the bus-device to think with

The following activities are designed to provide an intermediate or expert Cricket user with useful concepts in electronics and microcontroller programming. An introductory text of electronics is a necessary companion to this guide, such as the McGraw-Hill Benchtop Electronics Handbook or Practical Electronics for Inventors. Use these books to fill in the gaps or to learn more about a particular subject. A *Bus-Device to Think With* is needed, of course, along with a copy of the BDTTW Edition of Jackal. A digital mulitmeter is very useful and can be purchased cheaply at www.mpja.com. For users that push further into microcontroller programming, a few more resources are needed: A basic text of the C programming language, a CCS C compiler, and a Microchip PIC Start programmer. Users committed to the exercises presented here and willing to spend many hours hovering over solder fumes or in front of glaring CRTs may find themselves soon programming microcontrollers and even designing their own Bus Devices for the Cricket system.

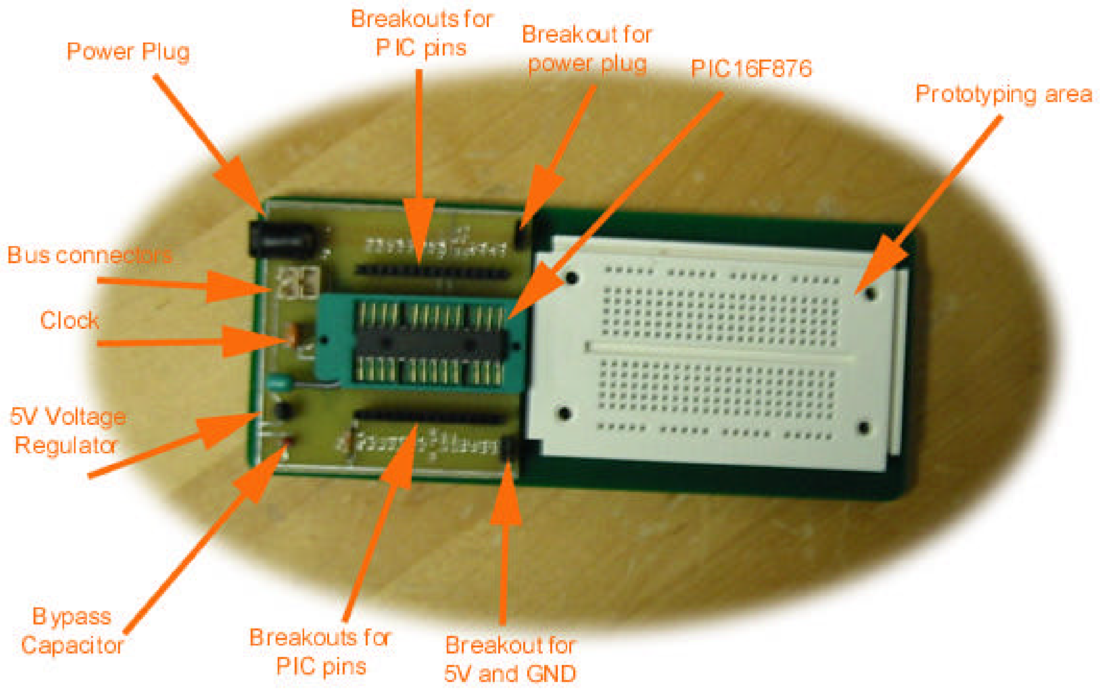
Modern microcontrollers are quite capable and flexible devices. The MicroChip PIC16F876, used in the Bus-Device to Think With, has 22 I/O pins, two PWM modules, three timers, a RS-232 port, and five analog-to-digital converters. The following exercises will discuss each of these capabilities and guide the user through their use. The goal is to provide enough experience with the basic concepts in microcontrollers and electronics[1] to let the user begin playing with ideas and to initiate them into the enormously helpful community of artists and engineers using these materials in their work. This is a heavy document that glances off many complex topics. Do not expect to understand all of it, but do not underestimate the time and commitment that these topics deserve. Most of all, build something fun with these exercises. It makes it all worthwhile.

Depending on your experience level, you may want to read up on the basic concepts in electricity and electronics (try to understand the concepts of voltage, current, resistance, circuits, capacitors, and the relationships between them). The activities that follow

[1] One important skill to master is the ability to read electronics specification sheets. One must wade through endless technical nonsense when all one really wants is a simple explanation. Included with this manual are spec. sheets for almost every component used. For some circuits in this manual, you might even have to read them.

assume little previous knowledge, but they also assume that you have some outside resources and motivation to seek out additional ones as needed.

The Bus Device to Think With

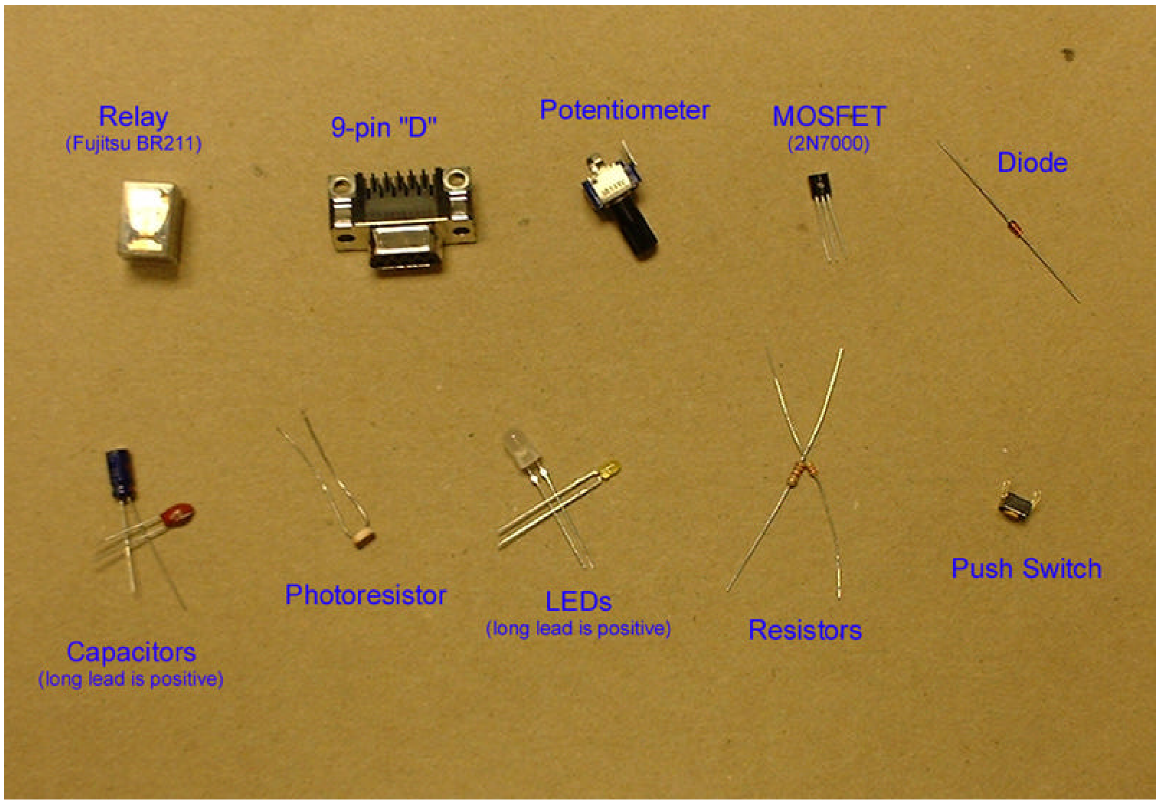


The Bus-Device to Think With is a device that allows you to play with the functionality of a PIC microcontroller through the familiar Cricket Logo language. The code running on the PIC16F876 is written in C for the CCS compiler (http://www.ccsinfo.com/picc-referall.shtml). It should be studied and tinkered with once you are comfortable with the basic functionality of the PIC. It contains two functions to handle communications with the Cricket, and a large table of commands. It compares the instruction sent from the Cricket to this table, then executes that function. Whenever possible, the CricketLogo commands are named and used similarly to the C commands, hopefully making the C code more understandable for the Cricket user.

The board itself contains only parts needed for the PIC to run (clock, MCLR resistor) , for power (5 volt regulator, bypass capacitor), and for Cricket communication (bus headers). The PIC is powered off of the power supplied by the Cricket Bus line. The 5V regulator cuts the 9V of the Cricket down to a safe 5V, and the bypass capacitor reduces the noise in the power supply. The bus headers contain the 9V power, ground, and the bus signal line. This signal line is connected to the RB0 (discussed later) on the PIC.

The board schematic is included in this document for your use. *Take notice to the breakout plugs on the bottom, labeled +5 and GND. These are your sources of 5V and ground whenever a circuit diagram calls for them. The external supply plug is not connected to anything other than the other two breakout plugs on the lower right side of the board. These are useful if you want to use a different voltage supply for the circuit you build, such as 9V, 12V, or 24V.*

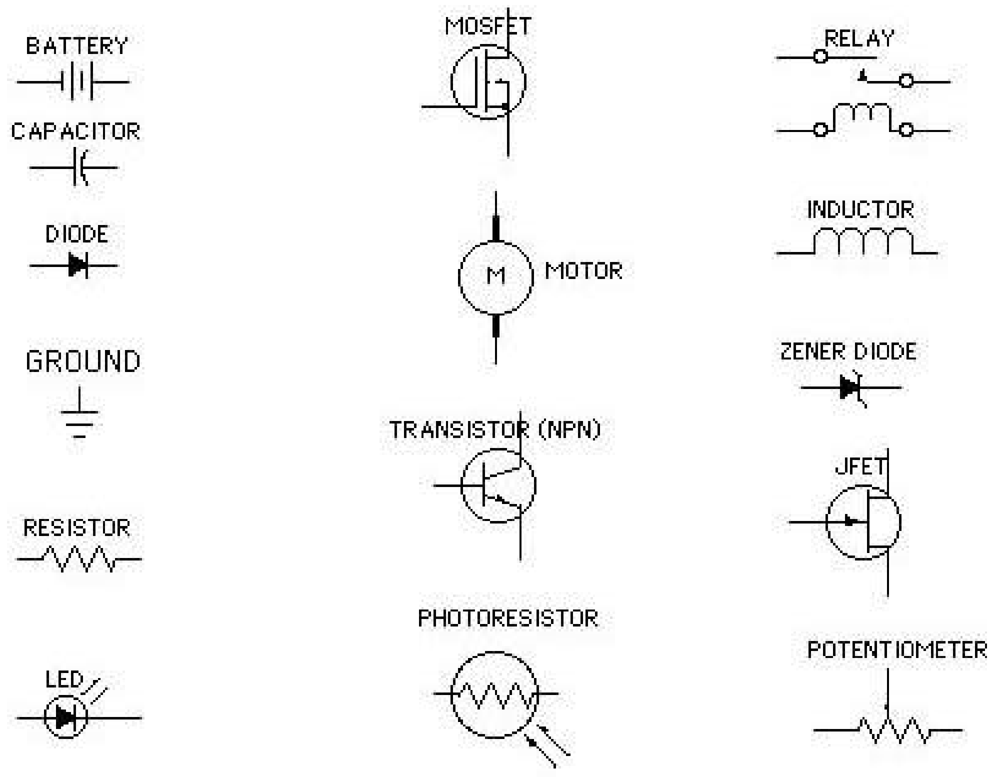
Parts:



All parts and data sheets from www.digikey.com.

Additional parts included: 9V motor, slide switch, multiple stripped wires of different lengths

Common symbols used in electronic schematics:



These parts can be found at any Radio Shack or online at www.digikey.com.

Activity One: LED array!

The PIC16F876 has 22 *pins* available for use as an input or an output (I/O), divided among three *ports.* A port is a group of pins, intended to give the user control over an entire collection of pins at a time. These ports are names A, B, C. For instance, pin 5 of the PIC16F876 is named RA4, meaning pin 4 of the A port, while pin 12 is named RC1, meaning pin 1 of the C port. Don't think too hard about this, its simply an imposed order. Before a pin can be used, we must tell the microcontroller whether it is an input or an output. We will start by making the pin RA2 an output with the function *bit_clear $A2*[2]. The name *bit_clear* has origins that we will discuss later, but now is not the time. For now, understand that *bit_clear* makes a pin an output (capable of producing 0 Volts or 5 Volts) and *bit_set* makes a pin an input (capable of reading 0 Volts or 5 Volts). Once the pin is set as an output, we can set its value low (0 volts) or high (5 Volts). This is accomplished with the function *pin_set*, which makes the pin high, and *pin_clear*, which makes the pin low. The simplest function looks like:

```
to turn_a_pin_on_briefly
    bit_clear $A2
    pin_set $A2
    wait 10
    pin_clear $A2
end
```

Lets add a LED to make this more visually exciting. We know that pin A2 will be set at 5 Volts. Most LED's will draw too much *current* at this voltage, so we will add a resistor to our circuit that will act as a *current limiter*. This will bring us to the first equation of the activity, the famous Ohm's Law:

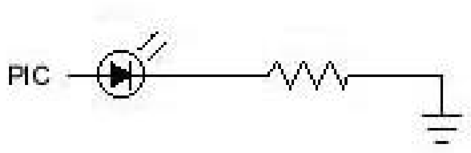
Voltage = Current * Resistance (V = I * R)   (Units: V = Volts, I = Amps, R = Ohms)

We know that pin A2 will be at 5 Volts. This forms the left hand side of the equation. LED's are rated by the amount of current they can draw so that they don't burn out. The included LED's are rated at 20 mA. This is the current term. Then, we can assume that the LED itself has negligible resistance and solve for R.

5 V = 20 mA * R
R = 250 Ohms

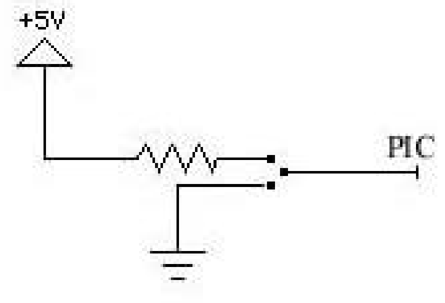
[2] What's the $ sign all about? See appendix on Hex numbers.

The schematic of the circuit you will build looks like the following:



Now, to turn this schematic into a circuit: On the prototyping area (known as a breadboard), use the small segments of wire included in you kit to connect pin on the PIC (A2 in this case) to the long lead (positive side, called the *anode*) on the LED. The take the resistor (in this case, 250 Ohms) and connect to short side of the LED (negative side, called the *cathode*) to the ground (Ground can be found on the bottom left corner of the board. One header is 5V, the other ground. If you look closely, they are labeled as such). After building this circuit, run the CricketLogo program. For more information on using a breadboard and creating circuits from schematics, please see the appendix on this topic.

Next, we will add a slide switch that will signal the microcontroller to turn on this LED. To do this, we will make pin A3 an input to check the state of our switch. We also will choose an input of 0V to be off and an input of 5V to be on, but this could flipped. Once configured, the status of an input pin is checked using the *pin_test* function. This function returns a 1 at 5V and a 0 and 0V. This is the convention used in most digital systems (1 = high voltage level, 0 = low voltage level). Before wiring the below circuit, use the multimeter[3] to probe the pins of the switch. To do this, use the resistance measuring function of the multimeter (called an *ohmmeter*, because it measures in ohms) to check which pins are connected with the switch in a given position, and which ones are connected when the switch is in the other position. Drawing a diagram of the switch might help. The pin that gets connected in either case will go to the PIC (probably the center pin). One of the remaining pins will go through a resistor to 5V, the other to ground.



3 See appendix for more details of using a multimeter

We write this function:

```
to lil_switch_program
   bit_set $A3                ;set A3 as an input
   bit_clear $A2         ;set A2 as an output for LED
   loop[
     ifelse ((pin_test $A3) = 1)
      [pin_set $A2]
      [pin_clear $A2]
     ]
end
```

Try it out!

As a challenge, figure out how to safely use the push button to turn on the LED. Hints: There should be a resistor between 5V and any given path to ground or the PIC. Probe the switch…unlike the slide switch, you only have two pins which get connected on push, and nothing which is connected otherwise. This circuit is included in the appendix .

Now you are own your own. Go crazy- you have 20 pins on 3 ports to use any way you wish. Warning: Four pins on the PIC16F876 are special: **1.** Pin A4 doesn't have the ability to go high by itself. We must 'tie it high', as they say. Here is how it works: We place a large resistor (~10 kiloOhm) between the pin and 5V. This will allow the pin to go into its high state. However, when the pin is set low at 0V, the resistor is large enough that not much current flows. In fact:

5V = I * 10 kOhm
I = .0005 amps

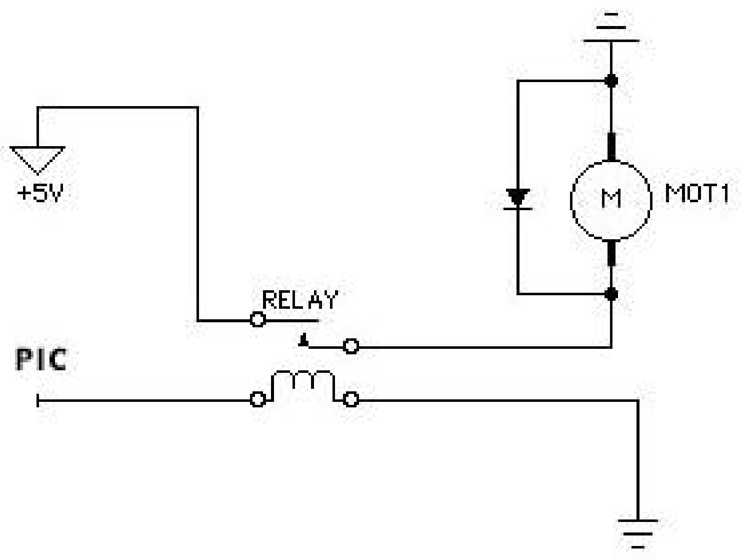
2. Pin B0 has been designated to serve at the communication line between the PIC16F876 and the Cricket. In fact, every Cricket and Bus Device use pin B0 as the communication pin. It is more than just protocol, however. Pin B0 is an 'interrupt' pin, which means that a change of voltage level will cause the microcontroller to pause the process that it is currently doing and execute some other functions. For instance, when a Cricket sends a command to a Bus-Device, the microcontroller on the Bus-Device pauses what it is doing to listen to the Cricket. It then either carries out a different task or continues on doing what it was doing before, depending on what the Cricket told it to do. This interrupt ability is a special trait of B0. Note that if you set B0 as an output, it can no longer listen to the Cricket and the microcontroller will no longer respond to the Cricket. You must then turn off the Cricket to reset pin B0's state, as it needs to be an input. Try it…I dare you. **3&4.** I have reserved RC4 and RC5 for serial communication. Normally, these pins are fully functional; however, the code residing on the PIC right now reserves these for the RS-232 exercise later on. These pins may not function properly for normal I/O as a result and should be avoided.

Activity Two: Relays!

Try the below circuit to turn on and off a motor with a *relay*. A relay is a switch turned on and off by a voltage applied to the *coil* of the relay. Basically, current flowing through the coil produces a magnetic field that closes two pieces of metal together, thus completing a separate circuit. This separate circuit contains the load to be switched, in our case a motor, and a power supply. The battery in the diagram can be any direct current source, the motor can be any load that falls within the specifications of the relay used. Remember, a PIC pin can only source 20mA, so choose you relay to fit within that specification. Also, make sure that the motor and power supply are compatible. The diagram below may be a bit confusing. Lets divide it into two parts: The first is the coil, the little loop-dee-loop thing. Find this part on the relay spec. sheet. Wire one side to the PIC, and one to ground. When you run the program below, you should here the 'click' sound of the switch flipping over.

```
to turn-on_and_off
    bit_clear $A2
    pin_set  $A2
    wait 20
    pin_clear $A2
end
```

Now, lets look at the other part of the schematic, which is the switch. Your relay has on pin that is the *lead*, one pin that is *normally open*, and may have one pin that is *normally closed*. Without current through the coil, the *lead* is connected to the *normally closed*. With current flowing through the coil, the *lead* is connected to the *normally open*. With schematic in-hand and mulitmeter fired up, verify this. Then wire up the rest of the circuit as shown below. The pins shown in the diagram are the lead (with the little arrow) and the normally open.



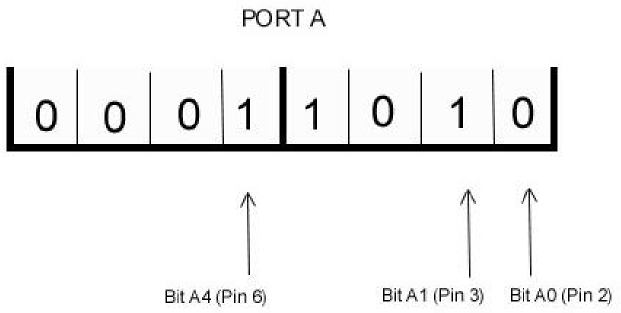
The diode that bridges the motor is to protect the components from harmful voltage that builds up when the motor switches off. This is because motors have a high *inductance*.

Relays are just a switch, making them useless if you want to control the speed of a motor or dim a light. However, the are very useful at switching AC loads or very large DC loads. We will control the speed of a motor in activity 4.

Activity Three: Another way to control pins!

For those new to binary numbers, please read the appendix regarding that topic first.

Imagine an array that looks like the following:



This array contains the I/O state of the pins on Port A. The ones represent inputs, zeros represent outputs. In the array above, A0, A2, and A6 are inputs, while A1, A3, and A4 are outputs[4]. This array is contained within the PIC and written to when setting up I/O pins on the PIC. It is possible to write to this *register* directly. We use the function *set_tris_a* to do this, giving us a big boost in speed as we only need one function to set up the eight pins, instead of setting each up individually. To do this, take the binary representation, starting with the high pin number on the right and low and the left, and convert this number to decimal or hex. The converted number is then used in the *set_tris* function. For example, 00011010 is 26 in decimal or $1A in hex. This step would not be necessary if we could send binary numbers with the Cricket. However, we are limited to sending numbers in decimal format (no pre-fix) or hexadecimal ($ as a prefix), so we much convert our port configuration from binary to decimal or hexadecimal.

We set-up Port A in the configuration above by simply writing *set_tris_a 52*. Another very similar register exists (it looks just like the diagram above) for controlling the output state of the pins (either 0V or 5V). To write to this register directly, we use the function *output_a*. This function works the same way as *set_tris_x*, except that instead of setting up inputs or outputs, we are setting the output state of the pins. We will use both of these functions to setup and turn on certain pins of the PIC, just as we did in the first LED activity. First, lets make all pins on C outputs. Then we turn the even ones on and the odd ones off:

```
to even-odd
  set_tris_c 0       ;set up all pins as outputs
```

[4] [4] You may have notice that the array has 8 slots, but Port A only has 6 pins. The highest two slots (bits) are unused in this port and do not effect its state. There are eight slots because 8 bits (slots) equals 1 byte, a common unit used in computer memories.

```
                       ;(111111111 in binary, $FF in hex)
  output_c   85        ;turn on even pin numbers, odds off
                         ;( 01010101 bin, $55  hex)
end
```

Hook up the bi-color LED, through a resistor between pins A0 and A1. (That is, run a wire from A0 to the bi-color LED, then a resistor from the other pin on the LED to A1.) Then try this program:

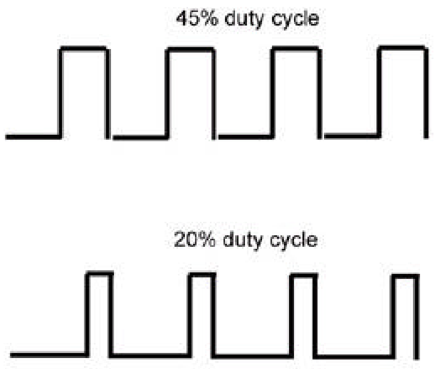
```
to tris_example
    set_tris_a 0
    output_a 1
    wait 10
    output_a 2
    wait 10
     loop [ output_a 1 output_a 2]
end
```

This concept takes a while to fully wrap your head around. It is a trick, mostly. It takes advantage of a way of representing numbers and uses that representation for a completely different reason, like setting up pins. There is no reason that tricks like that should make sense- you just accept them at first and then appreciate them later.

Activity Four: Motors and MOSFETS!

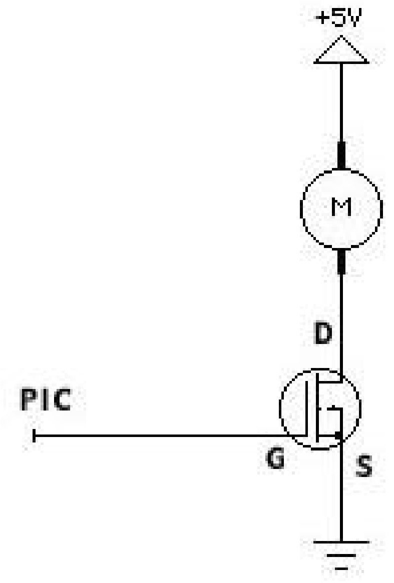
In this activity, we will use one of two available Pulse Width Modulation (PWM) modules to power and control a DC motor. Later, we will also use this module to produce a continuously variable DC analog signal (from 0-5V).

Pulse Width Modulation refers to a continuous series of square pulses produced by a pin on the PIC. The width of the pulses is variable, as illustrated below. The *duty cycle* of a PWM signal is the amount of time the pin is held high in each cycle. For instance, a duty cycle of 50% means the pin is low for the same amount of time as it is high, while a 10% duty cycle means that the pin is high for 10% of the time. A PWM signal also has a frequency, which is the number of high-low cycles per second. This frequency is not important in our examples. A PWM signal fed into a motor effectively turns the motor on and off. However, this is happening very fast when compared to how long the spindle of the motor takes to slow down. In the end, the motor 'averages' the highs and lows of the PWM signal, and its speed is proportional.



DC motors of reasonable size (big enough to do something useful with) are power-hungry devices. They draw anywhere from 100 mA to 10 A or more. A pin on our PIC can only *source* 20 mA. While the relays used in Exercise 1.5 can turn a motor of this type on, the mechanical switching action of the relay is not fast enough to keep up with the high-speed PWM signal. For this exercise, we will use a high-current MOSFET (metal oxide field effect transistor, not that it matters) that acts just like a relay, but can switch faster since it does not use moving parts to switch. The amount of current that can run through a MOSFET depends on the MOSFET, but we will use .4 A devices. Note that the pins of the MOSFET are labeled as D,G,S (in the spec. sheet). These stand for Drain, Gate, and Source, respectively. The Drain is the *drain of electrons*, or where the electrons flow to, which is the positive voltage supply. The source is the source of electrons, which is the ground. This is confusing! Current flows from positive voltage to ground, right? Yes, but only because current was defined wrong by Benjamin Franklin way back when. Current does flow from higher voltage to lower voltage (by definition), but what is actually flowing are electrons (from lower voltage (or ground) to higher voltage). To avoid confusion, always think of current flowing from high to low, except when reading spec. sheets and seeing the terms *drain* (or *Vdd*, which is the drain voltage) and *source* (*Vss*, source voltage). Don't blame Ben. He knew something was flowing and guessed wrong. He did bring us bifocals and chimneys, after all.

The PWM module on the PIC16F876 relies on an internal timer, specifically Timer2. The first step in using the PWM module is setting up this timer. We use to the function *setup_timer2* with a 1,2 or 3 passed as an argument. Passing a 1 will setup the timer near its fastest rate and 3 slowest, with 2 somewhere in between. Verify this with an oscilloscope once the PWM module is running. The next step is turning the PWM module on. The PIC16F876 had two modules, PWM1 on RC2 (pin 13) and PWM 2 on RC1 (pin 12). The functions to call are *setup_pwm1* or *setup_pwm2*. Then, we set the duty cycle of the module. The function is *set_pwm1_duty* or *set_pwm2_duty* followed by an argument 0-255. One would imagine that the duty cycle should be 0-100. I would think that too. Microchip doesn't. Use 0-255. The circuit is sketched below:



```
to motor_program
  setup_pwm2
  setup_timer2 1
  set_pwm2_duty 10
  wait 10
  set_pwm2_duty 50
  wait 10
  set_pwm2_duty 150
  wait 10
  set_pwm2_duty 200
end
```

After successfully driving the motor, remove it and the MOSFET. Place a 100 uF capacitor (mind the polarity) in its place, with the positive side of the cap on the PWM line and the negative side to ground. Write a CricketLogo program to sweep the duty cycle from 0 to 255 repeatedly. While running this program, measure the voltage level of the positive side of the capacitor[5] with a voltmeter or oscilloscope. Analog out! Neat trick, although it turns out it isn't necessarily useful in most circuit designs. Some interesting things *might* be possible, as there are voltage-controlled amplifiers and filters used in musical synthesizers that might be interfaced to a PIC using this technique. One more PWM trick: Replace the capacitor with an LED (wire a resistor and LED from the PWM pin to ground). Change your CricketLogo program to ramp from 0% duty to 30% and back down again. You should see a nice 'hearbeat' from the LED.

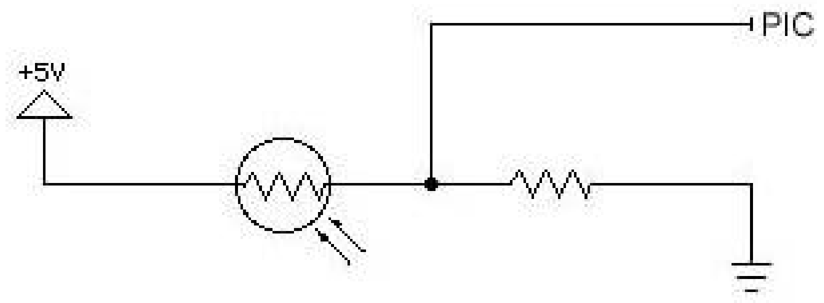
[5] The value of the capacitor here should be played with. Too small of a capacitor and you will still be able to see individual spikes of the PWM signal with an oscilloscope. Too big, and it will take a mighty long time to discharge itself. You can also place a resistor in series from the PIC to the capacitor and see what effect it has. Watch the difference on an oscilloscope.

Activity Five: Analog to Digital (A/D) Conversion:

Digital systems have many advantages and have led to many great advances. However, the sensors used to sense or detect an aspect of the physical world measure an analog signal, and we are forced to convert an analog signal to a digital one for it to be useful. Fortunately, this is largely done for us in the PIC. It contains 5 pins (A0, A1, A2, A3, A5) that can be configured to perform direct AD conversion on a signal between 0V and 5V. The functions we will use are *adc_setup* and *adc_read*. The function *adc_setup* gets passed a number (0,1,2,3,5) for the pin of Port A to perform the conversion on. The function *adc_read* returns a 1-byte number (0-255) proportionally to the signal, with 0 representing 0V and 255 representing 5V.

```
to sample-adc
   adc_setup 0
   loop[ display adc_read]
end
```

Many sensors use environmental factors to change the sensor's resistance, such as thermistors (temperature effects resistance), photoresistors (light effects resistance), or force-sensitive resistors (FSRs). To create an analog voltage signal from these devices, we create a voltage divider with the sensor and an ordinary resistor, as shown below.



Lets analyze this circuit. The total resistance from 5V to Ground is the resistance of the sensor (R_s) plus the resistance of the resistor (R)[6], in our case we will use a 30-50 k resistor. The current flowing through both of these resistors is I = 5V / (R_s + R). Now, since current cannot flow into the PIC (it has very high resistance, or *impedance*), the current flowing through the sensor is equal to the current flowing through the resistor (where else would it go?). We use this fact to see that:

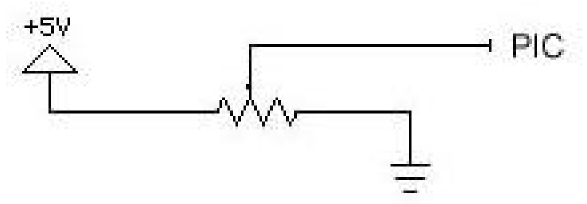
[6] How do you pick the value of this resistor? You should match this resistor to the change of resistance of your sensor in the conditions you are most interested in detecting. Just be certain that the total resistance at any time is bigger than a kiloOhm or more so that the current flow is small.

$(5V - V_{PIC}) = I * R_z$

Solving this for V_{PIC} gives us $V_{PIC} = 5V - 5V*R_s/(R_s + R)$

Note that the voltage at the PIC (V_{PIC}) is proportional to the resistance of the photoresistor (R_s). Just what we wanted! We have our analog voltage signal for the PIC to measure!

There are devices known as variable resistors that have a knob or slider that is used to vary the resistance of the device. Many of these devices are set up in a voltage divider configuration for easy AD conversion. These devices are known as a potentiometer. We use one below with the same AD functions.

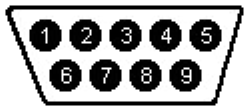


The center pin of a potentiometer is usually the *wiper*, or the point of variable resistance. This pin will be connected to the PIC, and the outer two pins to 5V and ground.

Exercise Six: Serial I/O

(Note: this exercise requires the installation of the Proce55ing program included on the CD)

Serial communication is a method of communicating between electronic devices. It refers to the passing of information bit-by-bit in time, in contrast to parallel communication, in which many bits are passed simultaneously. Many different protocols exist for this serial exchange. One very useful and ubiquitous protocol is known as RS-232 and it is used, amongst other things, for communication between computers and their peripherals (although this use is being rapidly replaces by USB protocol). The nice feature of RS-232 is that most of the work is done for us, either in the PIC itself of in the C compiler. We will be using RC4 and RC5 to send and receive.[7]



(Backside of the female D connector)

Setup the photoresistor circuit shown in Exercise Five. Connect pin two of the 9-pin D connector to RC4, pin three to RC5, and pin five to ground. Run the following program on the Cricket.

```
to send-data
adc_setup 0
 loop [
  send_serial (adc_read)         ;send the result of the
A/D conversion up the serial line
    ]
end
```

Once this is running, run Sketchbook/Standard/SimpleSerialDemo.pde within the Proce55ing environment and connect the serial cable to the computer. Proce55ing is a environment and libraries made specifically for graphics programming in Java. Once you have opened SimpleSerialDemo, click on the right arrow button to start. A small square should appear and change from black to white as the light level on the photoresistor changes. Serial data from the PIC might also be used for controlling graphics in Java,

[7] The PIC 876 has a 'hardware' serial port built in, called a USART (universal synchronous/asynchronous receiver/transmitter built in. However, this feature was meant to work with a *level converter*, which converts the 0-5V signal to –12 to 12V (most computers these days don't need these levels...0-5 works fine). These level converters also invert the signal, which means that the signal coming from the hardware USART is inverted to what the computer wants to see…thus we are forced to not use the hardware USART, unless we want to wire up a level converter. However, this only means slightly larger and slower code on the PIC, and nothing more.

video in Director, or sound in Max. Much information regarding this can be found on the Internet.

We can also receive serial on the PIC from the computer. There are two functions for this, *new_serial?*, which returns a one if there is new serial data and a zero if not, and *get_serial* which returns the new serial data and clears the *new_serial?* flag. An example might look like:

```
global [serial-data]

to serial-test
bit_clear $A1
  when[new_serial? = 1][
     setserial-data get_serial
     if (serial-data = 64)[pin_set $A1]
            if(serial-data = 65) [pin_clear $A1]
   ]
end
```

Moving to Other Microcontrollers

Take some time to build a few things with the BDTTW. Use these projects as excuses to try out some new circuits you have dug up, as well understanding further the functionality of a microcontroller. Dig for information regarding the various microcontrollers on the market, like the Basic Stamp, PIC, OOPIC, etc. Find the one that looks right for you. For those interested in using a Basic Stamp, see the web page cricket.media.mit.edu/basicstamp regarding the use of Bus Devices with the Basic Stamp. For those interested in the PIC, the BDTTW also serves as a development platform for programming and using the PICF876. The section below will take you through that process.

The BDTTW as PIC Development Platform

The first step in this direction is obtain the PICSTART Plus programmer, MPLAB and the CCS PICC compiler. The PICSTART Plus and MPLAB are available at http://www.microchip.com/. The CCS PICC compiler is available at http://www.ccsinfo.com/ccscorder.html.

Installing MPLAB and the CCS compiler and Using Activites1-6

1. After installing MPLAB on your computer and unzipping the CCS files into their own folder on your hard drive (put the unzipped folder on c:/, so that the path is simply c:/picc), open MPLAB. Choose Project/Install Language Tool. Now, choose CCS and the C-Compiler and then choose the executable PICC\CCSC.EXE (this is in the folder you unzipped, wherever you put it) You also want it Windowed and not Command Line.

2. Now we must tell the computer where to find the header files (the header files are the files specific to the type of PIC we are using that come included with the compiler). To do this (in Windows 2000), we click on the Start Menu and choose Settings/Control Panel. In Control Panel, choose System/Advanced/Environment Variables. Edit the System Variables section labeled Path and add the line c:/picc/EXAMPLES (yes, case matters). Do not erase or change any of the other variables. Now, restart your computer.

3.After copying the folder picc_examples (available on the BDTTW website) to you hard drive, select Project, New. Now find the picc_example folder, and create a new project with the name led.pjt in that folder (the project name should be the same as the source file, in this case ‘led’ (the source file is ‘led.c’). Hit OK. The Edit Project window should pop up. The Target File name should be led.hex. You can leave the other boxes within the Project heading blank. If the Development Mode box does not read Editor Only16F876, click on the Change button. Click the check box labled None (Editor

Only), and find the PIC16F876 in the Processor field. Hit OK. Now, change the Language Tool Suite to read CCS. A box will pop up to warn you that you will lose your 'target command line options'. That's quite fine, so hit OK. Now, highlight the temp [.hex] file in the Project File box. The Node Properties button should now be available. Click on this. Click on the box labeled PCM. It should now be checked. Hit OK. Now, one last step. Click Add Node. Find the file led.c and add it. Hit OK, then back in the Edit Project dialog box, hit OK.

4. Now, we need to enable the PICSTART programming board. Click on the Menu item PICSTART Plus. With the programming board plugged into your serial port, click on Enable PICSTART PLUS.

5. To compile the file led.c, we choose Project/Build All. The program has compiled successfully if the resulting window reads 'Build Completed Successfully'. If there are errors in our code, the window will contain the line number of our error (or perhaps the line before it) and some cryptic error message. The reference for the CCS compiler will explain these error messages a bit better, but not great. Also, since the compilers only good clue to what is wrong with your program is the line number, we must use a text editor that includes the line numbers. I use the EditPlus text editor, available as shareware. Note: After we change anything to the source file, led.c, we must save it and compile it again.

6. After placing the PIC to program in the programmer (depressed dot on the PIC should be at the top of the programmer), we hit Program. The yellow light on the programmer will light up, and a dialog box should read 'success' after the programming is finished. You are ready to rock and roll. Place the PIC back in the BDTTW and power it up (the Cricket powers the BDTTW, so just turn on the Cricket)

7. Now, wire up the LED just as in Activity One. You should see the familiar blink. Note: The Cricket is just powering the BDTTW. We could easily make our own cable to power the BDDTW without the Cricket, but the Cricket is handy so what the heck. Note that in Activities 1-6, the Cricket was running the show, telling the BDTTW what to do. Now the tables have turned, and the PIC is the star.

8. Go through all of the example PIC code provided (Activities 1-6). These programs should function similarly to the CricketLogo versions in the original activities. All of the wiring and circuitry is the same as the original activities. Pay close attention to the *adc.c* file, as in this example we run the Display Bus Device right from the PIC!

There are many web pages and discussion groups devoted to PIC's that you should utilize as you progress. Again, pick a project to focus your efforts and to make all of the frustration worthwhile. Good luck!

Building a Cricket Bus Device

For those interested in building their own bus device, open the *lib.txt* file in the Support folder of the BDTTW Edition of Jackal. Look at the various functions that the BDTTW uses, like *pin_set* or *output_a*. You will notice the function *bsend*. This function sends a number over the bus line. The first number it sends is $199[8]. This is the address of the BDTTW. Upon receiving this number, the BDTTW knows the Cricket is talking to it and then listens for more numbers. The next numbers it receives are instructions to do something else, like make a pin an input or set a pin high. This number-passing scheme is how all bus-devices work. Now open up the file BDTTW.c with a text editor such as EditPlus. Find the functions with names similar to the CricketLogo functions (they near the very bottom). These functions are what get executed with the similarly named CricketLogo function is run. Notice that there are a few extra steps in some cases, but are very similar none-the-less. Use this file as a guide to how to structure your bus device code.

[8] $199 is bigger than one byte! Ok, there is some slight of hand going on here. The Cricket actually sends 9 bits, not eight, with the extra bit signifying whether the other eight bits should be treated as instructions or data. In this case, the Cricket sends $99 as an instruction. If we had written *bsend $99*, it would send a $99 as data.

Appendix:

Binary and Hexadecimal numbers:

We use the decimal representation of numbers in ordinary math. However, computers operate with a binary representation. At some point, people realized that a hexadecimal representation was a good compromise between the two, allowing humans to better visualize numbers as the computer might see them, but not make them stare at ones and zeros either. We will briefly cover all three and how to convert between them.

In the decimal system, we count from 0 to 9, then add a place holder in the next space over to represent 10. We then continue counting from 0 to 9 again. In binary, we count from 0 to 1, placing a 1 in the next space over when we hit one in the first. In hexadecimal, we count from 0 to 15 before marking the next space. However, the number 15 takes two spaces! To solve this, we start counting in with letters after nine..ie, count from zero to F, going through 9 to A, then B, and so on.

Decimal: 0,1,2,3,4,5,6,7,8,9
Binary: 0,1
Hexadecimal: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Binary is denoted by a lower-case b, hexadecimal by a '$' or '0x' prefix or a 'h' postfix.

To convert, its best to use calculator! In time, you will be able to convert back and forth in your head. For now, try out Marin Steen's Hex Calculator at http://www.martin-steen.de/hexcalc.html or use a scientific calculator.

Pinout for 9-pin D connector

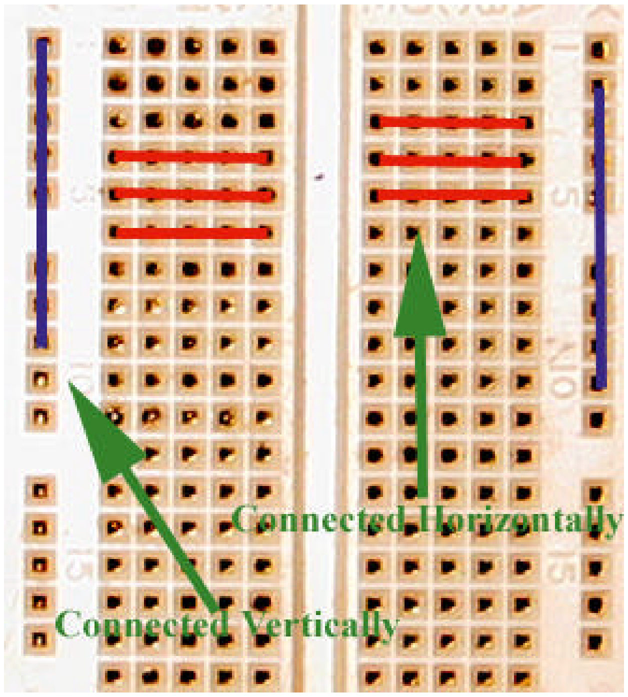
Pin No.	Function	Pin No.	Function
1	DCD (Data Carrier Detect)	6	DSR (Data Set Ready)
2	RX (Receive Data)	7	RTS (Request To Send)
3	TX (Transmit Data)	8	CTS (Clear To Send)
4	DTR (Data Terminal Ready)	9	RI (Ring Indicator)
5	GND (Signal Ground)		

Multimeters

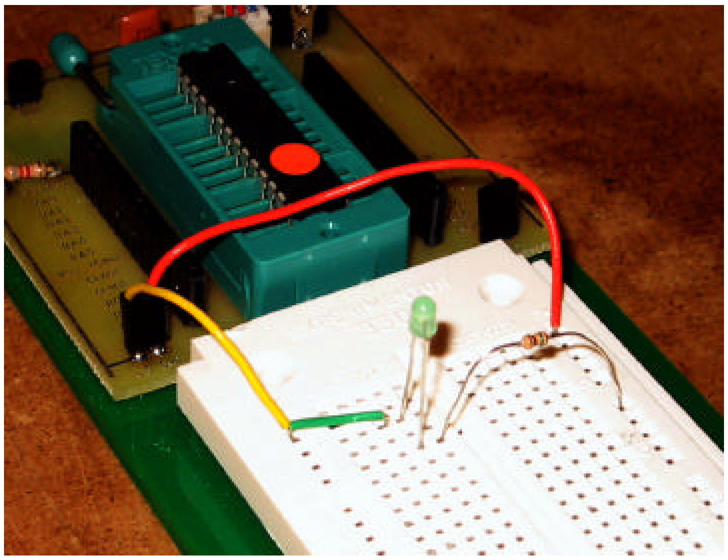
Mulitmeters are many measuring instruments packaged in one device. These instruments are usually a ohmmeter (measures resistance), a voltmeter (measures voltage), an ampmeter (measures current), and sometimes a faradmeter (measures capacitance). Also, some have a continuity tester, which beeps when a connection exists between the leads. This is a handy feature. The leads of the mulimeter may or may not have different plugs for the different functions. This is important to remember. Also, the ampmeter will certainly have a maximum input current it can accept. Do not ignore this. Also, there might be a switch for AC/DC coupling. Use DC for all purposes in this manual. For probing a switch, use the continuity tester (if possible) or ohmmeter on lowest setting. For measuring voltage, connect the black lead to ground and the red lead to the voltage being measured. For measuring amperage, connect the device in series with the circuit, so that current is flowing in the red lead and out the black. For measuring capacitors, make sure that you discharge them by touching their leads together before inserting them into the faradmeter.

Using Breadboards and Building Cicuits

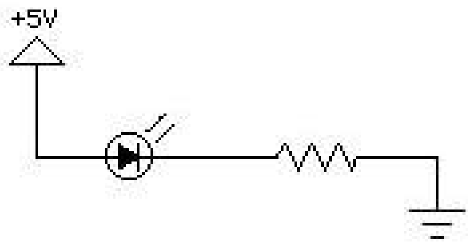
A breadboard (or solderless breadboard) is a device that allows for rapid creation of electronic circuits. Notice that there are two different types of hole patterns. The holes in the center area of the board are electrically connect horizontally. The holes the run down the edges of the board are connected vertically, as shown in the picture below.



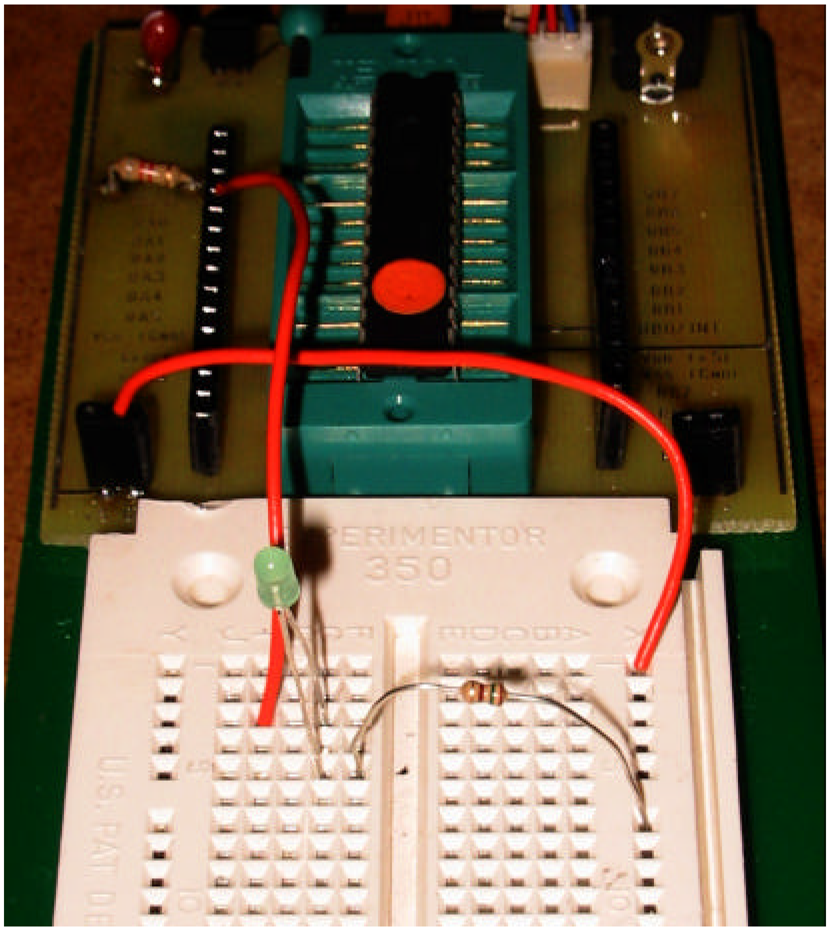
The outer edge holes (sometimes called rails) are usually used for 5V and ground (0V). By plugging in a wire to the 5V header and plugging in the other end to the left hand rail, we can now connect 5V into any circuit by utilizing the fact that all of its holes are connected vertically. Do a similar thing for ground. Now, plug a wire into the left hand rail (5V), and insert the other end into an arbitrary hole in the center area. Since these holes are connected horizonatally, we will now plug the long end of an LED (the anode, or positive voltage side) into any hole in that row. This lead will now be at 5V. Insert the short end of the LED into another arbitrary hole (anyone except one in the same horizontal row as the first!) Now plug one end of a 100 Ohm-1kOhm resistor into a hole in the same horizontal row as the short end of the LED. The other end of this resistor should now be plugged into the ground rail. Your circuit should look like this:



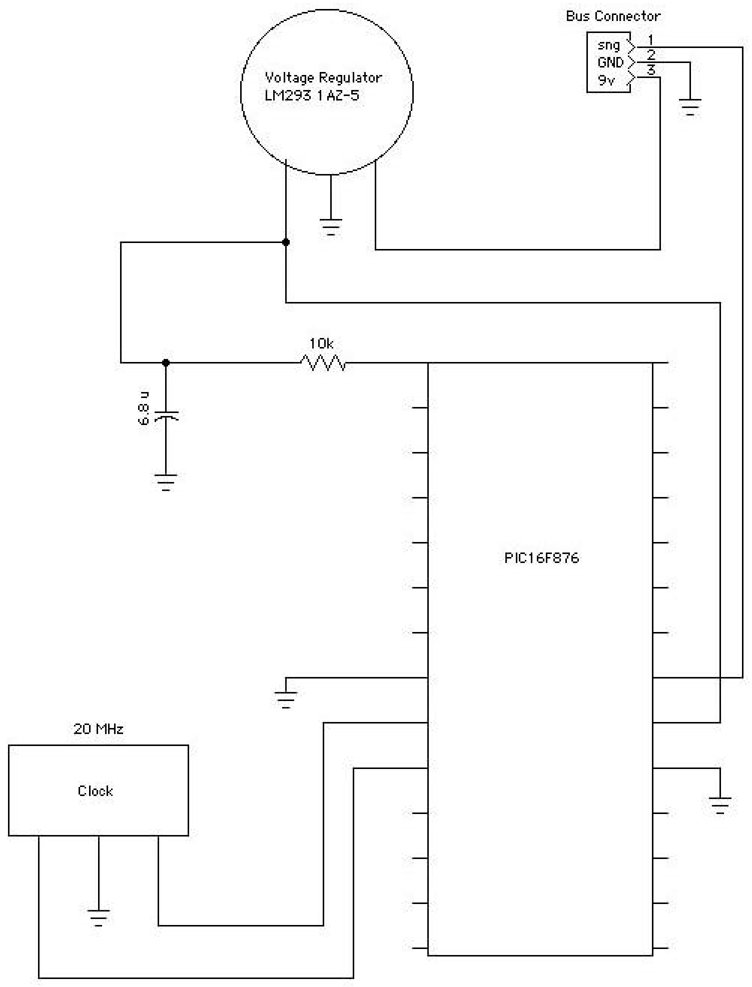
The schematic of this circuit looks like:



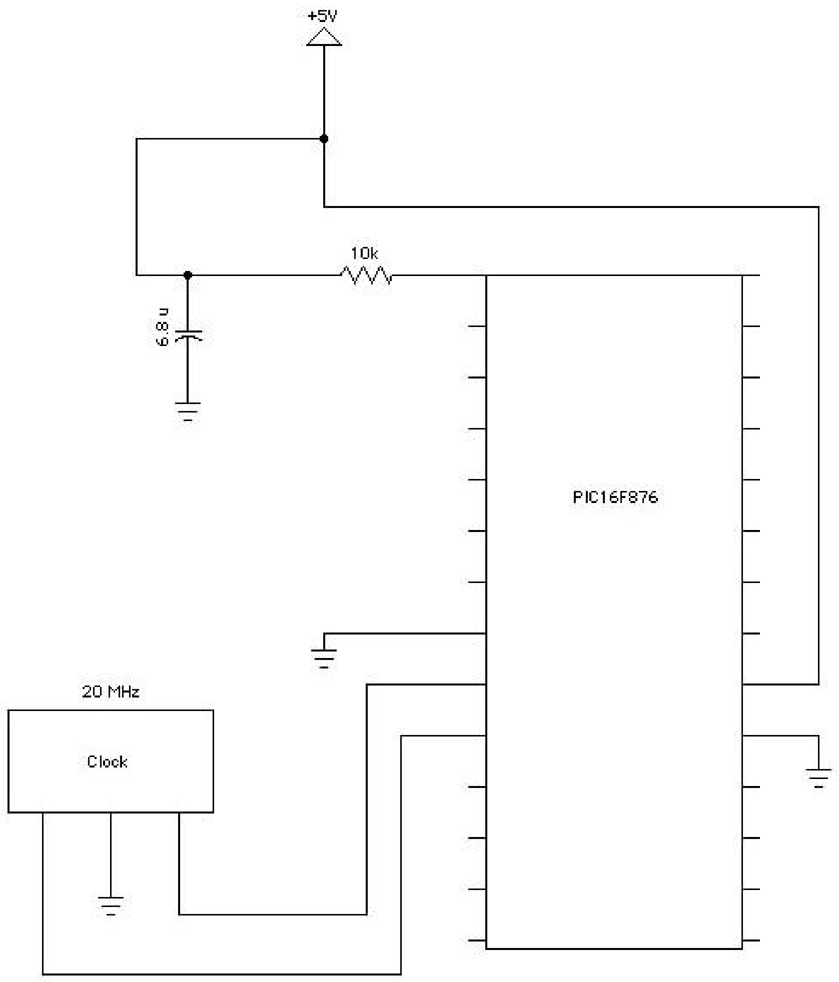
The schematic pictured in Activity One looks like this on the protoboard:



Schematic of BDTTW



Simplest Circuit Needed to Run the PIC16F876



Button Schematic