, so we want you to lay out the boards as we describe here. Otherwise you'll need to rip it apart and redo it later.

We want your breadboards set up as follows.

1. The breadboards should be side-by-side, with a little space remaining at the top.
2. The regulated power rails should be the two middle rails, and the unregulated power rails should be the two outside rails.
3. The ground rails should all be connected together.
4. Note how the header is mounted in the photo, this minimizes the number and length of jumpers used to connect the PICkit3 to the PIC.
5. The white arrow for the PICkit3 will be at the top of the header, near the top of the board.
6. The PIC should have pin 1 at the top. However, you don't have to use the same wire colors ☺.



Careful this board has Blue **(GND) on the left (which is different than your board)**
The goal of this image is ONLY to show you where to put the 6 pin header,
figure out the rest with the steps listed above.

Using the 6 pin header in this location, you need only 3 jumpers connecting a) pin 1 on PICkit to pin 1 on PIC, b) regulated power to PICkit pin 2, and c) ground to PICkit pin 3. The PGD and PGC connections are made directly on PICkit pins 4 and 5. These steps are described in more details here…

Next we will get the power and ground connections ready. Check off these steps:

1. Connect Pin 2 on the PICkit3 header to regulated Power.
2. Connect Pin 3 on the PICkit3 header to Ground.
3. Connect Pins 11 and 32 on the PIC to regulated Power.
4. Connect Pins 12 and 31 on the PIC to Ground.
5. Put a 0.1 µF(104) Decoupling Capacitor between power and ground next to the PIC, as shown in the photo. Put your larger Decoupling Capacitor between power and ground near the regulator chip.
6. Put a resistor (1K to 10K) between Power and Pin 1 (MCLR) on the PIC.
7. Add an LED circuit to show when the power is on.
8. Double check all of your wires and count carefully to make sure the V_{DD} and V_{SS} pins are connected correctly to the PIC.

Once these connections are all set you won't move them. The header and PIC will stay in these places until we finish our line following robots.

After you get everything connected and double-checked, turn on the power and make sure nothing gets hot.

Connecting to the PICkit3:

The next step is to plug in the PICkit3 and see if you can connect and download a program. Give it a go! Download the same program we were using in the last part on the green board.

___ We can connect to the PIC and it appears to download a program. (Self check-off)

Of course, we won't really know if the program is working unless we have some LEDs to light up. Use a Darlington chip and your knowledge from earlier in the course to set up a Darlington circuit to run 5 LEDs. (You may find it useful to look at notes from previous labs to recall how to do this.)

Next, connect the RC0 thru RC4 lines on the PIC to the Darlington inputs, in order to control the 5 LEDs. Get the five LEDs to run just like the 4 LEDs on the green board (plus one!).

When you get the program running, with the LEDs blinking properly, call your instructor over to check you off on the front page

Amazing! We don't need the green board anymore!

Part (D) Breadboard PIC Running a Stepper Motor

In this part of the lab, we want to use our breadboarded PIC to actually run a stepper motor. You already have all of the knowledge you need to do this, so we have deliberately made the instructions terse.

First, in addition to controlling the LEDs through the Darlington, use an H-bridge circuit to drive the stepper motor. Remember that the H-bridge is **NOT the 74LS47** chip. Run the stepper motor at 1 step per 0.1 seconds, and then try a few other speeds. Find a speed that causes the motor to turn at 1 revolution per second. Be sure that you set the H-bridge up so that the power for the stepper motor comes from the unregulated line.

Self check-off: _____ Our motor turns at 1 revolution per second. It takes _____ steps of the stepper motor to make one full turn.

Just turning the motor one way at a constant speed is dull. We can do a lot more now that we have a microprocessor controlling the motor. We want you to program these three operating stages to happen in sequence:

Table 3: Stepper motor program

Stage	Direction	Speed	Revolutions
1	CW	1 rps	2
2	CCW	1 rps	3
3	CW	0.5 rps	1

So, we want to make the motor spin clockwise (CW), turning at 1 revolution per second for 2 revolutions. After finishing those 2 clockwise revolutions, the motor needs to start spinning counterclockwise at 1 revolution per second for 3 revolutions. Finally, the motor should spin at 0.5 revolutions per second for 1 revolution. (Then, the motor should stop.)

When you have finished this part, call your instructor over to check off this part on the front page.

Part (E) Simple Interrupts

If haven't watched the video on "Interrupts, Day 1 of 2", you should do that before you work on this lab. You can watch it in class if that works for you.

For this part we'll move back to the green board for another interrupts example (one more quick exercise on the green board before moving off for good.)

Here, our goal is the same as it was in Lab 4, Part E—write your name and age to the LCD screen, and have yourself age as a button is pressed. However, this time we are going to use the button RB0 with interrupts to trigger the "aging process".

Make a new project folder for this project, and copy over your completed files from Lab4 Part E. Make sure the new project still works the way it did before—printing to the screen and aging when you press RB0.

You will need a ".c" file with some examples of how to code interrupts. You can use the "template_with_interrupts.c" file from the courseware page of the ME430 website.

Now create a program which uses interrupts and button RB0 to write your name and age to the LCD screen and have yourself age when the button is pressed. Move all of the code for updating the LCD into the high interrupt service routine (isr) ``high_isr''. The main routine should have an empty `while(1)` loop. (When the RB0 button is pressed, the PIC should go to the interrupt service routine and update the LCD. Then the PIC will go back to "while-ing" away time in the main routine.) Don't forget to take care of the interrupt flag once you get into the isr. Also don't forget to

- make all pins digital
- make the LCD pins outputs
- make the RB0 pin an input
- initialize the interrupts

When you have this working, show it to your instructor and have it checked off on the front page. (Be prepared for your instructor to check whether you actually used an interrupt ☺)

Part (F) Breadboard PIC Running a Servo Motor

In this part of the lab, we want to use our **breadboarded PIC** to run a servo motor connected to pin RB0. You will want to go back and review how servo motors work (the "Motors" video lecture). Use **un**regulated power for the servo red wire.

To tell the servo motor the angle we want to move to, we will add four buttons (basic switch circuits) to our breadboard, and connect them to pins RA0 through RA3 on the PIC. Here is our goal:

- When we press the button connected to RA0, the servo should go to 0 degrees.
- When we press the button connected to RA1, the servo should move to 45 degrees.
- When we are not pressing any buttons, the servo should always go back to 90 degrees.
- When we press the button connected to RA2, the servo should move to 135 degrees.
- When we press the button connected to RA3, the servo should move to 180 degrees.

A servo motor is similar to a stepper motor in that it can be easily controlled with Timer interrupts and it can be moved to a certain position. The similarities end there, though-- internally the two are totally different. A servo motor is really a DC motor with a potentiometer, as discussed in the video lecture on motors.

Get a servo from the cabinet. Remember that we have a limited number of them, so please return it to cabinet before you leave this room.

Download the starter code "ServoMotor.c" from the labs page, build a new MPLAB project with that code, and open up the .c file. A servo motor requires a 50 Hz (up to 60 Hz) signal, so we have selected the WriteTimer0 function that gives 50 "steps" per second, or an interrupt that happens at 50 Hz. If you look in the interrupt service routine (ISR) you will see that we have only three lines:

```
If (INTCONbits.TMR0IF)
{
        INTCONbits.TMR0IF = 0;   // Clear interrupt flag for TIMER Zero
        WriteTimer0(64911);  // 50 step every second

        // We'll be adding servo code here
}
```

Within this interrupt, we will set RB0 high, delay for an appropriate amount, and then set RB0 low. The table on the next page shows the appropriate (approximate) delays for each angle:

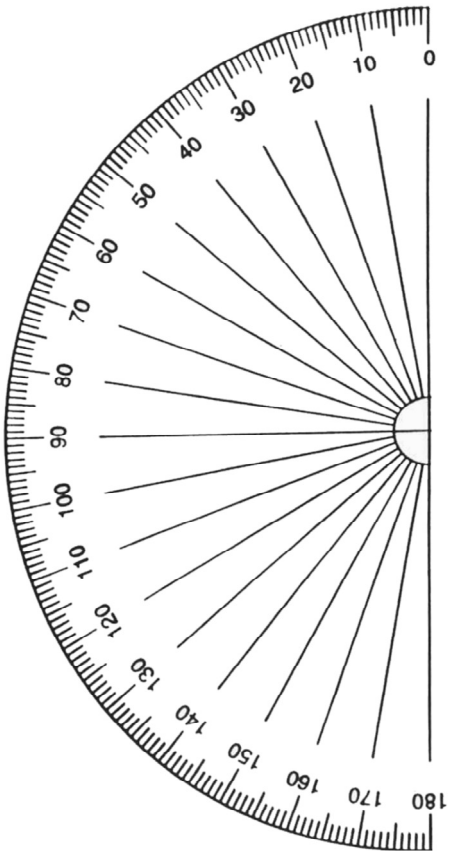
Button Pressed	Desired Angle	Time Delay
RA0	0^o	0.5 ms
RA1	45^o	1.0 ms
none	90^o	1.5 ms
RA2	135^o	2.0 ms
RA3	180^o	3.0 ms

You will need to add the switches to your breadboard and modify the interrupt code.

Test your code using the protractor below. It doesn't have to be perfect, though. (Note: You are also learning about PWM with the PIC, and it might seem as though PWM would be a wonderful way to control a servo motor. It turns out that the PWM on the PIC chip we are using is not accurate enough for good servo motor control. You can do it, but you will have to get an oscilloscope and check the signal that the PIC is sending out the PWM line. Then you can play around with it until you get a signal that is correct.)

Get this checked off on the front page by your instructor when you get this working.

When you are done with this, you can remove the 4 switch circuits.


0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180