


















- **Getting Started**
-  **Handling the Components**
-  **Building a Circuit on a Breadboard**
- **Introductory Projects**
-  **Introduction**
-  **1. Lighting an LED**
-  **2. Using a Button**
-  **3. Making Sounds**
-  **4. Sensing Light**
-  **5. Controlling the LCD**
- **Advanced Projects**
-  **6. Digital Stopwatch**
-  **7. Digital Thermometer**
-  **8. Music Synthesizer**
-  **9. Memory Game** (Skip)
-  **10. Text Messenger** (Skip)
- **Robotics Projects**
-  **11. Basic Robot**
-  **12. Sensor Robot**
-  **13. Remote Control**
-  **14. DC Motors**

## Breadboard Electronics Components

Although the components used in breadboard electronics projects may appear fragile, most are surprisingly sturdy. These components will provide many years of service, provided you follow some simple safe handling procedures.



Figure 1. Breadboard Microcontroller Starter Kit components.

## Keeping Batteries Charged

Machine Science's breadboard-based projects run on rechargeable AA batteries, which are included in the project kits. Since the batteries must be charged before you begin your projects, it is a good idea to charge the batteries immediately upon receiving your kit. Each set should be charged for at least 2 to 3 hours. Battery chargers, like the one shown in Figure 2, are widely available in retail stores as well as from Machine Science.

**NOTE: Never use regular alkaline batteries for breadboard-based projects, as their voltage differs from the rechargeables.**



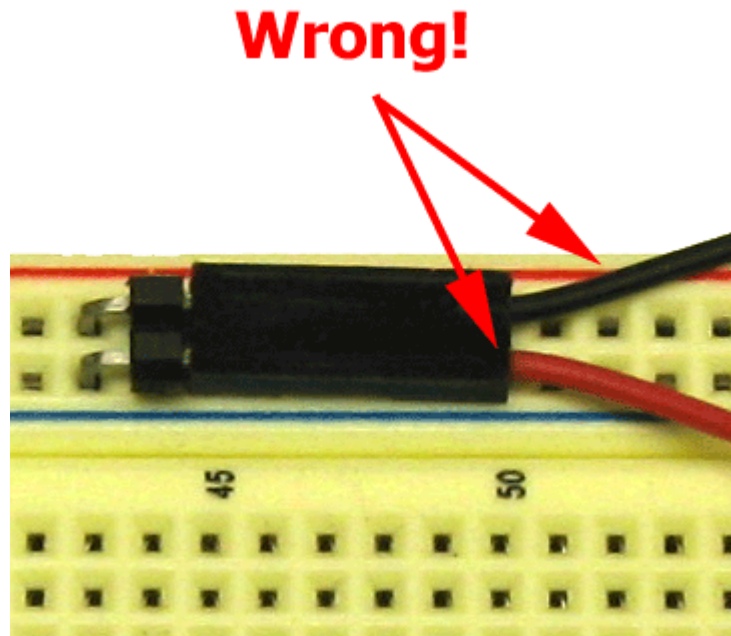
**Figure 2. Battery charger.**

Usually, the batteries will last for several weeks on a single charge. When batteries get weak, projects may not work properly. Therefore, it is a good idea to charge batteries immediately upon receiving your breadboard kits and to recharge them on a regular basis.

### **Avoiding Short Circuits**

The number one threat to electronic components is a *short circuit*--an arrangement of wires or components that creates a direct connection between power and ground. In a short circuit, electricity flows freely through one or more of the components, causing their internal circuitry to heat up. If the short circuit lasts for more than a few seconds, the components can get very hot, and they may be permanently damaged.

If you notice any components getting warm to the touch or detect a burning smell, ***immediately turn off the battery pack***, or disconnect it from the breadboard entirely. (It's a good idea to turn off the battery pack whenever anything seems amiss on the board.)



**Figure 3. Reversing the battery leads is a common cause of short circuits.**

The best way to avoid short circuits is to pay careful attention to directions when inserting jump wires and components into the board. Particularly close attention should be paid when connecting or reconnecting any of the following components:

- The **battery pack**. Be sure that red lead goes to power, and the black lead goes to ground. This is probably the leading cause of short circuits.
- The **microcontroller**. Make sure pins 11 and 32 go to power, and pins 12 and 31 go to ground. Reversing these connections can cause permanent damage to the microcontroller.
- The **LCD**. Make sure that the LCD's 10 pins are properly aligned on the XBoard. Since these are a little hard to see, they occasionally become misaligned when the LCD is removed and replaced.
- **Preventing Pin Damage**
  - After short circuits, the next biggest threat to the project components is *pin damage*--the bending or breaking of the pins that connect components to the board. The microcontroller is particularly vulnerable to pin damage, especially when it is being inserted or removed. If the pins become badly bent, it will be impossible to properly connect the microcontroller, and it will have to be replaced.
  - The microcontroller will need to be removed only in rare instances, but extreme care should be taken whenever removal is necessary. Figure 4



shows one safe way to remove a microcontroller from a breadboard. Place the tip of a flat-head screwdriver under one end of the microcontroller, and gently pry the chip away from the board. Alternate between the right and left side of the microcontroller, until the chip is free of the board. When inserting the microcontroller, or any other component, make sure that its pins are properly aligned before applying pressure to the device.

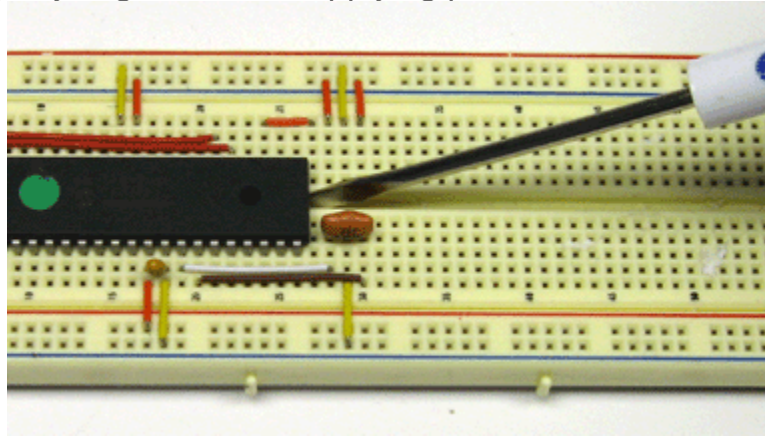


Figure 4. Safely removing the microcontroller from the breadboard.

- **Reducing Static Shocks**

- The microcontroller and the programming board are vulnerable to static electricity. A spark of static electricity can cause permanent damage to these components. To prevent this from happening, always touch a metal surface, such as a door knob or table leg, to discharge static electricity before handling the electronics projects.

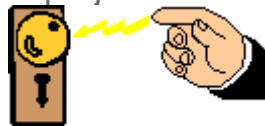


Figure 5. Discharging static electricity.

- The working environment can affect the threat of static electricity. In areas with low humidity and carpets on the floor, static electricity tends to build up rapidly. If your environment is particularly prone to static build-up, you may want to consider wearing an anti-static wrist band, which continuously discharges static electricity as you work on your projects.



- **Figure 6. Anti-static wrist band.**

## Building a Circuit on a Breadboard

Think about all the electronic devices you use every day--computers, cell phones, video games, digital music players (Figure 1). It's hard to imagine life without them, isn't it? If you're like most people, you don't know how these devices are made or how they work. Maybe you've seen the maze of electronic stuff inside a computer and thought: how could anyone make sense of all that? Well, the truth is that you don't have to be a genius to understand how electronic devices are made and how they work. Anyone can do it, and Machine Science's engaging hands-on projects will show you how.



**Figure 1. Common electronic devices.**

So, what exactly is all that stuff inside your computer? The simple answer is electronic circuits, which are arrangements of metal wires and electronic components (Figure 2). The wires carry electricity between the components, and the components do things such as store electricity and limit its flow. You don't need to understand all this yet. Just remember that electronic circuits are made up of electronic components connected by wires.



**Figure 2. Electronic circuits.**

## The Challenge: Build a Circuit



In this challenge, you will start building your own circuits, using a professional engineering tool called a breadboard. You will attach a battery pack to your breadboard, insert a few wires and components, and learn how to supply power to these components. You will also learn how to read schematic diagrams, which are the "blueprints" that engineers use to design circuits.

## The Challenge: Build a Circuit Collecting Your Components

The components for this activity are listed in the table below and shown in Figure 3.

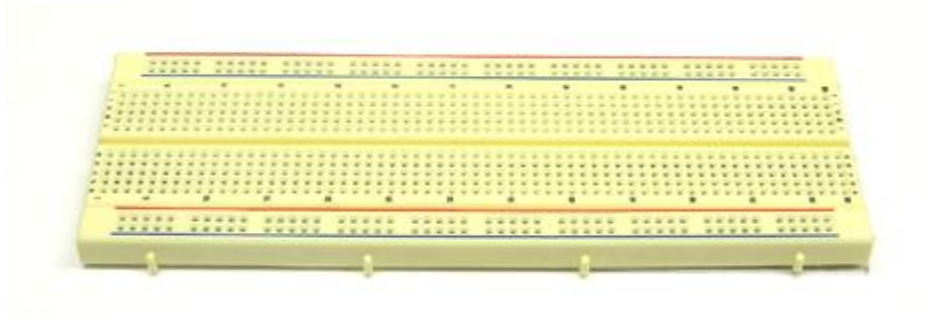
Part	Quantity	Description
A	1	Breadboard
B	1	Seven-segment display
C	2	Resistors (390 Ohm)
D	2	Jump wires (long red)
E	7	Flexible jump wires
F	1	Battery pack, with 4 AA rechargeable batteries
G	1	Bent connector (two prong)



Figure 3. Components for circuit-building activity.

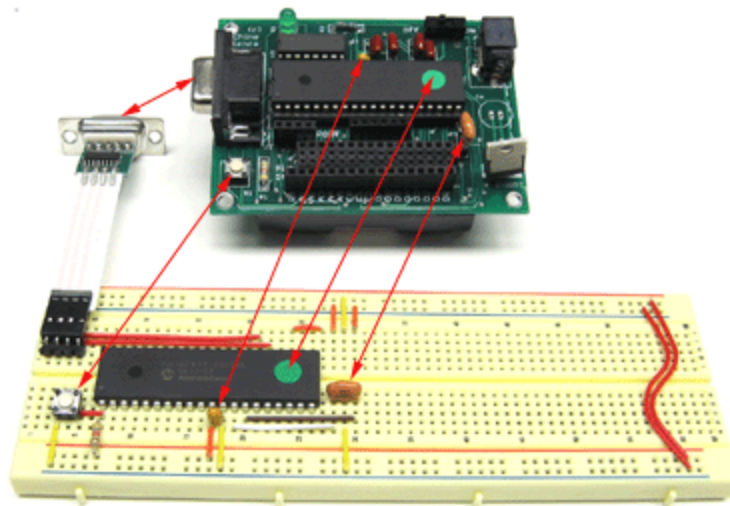
## The Challenge: Build a Circuit About the Breadboard

The circuits inside modern electronic devices are permanently attached to circuit boards, but they didn't start out that way. Every circuit in use today began its life as an experimental arrangement of wires and components. One tool that engineers use to create experimental electronic circuits is called a *breadboard* (Figure 4). Although the breadboard is a professional tool, it is very easy to use. In fact, you will create all of the circuits you need for your Machine Science projects by plugging wires and components into your breadboard.



**Figure 4. Breadboard.**

Figure 5 shows a circuit designed on a breadboard next to a circuit board with a very similar circuit. Note the correspondence between the components on the breadboard and the components on the circuit board. Remember: when you are using the breadboard, you are using a tool that engineers use to design real circuits!

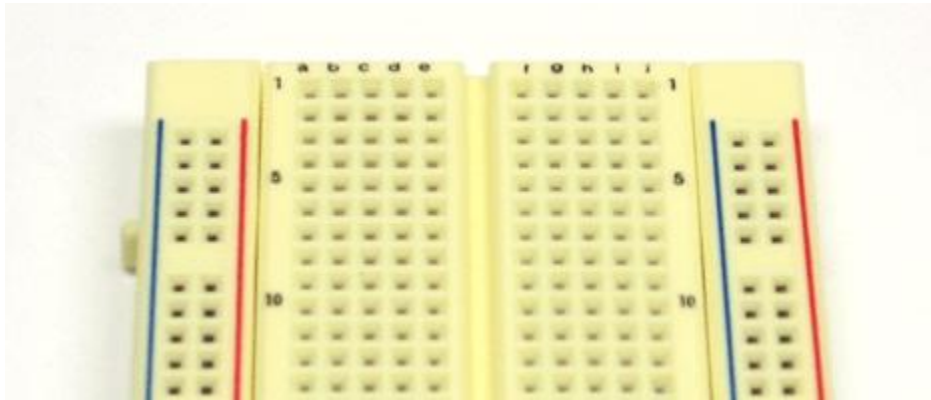


**Figure 5. Breadboard and circuit board.**



## The Challenge: Build a Circuit About Rows and Columns

The breadboard has hundreds of tiny holes that hold wires and electronic components. Figure 6 shows a close-up view of one end of the breadboard. Notice that each row of holes is assigned a number, and each column is assigned a letter. This way, the exact position of any hole on the breadboard can be specified by a combination of a letter and a number. For example, in Figure 6, the hole in the upper-left corner is hole A1.



**Figure 6. Close-up view of breadboard.**

Metal strips inside the breadboard make electrical connections between the wires and components that you plug into certain holes. Look closely at the long edges of the breadboard. On each side, there are two columns--one marked with a red line, and the other marked with a blue line. All of the red line holes on a side are connected to one another, and all of the blue line holes on a side are connected to one another.

Now, look at the breadboard's numbered rows of holes. Notice that each row has 10 holes--five in columns A to E, and five in columns F to J--separated down the middle by a shallow groove. Within each row, the five holes in columns A to E are connected, and the five holes in columns F through J are connected. However, the breadboard has no built-in connections across the central groove. The internal connections at one end of the breadboard are marked with green lines in Figure 7.

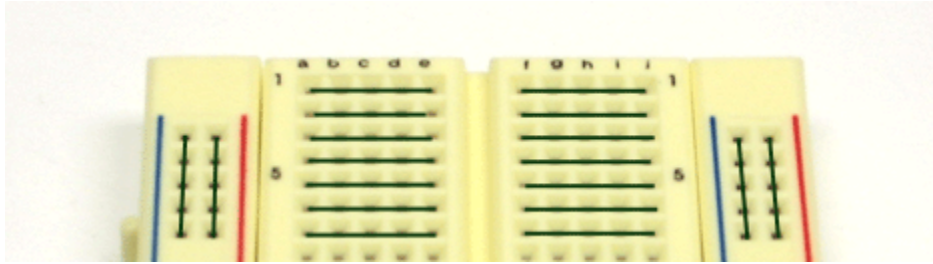


Figure 7. The breadboard's internal connections, highlighted in green.

## The Challenge: Build a Circuit

### Preparing the Breadboard

Before you can build a practice circuit, you need to secure a metal plate to the bottom of the board. The metal plate will help all of your components function more consistently. *NOTE: If your board already has a metal plate on the bottom, you can skip this step.*

1. Peel the backing from the breadboard, leaving the sticky foam layer in place, as shown in the video on in Figure 8.
2. Apply the metal plate to the bottom of the breadboard, as shown in the video and in Figure 8.

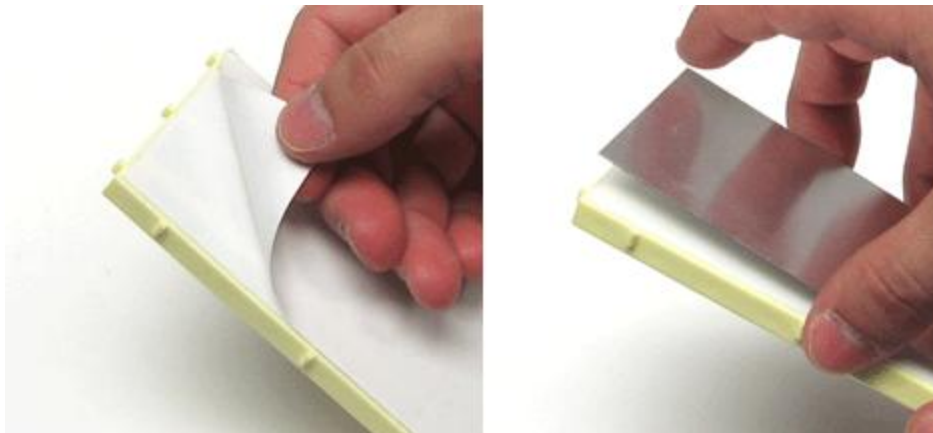


Figure 8. Peeling the backing from the breadboard and applying the metal plate.

## The Challenge: Build a Circuit

### About Power and Ground

The holes marked with red and blue lines along the edges of the breadboard are called *power* holes and *ground* holes, respectively. These holes have a special purpose: making it easy to supply electricity to the components in your circuit. No



matter where you put a component, there's always a power hole and a ground hole nearby. Think of the power and ground holes as a "power strip" for your breadboard.

In the next step, you will set up this "power strip" by connecting the battery pack to the power and ground holes, and using two jump wires to make connections across the board. The battery pack holds four rechargeable AA batteries. Each battery supplies about 1.3 volts, so together the four batteries provide just over 5 volts for your breadboard.

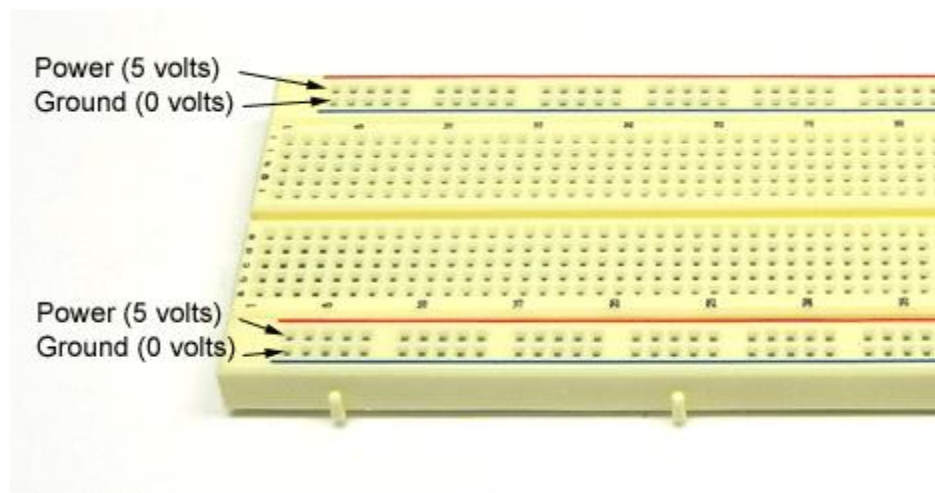


Figure 9. Power holes and ground holes.

### **The Challenge: Build a Circuit Connecting the Battery Pack**

The battery pack is connected to the breadboard by short lengths of red and black wire. The red wire plugs into one of the holes marked with a red line, and the black wire plugs into one of the holes marked with a blue line. Two long red jump wires carry these connections across the board.

- 1. Insert the batteries into the battery pack, as shown in the video to the right, taking care to align the positive and negative terminals as marked inside the pack. *NOTE: The batteries should be charged for several hours before their first use. Never use regular alkaline batteries for Machine Science projects.***
- 2. Make sure the power switch on your battery pack is in the OFF position.**
- 3. Using pliers or scissors, break off a two-segment piece of the 36-prong bent connector included in the Starter Kit, making a two-prong connector, as shown in Figure 10A. *NOTE: Be sure to keep the remaining segments for future projects.***

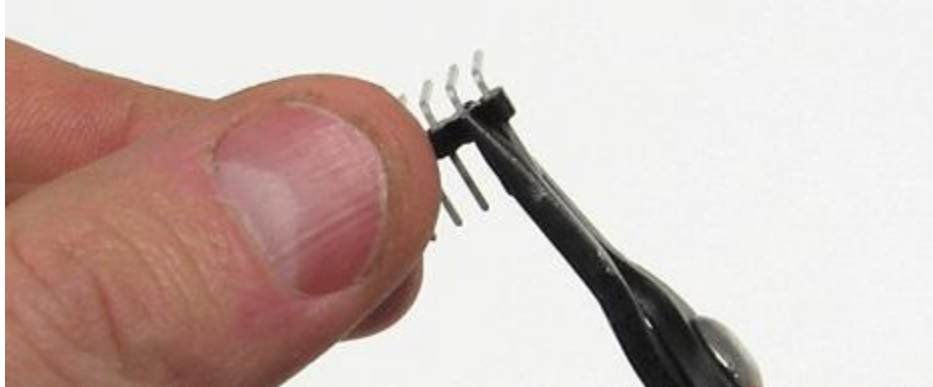


Figure 10A. Jump wires connecting power and ground from one side of the breadboard to the other.

4. Insert the two-prong connector in the holes next to hole J61, as shown in Figure 10B. **Align the red and black wires EXACTLY as shown.** *NOTE: If the red wire is on the wrong side, then the connector may be upside-down. Pull it off, turn it over, and reconnect it to the wires.*

5. Connect the red line holes and the blue line holes on each side of the board by inserting two long red jump wires, as shown in Figure 10B. *NOTE: You will need to bend or trim the wires to get them to fit.*

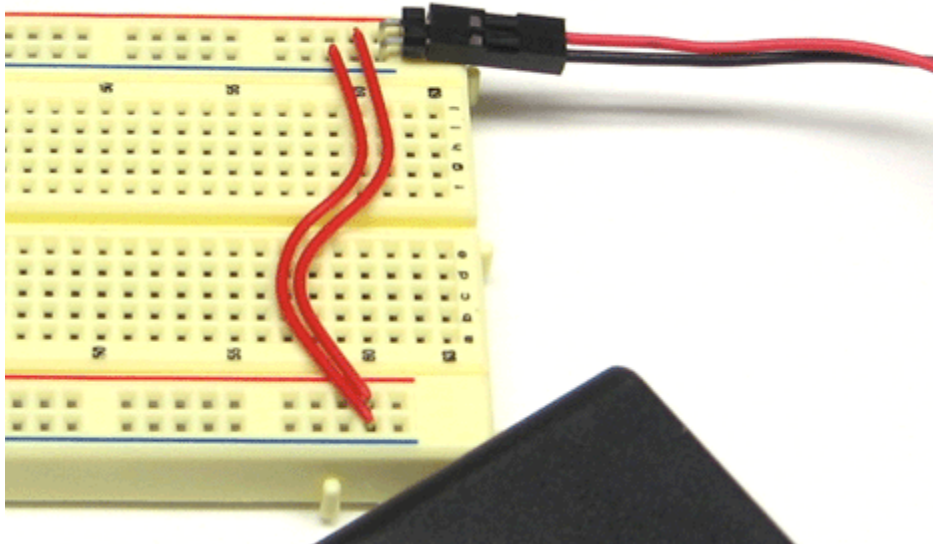


Figure 10B. Jump wires connecting power and ground from one side of the breadboard to the other.

### **The Challenge: Build a Circuit About the Seven-Segment Display**

In the next step, you will add a component to the breadboard called a seven-segment display. These components are widely used in the numerical displays on digital clocks, digital thermometers, and other home appliances, such as DVD players and microwave ovens. Their name derives from the number of individual light segments required to produce each number on the display. If you look closely at the devices shown in Figure 11, you can count the individual segments.



Figure 11. Devices that have seven-segment displays.

### **The Challenge: Build a Circuit About Pin Numbers**

Figure 12 shows the seven-segment display that you will use in this project. Note that the component has seven different segments and a decimal point on its top face. Each segment and the decimal point can be lit individually to produce different numbers on the display.

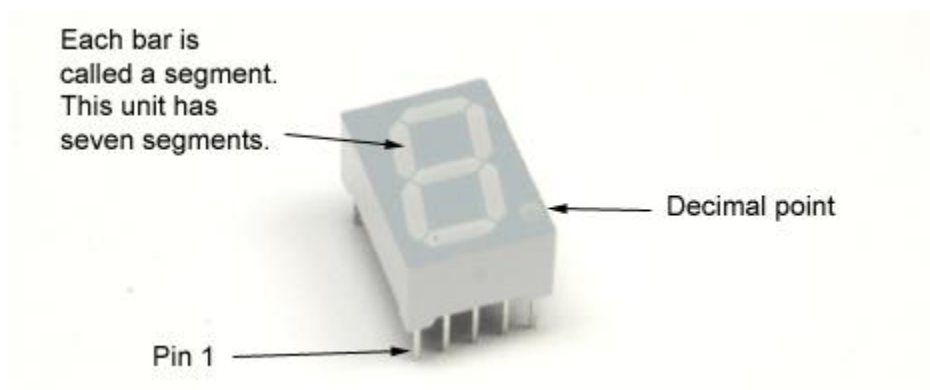


Figure 12. Seven-segment display.

Like many electronic components, the seven-segment display has pins on its underside that connect it with power, ground, or other components. On the seven-segment display, the pins are numbered from 1 to 10. With the display oriented so the decimal point is in the lower-right corner of the top face, pin 1 is under the lower-left corner. The remaining pins are numbered counterclockwise starting from pin 1, as shown in Figure 13.



Figure 13. Pin numbering on the seven-segment display.

### The Challenge: Build a Circuit Adding the Seven-Segment Display

1. Orient the seven-segment display so the decimal point is in the lower-right corner of the top face. In this orientation, pin 1 is under the lower-left corner.
2. Plug pin 1 in hole C27, as shown in Figure 14. The remaining pins along the lower edge of the display should go in holes C28, C29, C30, and C31. **NOTE: Be careful not to bend the pins as you insert the component into the breadboard.**

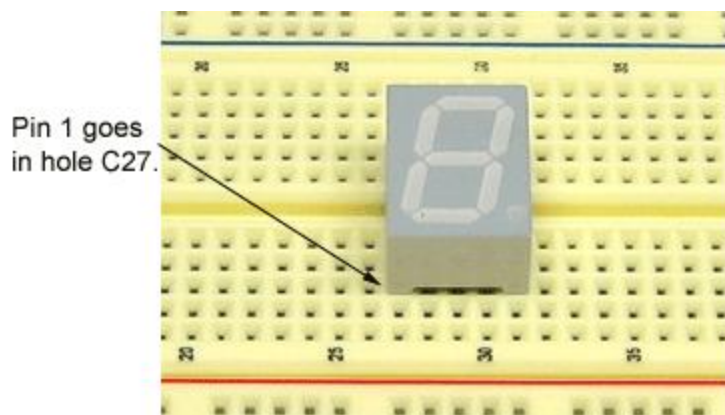


Figure 14. Seven-segment display on breadboard.

### The Challenge: Build a Circuit About Resistors

In the next step, you will connect the seven-segment display to ground, using a component called a *resistor*. A resistor limits the flow of electricity in an electronic

circuit. The resistors in your practice circuit will keep your seven-segment display from "burning out," like an old light bulb.

Resistors vary widely in terms of their *resistance*--i.e., how much electricity they allow through. You can determine a resistor's resistance by looking at the colored bands on its side. Resistance is measured in units called *Ohms*. To learn more about resistors, including how to determine a resistor's resistance, see the [Resistors Quick Reference](#) document. For this activity, you just need to make sure to use 390-Ohm resistors. Figure 15 shows two ways to identify this resistor.



Figure 15. Two ways to identify a 390-Ohm resistor.

### **The Challenge: Build a Circuit Connecting the Display to Ground**

Before you can light the display, you have to connect it to ground. Fortunately, this is easy to do, since you've already set up your power strip. Remember: all the holes marked with a blue line are ground holes.

1. Make sure the power switch on your battery pack is in the OFF position.
2. Insert a 390-Ohm resistor into the breadboard, connecting hole B29 to ground, as shown in the video and in Figure 16.
3. Insert a second 390-Ohm resistor into the breadboard, connecting hole H29 to ground.



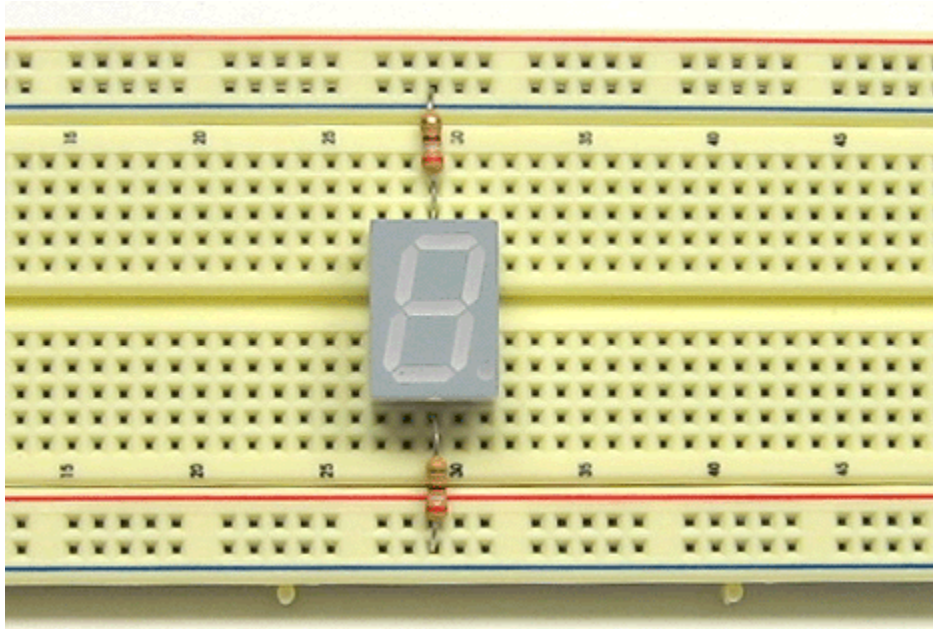


Figure 16. Resistors connecting seven-segment display to ground.

### **The Challenge: Build a Circuit Lighting One Segment**

1. Move the power switch on the battery pack to the ON position.
2. Insert one end of a flexible jump wire into any open power hole (any hole marked with a red line).
3. Insert the other end into any open hole aligned directly above or directly below a pin on the seven-segment display, as shown in the video and in Figure 17. **NOTE: Do not let the tip of the jump wire touch either resistor - this may cause a short circuit!**

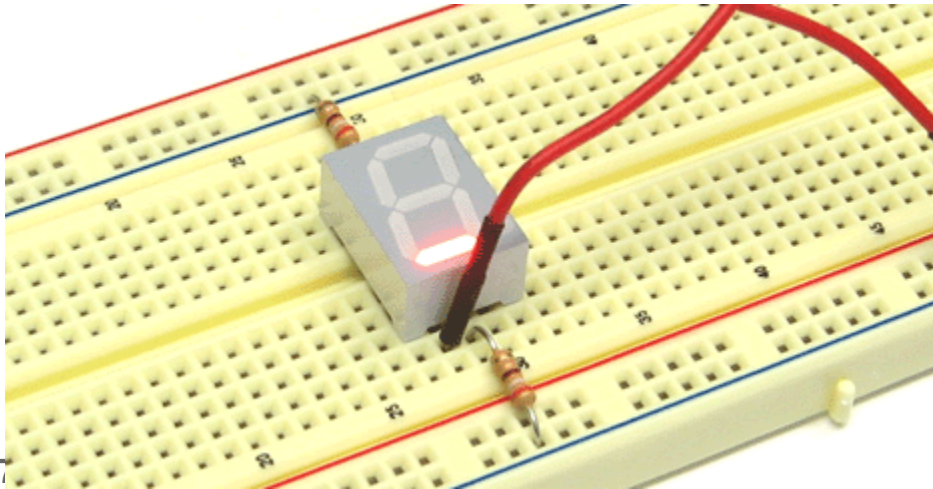


Figure 17

segment to light.

sing one

### The Challenge: Build a Circuit Making Numbers and Letters

1. Using a second flexible jump wire, light another segment of the display.  
**REMEMBER:** *Never insert the jump wire into a ground hole, and never touch any part of the resistor.*
2. Using additional jump wires, light combinations of segments to produce the numbers from 0 to 9 on the seven-segment display.
3. Arrange the jump wires to display one of your initials. **NOTE:** *Some letters cannot be made, and some can be produced only in their lowercase form.*

### The Challenge: Build a Circuit Reading Schematic Diagrams

In order to make a record of their circuits, engineers often draw "blueprints" of their circuits, called *schematic diagrams*, or just *schematics* for short. In a schematic, engineers use pictures to represent each component. Figure 18 shows some of the components you learned about in this unit.







			
Resistor	Seven-Segment Display	Power	Ground

Figure 18. Schematic representations of circuit components.

In Figure 19, these components are linked together in one diagram. Can you identify each element of the diagram?

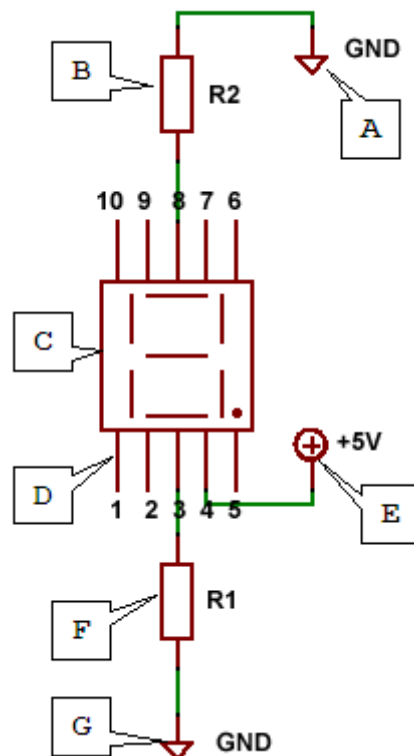


Figure 19. Seven-segment display circuit.

## The Challenge: Build a Circuit Cleaning Up the Breadboard

Congratulations! You've begun building electronic circuits, using the same tools professional engineers use to create modern electronic devices. Before moving on, do a little clean up.

**1. Move the switch on the battery pack to the OFF position. NOTE: Do NOT**

**disconnect the battery pack from the breadboard. You will need the battery pack in the next unit.**

**2. Remove the flexible jump wires from the breadboard. NOTE: Leave the two red wires connecting power and ground across the board.**

**3. Remove the two resistors from the breadboard.**

**4. Remove the seven-segment display from the breadboard.**

**5. Store the components in their original packaging.**

## Attaching the Arduino Uno to the Bread Board

The Machine **Arduino Board** is a configuration of wires and components on the breadboard that will allow you to rapidly build sophisticated electronics and robotics projects. The Arduino Board's key component is a ATMEGA microcontroller: a tiny computer on a chip. The microcontroller can be programmed with a set of instructions, called code, which determine how it behaves. The Arduino Uno will be the control center for many Machine Science projects, from digital stopwatches to remote-controlled robots! Figure 1 shows the completed Arduino Board.

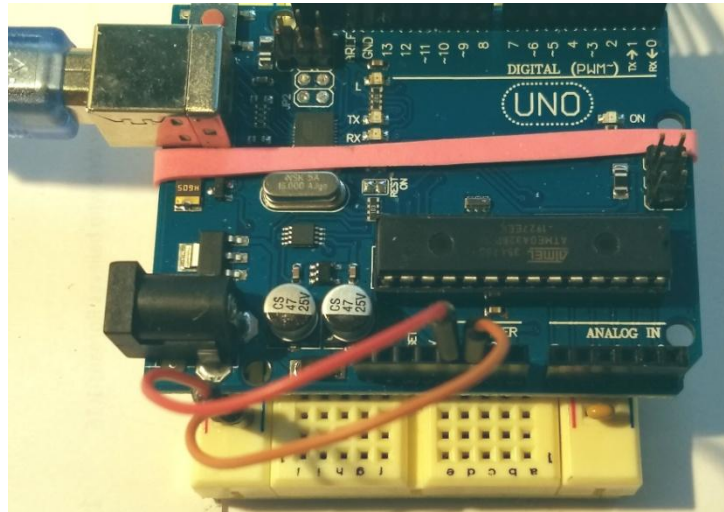


Figure 1. The Arduino Uno attached and wired.

### The Challenge: Attach the Arduino Uno



In this challenge, you will attach and wire an Arduino Uno to a breadboard. In the next unit, you will practice programming the Arduino, using an existing code file.

### The Challenge: Attach the Arduino Uno Collecting Your Components

The components needed to build the Arduino Board are listed in the table below and shown in Figure 2.

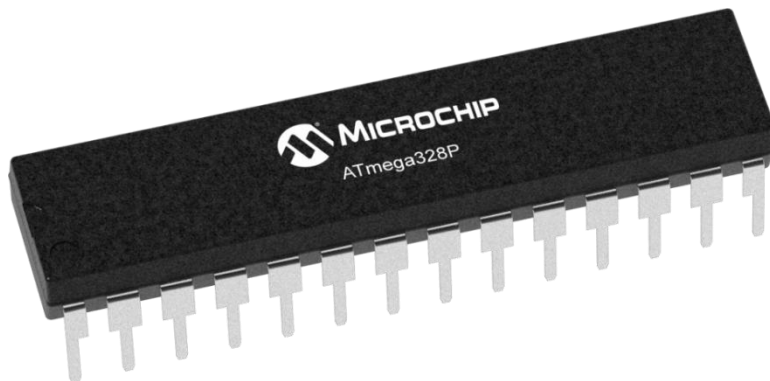
Part	Quantity	Description
A	1	Arduino Uno (ATMEGA 328P)
B	1	Broccoli Rubber Band

C	2	Flexible jump wires (2 long orange)
D	1	1 uf Capacitor

**Figure 2. XBoard components.**

### **The Challenge: Attach the Arduino Uno About the Microcontroller**

The *microcontroller* is the Arduino Uno's "brain," a tiny computer on a chip, which can be programmed to perform specific tasks. The Arduino's microcontroller, shown in Figure 3, is just like those found in many commercial devices. It has 28 pins, which plug directly into the Arduino board ports.



**Figure 3. The Arduino's microcontroller.**

In recent years, microcontrollers have become smaller, more powerful, and less expensive. As a result, they have been incorporated into more and more everyday products, from cell phones to home appliances and automobiles. Figure 4 shows some of the systems in modern automobiles that rely on microcontrollers.

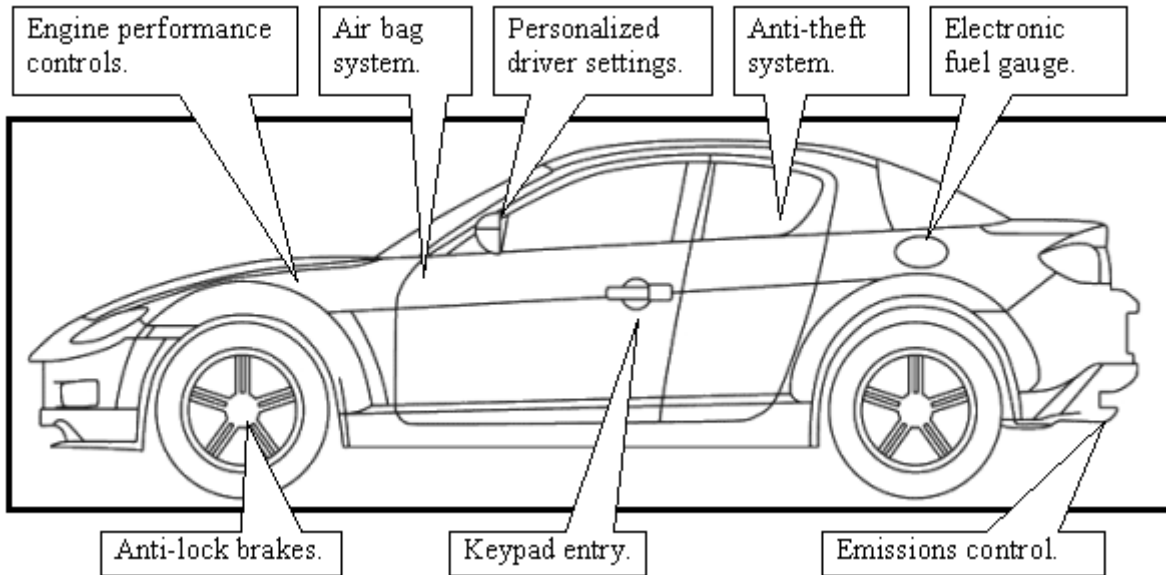


Figure 4. Automobile systems that rely on microcontrollers.

Figure 5. Top view of the breadboard with the microcontroller installed.

### The Challenge: Attach the Arduino Uno About the Capacitor

Like many electronic devices, the microcontroller is sensitive to surges in power, which can cause it to malfunction. Occasionally, the battery pack's power surges. To even out these surges, you will attach a component called a **capacitor**, which stores the extra electricity during a surge and then releases it slowly. The capacitor protects the microcontroller, just as a surge protector protects your computer from power surges in your home. Figure 8 shows the capacitor.



Figure 8. The 1 uf capacitor (left) and a home surge protector (right).

### The Challenge: Attach the Arduino Uno Installing the Capacitor

1. Attach your Arduino to the breadboard “the long way” using the **rubber band** provided in your kit covering holes 5 through 24, with the digital ports facing the high numbers on the breadboard, as seen in Figure 10.

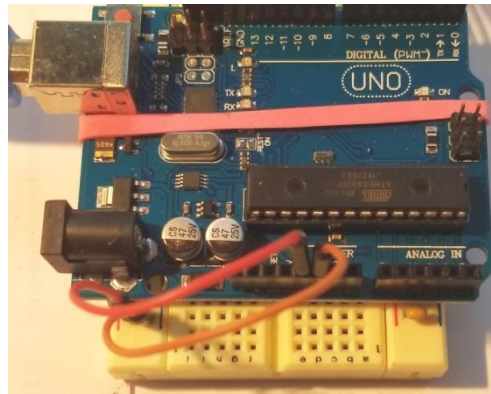


Figure 9. Arduino attached and wired up to the breadboard.

2. Install the capacitor on the board so it connects the blue rail to the red rail to protect the microcontroller. See *Figure 10*.

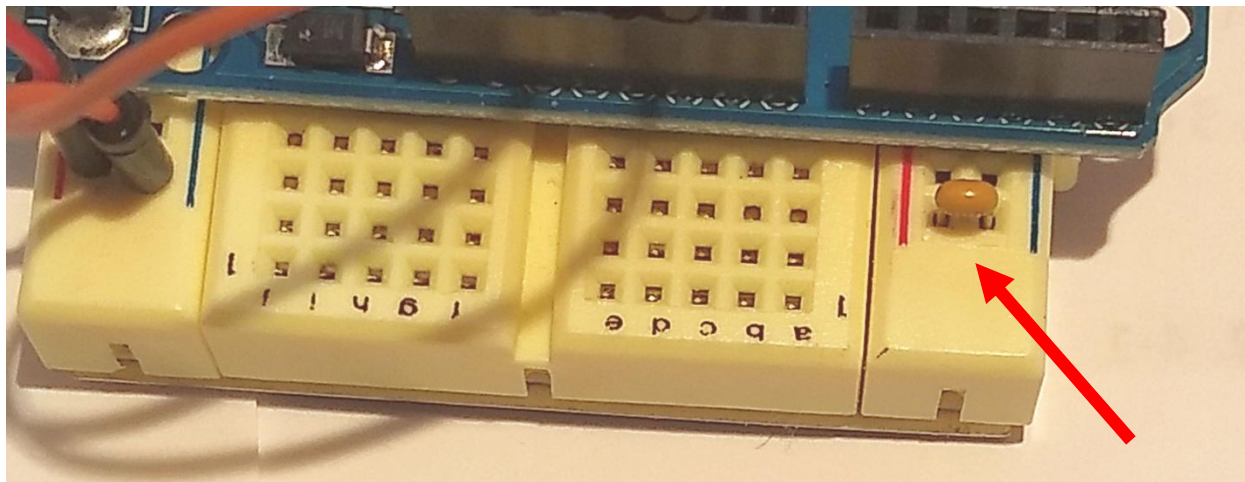


Figure 10. Capacitor connected to blue and red rails.

3. Connect the 5v port to the **red power rail** of the breadboard using a flexible jumper wire, and connect one of the GND1 ports on the Arduino to the **blue ground rail** of the breadboard. **Be sure you do not get these wires confused. Getting these backwards could destroy your Arduino.** See Figure 11.

---

<sup>1</sup> GND is an abbreviation for ground. It is the negative side of batteries or power supplies.



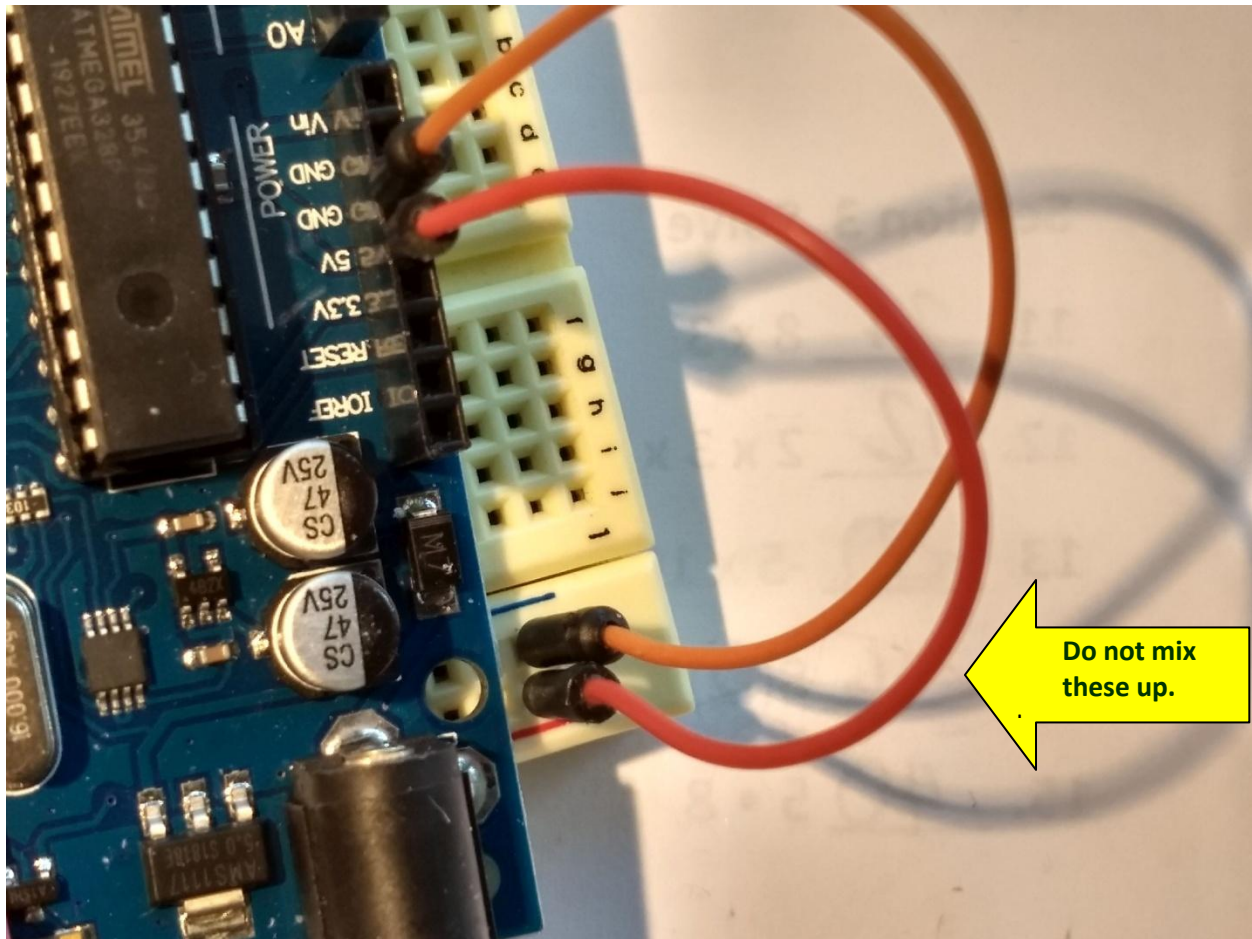


Figure 11. Arduino 5V and GND ports connected to the breadboard.

### **The Challenge: Attach the Arduino to a Computer and its electronics** **Identifying Your Programming Hardware**

To program the microcontroller, you must establish a linkage between your computer and the Arduino. The Arduino came with **USB B port** and with two rows of **female headers** that plug into the breadboard using jump wires.



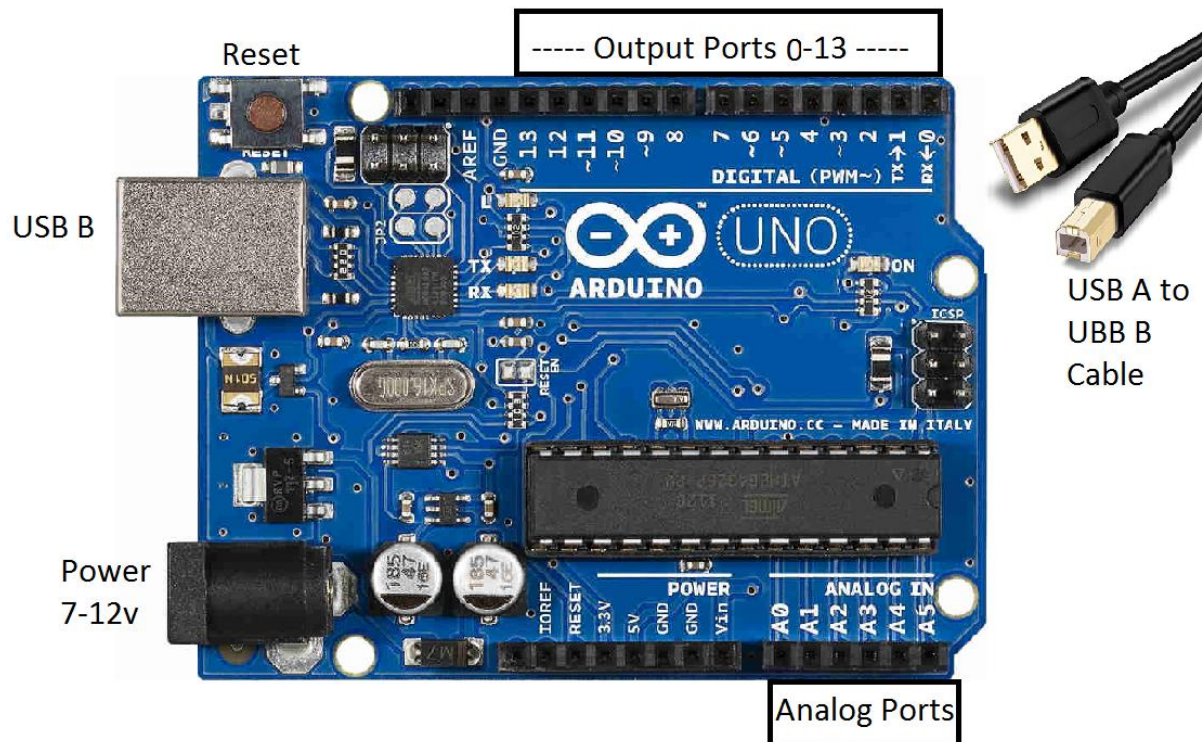


Figure 12. Arduino and USB cable.

## Features of the Arduino Board

Because it has an internal voltage regulator, the Arduino Uno can be powered by variety of power sources. Any power source that provides 7 to 12 volts, and has a Japan Jack also known as an EIAJ connector will work.<sup>2</sup>

Arduinos are easy to program; they connect up to the USB port of a computer or device using a USB cable.<sup>3</sup> The USB connection also provides power to the Arduino and small device is connected up to the Arduino. It is very important that large devices like Motors not be connected directly to the Arduino. Doing so would damage your Arduino's delicate circuitry.

The reset button on the top of the Arduino is used to restart the program resident in the microcontroller. Pressing it will short out the microcontroller chip and make it restart. Use this to run your circuits.

The output ports shown at the top of figure 12 are the standard digital output

<sup>2</sup>. Japan Jacks are barrel jacks that are 5. 5 mm outside and 2. 1 mm inside.

<sup>3</sup>. You need a USB A to USB B, male to male cable.

ports used to program devices. They can be programmed to either be on or off, or detect a voltage.

Along the bottom of figure 12 are analog input ports. These ports can sense I voltage between 0 and 5 volts and turn it into a digital number in the program that will equal between 0 and 100.



Before moving on to the next challenge, take a few minutes to check your hardware set-up. The following checklist will help you ensure that your XBoard is ready for programming.

- **GND Port:** Should be connected to the BLUE rail on the breadboard.
- **5V Port:** Should be connected to the RED rail on the breadboard..
- **Capacitor:** The capacitor's pins should be bridging the blue and red rail of the breadboard.
- **Rubber Band:** Straps your Arduino on to the breadboard long-wise, with the digital ports facing the high numbers on the breadboard.

## Introduction: Building Blocks

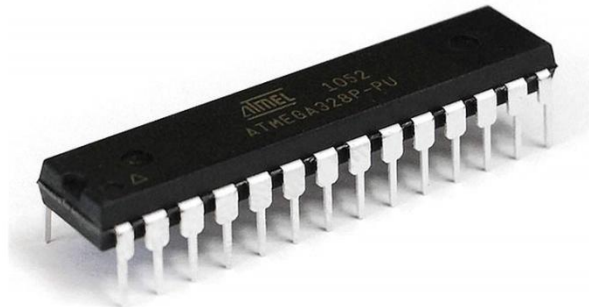
By now, you have set up the microcontroller on the breadboard and learned how to transfer code from your computer to the chip. In each of the following activities, you will add basic electronic components to the breadboard--an LED, a button, a speaker, a sensor, and an LCD--and then write a few lines of code to get them working. These activities are simple, but there's a lot to learn here, so take your time and read carefully. The skills you learn in this section will be very important later on. Figure 1 shows some electronic devices that have LEDs, buttons, speakers, sensors, and LCDs.



Figure 1. Electronic devices with LEDs, buttons, speakers, sensors, and LCDs.

## About Outputs and Inputs

Before you get started, take a close look at the microcontroller. A photo of the chip, taken out of the breadboard, is shown in Figure 2.



**Figure 2. Microcontroller.**

The microcontroller may seem like a mysterious device, but actually, it interacts with other electronic components on the breadboard in only two, very simple ways:

- #1 It can set the voltage on any pin HIGH (5 volts) or LOW (0 volts).**
- #2 It can detect whether the voltage at any pin is HIGH (5 volts) or LOW (0 volts).**

It also knows a few other tricks, but that's basically all the chip does: set the voltage on certain pins, and detect the voltage on others. Coupled with the ability to store and execute code, these two simple abilities make the microcontroller a very powerful device.

Consider #1 first. If you connect an output device, such as an LED, to a particular microcontroller pin, **the microcontroller can turn the device on or off by raising and lowering the voltage** on that pin. Remember: electric current flows whenever there is a voltage difference between one side of a component and the other. In similar fashion, the chip can control more complex output devices, like speakers, motors, and the LCD, by raising and lowering the voltage on its pins in precisely timed patterns.

Now consider #2. **The microcontroller can detect whether the voltage at any of its pins is 5 volts or 0 volts.** Imagine that you connect a simple input device, such as a button switch, to a particular microcontroller pin, and set up the circuit so that the button connects that microcontroller pin to power (5 volts) when it is pressed and ground (0 volts) when it is not pressed. The microcontroller can tell whether the button is pressed or not pressed, simply by detecting whether the voltage at the pin is high or low.

That, in a nutshell, is how all modern electronic devices work. Think of all the times you have seen an LED light on your computer keyboard, read text on an LCD, pressed a cell phone key, or hit a game controller button. Inside those devices, there were computer chips, raising and lowering the voltage on some

pins, and detecting high and low voltage on others.

The microcontroller's code is simply the logic that connects the inputs and the outputs. In this way, a particular input; like hitting the jump button on a game controller; can trigger a particular output; your game character moves on the screen.

## Identifying Microcontroller Pins

In order to connect new components to the microcontroller, you need to know how to identify its ports. Figure 3 shows the microcontroller, with a map of its ports.

(For help finding pins, you can always return to this page or check the [Arduino reference page](#), which is listed along with other useful Quick References near the bottom of the project guides list.)

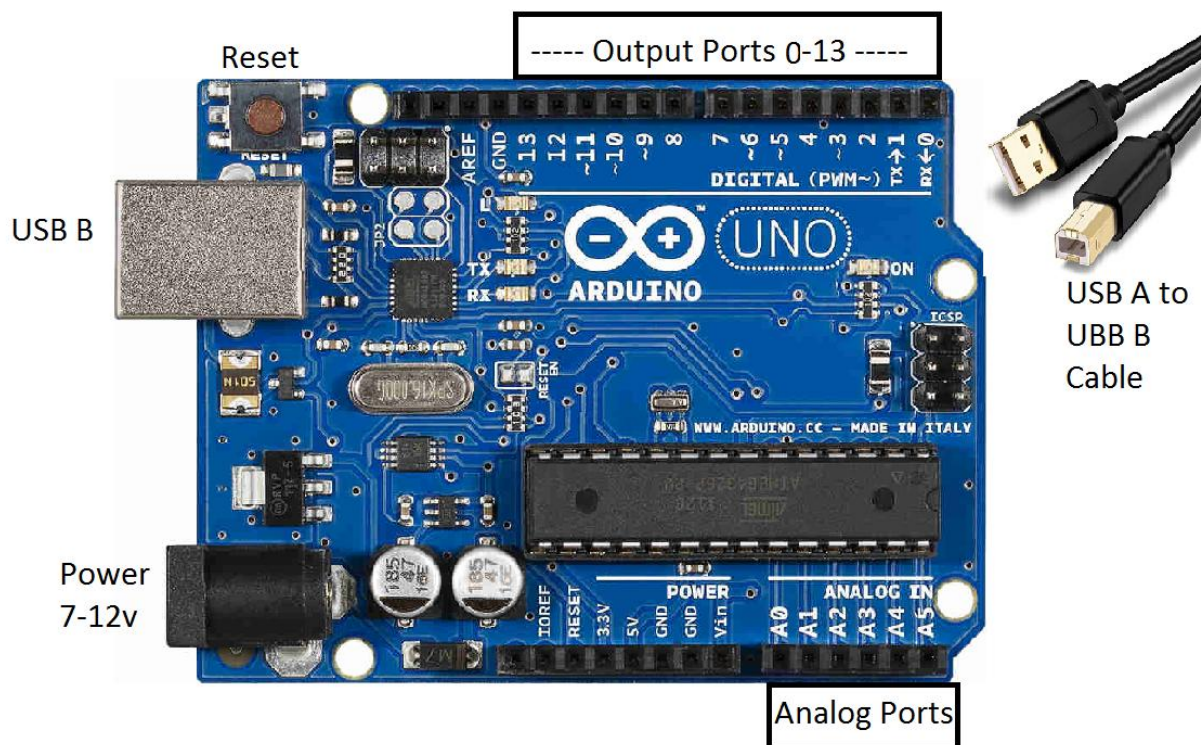
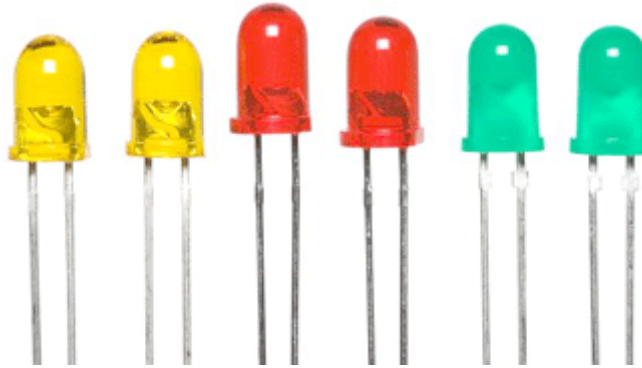


Figure 4. Arduino and USB cable.

## 1 Lighting an LED



Light emitting diodes (LEDs) are like tiny light bulbs that emit light when electricity passes through them. Figure 1 shows yellow, red, and green LEDs.



**Figure 1. LEDs.**

Because LEDs require very little electricity and last for thousands of hours, they are becoming increasingly common in electronic devices and other applications. Your computer keyboard probably has a few LEDs to indicate when the Caps Lock or Num Lock keys have been pressed. Grouped together, LEDs can produce enough light to be used as a source of household illumination, as in the lamp shown in Figure 2.



**Figure 2. LED lamp.**

### **Challenge 1: Build a Circuit with an LED**



To start this activity, you will build a circuit on the breadboard with an LED, a resistor, and a flexible jump wire.

#### **Challenge 1: Build a Circuit with an LED**

## Collecting Your Components

You will need the following components for this activity (shown in Figure 3):

Part	Quantity	Description
A	2	LEDs
B	2	Flexible jump wires
C	2	220-Ohm resistors



Figure 3. Components for the LED activity.

### Challenge 1: Build a Circuit with an LED About LEDs

Every LED has a specific polarity, meaning one of its two pins must connect to the power side of the circuit and the other pin must connect to the ground side. A tiny flat segment on the round rim of the LED's circular base marks the ground pin. The power pin is typically longer than the ground pin, but this is an unreliable indicator, since the pins may be trimmed during installation. Figure 4 shows these distinguishing features.

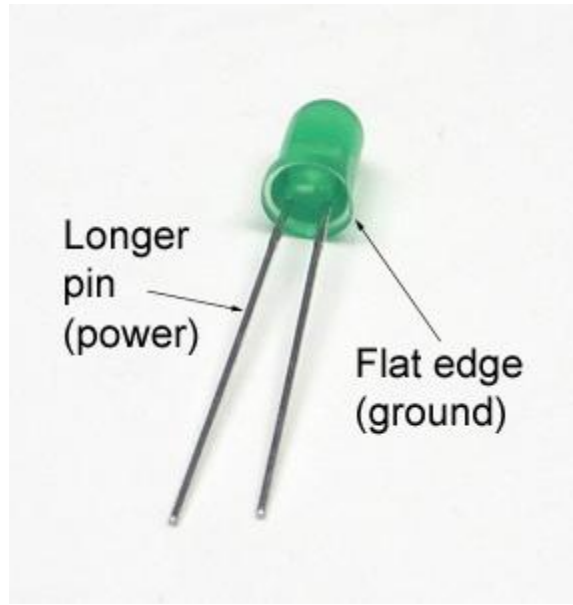


Figure 4. LED polarity.

### Challenge 1: Build a Circuit with an LED

#### Adding the LED

The LED's ground pin is connected to ground by a 220-Ohm resistor, and its power pin is connected to the microcontroller by a flexible jump wire.

**1. Using a 220-Ohm resistor, connect hole J31 to ground. Then insert the LED, with its power pin in hole G30 and its ground pin in hole G31, as shown in Figure 7.**

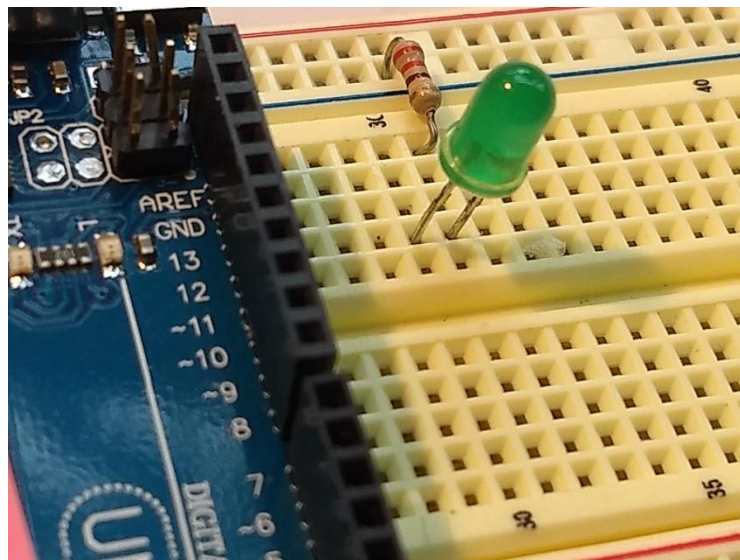




Figure 7. Adding the LED and resistor.

2. Using a flexible jump wire, connect the LED's power pin (hole I30) to Port 7 on the Arduino, as shown in Figure 8. (If you need help finding Port 7, consult the [Arduino Quick Reference](#) page at the end of this document.)

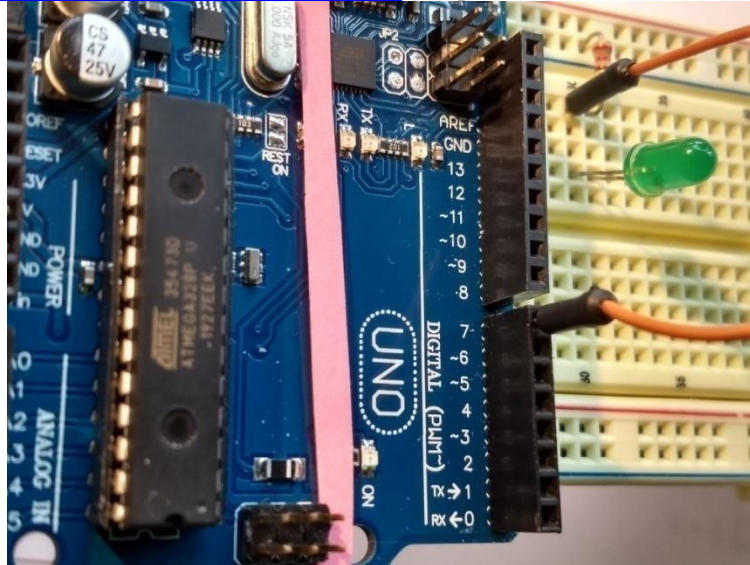


Figure 8. Connecting the LED to the Arduino.

## Challenge 2: Write Code to Control the LED



With your LED hooked up to the Arduino, you can control it automatically; turning it on, turning it off, and making it flash. This will be your first chance to write your own C code for the microcontroller.

## Challenge 2: Write Code to Control the LED About C Code

A **C program** is a series of instructions that tell the microcontroller in the Arduino what to do. The microcontroller on the Arduino board executes these instructions in sequence from top to bottom. Statements can be combined into larger groups, called functions, which enable the microcontroller to perform more complex operations. In order for the compiler to understand your instructions, the program must always be "wrapped" with a few specific lines of code. Figure 9 shows a simple C program, with these important "wrapping" parts identified. You don't have to get this program working yet; just look at it and try to understand its basic structure.

This is example named **BareMinimum** is found under file, examples and 01.basics.

```
void setup() {  
  // put your setup code here, to run once:  
  
}  
  
void loop() {  
  // put your main code here, to run  
  repeatedly:  
  
}
```

The word **void** tells the microcontroller not to provide an answer.

**Functions**, like **setup()** and **loop()** always begin and end with brackets. All commands between the brackets are executed when the function is executed.

Text preceded by slashes, **//** are comments. They are ignored by the computer. Programmers use them to write notes to the people reading the program.

Pull this program up to make it easier to write programs for your Arduino.

If you don't understand every line of the program, don't worry. Just remember three things:

Every main function must be contained in curly braces: { and }

Every Arduino program has a *setup()* function and a *loop()* function.

Everything after **//** is ignored, so programmers can write notes there.

## Challenge 2: Write Code to Control the LED

### Turning the LED On

OK. You are ready to write a simple program to turn on the LED. Your program has two control statements: **setup()** and **loop()**. Commands placed in between *setup()*'s brackets {}, are run one time, right as the program starts. Commands placed in between *loop()*'s brackets {}, are run over and over, after *setup()* is done. In our program, *setup()* sets up Port 7 (the pin connected to the LED) as an output pin, meaning electricity can flow out of it; the second statement, in *loop()*, sets the value of the Port 7 pin high, meaning its voltage is 5 volts.

1. Open the Arduino IDE.

2. Type the following code in the window. **NOTE: You don't have to type the comments, but it is good practice to do so.**

```
// Program 1-1LED
```

```
// Lights an LED connected to port 7

// the setup function runs once when you press reset or power the board
void setup() {
  pinMode(7, OUTPUT); //set up port 0 to output 5 volts
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(7, HIGH); // turn the LED on (HIGH is voltage 5 volts)
}
```

3. Save your code file as ledon.

4. Compile and test your code by clicking the run arrow.



Your LED should now light and stay lit. Electricity is flowing from the Port 7 pin, which has a value of 5 volts, through the LED and the resistor to ground, which has a value of 0 volts.



### Programming Challenge

Move one end of the flexible jump wire from Port 7 to Port 8, leaving the other end in hole H30. Note what happens. Now, leaving the flexible jump wire in place, change the code in ledon to make the LED light up again. HINT: You will need to modify two lines in your program. When you have finished, return the jump wire to its original position.

## Challenge 2: Write Code to Control the LED Turning the LED Off

Turning an LED off is as simple as turning it on. You simply set the pin it is connected to low (0 volts), instead of high (5 volts). This way, the voltage at the pin is the same as the voltage at ground, and no electricity flows through the circuit.

1. Rename your code file ledoff.c.

2. Modify your code file, as follows:

```
// program ledoff
```

```
// the setup function runs once when you press reset or power the board
void setup() {
  pinMode (7, OUTPUT); //set up port 0 to output 5 volts
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite (7, LOW); // turn the LED on (LOW is voltage 0 volts)
}
```

3. Compile and test your new code.

## Challenge 2: Write Code to Control the LED Turning the LED On and Off

So now you know how to turn an LED on and off. How about doing both of these things these things in one program? To do this, you will need to introduce a new instruction in your code, called a **delay()**. A delay statement tells the microcontroller to wait for a specific period of time before executing the next instruction. Depending on what type of delay statement you use, these intervals are measured in milliseconds (1/1000ths of a second) or microseconds (1/1,000,000ths of a second), as shown in Figure 10.

Two commands to wait.

**delay(ms)** – Where ms is the number of milliseconds. (1ms = 1/1,000 second = 0.001 second)

**delayMicroseconds(us)** – Where us is the number of microseconds. ( 1us = 1/1,000,000 second = 0.000001 second)

Figure 10. Two types of delay statements.

## Challenge 2: Write Code to Control the LED Making the LED Flash

1. Rename your code file ledflash.c.
2. Modify your code file, as follows:

```
// the setup function runs once when you press reset or power
the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
```

```

pinMode(7, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(7, HIGH); // turn the LED on
  delay(1000);           // wait for a second
  digitalWrite(7, LOW);  // turn the LED off
  delay(1000);           // wait for a second
}

```

#### **Blink**

Show your work to the instructor for a grade.

### 3. Compile and test your new code.



#### **Programming Challenge**

Change your delay statements to make the LED flash even faster. How fast can you go before your eyes can't even tell that it's flashing? Change the condition for your while...loop to something else that's always true, such as `5==5`, or `9>6`. Does this affect your program at all?

### **Challenge 2: Write Code to Control the LED** **Add a Second and third LED**

Before moving on to the next activity, for even more credit, add a second LED to the breadboard, placing its pins in holes G36 and G37. As before, use a 220-Ohm resistor to connect the LED to ground, and use a flexible jump wire to connect the power pin to Port 8 on the microcontroller, then add a third LED and connect it to port 9.

#### **LED SEQ EC**

Show your work to the instructor for an extra credit grade.

Modify your program to light the LEDs in sequence.

## 2 Using a Button

A button is a switch that completes or interrupts an electric circuit, much like a switch for a household appliance. Buttons are commonly used to provide input to electronic devices, such as the texting device and game controller shown in Figure 1.



Figure 1. Electronic devices with buttons.

### Challenge 1: Build a Circuit with a Button



In this Challenge, you will build a circuit on the breadboard that allows you to control the LED with a button.

### Challenge 1: Build a Circuit with a Button Collecting Your Components

For this activity, you will need the following components (shown in Figure 2):

Part	Quantity	Description
A	1	Button switch
B	1	Flexible jump wire
C	1	10,000-Ohm resistor
D	1	Pre-bent jump wire (yellow)



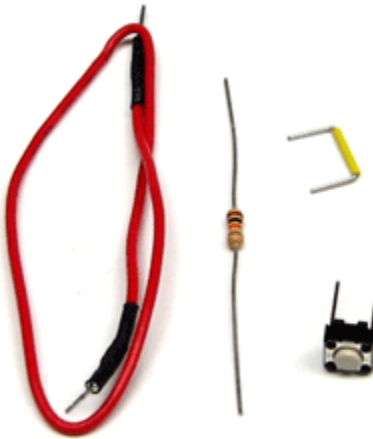


Figure 2. Components for the button activity.

### Challenge 1: Build a Circuit with a Button Adding the Button

Figure 3 shows the type of button included in the Machine Science Starter Kit. These are referred to as **momentary switches**, because their two pins are connected only while the button is pressed and held down. As soon as you take your finger off the button, the connection is broken.



Figure 3. Button switch.

1. Add the button to the breadboard, putting its pins in holes F41 and F43. Connect one side to ground with a 10,000-Ohm resistor, and connect the other side to power with a yellow jump wire, as shown in Figure 4.

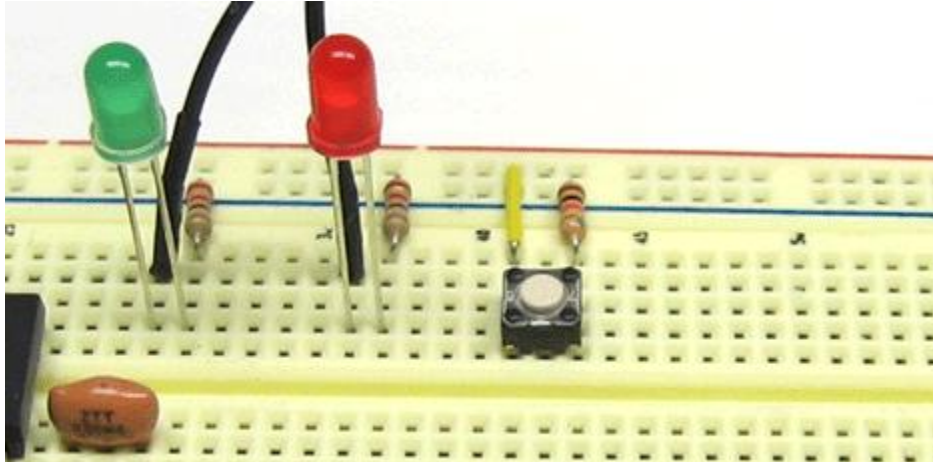


Figure 4. Adding the button.

2. Using a flexible jump wire, connect the ground side of the button to Port 6 on the Arduino, as shown in Figure 5. (If you need help finding Port 6, consult the [Arduino Quick Reference](#) at the end of this document.)

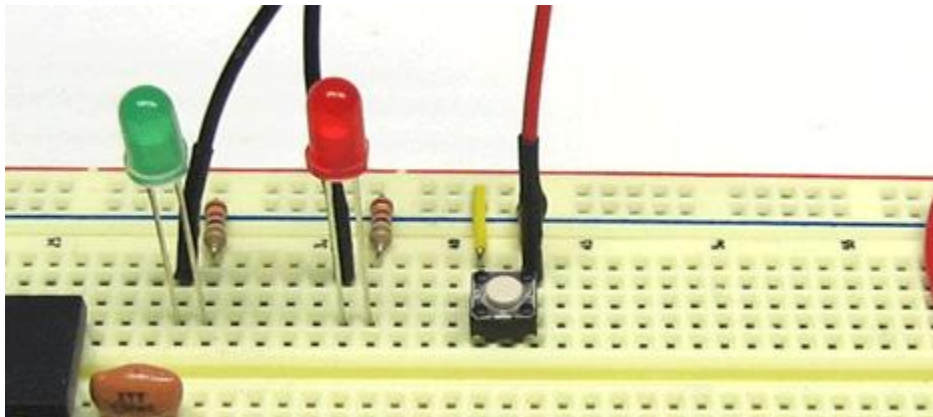


Figure 5. Connecting the button to the microcontroller.

## Challenge 1: Build a Circuit with a Button

### How the Button Works

Look closely at the button connections on the breadboard. One side of the button is tied to ground, through a resistor, and the other side is tied to power. On the ground side, there is also a flexible jump wire connecting to Port 6 on the Arduino. This means that, when the button is not pressed, Port 6 will be at 0 volts, i.e. low. When the button is pressed, Port 6 will be tied to 5 volts, i.e. high. Figure 6 shows this relationship.

Button State	Port 6 State
--------------	--------------

Not Pressed	0 Volts (LOW)
Pressed	5 Volts (HIGH)

Figure 6. Button state and Port 6 state.

## Challenge 2: Write Code for the Button



In this challenge, you will write code so that you can use the button to control the LED.

## Challenge 2: Write Code for the Button Turning the LED On

To use the switch to turn on the LED, you need to introduce a new programming structure into your code; an **if... statement**. An if... statement tells the microcontroller to execute a segment of code if a certain condition is true. In this case, your if... statement will tell the microcontroller to light the LED if the voltage at Port 6 is HIGH (in other words, if the button is pressed.)

1. Rename your code file `buttonpress.c`.
2. Modify your code file, as follows:

```
int button = 0;    // Used to store the button state

void setup() {
  pinMode(13, OUTPUT); // The built in LED
  pinMode(6, INPUT);   // The switch
}

void loop() {
  button = digitalRead(6);

  if (button == HIGH) {
    digitalWrite(13, HIGH); // LED on
  }
}
```

3. Compile and test your new code.

## Challenge 2: Write Code for the Button Turning the LED On and Off

It is also possible to extend your *if...* statement, so that the microcontroller executes another segment of code if the condition is not true. This is known as an **if...else...** statement, because it causes the microcontroller to do one thing *if* a condition is true, or *e/*se it does another thing. In this case, your code will light the LED if the button is pressed, or else it will turn the LED off.

1. Rename your code file `buttononoff.c`.
2. Modify your code file, as follows:

```
int button = 0;

void setup() {
  pinMode(13, OUTPUT);
  pinMode(6, INPUT);
}

void loop() {
  button = digitalRead(6);

  if (button == HIGH) {
    digitalWrite(13, HIGH);  // LED on
  } else {
    digitalWrite(13, LOW);   // LED off
  }
}
```

### Button

Show your work to the instructor for a grade.

3. Compile and test your new code.



### Programming Challenge

Modify your code so that the green LED turns on if the button is pressed, and the red LED turns on if the button is not pressed.

### Button 2LED EC

Show your work to the instructor for an extra credit grade.

### 3 Making Sounds

The headphones used to produce sounds from portable MP3 players, like Apple's iPod are just tiny speaker's controlled by signals sent from a computer chip. Figure 1 shows some typical "earbud" style speakers.



Figure 1. "Earbud" style headphone speakers.

#### Challenge 1: Build a Circuit with a Speaker



To start this activity, you will add a speaker to the breadboard.

#### Challenge 1: Build a Circuit with a Speaker Collecting Your Components

For this activity, you will need the following components (shown in Figure 2):

Parts	Quantity	Description
A	1	Piezo speaker
B	1	Bent connector (two prong)
C	1	Flexible jump wire
D	1	Pre-bent jump wire (orange)





Figure 2. Components for the speaker activity.

### Challenge 1: Build a Circuit with a Speaker

#### Adding the Speaker

The speaker connects to ground with an orange jump wire and to Port 2 on the microcontroller with a flexible jump wire.

1. Add a two-prong connector, an orange jump wire, and the speaker to the breadboard, as shown in Figure 3. **NOTE: Be sure to align the black speaker wire with the jump wire to ground.**

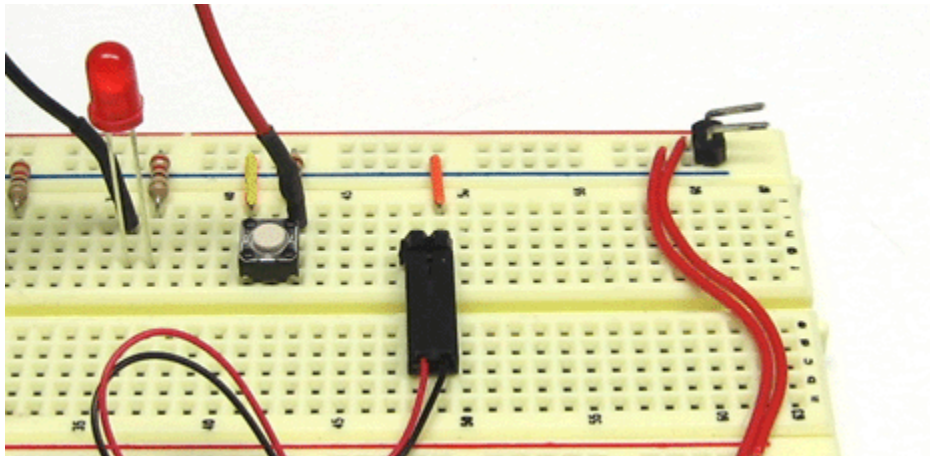


Figure 3. Adding the speaker.

2. Using a flexible jump wire, connect the red wire side of the speaker to Port 2 on the Arduino as shown in Figure 4. (If you need help finding Port 2, consult the [Arduino Quick Reference](#) at the end of this document.)

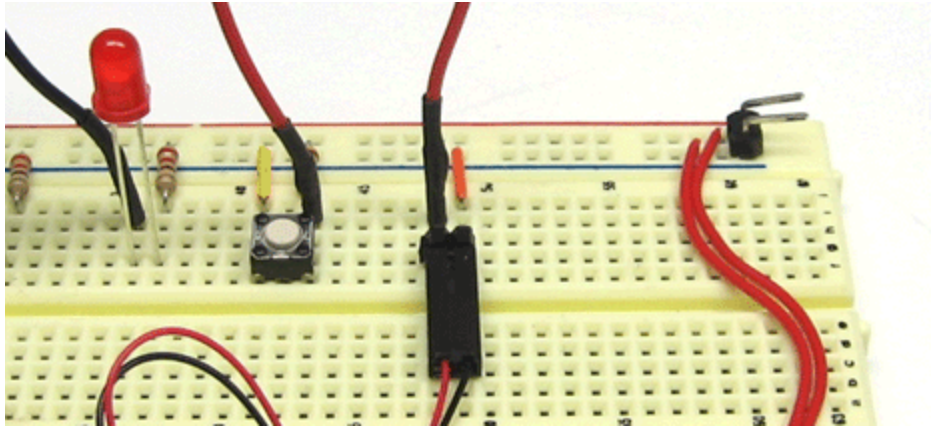


Figure 4. Connecting the speaker to the microcontroller.

### Challenge 1: Build a Circuit with a Speaker About the Speaker

The speaker used in this activity is called a **piezo-electric speaker**. The key component in the speaker is a material that bends slightly when you supply electricity to it. If the electricity is sent in pulses, the element vibrates back and forth. Just like the movement of a saxophone reed or a guitar string, this vibration produces audible sound waves in the air. To make the piezo-electric element vibrate, you need to raise and lower the voltage on Port 2, alternating between 0 volts and 5 volts. Figure 5 shows how the voltage signals make the piezo-electric element move.

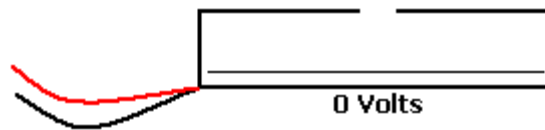


Figure 5. Voltage signals making piezo element move.

### Challenge 2: Write Code to Control the Speaker



In this challenge, you will learn to control the speaker. You will be able to make a simple click, a sustained tone, and many different tones. This will be another opportunity to write more C code for the microcontroller.

## Challenge 2: Write Code to Control the Speaker Making a Click

Using simple program, much like your LED control code, you can cause the speaker to make a clicking sound.

1. Rename your code file click.
2. Modify your code file, as follows:

```
// Program click

void setup() {
  pinMode(2, OUTPUT); //set up port 2 to output 0 or 5 volts
}

void loop() {
  delay(1000);
  digitalWrite(2, HIGH); // HIGH is voltage 5 volts LOW is 0 volts
}
```

3. Compile and test your new code.
4. Press the reset button a few times, while pressing the speaker up to your ear, you should hear a clicking sound each time.

## Challenge 2: Write Code to Control the Speaker Making a Sustained Tone

By putting a bunch of clicks together in a while... loop, you can make the speaker produce a sustained tone. Note that in the code file shown here, the delay is in microseconds (**delayMicroseconds**), not in milliseconds (**delay**).

1. Rename your code file sustained.
2. Modify your code file, as follows:

```
// Program sustained

void setup() {
  pinMode(2, OUTPUT);
}
```

**Sound**  
Show your work  
to the instructor  
for a grade.

```
void loop() {
  digitalWrite (2, LOW); // 0 volts
  delayMicroseconds (1000);
  digitalWrite (2, HIGH); // 5 volts
  delayMicroseconds (1000);
}
```

### 3. Compile and test your new code.



#### Programming Challenge

Modify your code file to produce different tones. HINT: The value in the delay\_us statement is the key.

### Challenge 2: Write Code to Control the Speaker Using the Tone Function

Because controlling the speaker is so much fun, BCMI, the company that makes Arduinos, has created a special function just for that purpose. This function, called **tone()**, makes it very easy to control the speaker. Each time you use the tone function you have to give it three numbers: the first number is the pin number; the second is the frequency (in Hertz) of the note you want; the third number is the duration in milliseconds. For example, the statement `tone(440, 200);` will produce a note of 440 Hertz for 200 milliseconds. Duration can be left off, leaving only the pin and the frequency.

You can stop the sound using the `notone` function. It needs only one piece of information, the pin number.

**tone (pin, frequency, [duration])**  
**notone (pin)**

1. Rename your code file `tonefunction`.
2. Modify your code file, as follows:

```
// Program tonefunction
```

```
void setup() {
  pinMode(2, OUTPUT);
}
```

```
void loop() {  
  tone(2, 988, 320);  
  delay(1000);  
}
```

3. Compile and test your new code.



### Programming Challenge

Modify your code file to change the frequency and the duration of the tone produced by the speaker. Try adding some additional notes.

## Challenge 2: Write Code to Control the Speaker Putting Tones in a Loop

With a while loop, you can make any sequence of tones repeat over and over.

1. Rename your code file **twotones**.
2. Modify your code file, as follows:

**// Program 1-3twotones**

```
void setup() {  
  pinMode(2, OUTPUT);  
}  
  
void loop() {  
  
  tone (2, 440);  
  delay (200);  
  
  tone (2, 535);  
  delay (100);  
}
```

3. Compile and test your new code.

## Challenge 2: Write Code to Control the Speaker Using a For... Loop



With a different kind of loop, you can make the frequency of the tone rise or fall. To do this, you will need to introduce two new elements into your code. The first is a **variable**. A variable is letter or phrase that is assigned a numerical value, which can change over time. In this case, your variable will be called x. The second is a **for... loop**. A for... loop repeats a segment of code a specific number of times and then stops.

1. Rename your code file forloop.c.
2. Modify your code file, as follows:

```
// Program forloop

int x;
void setup() {
  pinMode(2, OUTPUT);
}

void loop() {
  for (x=250; x<750; x=x+1){
    tone(2,x);
    delay (2);
  }
}
```

**Siren**  
Show your work  
to the instructor  
for a grade.

3. Compile and test your new code.



#### Programming Challenge #1

Modify your code so that the tone falls instead of rising each time through the loop. HINT: You will need to change three things in the statement setting up the for loop.



#### Programming Challenge #2

Wrap your for loop inside a while loop so that the each rising or falling tone sequence repeats over and over, like a siren.



### Programming Challenge #3

Add code to make the LEDs flash in time with the speaker.

## Challenge 2: Write Code to Control the Speaker Making a Melody

By stringing together individual tone statements, you can program the microcontroller to make a simple melody.

1. Rename your code file melody.
2. Program the microcontroller to play the first seven notes of *Mary Had A Little Lamb*, using the following table of frequencies and durations as your guide:

Note	Frequency	Duration		Note	Frequency	Duration
B	494	200		B	494	200
A	440	200		A	440	200
G	392	200		G	392	200
A	440	200		A	440	200
B	494	200		B	494	200
B	494	200		B	494	200
B	494	400		B	494	200
A	440	200		B	494	200
A	440	200		A	440	200
A	440	400		A	440	200
B	494	200		B	494	200
D	587	200		A	440	200
D	587	400		G	392	800

3. Compile and test your new code.

#### Mary EC

Show your work to the instructor for an extra credit grade.

## 4 Sensing Light

Because they do not flash or make noise, sensors are less noticeable than the other electronic devices in the world around us, but they are becoming increasingly common. Often, they are used to monitor the environment and trigger actions that human beings may forget or neglect to do--for example, turning a car's windshield wipers on when it starts to rain, or flushing a toilet automatically when a user exits a stall. In other cases, sensors are used to maintain desired conditions when human beings are not around at all--for example, activating the furnace when a house's temperature drops, or turning on a lamp when it gets dark. Figure 1 shows some of these common sensors.



Figure 1. Electronic devices with sensors.

### Challenge 1: Build a Circuit with a Light Sensor



In this challenge, you will add a light sensor, also known as a photoresistor (or phototransistor) to the breadboard.

### Challenge 1: Build a Circuit with a Light Sensor Collecting Your Components

For this activity, you will need the following components (shown in Figure 2):

Part	Quantity	Description
A	1	Light sensor (photoresistor)
B	1	Flexible jump wire
C	1	Orange jump wire
D	1	Resistor (10,000 Ohm)

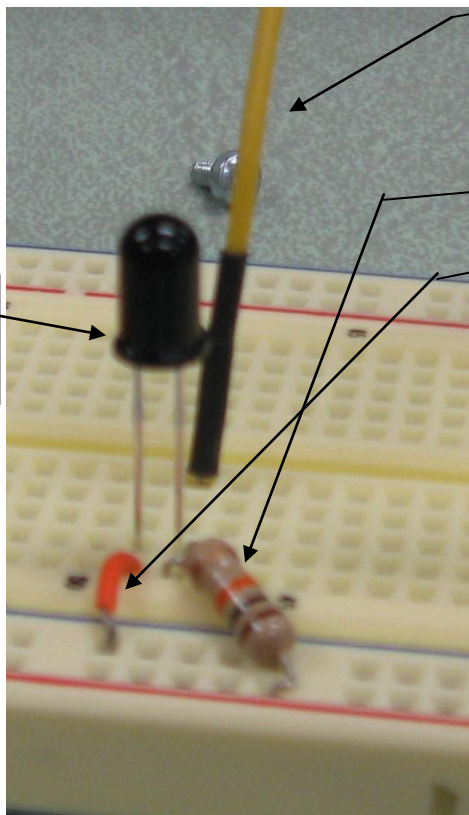


Figure 2. Components for the sensor activity.

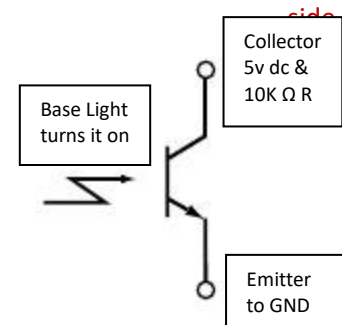
## Using a Phototransistor as a Sensor (Instead of Photoresistor)

If you have a photoresistor, click [here](#).

The QSD123 NPN Phototransistor will send a signal when light strikes it and stop when the light stops. The phototransistor looks like a black LED. The yellow jump wire is connected to the long lead on the



transistor on the opposite side to the cut off rim. Both these wires are connected to power using a 10K  $\Omega$  resistor that gives the circuit TTL voltage. The wire on the side with a cut off rim is connected to the ground.

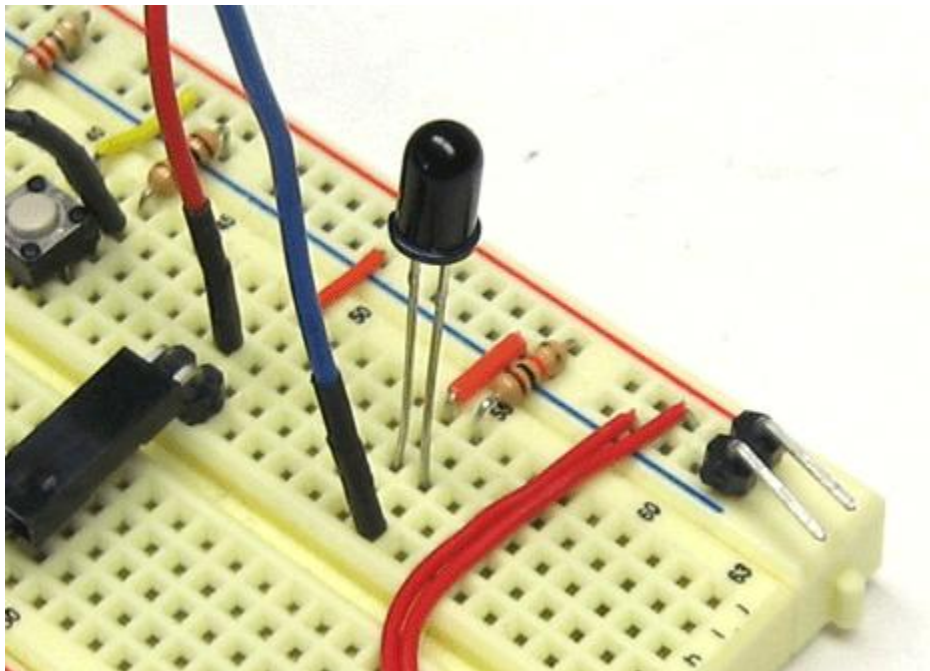


This sensor will respond to sunlight and incandescent light, but does not respond to light produced by florescent lighting very well. Click [here](#) to continue.

## Challenge 1: Build a Circuit with a Light Sensor (Photoresistor only) Adding the Sensor to the Board

The installation of the light sensor is similar to that of the button, with one side connected to ground, the other side connected to power, and a flexible jump wire spanning to the microcontroller.

**1. Insert the light sensor's two pins into holes H54 and H55. The ground pin, marked by the flat edge on the rim of the sensor, should go in hole H54 and be connected to ground by an orange jump wire. The sensor's power pin should go in hole H55 and be connected to power with a 10,000-Ohm resistor. A flexible jump wire should then be connected to the power side of the sensor (hole F55), as shown in Figure 3.**



**Figure 3. Adding the light sensor, resistor, and jump wire.**

**2. Connect the loose end of the flexible jump wire to Port A0 on the microcontroller.**

### **Challenge 1: Build a Circuit with a Light Sensor (Both resistor and transistor) About the Light Sensor**

At the beginning of this section, you learned that the microcontroller has two essential abilities:

**#1 It can *set* the voltage on any of its pins HIGH (5 volts) or LOW (0 volts).**

**#2 It can *detect* whether the voltage at any of its pins is HIGH (5 volts) or LOW (0 volts).**



In fact, on certain pins, the microcontroller can measure the exact voltage at the pin, as long as the voltage is between 5 volts and 0 volts. These pins are called **analog ports**, because they can measure a range of values, rather than just detecting high or low values.

The light sensor is a light-sensitive resistor, whose resistance decreases with the amount of light falling on it. When you added the light sensor to the breadboard, you tied one side of the component to ground and the other side to power and an analog port (Port A0). Configured in this way, the light sensor effectively puts a voltage at Port A0 that ranges from 0 volts to 3 volts, depending on how much light the sensor detects. The microcontroller then converts this voltage value into a number ranging from 0 to 1024, using its built-in *analog-to-digital converter*, or ADC. Figure 4 shows approximately how these values are related.

Light Level	Voltage at Port A0	ADC Value
Very Bright	0.00 volts	0
Bright	0.75 volts	256
Medium	1.50 volts	512
Dim	2.25 volts	768
Very Dim	3.00 volts	1024

**Figure 4. Relationship among light level, voltage, and ADC value.**

The light sensor included with the Starter Kit can sense only certain types of light. It sees incandescent and sunlight well, but it cannot detect infrared light, as shown in Figure 5.

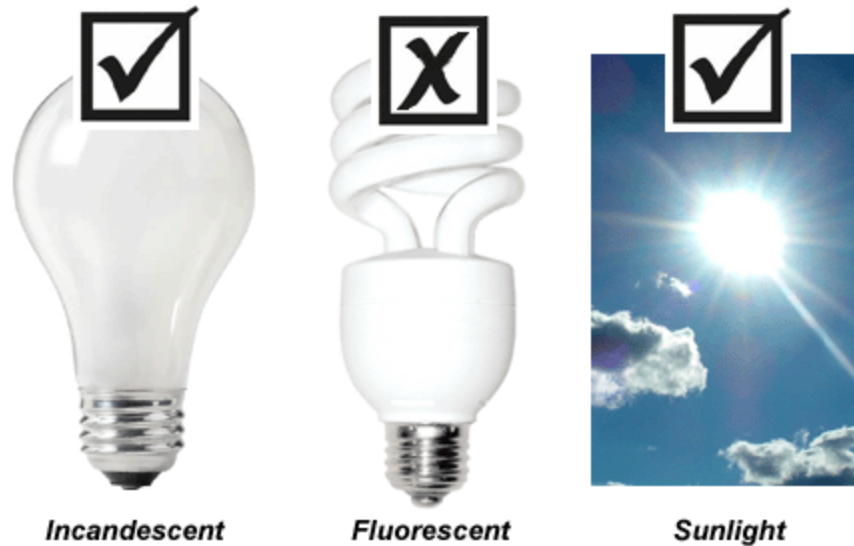


Figure 5. Sources of light detected by the sensor.

## Challenge 2: Write Code for the Light Sensor



In this challenge, you will write code to gather input from the light sensor.

## Challenge 2: Write Code for the Light Sensor Controlling the Speaker

Sensors are not very useful unless they are used to control output devices. In this activity, you will use the sensor to control the tone produced by the speaker. To do this, you will again need to have a **variable** in your code, called *reading*. You will set the value of *reading* equal to the analog value from Port A0, and then pass this value to the computer using **serial.print()**, a command that sends information back to the computer using the USB wire.

1. Rename your code file **sensorspeaker.c**.
2. Modify your code file, as follows:

```
//Project 5.01 Read the phototransistor
```

```
int reading = 0; // a variable to store the light sensor reading
```

```

void setup(){
  pinMode(A0,INPUT);
  Serial.begin(9600);
}

void loop(){
  reading = analogRead(A0); // Get analog reading
  Serial.println(reading); //Send reading to computer
  delay(100);
}

// To see the reading, go to: Tools, then Serial Monitor

```

### 3. Compile and test your new code.



#### Programming Challenge

Modify your code so that the value returned by the light sensor affects the duration of the tone, rather than the frequency. HINT: You will have to set the frequency to a fixed value, such as 440.

### Challenge 2: Write Code for the Light Sensor Controlling the LED

Now imagine that the LED is a light in your house, and you want to turn it on whenever it gets dark in the room. The code in this section shows one way of doing this.

1. Review how to [light up an LED](#).
2. Review the [tone function](#).
3. Write a program that converts the light level into tones.

1. Rename your code file sensorled.
2. Modify your code file, as follows:

```

//Night Light Program

int reading = 0;

void setup(){
  pinMode(A0,INPUT);
  pinMode(13,OUTPUT); // Port 13 with a built-in LED
}

```

```
void loop(){  
  reading = analogRead(A0); // Get analog reading  
  
  if (reading>512){           // High numbers are dark  
    digitalWrite(13, HIGH); } //Turn on LED  
  
  delay(1);  
} // end loop
```

#### 4. Compile and test your new code.



##### Programming Challenge

Modify your code so that the value returned by the light sensor affects the duration of the tone, rather than the frequency. HINT: You will have to set the frequency to a fixed value, such as 440.



##### Programming Challenge

Program the board so that it makes a continuous tone in the dark, and goes silent when there is light. What would happen if you put a board programmed this way in the refrigerator?

## 5 Controlling the LCD

These days, digital text displays are everywhere, from the screens on MP3 players to electronic book readers, such as those shown in Figure 1. Have you ever wondered how these displays work? In this activity, you will learn to send text to an LCD screen, position this text on the screen, and even make your text flash and move!



Figure 1. Text display screens.

### About the LCD

A *liquid crystal display* (LCD) contains a thin layer of liquid crystal that changes from light to dark when electricity is supplied to it. LCDs can display text and graphics, like a miniature computer screens. LCDs are everywhere--in digital watches, household appliances, and handheld video games. Figure 16 shows the most common type of LCD we will be using.

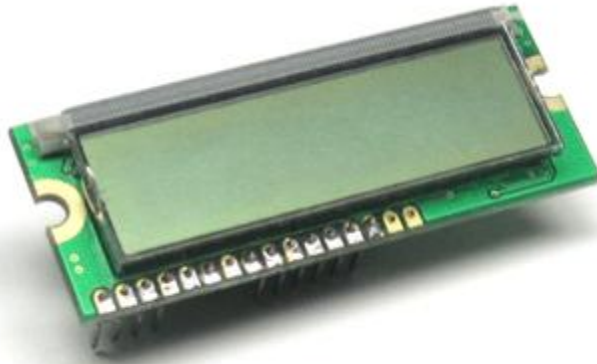


Figure 16. Liquid crystal display (LCD).

### Challenge 1: Install the LCD



In this section you will install and program a standard LCD. To do this, you may have to move other components that you added in other activities. You can either remove these components, or move them to another section of the board. Remember: as long as components are connected to power, ground, and the microcontroller, it doesn't matter where you put it on the board.

### Challenge 1: Re-Install the LCD Collecting Your Components

This project makes use of a 16x2 LCD display, which is shown in Figure 2. The LCD is connected to the breadboard by 10 pins. These pins supply power to the LCD and transmit data from the microcontroller, controlling the text on the screen.

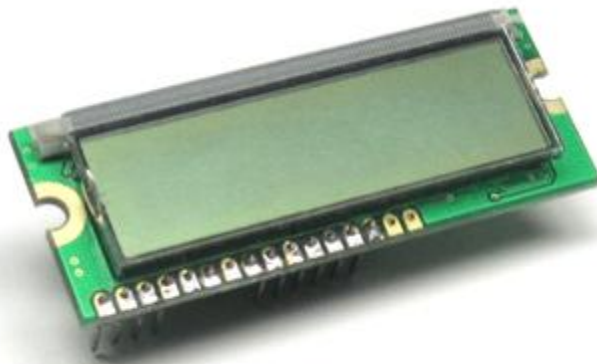


Figure 2. LCD.



## Challenge 1: Install the LCD Putting the LCD Back

1. Use yellow jump wires to connect A42, A45 and A49 to ground; use an orange wire to connect A46 to power, and connect E42 to E47 using a green jump wire, as shown in Figure 3.

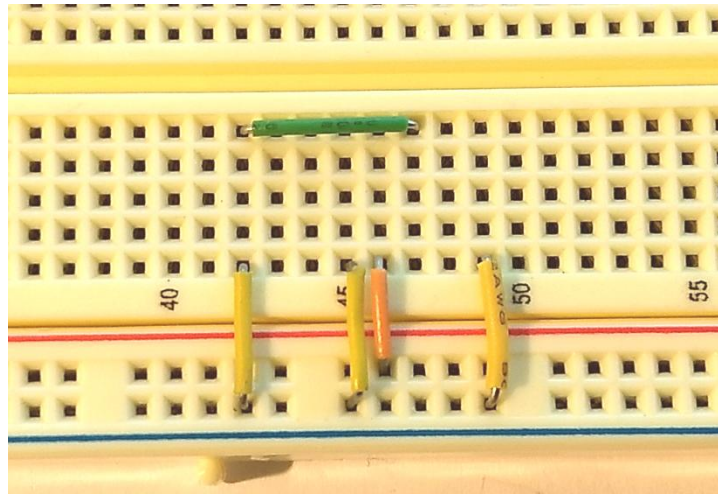


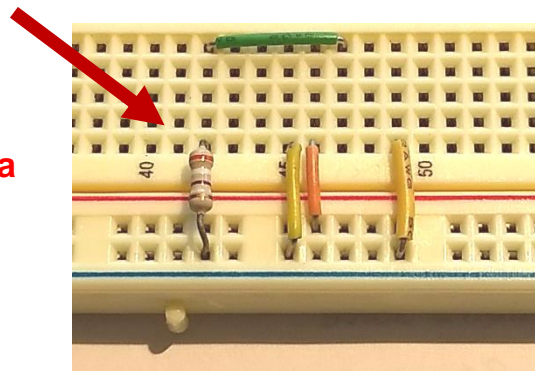
Figure 3. Connecting power and ground for the LCD.

2. Use long orange jump wires to connect port 12 on the Arduino to E48, and port 11 to E50
3. Use four long yellow jump wires to connect port 2 to E58, port 3 to E57, port 4 to E56 and port 5 to E55, as shown in Figure 4.

### If you have WCS's new BC1602HGPLEH\$ LCD

Remove the orange jump wire the red arrow is pointing at from A42 to the ground, and replace it with a 4.7 K ohm resistor.

**Caution:**  
Make this change  
only if your kit has a  
BC1602HGPLEW\$  
LCD.



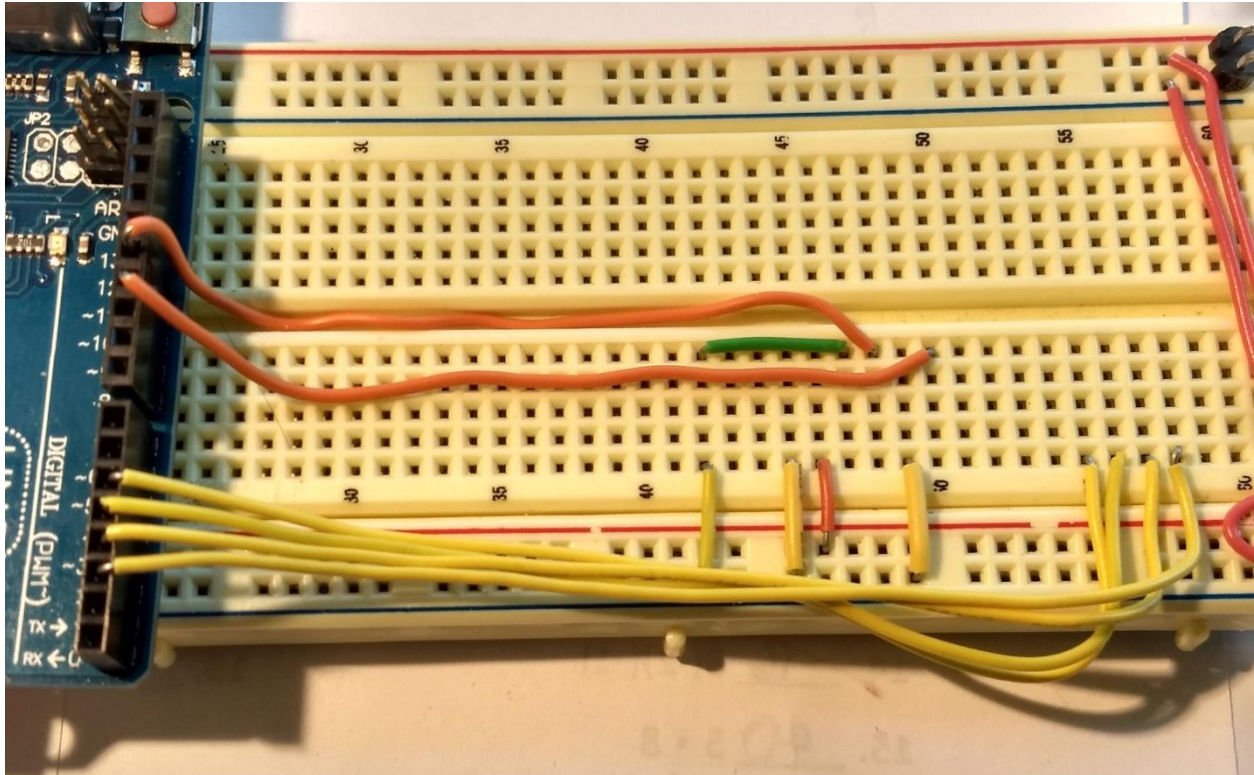


Figure 4. Connecting the Arduino to the breadboard.

Figure 17. XBoard with jump wires for LCD installed.

2. Insert the LCD's pins into holes I17 to I20 and holes I25 to I30, as shown in Figure 18. **(Westminster's new BC1602HGPLEH\$ LCD inserts the same way)**

4. Attach the LCD in row C pushing its pins into holes C45 through C58, as shown in Figure 5. It is very important that the wires line up correctly with the wires on the breadboard. Check to make sure the

right most pin on the LCD is in hole C58, before connecting your Arduino to the USB port.

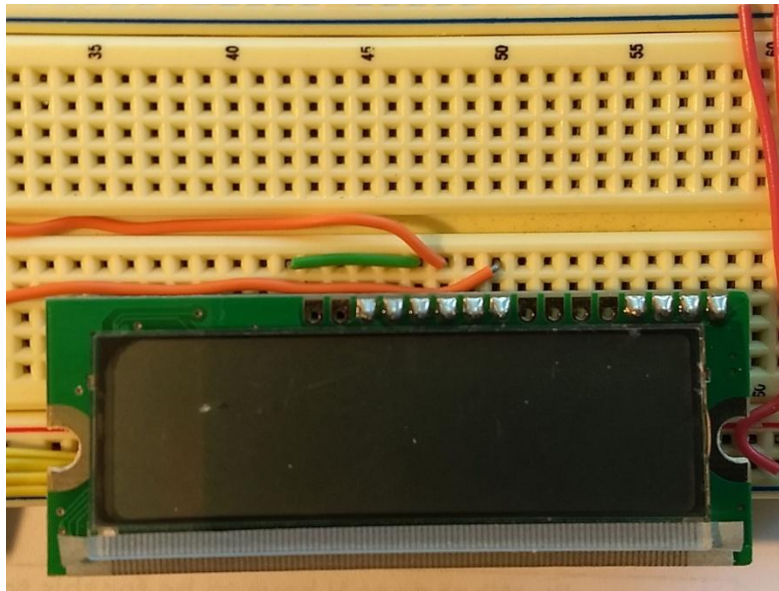


Figure 5. Attaching the LCD.

## Challenge 2: Write Code for the LCD



In this challenge, you will learn to control an LCD, which is just like the ones found in hand-held electronic devices. First, you will learn to send simple lines of text to the LCD. Then, you will learn to position text on the LCD. Finally, you will learn to make text flash and move.

## Challenge 2: Write Code for the LCD

### Sending Text to the LCD

Your first C program will display a simple line of text on the LCD: "Hello World!"

1. Rename your file `text.c`.
2. Modify your code file, as follows:

```
// Hello World Program  
  
#include <LiquidCrystal.h>
```

```
//Tell Arduino which ports to use for the LCD
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {
  lcd.begin(16, 2); //Command to active the LCD
  lcd.print("hello, world!");
}

void loop() { }
```

3. Compile and test your new code.

4. Check out the LCD. It should be displaying "Hello World!"



#### Programming Challenge

Change your code to get the microcontroller to display different text (e.g., your name). Try a few different lines of text.

### Challenge 2: Write Code for the LCD About the LCD's Cursor

Although you cannot see it, the LCD has a cursor, which determines where text will be displayed each time you use the **lcd.print** command. If you look closely at the LCD, you will notice that it has two lines of text, with 16 characters in each row--a total of 32 character positions in all. Figure 6 shows the LCD with a different numeral in each position. Using the **lcd.setCursor(Column,Row)** command, and the **lcd.begin(16, 2)** command that tells the Arduino how many columns and rows your LCD has, you can position the cursor at any one of these 32 positions.



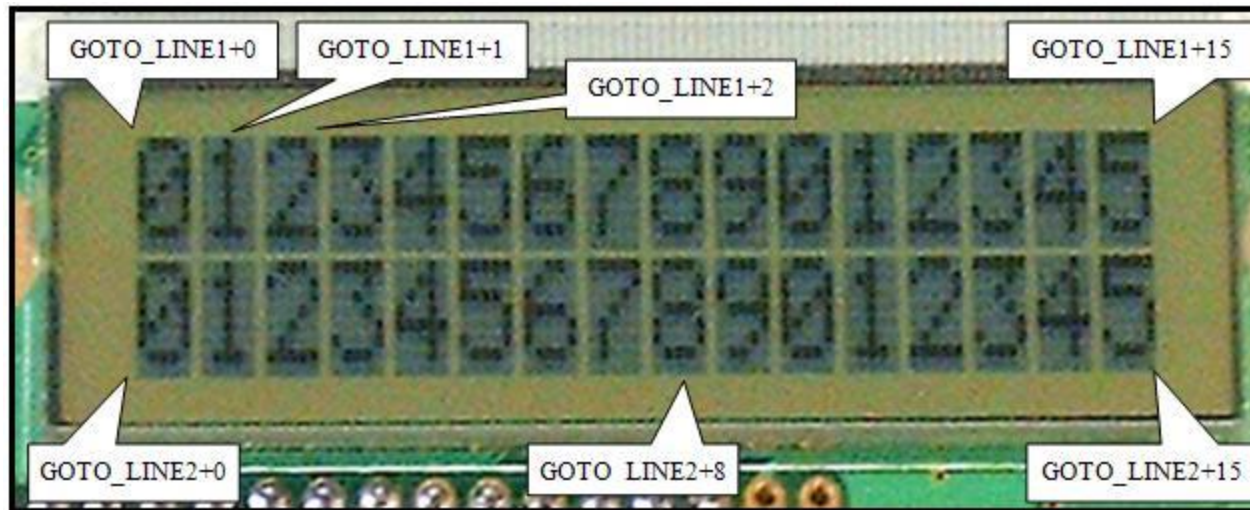


Figure 6. LCD with a few character positions labeled.

## Challenge 2: Write Code for the LCD Positioning Text on the LCD

With an `lcd.setCursor()` statement, you can control the precise location of text on the LCD.

1. Rename your code file `GodRules`.
2. Modify your code file, as follows:

```
// God Rules Program
#include <LiquidCrystal.h>

//Tell Arduino which ports to use for the LCD
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {
  lcd.begin(16, 2); //Command to active the LCD
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  lcd.print("hello, world!");
}

void loop() {

  lcd.setCursor(0, 1); // Go to 0 across and 1 down
                        // note: LCDs count starting from zero
  lcd.print("God rules");
```

```
}
```

### 3. Compile and test your new code.



#### Programming Challenge

Change the values in the `lcd_instruction(GOTO)` statement to reposition "Hello World!" in the lower right hand corner of the LCD. Then, place one line of text on Line 1 of the LCD and another line of text on Line 2 of the LCD.

## Challenge 2: Write Code for the LCD Making Text Flash

By displaying a line of text, clearing the screen, and then displaying the text again, you can make your text flash on and off. With delay statements, you can control the precise rate of flashing.

#### PROGRAMMING NOTE



You can cut and paste text in the Programming Portal, just as you would in a normal word processing program. This feature may come in handy in the next step, since there are many repeated lines of code. (The repeated segment is highlighted.)

1. Rename your code file flash.
2. Modify your code file, as follows:

```
// Flash program

// Flash Program

#include <LiquidCrystal.h>

//Tell Arduino which ports to use for the LCD
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {
  lcd.begin(16, 2); //Command to active the LCD
  // set up the LCD's number of columns and rows:
  lcd.print("hello, world!");
}
```



```

void loop() {

  lcd.setCursor(5, 0); // 6 over, top line
  lcd.print("Jesus");
  lcd.setCursor(4, 1); // 5 over bottom line
  lcd.print("is Lord");
  delay(250);

  lcd.clear();
  delay(250);
}

```

3. Compile your code and download it to the microcontroller.
4. Check out the LCD. "Jesus is Lord" should flash on the screen



#### Programming Challenge

In your flash.c file, change all the delays to 5,000 milliseconds. Try a 5,000-microsecond delay, then a 200-millisecond delay.

## Challenge 2: Write Code for the LCD Making Text Move

Text on the LCD can be made to move to the left or the right, using the following two statements: **lcd.scrollDisplayRight()** and **lcd.scrollDisplayLeft ()** Each of these statements takes all of the text on the LCD and moves it one character to the left or right. By pairing these statements with delay statements, you can make your text appear to scroll--like the "tickers" that are often used to display sports scores, stock prices, and news headlines.

1. Rename your code file **scroll.c**.
2. Modify your code file, as follows:

```

// Scroll Program
#include <LiquidCrystal.h>

//Tell Arduino which ports to use for the LCD
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {

```

```

lcd.begin(16, 2); //Command to active the LCD
// set up the LCD's number of columns and rows:
lcd.print("God is love");
}

void loop() {

  lcd.scrollDisplayRight();
  delay(100);
  lcd.scrollDisplayRight();
  delay(100);
  lcd.scrollDisplayRight();
  delay(100);
  lcd.scrollDisplayRight();
  delay(100);
  lcd.scrollDisplayRight();
  delay(100);

  lcd.scrollDisplayLeft();
  delay(100);
  lcd.scrollDisplayLeft();
  delay(100);
  lcd.scrollDisplayLeft();
  delay(100);
  lcd.scrollDisplayLeft();
  delay(100);
  lcd.scrollDisplayLeft();
  delay(100);

} // end loop

```

3. Compile your code and reprogram the microcontroller.
4. The words should scroll to the right and then to the left. To restart your code, just press the reset button.



### Programming Challenge

Change the `lcd_instruction` statements to move the text in new ways. What happens to text when it moves off the screen? Does it ever come back? Add code to display text on both lines of the LCD. Do both lines move in the same way? Change the `delay_ms` values to speed up or slow down the rate at which the text scrolls.

## Challenge 2: Write Code for the LCD Practice Debugging

Looking for errors in your code is called debugging. Debugging is an important skill. By this point, you have probably already encountered at least one error or "bug" in your code. If you haven't already, you will soon! As practice, take a look at the code below. It has four errors. Can you find them?

```
// Hello World Program

#include <LiquidCrystal.c>

LiquidCrystal lcd(12, 11, 5, 4, 3, 2)

void setup() {
    lcd.begin(16, 2); //Command to active the LCD
    lcd.print("hello, world!");
}

void loop()
    lcd.setCursor(0, 1); // Go to 0 across and 1 down
    lcd.print("God rules");
}
```

## Digital Stopwatch

In this project, you will use the Arduino to create a digital stopwatch. This is the first of many electronic devices that you can build with an Arduino Uno. Your stopwatch will be accurate to within 1/10th of a second and feature Start, Stop, and Reset functions, just like the one in Figure 1.



Figure 1. A digital stopwatch.

The stopwatch project has two challenges. In Challenge 1, you will add buttons, wires, and resistors to your breadboard to control your stopwatch. In Challenge 2, you will write code to make the microcontroller keep track of time and display this information on the XBoard's LCD.

### Challenge 1: Install the Stopwatch Buttons



To begin the stopwatch project, you must add several new components to your board, including two more buttons, two more resistors, and four more jump wires. These components will allow you to control the stopwatch. In this challenge, your task is to gather these components together and install them on your breadboard. If you removed the first button, or never installed it click [here](#) to go to the instructions for installing and programming it.

### Challenge 1: Install the Stopwatch Buttons Collecting Your Components

The components for your stopwatch are listed below and shown in Figure 2:

Part	Quantity	Description
A	2	Buttons
B	2	Resistors (10,000 Ohm)
C	2	Short yellow jump wires
D	2	Flexible jump wires

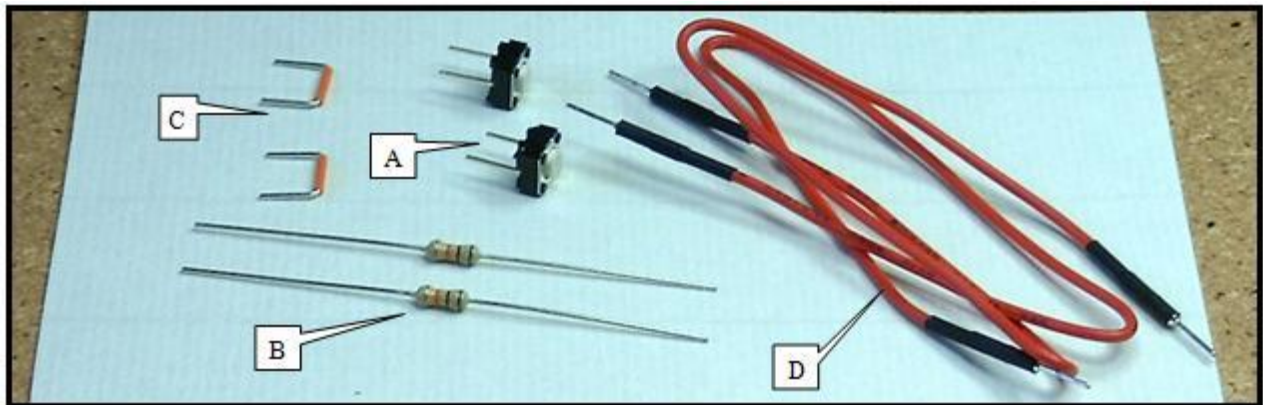


Figure 2. Components for stopwatch.

### Challenge 1: Install the Stopwatch Buttons Adding the Buttons

Two buttons will be needed to control your stopwatch. The installation procedure for each button is exactly the same: a resistor connects one side of the button and one end of the flexible jump wire to ground, and a short orange jump wire connects the other side of the button to power.

1. Make sure that your Arduino is not plugged into your computer, so you will not damage any of its components.
2. Connect the buttons, two flexible jump wires, and two short orange jump wires, and 10,000-Ohm resistors to your breadboard, as shown in Figure 3. Connect holes F46 to F48, and F51 to F53 using switches. Connect J48 and J53 to the ground using 10K ohm resistors. (if you have large resistors use H48 and H53.) Connect rows J46 and J51 to power using yellow jump wires.

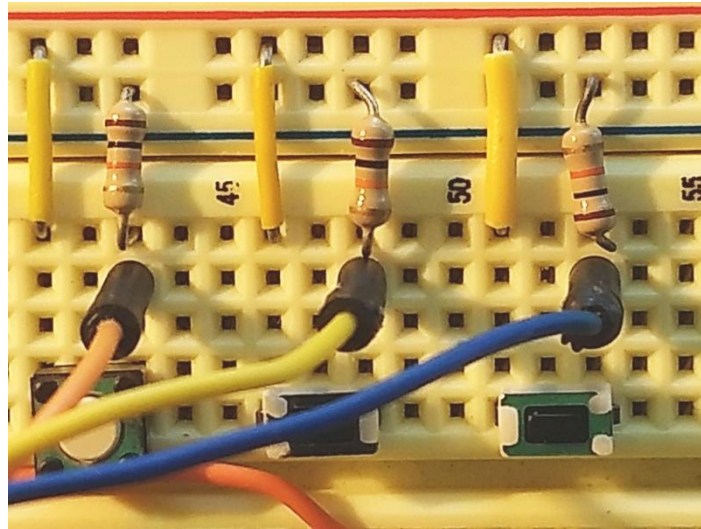


Figure 3. Buttons, resistors and jump wires installed.

### Challenge 1: Install the Stopwatch Buttons Identifying Microcontroller Pins

The loose ends of the flexible jump wires will connect the buttons to pins on the microcontroller. In order to connect these wires properly, you will need to be able to locate specific pins on the microcontroller. Figure 4 shows the Arduino..



Figure 4. Arduino board



## Challenge 1: Install the Stopwatch Buttons

### Connecting the Buttons to the Microcontroller

The buttons for your stopwatch are connected to the microcontroller by the flexible jump wires. The button on the left connects to Port 7, and the button on the right connects to Port 8.

Connect the jump wire from I48 to port 7 on the Arduino, and the wire from I53 to port 8, as shown in Figure 5. NOTE: I43 should already be connected to port 6, a switch and a 10K ohm resistor.

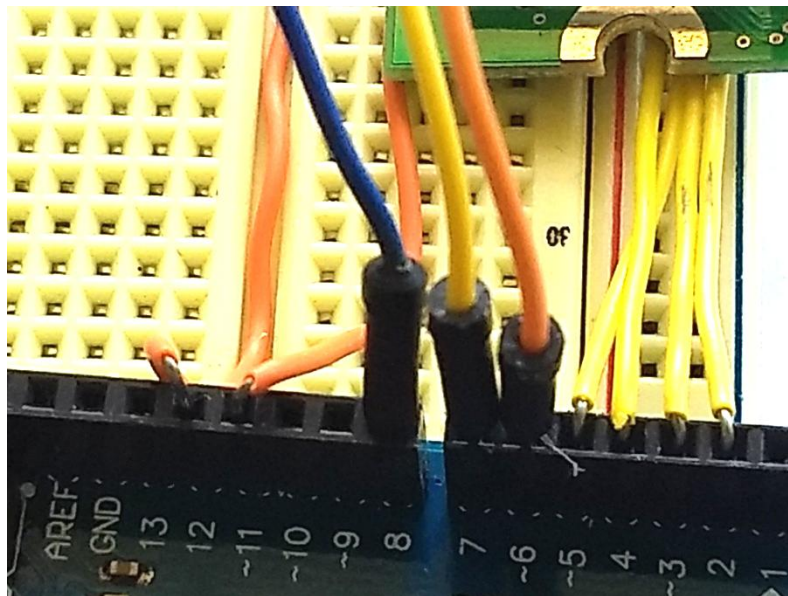


Figure 5. Jump wire connections on the Arduino.

## Challenge 1: Install the Stopwatch Buttons

### Final Hardware Check



Before moving on to the next challenge, take a few minutes to check your hardware set-up. The following checklist will help you ensure that your stopwatch is ready for programming.

- **Buttons:** Left most button pins should be in hole F41 and F42; center button pins should be in hole F46 and F48; right button pins should be in holes F51 and F53.
- **Resistors:** Left resistor should connect hole J43 to ground; center resistor should connect hole J48 to ground; right resistor should connect hole J53 to ground.

- **Yellow jump wires:** Left wire should connect hole A41 to power; center wire should connect hole J46 to power; right wire should connect hole J51 to power.
- **Flexible jump wires:** Left wire should connect hole I43 to Port 6 on the microcontroller; center wire should connect hole I48 to Port 7 on the microcontroller.; right wire should connect hole I53 to Port 8 on the microcontroller.

## Challenge 2: Program the Stopwatch



Now that you have installed the buttons on your breadboard, you can start writing code to turn your Arduino into a digital stopwatch. First, you will learn how to work with **variables**. Then, you will create loops in your code to track minutes, seconds, and fractions of a second. Later, you will program the microcontroller to receive input from the buttons. By the end of this unit, you will have a working stopwatch, accurate to within 1/10th of a second.

### Challenge 2: Program the Stopwatch Working with Variables

In order to keep track of time, you will need to introduce a *variable* into your code. A variable is a letter or word that can be assigned a specific numerical value. In math class, you may have worked with variables in equations, such as  $2x = 6$ , or  $x^2 = 9$ . Using C, you can program the microcontroller to store one or more variables, assign specific values to variables, change the values of variables, perform mathematical operations with variables, and display the values of your variables on the LCD.

Any time you want to have a variable in your code, you must declare the variable. Declaring the variable assigns a name to it, and tells the microcontroller what type of values the variable will have. For now, all of your variables will be integers (i.e. whole numbers, such as 0, 1, 2, etc...). To declare an integer variable, you will use an **int statement**, and to display the variable's value on the LCD, you will use an `lcd.print` statement. For example, the statement `int x;` declares a integer variable called x, and the statement `lcd.print(x)` displays the value of x on the LCD.

C has many types of variables besides integers. Other types include *char* variables, *float* variables, and *long* variables. You will learn more about

these variable types later. For now, any time you want to declare a variable, just use an *int* statement.

## Challenge 2: Program the Stopwatch

### Declaring a Variable in Your Code

1. Open the Programming Portal and create a new code file. Save your code file as variable.

2. Enter the following code:

```
// Variable Program

#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2); // define ports

int x; // x is a variable that can be set to any number

void setup() {
  lcd.begin(16, 2); // Activate LCD as 16 across by 2 down
  x=100;           // Change the value of x to 100
  lcd.print (x);
} // end setup

void loop() {
} // end loop
```

3. Compile and test your new code.



#### Programming Challenge

Assign a different value to x in your code and reprogram the microcontroller. What's the largest value you can assign to x? Can you use negative numbers? Use a different name for your variable and reprogram the microcontroller. (Hint: You can use any letter you want, or you can use a whole word.)

## Challenge 2: Program the Stopwatch

### Creating a While... Loop

In order to track time with your x variable, you need to make x increase in value over time. To do this, you will take advantage of the **while loop** in your code. A loop is a programming structure that causes a certain section of code to be

repeated over and over. One common type of loop in C is the *while ...* loop. The *while ...* loop tells the microcontroller to repeatedly execute a segment of code while a certain condition is met. If the *while* command is followed by a statement that is always true, then the *while ...* loop will repeat indefinitely. This is what the *loop()* function in your program does automatically. We will make our own loop in the next program.

**While loop**  
**While (condition){ ... }**  
**Example:**  
**while (x<300){**  
**x=x+1; }**

1. Using the **Save As** command, rename your code file **loop**.

2. Modify your code file, as follows:

```
// Loop Program

#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2); // define ports

int x; // x is a variable that can be set to any number

void setup() {
  lcd.begin(16, 2); //Activate LCD as 16 across by 2 down
  x=0;             // Change the value of x to 100
  while (1==1){
    lcd.clear();   // Clean off the information on the LCD
    lcd.print (x);
    delay(100);    // wait one tenth of a of a second
    x=x+1;         // Make x equal to what is was before plus 1
  }               // end while loop
}                // end setup

void loop() {
}                // end loop
```

3. Compile and test your new code.

## Challenge 2: Program the Stopwatch About Logical Operators

In C, the equals sign (=) and the double equals sign (==) have different uses. The equals sign (=) sets a variable equal to a specific value. For example, the statement `x=0` sets the value of `x` equal to 0. The double equals sign (==) compares two values, and determines whether the statement is true or

**Tenths**  
Show your work  
to the instructor  
for a grade.

false. For example, the statement (1==1) is always true. The statement (x==1) could be true or false, depending on the value of x. With other statements, you can make other comparisons, determining if one value is greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=), or not equal to (!=) another value. These types of symbols are called *logical operators*.

## Challenge 2: Program the Stopwatch Creating a Finite While... Loop

In your final stopwatch code, the variable x will represent 1/10ths of a second. You will not want this variable to increase indefinitely; instead, you will want it to count from 0 to 9 once every second and then reset to 0. In order to do this, you will need to create a finite loop in your code. Right now, your while loop is tied to a condition that is always true: (1==1). This creates an infinite loop. By tying your while loop to a condition that can be either true or false, you can create a finite loop. In this section, you will tie the while loop to the value of x, so that the loop repeats when x is less than or equal to 9 and stops when x reaches 9.

1. Using the Save As command, rename your code file endwhile.

2. Modify your main function, as follows:

```
// EndWhile Program

#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

int x;

void setup() {
  lcd.begin(16, 2);
  x=0;
  while (x<=9){ // Repeat as long as x is less than or equal to 9
    lcd.clear();
    lcd.print (x);
    delay(100);
    x=x+1;
  } // end while loop
} // end setup

void loop() {
} // end loop
```

### 3. Reprogram the microcontroller and observe the LCD.

#### PROGRAMMING NOTE



The `delay()` statement in your *while ...* loop makes each cycle of the loop last approximately 100 1/1000th (or 1/10th) of a second. This means each change in the value of *x* takes about 1/10th of a second, and overall, it takes a total about one second for *x* to increase from 0 to 9. (Actually, this takes a little longer, since every line of code takes some time to execute. Later in this unit, you will adjust the delay statement to fine tune your stopwatch.)



#### Programming Challenge

Change your code so that *x* counts from 0 to 200 on the LCD. Change your code so that *x* counts from 10 to 50 on the LCD. Change your code so that *x* counts from 10 to 1000, increasing by 5 each time through. Try making *x* count down from 1000 to 0.

### Challenge 2: Program the Stopwatch About For... Loops

Since you will frequently want to create loops that repeat a finite number times, C has another type of loop designed specifically for this purpose--the *for ...* loop. The *for ...* loop repeats a section of code a specified number of times, represented by a variable. Figure 7 shows the format for setting up a *for ...* loop.

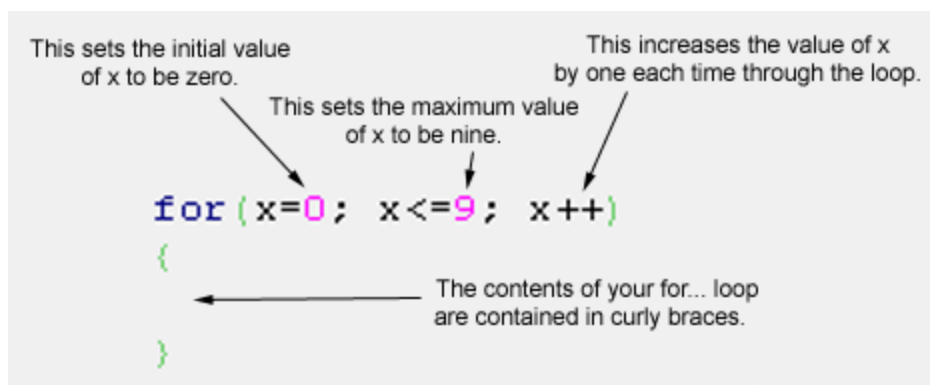


Figure 7. Setting up a *for ...* loop.

### Challenge 2: Program the Stopwatch Adding a For... Loop to Your Code



1. Using the Save As command, rename your code file forloop.
2. Modify your code file, as follows:

```
// EndWhile Program

#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

int x;

void setup() {
  lcd.begin(16, 2);

  for (x=0;x<=9;x++){
    lcd.clear();
    lcd.print (x);
    delay(100);
  } // end for loop
} // end setup

void loop() {
} // end loop
```

3. Reprogram the microcontroller and observe the LCD.



#### Programming Challenge

Change your code so that the final number displayed on the LCD is larger or smaller than 9. Next, modify your code so that x starts at 99 and decreases to 10. (Hint: In the line setting up the for... loop, what needs to change so that x starts at 99 and decreases each time, ending at 10?)

### Challenge 2: Program the Stopwatch Adding a For... Loop for Seconds

What you have created so far is a digital counter, capable of counting up to 9 1/10ths of a second. As a next step, you will need to declare a separate variable to track seconds. This variable will increase by 1 every time x reaches 9. In order to do this, you will create a new *for ...* loop for your seconds variable, and nest, your existing *for ...* loop inside of the new loop. Since there are 60 seconds in a minute, your seconds variable should increase from 0 to 59.

1. Using the Save As command, rename your code file secondloop..
2. Enter the following code into the Editor window:

```
// SecondLoop Program

#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

int x;
int sec=0; // Start sec with a value of 0

void setup() {
  lcd.begin(16, 2);

  for(sec=0;sec<=59;sec++){
    for (x=0;x<=9;x++){

      lcd.clear();
      lcd.print(sec); // Display seconds
      lcd.print("."); // Decimal point
      lcd.print (x); // Display tenths of seconds
      delay(100); // Wait one tenth of a second

    } // end for x loop
  } // end for sec loop
} // end setup

void loop() {
} // end loop
```

3. Reprogram the microcontroller and observe the LCD.

## **Challenge 2: Program the Stopwatch**

### **Adding a For... Loop for Minutes**

By adding a third loop to your code, you can track minutes as well as seconds. Just as you did for the seconds loop, you will have to declare a new variable to track minutes, and nest the seconds loop and the x loop inside your new loop.

1. Using the Save As command, rename your code file minuteloop.
2. Modify your code file, as follows:

### // MinuteLoop Program

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
```

```
int x=0;
int sec=0;
int min=0;
```

```
void setup() {
  lcd.begin(16, 2);
```

```
  for (min=0;min<=59;min++){
    for(sec=0;sec<=59;sec++){
      for (x=0;x<=9;x++){

        lcd.clear();
        lcd.print(min);  // minutes
        lcd.print(":");
        lcd.print(sec);  // seconds
        lcd.print(".");
        lcd.print (x);   // tenths
        delay(100);

      }    // end for x loop
    }    // end for sec loop
  }    //end for min loop
}    // end setup
```

```
void loop() {

} // end loop
```

### 3. Compile and test your new code.



#### Calibration Exercise

With the addition of each new line of code, you increase the amount of time required for each cycle of your for... loops. As a result, at this stage, your stopwatch probably is not keeping very accurate time. Fortunately, you can easily fine tune your stopwatch by adjusting the delay statement at the bottom of the last for... loop. Making the delay shorter will speed up

your stopwatch, and making the delay longer will slow down your stopwatch. Test the accuracy of your stopwatch by using a wristwatch or wall clock. Is your stopwatch fast or slow? By changing the delay statement in your code, adjust your stopwatch for better accuracy. (Hint: With a shorter delay, your stopwatch will run faster.)

## Challenge 2: Program the Stopwatch

### About the Stopwatch Buttons

Look closely at the flexible jump wires connecting the buttons to the microcontroller. When the buttons are not pressed, Ports B4 and B5 are tied to ground through a resistor. When the buttons are pressed, Ports B4 and B5 are tied to power. To the microcontroller, this means that when the buttons are not pressed, Ports B4 and B5 are LOW, and when the buttons are pressed, Ports B4 and B5 are HIGH. Table 1 summarizes this information.

Table 1. Relationship between buttons and Ports 7 and 8.

	Center Button	Right Button
Not Pressed	Port 7 == LOW	Port 8 == LOW
Pressed	Port 7 == HIGH	Port 8 == HIGH

Ports on the microcontroller can be connected to *input* devices, such as buttons and sensors, or *output* devices, such as the LCD or a speaker. In your code, you need to specify whenever a port is going to be used as an input or an output. This is done with a *pinMode* statement. For example, the statement `pinMode(7, INPUT);` sets up Port 7 as an input. The statement `pinMode(7, OUTPUT);` sets up Port 7 as an output. `TRISB5 = OUTPUT` sets up Port 5 as an output.

#### PROGRAMMING NOTE



The words **HIGH**, **LOW**, **INPUT**, and **OUTPUT** are shorthand for values. HIGH is equal to 1, and LOW is equal to 0. INPUT is equal to 1, and OUTPUT is equal to 0. This shorthand is used interchangeably in the Machine Science code examples, but it is easy to remember. Just think: 1 = **I** for **I**np, and 0 = **O** for **O**utput.

## Challenge 2: Program the Stopwatch

### Programming the Start Button

As a first step, you will program the center stopwatch button as a start button. In order to do this, you will have to set up port 7 as an input, using a pinMode statement. Then you will need to introduce a while... statement that holds the microcontroller in a loop until the left button is pressed.

1. Using the Save As command, rename your code file startbutton.

2. Modify your code file, as follows:

```
// StartButton Program

#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

int x=0;
int sec=0;
int min=0;

void setup() {
  lcd.begin(16, 2);
  pinMode(7, INPUT); // Button

  while (digitalRead(7)==0);    //Wait until button press

  for (min=0;min<=59;min++){
    for(sec=0;sec<=59;sec++){
      for (x=0;x<=9;x++){

        lcd.clear();
        lcd.print(min);  // minutes
        lcd.print(":");
        lcd.print(sec);  // seconds
        lcd.print(".");
        lcd.print (x);   // tenths
        delay(100);

      }    // end for x loop
    }    // end for sec loop
  }    //end for min loop
}    // end setup

void loop() {
} // end loop
```

### 3. Compile and test your new code.

## Challenge 2: Program the Stopwatch Programming the Stop Button

With the left stopwatch button serving as a Start button, you are ready to program the right stopwatch button to serve as a Stop button. This will involve adding two new types of statements to your code--an **if** statement and a **break** statement. An *if ...* statement tells the microcontroller to execute a specific piece of code if a certain condition is met. A *break* statement tells the microcontroller to immediately break out of whatever loop it is currently executing.

1. Using the Save As command, rename your code file stopbutton.

2. Modify your main function, as follows:

```
// StopButton Program

#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

int x=0;
int sec=0;
int min=0;
int rbutton=0; // Stop button variable

void setup() {
  lcd.begin(16, 2);
  pinMode(7, INPUT); // Start - Center Button
  pinMode(8, INPUT); // Stop - Right Button

  while (digitalRead(7)==0);    //Wait until button press

  for (min=0;min<=59;min++){
    for(sec=0;sec<=59;sec++){
      for (x=0;x<=9;x++){

        lcd.clear();
        lcd.print(min);  // minutes
        lcd.print(":");
        lcd.print(sec);  // seconds
        lcd.print(".");
        lcd.print (x);   // tenths
```



```

    delay(100);
    rbutton = digitalRead(8); // If Stop button press – breakout
    if (rbutton==1) break; // Stop button- get out of x loop
  } // end for x loop
  if (rbutton==1) break; // Stop button- get out of sec loop
} // end for sec loop
if (rbutton==1) break; // Stop button- get out of min loop
} //end for min loop
} // end setup

void loop() {
} // end loop

```

3. Compile and test your new code.

## **Challenge 2: Program the Stopwatch** **Adding a Reset Function**

In this step, you will program the right stopwatch button to reset the stopwatch display. To do this, you will add a *while...* loop and another gateway *while...* statement, as well as `lcd.clear()` and `lcd.print` statements. Use the left most button, attached to port 6 to clear the display

1. Using the Save As function, rename your code file `resetbutton`.

2. Modify your main function, as follows:

```

// ResetButton Program

#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

int x=0;
int sec=0;
int min=0;
int rbutton=0; // Stop button

void setup() {
  lcd.begin(16, 2);
  pinMode(6, INPUT); // Start - Left Button
  pinMode(7, INPUT); // Start - Center Button
  pinMode(8, INPUT); // Stop - Right Button

```

```

while(1==1){//Gateway loop

//cbutton = digitalRead(7);// Start button = center button
while (digitalRead(7)==0);    //Wait until start button press

for (min=0;min<=59;min++){
  for(sec=0;sec<=59;sec++){
    for (x=0;x<=9;x++){

      lcd.clear();
      lcd.print(min);  // minutes
      lcd.print(":");
      lcd.print(sec);  // seconds
      lcd.print(".");
      lcd.print (x);   // tenths
      delay(100);
      rbutton = digitalRead(8); //Stop button
      if (rbutton==1) break;
    }    // end for x loop
    if (rbutton==1) break;// Stop button- get out of sec loop
  }    // end for sec loop
  if (rbutton==1) break;// Stop button- get out of min loop
}    //end for min loop

  while (digitalRead(6)==0); //wait for reset press - left button

  lcd.clear();
  lcd.print("0:0.0");

} // end Gateway Loop
}    // end setup

void loop() {
} // end loop

```

3. Compile and test your new code.

## **Challenge 2: Program the Stopwatch** **Fixing the Number Display**

**Stop Watch**  
Show your work  
to the instructor  
for a grade.

If you pay careful attention, you will notice an error on the stopwatch display during the first 10 seconds of each minute single digits print instead of double digits. For example, in the third second, your display shows 0:3.0 rather than

0:03.0. To fix this problem, you need add an *if* statement that prints a zero, when the number of seconds is less than 10. You will need to fix the minute display, as well. Additionally, after the `lcd.clear` statement, you will need to print "0:00.0", instead of "0:0.0".

1. Using the Save File button, rename your code file `twodigits.c`.

2. Insert the *if* statements in your code file, following the example below, and fix the reset `lcd.print` statement.

```
if (min<10) lcd.print ("0");
```

3. Compile and test your new code.

## Challenge 2: Program the Stopwatch Finishing Touches

With a few finishing touches, your stopwatch will work even better.

1. With all of your new code, your stopwatch may have lost some accuracy. Adjust the delay statement in the central for...loop to fine tune the watch's accuracy.

2. Add an `lcd.print` statement near the top of your code file, so that the stopwatch displays "00:00.0" before you press the Start button.

3. Using `lcd_instruction(GOTO)` and `lcd.print` statements, personalize your stopwatch by adding text to the stopwatch display. For example, you could put your name on Line 1 and move the time display to Line 2 of the LCD. This [link](#) will take you to a page that explains LCD commands.

## Challenge 2: Program the Stopwatch Cleaning Up

Congratulations! You have built a working stopwatch, using the same software and parts that professional engineers use. Before moving on to the next unit, take a little time to clean up your breadboard and store the stopwatch components.

**Stopwatch +**  
Show your work to  
the instructor for  
EXTRA CREDIT.

1. Remove the two flexible jump wires from the breadboard.

2. Remove the jump wire and resistor for each button.

**3. Remove the two buttons from the breadboard.**

## Digital Thermometer

This project will transform your Atmega into a working digital thermometer. Your thermometer will display temperature in units of Celsius and Fahrenheit, just like the ones shown in Figure 1.



Figure 1. Digital thermometers.

Like the stopwatch project, the thermometer project has two challenges. Challenge 1 is a *hardware* challenge--adding jump wires and a temperature sensor to the breadboard. Challenge 2 is a *software* challenge--writing code to make the microcontroller keep track of temperature and display this information on the LCD.

### Challenge 1: Install the Temperature Sensor



As a first step, you need to add a temperature sensor, another resistor, and a few more jump wires to your Arduino. Be careful when installing the temperature sensor, since it can be easily damaged if it is oriented incorrectly. Once you have completed this challenge, you will be ready to write code to gather temperature data from the sensor.

### Challenge 1: Install the Temperature Sensor Collecting Your Components

The components for your digital thermometer are listed below and shown in Figure 2:

Part	Quantity	Description
A	1	Analog voltage temperature sensor (TMP36GT9)

B	1	Short orange jump wire
C	1	Short yellow jump wire
D	1	Flexible jump wire

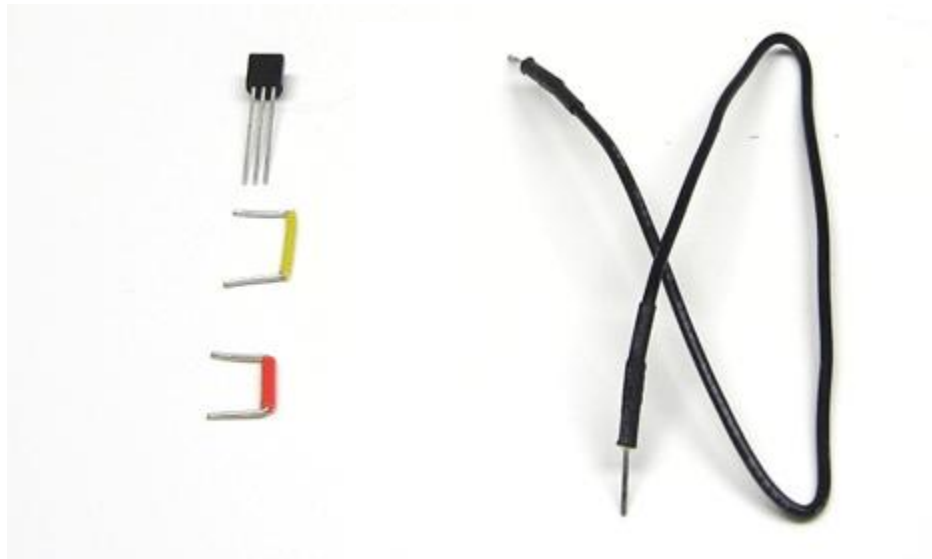


Figure 2. Components for the digital thermometer.

### Challenge 1: Install the Temperature Sensor

#### Adding the Temperature Sensor

The temperature sensor is a high-precision sensor that is accurate to within  $\pm 0.5^{\circ}\text{C}$ , with a range of  $-55^{\circ}\text{C}$  to  $+125^{\circ}\text{C}$ . The sensor has three prongs. The outer prongs connect to power and ground via short jump wires, and the center prong connects both to the microcontroller and to power, via a resistor.

1. Make sure that the power switch on your battery pack is in the OFF position.
2. Connect hole I31 to the ground with an orange jump wire, then connect hole I29 to power with a yellow jump wire, as shown in Figure 3
2. Connect the flexible jump wire to the Atmega, as shown in Figure 3.



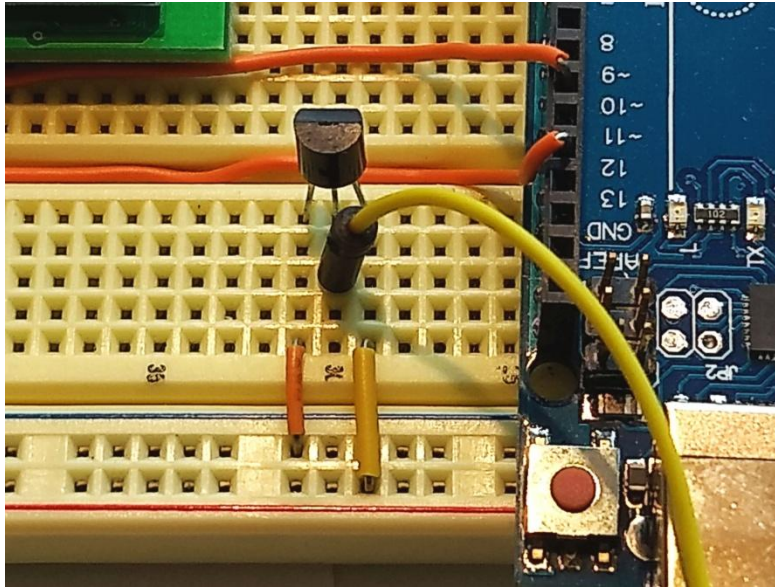


Figure 3. Sensors and jump wires installed.

3. Connect the temperature sensor to the breadboard, aligning its three pins with holes F29, F30, and F31. **NOTE: Be sure to orient the flat face of the temperature sensor as shown, away from the jumpers.**
4. Connect the loose end of the flexible jump wire to Port A2 on the microcontroller. **NOTE: If you need help finding Port A2, refer to the [Arduino Quick Reference](#) page at the end of this document.**

## Challenge 2: Collect Data from the Sensor



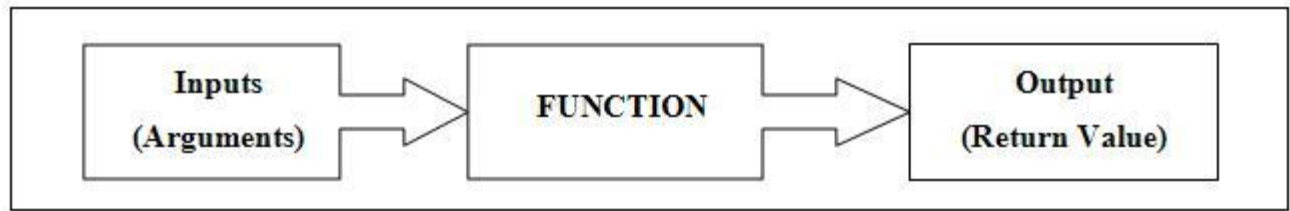
With the temperature sensor installed, you are ready to write code to turn your Atmega into a working digital thermometer. First, you will learn how to call an existing **function** that gathers data from the temperature sensor. Then, you will write your own function to convert Celsius temperatures into Fahrenheit temperatures.

## Challenge 2: Collect Data from the Sensor

### Understanding Functions

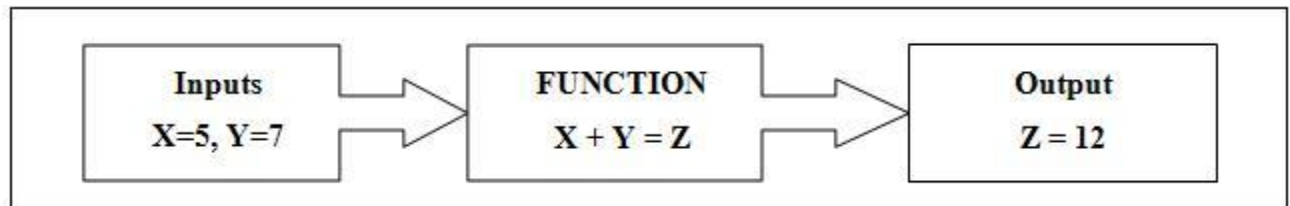
Functions are one of the most powerful tools in C. With a function, you can group several lines of code together, give them a name, and then execute them later on with a single line of code. Executing a function is usually referred to as *calling* the

function. Sometimes, functions have inputs (called *arguments*) and an output (also called a *return value*). The input-output feature of functions is shown in Figure 4.



**Figure 4. A function with inputs (arguments) and an output (return value).**

For example, let's say you wanted to create a function that adds two numbers together. The inputs (or arguments) would be the two numbers you want to add, the function would be a mathematical equation adding the two numbers together, and the output (or return value) would be the result of the equation. Figure 5 illustrates how this function would work.



**Figure 5. A function that adds two numbers together.**

In C code, the function shown in Figure 5 would contain the statements shown in Figure 6.

```
int add(int x, int y)
{
    int z;
    z = x + y;
    return(z);
}
```

This says the function has a return value.

This is the function's name.

These are the function's two inputs: x and y.

This declares a variable called z.

This sets z equal to the sum of x and y.

This returns the value of z.

**Figure 6. A C function that adds two numbers together.**

After a function has been defined, it can be called at any time with a single line of code. For example, the code shown in Figure 7 would add the numbers 5 and 7 together and display the result on the LCD.

```
void setup(){
  Serial.begin(9600);
}

void loop() {
  int i = 2;
  int j = 3;
  int k;

  k = myMultiplyFunction(i, j); // k now contains 6
  Serial.println(k);
  delay(500);
}

int myMultiplyFunction(int x, int y){
  int result;
  result = x * y;
  return result;
}
```

**Figure 7. A code file calling the add function.**

To help get you started, a number of functions have been defined for you. In fact, you have already been using some of these functions, without even realizing it. The statements `lcd_text`, `lcd_decimal`, `delay_ms`, and `delay_us` are all functions that were developed to make writing your first programs a little easier. These functions are defined in the files called **mxapi.h** and **mxapi.c**, which you have included at the top of all of your code files.

## **Challenge 2: Collect Data from the Sensor**

### **Calling a Function in Your Code**

Another function that has already been defined for you is called `temp_sensor_read`. The `temp_sensor_read` function gathers data from the XBoard's temperature sensor and sends back a Celsius temperature as a return value. In this section, you will call the `temp_sensor_read` function and display the return value on the XBoard's LCD.

1. Open the Programming Portal and create a new code file. Save your code file as `celsius.c`. (NOTE: Remember to include the `.c` filename extension.)

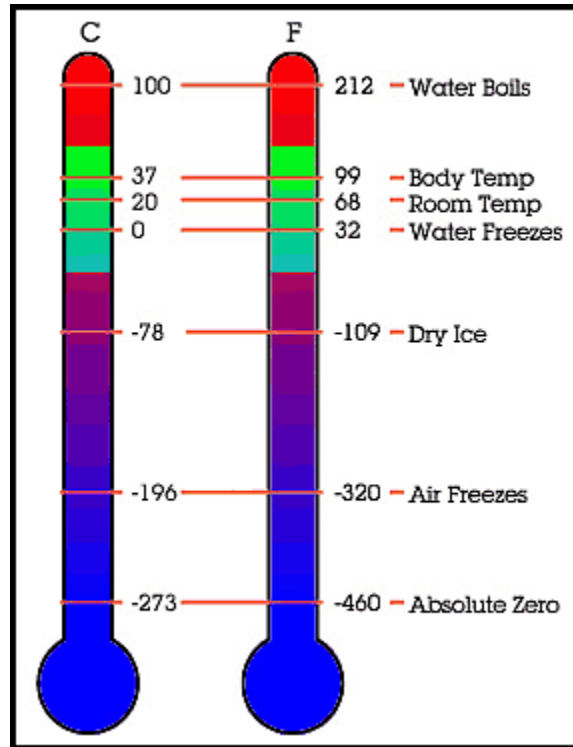
2. Enter the following code:

```
#include "mxapi.h"
void main(void)
{
    int temp_c=0;
    lcd_init();
    while(1==1)
    {
        temp_c=temp_sensor_read(TEMPERATURE);
        lcd_decimal(temp_c);
        delay_ms(200);
        lcd_instruction(GOTO_LINE1+0);
    }
}
```

3. Compile and test your new code.

## Challenge 2: Collect Data from the Sensor About Temperature Scales

The `temp_sensor_read` function's return value is a *Celsius* temperature. Celsius is a temperature scale based on the physical properties of water: 0 degrees Celsius (0°C) is the temperature at which water freezes, and 100°C is the temperature at which water boils. Figure 8 shows the relationship between Celsius and the more familiar *Fahrenheit* temperature scale.



**Figure 8. Thermometers showing Celsius and Fahrenheit temperatures.**

You can convert Celsius temperatures to Fahrenheit temperatures using the following formula, where temp\_c is a Celsius temperature and temp\_f is a Fahrenheit temperature.

$$\text{temp\_f} = ((9 * \text{temp\_c}) / 5) + 32$$

For example, to convert 0°C to Fahrenheit, your calculations would go as follows:

$$\begin{aligned} \text{temp\_f} &= ((9 * 0) / 5) + 32 \\ \text{temp\_f} &= 0 + 32 \\ \text{temp\_f} &= 32 \end{aligned}$$

Likewise, you can convert Fahrenheit to Celsius, using the following formula:

$$\text{temp\_c} = ((\text{temp\_f} - 32) * 5) / 9$$

For example, to convert 86°F to Celsius, your calculations would go as follows:

$$\begin{aligned} \text{temp\_c} &= ((86 - 32) * 5) / 9 & \text{temp\_c} &= ((54) * 5) / 9 \\ \text{temp\_c} &= (270) / 9 \\ \text{temp\_c} &= 30 \end{aligned}$$

## Challenge 2: Collect Data from the Sensor

### Writing a Function to Convert Celsius to Fahrenheit

Using a function of your own, you can have the microcontroller do the math to convert Celsius temperatures into Fahrenheit temperatures. Remember that every C function has four parts, which are explained below and shown in Figure 9:

- *A name.* In this case, your function will be called *fahrenheit\_conversion*.
- *Arguments.* These are the inputs that you pass to the function. You will pass one variable to your *fahrenheit\_conversion* function--an integer called *c\_temp*.
- *A return value.* This is the output of the function--a number that your function will return when it is called. Your function will return a Fahrenheit temperature called *f\_temp*.
- *Contents.* The contents of the function are contained in curly braces and tell the microcontroller what to do when the function is called.

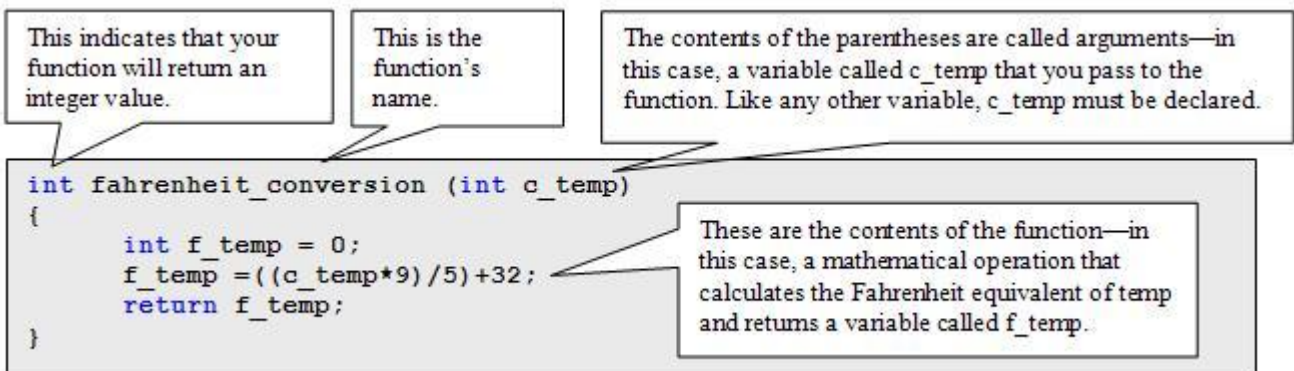


Figure 9. Setting up a function in C.

1. Using the Save As command, rename your code file `fahrenheit.c`.
2. Modify your code file, as follows:

```
#include "mxapi.h"

int fahrenheit_conversion(int c_temp)
{
    int f_temp=0;
    f_temp=((c_temp*9)/5)+32;
    return f_temp;
}

void main(void)
```



```

{
  int temp_c=0;
  int temp_f=0;
  lcd_init();
  while(1==1)
  {
    temp_c=temp_sensor_read(TEMPERATURE);
    lcd_decimal(temp_c);
    temp_f=fahrenheit_conversion(temp_c);
    lcd_instruction(GOTO_LINE2+0);
    lcd_decimal(temp_f);
    delay_ms(200);
    lcd_instruction(GOTO_LINE1+0);
  }
}

```

### 3. Compile and test your new code.

## Challenge 2: Collect Data from the Sensor Finishing Touches

With a few finishing touches, your display will be easier to read.

1. Add an *lcd\_text* statement to your code file, so that the thermometer displays "C Temp = " before the Celsius temperature on line 1 of the LCD.
2. Using an *lcd\_text* statement, display "F Temp = " before the Fahrenheit temperature on line 2 of the LCD.

## Challenge 2: Collect Data from the Sensor Cleaning Up

Congratulations! You have built a digital thermometer, using the same parts and code that professional engineers use. Before moving on to the next project, take some time to clean up the breadboard and store the thermometer components.

1. Remove the flexible jump wire from breadboard.
2. Remove the resistor and jump wires for the temperature sensor.
3. Remove the temperature sensor.

## Unit 1: Making Sounds

Digitally synthesized music is everywhere. Nearly every popular song on the radio today features tracks produced by a digital synthesizer, such as the one shown in Figure 1. But you don't need an expensive keyboard to make digital music--any device with a speaker and a microcontroller can produce a range of musical melodies. In fact, you hear these digitally synthesized melodies every day, whenever someone's cell phone rings. Inside every cell phone is a tiny speaker, which is connected to a microcontroller. The microcontroller is programmed to generate the ring tone melody when the phone receives a call.



Figure 1. A synthesizer keyboard (left) and a cell phone (right).

In Unit 1, you will take the first steps towards transforming your XBoard into a digital music synthesizer. Unit 1 has two challenges. In Challenge 1, you will add a speaker to the XBoard. In Challenge 2, you will program the speaker to make some basic sounds.

### Challenge 1: Install the Speaker



Challenge 1 is a simple hardware task--adding a speaker to the XBoard, using a two-prong connector and two jump wires.

### Challenge 1: Install the Speaker Collecting Your Components

In order to complete this challenge, you will need the following components (shown in Figure 2):

Part	Quantity	Description
------	----------	-------------

A	1	Piezo-electric speaker
B	1	Jump wire (yellow)
C	1	Bent connector (two-prong)
D	1	Flexible jump wire

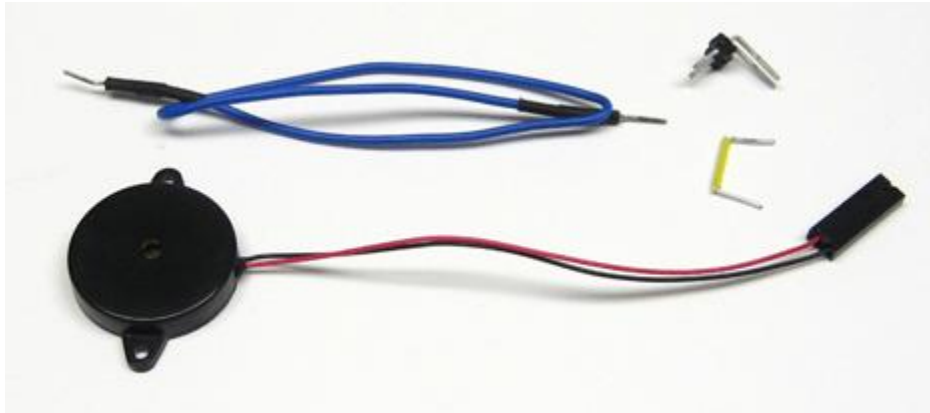
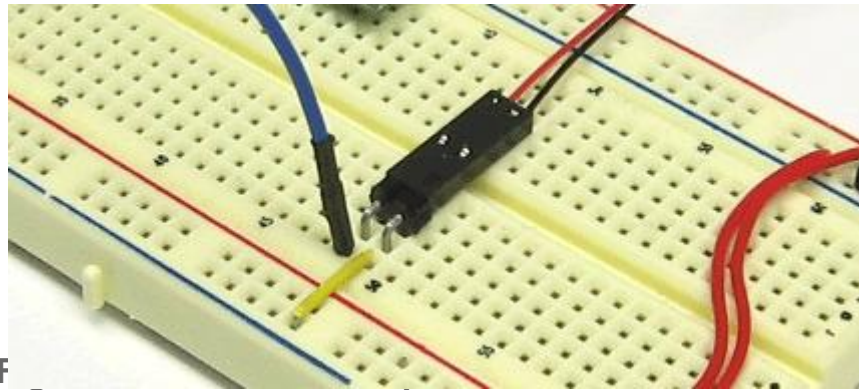


Figure 2. Components for Digital Music Synthesizer project.

### Challenge 1: Install the Speaker Adding the Piezo-Electric Speaker

The piezo-electric speaker should be connected to Port C2 on the microcontroller and to ground, as shown in the video to the right and in Figure 3.

1. Using a short yellow jump wire, connect hole A49 to ground.
2. Attach a bent two-prong connector to the speaker leads.
3. Insert the bent two-prong connector into the board, in holes B48 and B49. Make sure the black speaker lead aligns with the yellow jump wire. *NOTE: You may have to turn the two-prong connector over in order to get it to fit as shown in Figure 3.*
4. Using a flexible jump wire, connect hole A48 to Port C2 on the microcontroller (pin 17). *NOTE: If you need help finding Port C2, refer to the microcontroller pin diagram in the Microcontroller Quick Reference document.*



## Challenge 2: Produce Sounds from the Speaker



Challenge 2 is a straightforward programming task--sending pulses of electricity to the speaker to produce sounds. First, you will generate simple clicking sounds. Then, you will write a loop to produce a sustained tone. Finally, you will create a *for...* loop to produce a tone of a finite duration.

## Challenge 2: Produce Sounds from the Speaker About the Piezo-Electric Speaker

The speaker you will use in this project is called a piezo-electric speaker. The key component in the speaker is a piezo-electric element--a material that bends slightly when you supply electricity to it. If the electricity is sent in pulses, the element vibrates back and forth. Just like the movement of a saxophone reed or a guitar string, this vibration produces audible sound waves in the air. To make the piezo-electric element vibrate, you need to send pulses of electricity to the speaker, alternating between 0 volts and 5 volts. Figure 4 shows how the voltage signals make the piezo-electric element move.

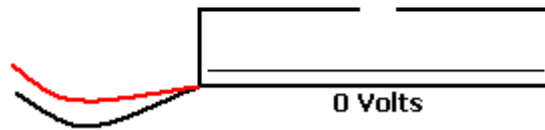


Figure 4. The piezoelectric element moving back and forth to create sounds.

## Challenge 2: Produce Sounds from the Speaker Making a Click

Your first programming task is to get the speaker to make a clicking sound by raising and then lowering the voltage on Port C2 of the microcontroller. Figure 5 shows the voltage pattern that your code should create--5 volts for 1 millisecond, and then 0 volts for 1 millisecond.

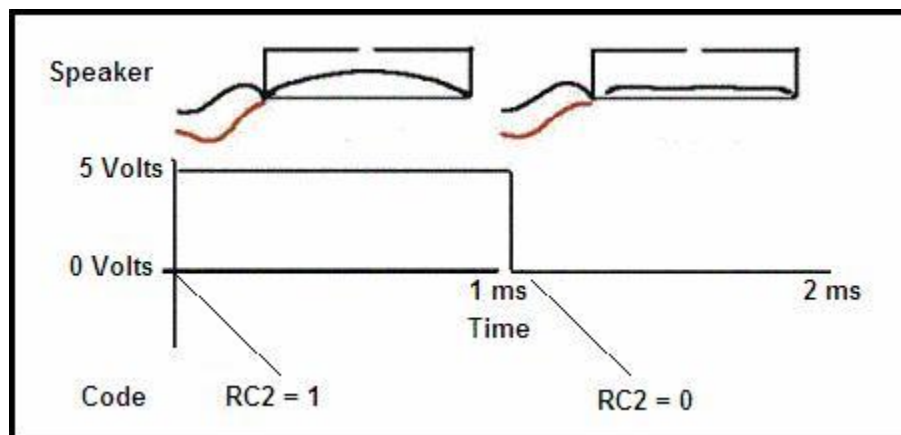


Figure 5. Voltage pattern (with speaker above and code below).

1. Open the Programming Portal, create a new code file, and save it as click.c.

2. Enter the following code:

```
#include "mxapi.h"
void main(void)
{
    TRISC2 = 0; //Sets Port as output
    RC2 = 1; //Sets Port high
    delay_us(1000); //Wait 1000 microseconds
    RC2 = 0;
```

```
    delay_us(1000);  
    end();  
}
```

**3. Click the Compile Button.**

**4. Click the Download button.**

**5. Listen to the speaker. It should produce a click.**

**6. Press the reset button on the XBoard a few times to hear the clicking noise.**

## **Challenge 2: Produce Sounds from the Speaker Producing a Sustained Tone**

To make a more interesting sound, like a sustained tone, it is necessary to make the piezo-electric element inside the speaker vibrate back and forth repeatedly.

**1. Create a new file by saving your code as tone.c.**

**2. Change your code so that the piezo-electric element moves back and forth repeatedly.**

```
#include "mxapi.h"  
void main(void)  
{  
    TRISC2 = 0;  
    while(1==1)    //Sets the while... loop  
    {  
        RC2 = 1;  
        delay_us(1000);  
        RC2 = 0;  
        delay_us(1000);  
    }  
    end();  
}
```

**3. Compile and test your new code. NOTE: When you are tired of listening to the tone, just switch off your battery pack!**

## **Challenge 2: Produce Sounds from the Speaker Playing Individual Tones**



With `tone.c`, you produced a sustained tone that plays indefinitely. For this project, it would be useful to play individual tones (tones that play for a short period and then stop).

**1. Using the Save File button, rename your code file `one_tone.c`.**

**2. Modify your code so that the tone plays for a short period and then stops.**

```
#include "mxapi.h"
void main(void)
{
    int x;                //Declare a variable called x
    TRISC2 = 0;
    for(x=0; x<200; x++)  //Sets the for.. loop
    {
        RC2 = 1;
        delay_us(1000);
        RC2 = 0;
        delay_us(1000);
    }
    end();
}
```

**3. Compile and test your new code.**

## Unit 2: Making Notes and Melodies

Once you know how to make sounds from the speaker, you are ready to take the next step: making musical *notes* and *melodies*. A musical note is just a tone with a specific *frequency* and a specific *duration*. A melody is a sequence of different notes. A song's melody is its central musical theme. If the song has lyrics, each note in the melody usually corresponds to a word or a syllable in the lyrics.



**Figure 1. Digital devices for measuring musical frequency and duration.**

Unit 2 has two challenges. In Challenge 1, you will learn to control the frequency of the tones produced by your XBoard. In Challenge 2, you will write a function to control both the frequency and the duration of each note, in order to streamline the making of musical melodies.

## Challenge 1: Control Frequency



A note's frequency is also referred to as its *pitch*. Some notes--such as those produced by a flute or a violin--have high frequencies, while other notes--such as those produced by a cello or a bass guitar--have low frequencies. Even someone who doesn't know anything about music can hear the difference between high-frequency notes and low-frequency notes. In this challenge, you will learn how to calculate a note's frequency and how to control the frequency of the notes produced by your XBoard.

## Challenge 1: Control Frequency About Frequency

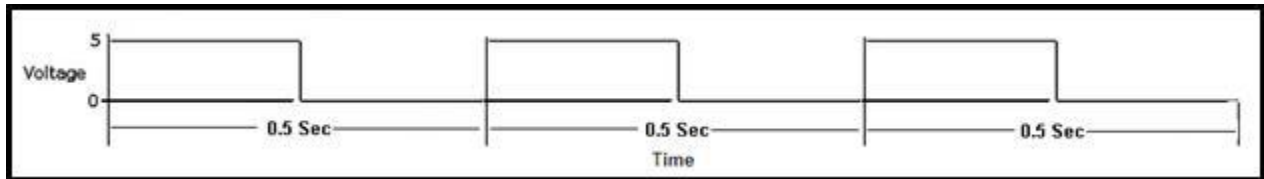
OK, so you can hear the difference between a high note and a low note, but how do you calculate a note's frequency? Actually, this is very easy to do, once you understand that the movement of the piezo-electric element is a type of wave, like the one shown in Figure 2. Like every wave form, the movement of the element has two important properties:

\* *Cycle*. A *cycle* is a complete movement of the piezo-electric element up to 5 volts and back down to 0 volts.

\* *Period*. *Period* is the amount of time required to complete one cycle.

*Frequency* is the number of cycles that happen during a one-second period, or simply the number of cycles per second.

Figure 2 shows the voltage sent to the XBoard's speaker over three cycles. A cycle starts when the voltage goes to 5 volts, continues as the voltage drops back to 0 volts, and ends just as the voltage is about to go back up to 5 volts.



**Figure 2. Voltage over three cycles.**

### **Challenge 1: Control Frequency**

#### **Calculating a Tone's Frequency**

Frequency can be calculated two different ways:

$$\begin{aligned} \text{frequency} &= 1 / \text{period} \\ \text{-or-} \\ \text{frequency} &= \text{cycles} / \text{second} \end{aligned}$$

Both equations produce the same result. Consider the example shown in the

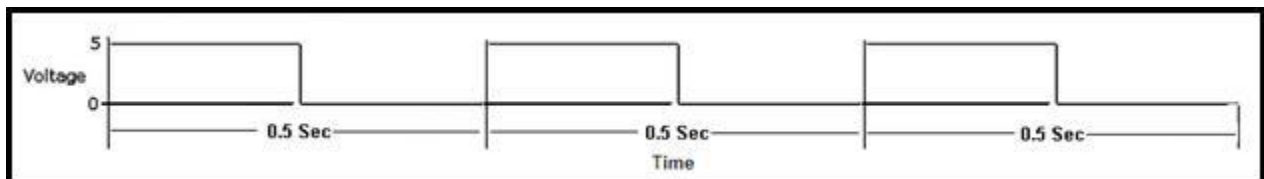


figure below:

$$\begin{aligned} \text{frequency} &= 1 / \text{period} \\ \text{frequency} &= 1 / (0.5 \text{ seconds}) \\ \text{frequency} &= 2 \text{ cycles} / \text{second} \\ \text{-or-} \\ \text{frequency} &= \text{cycles} / \text{second} \\ \text{frequency} &= (3 \text{ cycles}) / (1.5 \text{ seconds}) \\ \text{frequency} &= 2 \text{ cycles} / \text{second} \end{aligned}$$

The most common unit of frequency is the Hertz. One cycle per second is equal to one Hertz.

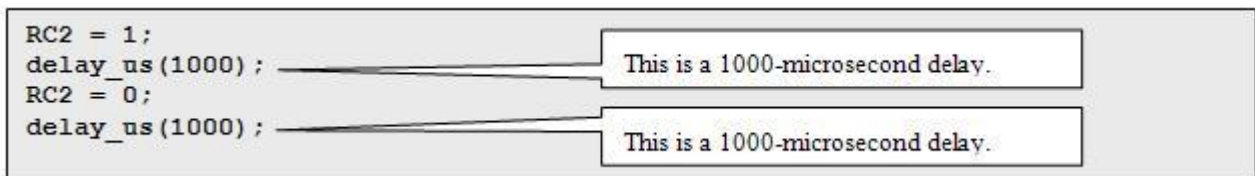
$$1 \text{ Hertz} = 1 \text{ cycle} / \text{second}$$

Therefore, in the example shown above, the frequency is 2 Hertz.

## Challenge 1: Control Frequency

### Calculating a Tone's Frequency from your Code

Now that you know how frequency and period are related, it is possible to determine the frequency of the tone your speaker will produce, just by looking at your code. The period of the tone will be the sum of the two delays in each cycle of your for... loop. For example, in the following code, the delays are 1,000 microseconds each:



$$\begin{aligned} \text{frequency} &= 1 / \text{period} \\ \text{frequency} &= 1 / (\text{delay1} + \text{delay2}) \\ \text{frequency} &= 1 / (1000 \text{ microseconds} + 1000 \text{ microseconds}) \\ \text{frequency} &= 1 / (2000 \text{ microseconds}) \\ \text{frequency} &= 1 / (0.002 \text{ seconds}) \\ \text{frequency} &= 500 \text{ hertz} \end{aligned}$$

## Challenge 1: Control Frequency

### Playing Specific Frequencies

By adjusting the delays in your for... loop, you can program the microcontroller to produce a tone with a specific frequency--in other words, a musical note. The table below shows seven musical notes, together with their frequencies.

Note	Frequency
G	784
F	698
E	659
D	587
C	523
B	494

A	440
---	-----

The note A has a frequency of 440 cycles per second; the note B has a frequency of 494 cycles per second; and so on.

1. Open the file `one_tone.c`. Using the Save File button, rename your code file `a440.c`.
2. Calculate the period and the delays required to produce the note A.
3. Adjust the delays in your code to produce that note.
4. Compile and test your new code.

### Challenge 1: Control Frequency Creating a Simple Melody

When notes are put together, they form melodies. By adding multiple loops to your code, you can create a simple melody, with one loop per note.

1. Using the Save File button, rename your code file `first_song.c`.
2. Calculate the period for the notes you need—for example: E, D, C, D, E.
3. Add loops to your code, and adjust the delays in each loop, to produce the notes you want.
4. Compile and test your new code.

#### PROGRAMMING NOTE



You can cut and paste text in the Programming Portal, just as you would in a normal word processing program. This feature may come in handy as you make melodies on the XBoard. You can cut and paste however many loops you need, and then change the delays to create different notes for your melody.

### Challenge 2: Use Functions to Make



With the code that you have written so far, each tone requires many lines of code, making it cumbersome to produce even a very simple

melody. This can be greatly streamlined by setting up a function in C to produce each tone. As you learned in Introductory Project 3, a function is a block of code that is given a name. Once you define the function, you can execute that block of code whenever you need it, with a single statement. A useful feature of functions is that you can pass one or more variables to the function, to control how the function operates. In this Challenge, you will define a function called beep, which will cause the speaker to generate a tone every time you call the function. You will pass a variable to the beep function, controlling the tone's frequency.

## Challenge 2: Use Functions to Make Setting Up the Beep Function

1. Using the Save File button, rename your code file pulse\_function.c.
2. Insert the following lines to your code file:

```
#include "mxapi.h"
void beep(int pulse)
{
    int x;
    for(x=0; x<200; x++)
    {
        RC2 = 1;
        delay_us(pulse); //Wait for pulse microseconds
        RC2 = 0;
        delay_us(pulse);
    }
}
```

3. Now, in your main function, any time you want to produce a tone, you just have to call the beep function, as shown below:

```
void main(void)
{
    TRISC2 = 0;
    beep(759); //Calls a beep function
    beep(852);
    beep(956);
    beep(852);
    beep(759);
    end();
}
```

4. Compile and test your new code.

## Challenge 2: Use Functions to Make Calculating Pulse Length from Frequency

It's much easier not having to write all the code for each note, but it would be even better if your program could take care of calculating the pulse length required for a specific frequency. That way, you could just enter frequencies and get the correct tone. In the next step, you will modify your function to convert frequencies to pulses automatically.

The calculation is accomplished as follows:

```
frequency = 1 / period
period * frequency = 1
period (in seconds) = 1 / frequency
period (in microseconds) = 1000000 * (1 / frequency)
period (in microseconds) = 1000000 / frequency
pulse (in microseconds) = period / 2
```

## Challenge 2: Use Functions to Make Passing a Frequency Variable to Your Beep Function

By incorporating the pulse calculations into your beep function, you can pass frequency directly to the function. In order to do this, you will need to introduce a new type of variable--a *long* variable, which can store larger numbers than an *int* variable. A long variable is needed in this case, because period and pulse are measured in microseconds and can reach values up to 1,000,000. An int variable can only go up to 32,768, whereas long variables can store values higher than 2,000,000,000 (2 billion).

1. Using the Save File button, rename your code file `freq_function.c`.

2. Enter the following code:

```
#include "mxapi.h"
void beep(int frequency)
{
    long period;           //declare variable called period
    long pulse;           //declare variable called pulse
    int x;
    period = 1000000 / frequency;
    pulse = (period / 2);

    for(x=0; x<200; x++)
    {
```



```

    RC2 = 1;
    delay_us(pulse);
    RC2 = 0;
    delay_us(pulse);
}
}

```

**3. Now in your main function, insert the following lines. Note that you can pass a specific frequency to the beep function, each time you call it.**

```

void main(void)
{
    TRISC2 = 0;
    beep(523);
    beep(587);
    beep(659);
    beep(698);
    beep(784);
    end();
}

```

**4. Compile and test your new code.**

## Challenge 2: Use Functions to Make Using the Function to Control Duration

In music, controlling a note's duration is as important as controlling its frequency. By adding another variable to the beep function, you can control the number of times your for... loop repeats each time the beep function is called.

**1. Using the Save As command, rename your code file cycles\_function.c.**

**2. Add a new variable, called cycles, to your beep function, as follows:**

```

void beep(int frequency, int cycles)

```

**3. Use the variable to control how many times the for... loop repeats, as follows:**

```

for(x=0; x<cycles; x++)

```

**4. Add the number of cycles each time you call the beep function, as shown below.**

```

beep(400, 200);
beep(440, 400);
beep(494, 200);

```

```
beep(494, 400);  
beep(523, 200);
```

## 5. Compile and test your new code.

### Challenge 2: Use Functions to Make Calculating Cycles from Duration

If you listen closely to two notes with the same number of cycles but different frequencies, you will notice that the higher note has a shorter duration than the lower note. The reason for this is simple: higher frequencies have shorter periods. Two hundred cycles of a short period takes less time to complete than 200 cycles of a longer period.

To address this problem, it is necessary to adjust the number of cycles depending on the frequency of the note. This adjustment can be calculated using the following facts: 1) the duration of a note is equal to the number of cycles multiplied by the period of each cycle; and 2) the period of each cycle (in milliseconds) is equal to 1000 divided by the note's frequency.

$$\begin{aligned}\text{duration} &= \text{cycles} * \text{period} \\ \text{duration} &= \text{cycles} * (1000 / \text{frequency}) \\ \text{frequency} * \text{duration} &= \text{cycles} * 1000 \\ \text{cycles} &= \text{duration} * \text{frequency} / 1000\end{aligned}$$

### Challenge 2: Use Functions to Make Fine Tuning the Duration

By incorporating the cycles calculations into the beep function, you can fine tune the duration of each note.

1. Using the **Save As** command, rename your code file **full\_function.c**.
2. In your beep function, replace the **cycles** variable with a new variable, called **duration**, as follows:

```
void beep(int frequency, int duration)
```

3. In your beep function, initialize a new variable, called **cycles**, just below the statements initializing your **period**, **pulse**, and **x** variables, as shown below. (REMEMBER: All statements initializing variables always must be at the top of your function.)

```
void beep (int frequency, int duration)
{
    long period;
    long pulse;
    int x;
    long cycles;
```

4. Below the statements that calculate *pulse* from *frequency*, use the *duration* and *frequency* variables to compute the number of *cycles*--i.e., the number of times the *for...* loop repeats each time the beep is called, as follows:

```
period = 1000000 / frequency;
pulse = period / 2;
cycles = duration * (frequency / 1000.00);
for(x=0; x<=cycles; x++)
```

5. In your main function, each time the beep function is called, replace the number of cycles with the duration of the note (in milliseconds).

```
beep(440,100);
```

6. Compile and test your new code.

#### PROGRAMMING NOTE



In C, when you divide one integer by another, any fractional remainder is automatically left off. For example, if you divide 2500 by 1000, you would expect to get 2.5, but in C, the 0.5 fractional remainder is left off, leaving 2. If you don't want this to happen, you have to change your code slightly. Notice that in the beep function above, you divide frequency by 1000.00. This indicates that you want to keep the fractional remainder, allowing more precise computations.

### Challenge 2: Use Functions to Make Making a Melody with the Beep Function

With your new, fine-tuned beep function, you can make melodies much more accurately and efficiently than you could when you were coding each note by hand.

1. Using the **Save As** command, rename your code file **new\_song.c**.

2. In your main function, use the beep function to create the following melody:

Note	Frequency	Duration		Note	Frequency	Duration
B	494	200		B	494	200
A	440	200		A	440	200
G	392	200		G	392	200
A	440	200		A	440	200
B	494	200		B	494	200
B	494	200		B	494	200
B	494	400		B	494	200
A	440	200		B	494	200
A	440	200		A	440	200
A	440	400		A	440	200
B	494	200		B	494	200
D	587	200		A	440	200
D	587	400		G	392	800

3. Compile and test your new code.

## Challenge 2: Use Functions to Make About the Musical Scale

In Western music, the complete range of musical tones is divided into groups of seven *full tones* --A, B, C, D, E, F, G--and five *half tones*, which fall in between the full tones. Each half tone has two names--for example, the half tone between A and B is called A Sharp or B Flat. The four remaining half tones are called C Sharp/D Flat, D Sharp/E Flat, F Sharp/G Flat, and G Sharp/A Flat. A complete set of whole tones and half tones is called an *octave*. Two notes that are exactly an octave apart always sound fundamentally similar and have the same name, while all of the notes in between sound distinctly different.

The sharps and flats may seem confusing, but just remember that each is an individual note with two different names. For example, C Sharp and D Flat have the exact same frequency. On a piano keyboard, the sharps and flats are the black keys, as shown in Figure 3 below.

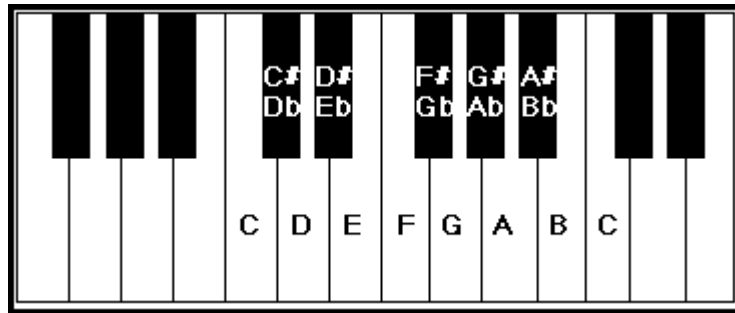


Figure 3. Piano keyboard with notes marked.

## Challenge 2: Use Functions to Make Using Define Statements for Musical Frequencies

The table below presents a matrix of notes and frequencies, grouped into octaves. The octaves shown shaded in gray are those most easily produced by the XBoard's speaker. A wide range of melodies can be written using these tones. Notice that the frequency of a note in one octave is twice the frequency of the corresponding note in the previous octave, and half the frequency of the corresponding note in the next octave.

Note	1	2	3	4	5	6	7	8
B	62	123	247	494	988	1976	3951	
A sharp/B flat	58	117	233	466	932	1865	3729	
A	55	110	220	440	880	1760	3520	
G sharp/A flat	52	104	208	415	831	1661	3322	
G	49	98	196	392	784	1568	3136	
F sharp/G flat	46	92	185	370	740	1480	2960	
F	44	87	175	349	698	1397	2794	
E	41	82	165	330	659	1319	2637	
D sharp/E flat	39	78	156	311	622	1245	2489	4978
D	37	73	147	294	587	1175	2349	4698
C sharp/D flat	35	69	139	277	554	1109	2217	4434
C	33	65	131	262	523	1047	2093	4186

The beep function gives you complete control over each note's frequency. However, when making melodies, you don't need to produce every possible frequency. You need only those frequencies that are found in music. With define statements, you can store these commonly used frequencies, to make it easier to create songs. That way, when you call your beep function, you won't have to look up the note's frequency. You can just type the note by name.

### 1. Using the Save As command, rename your code file new\_song2.c.

2. For each frequency shaded in gray in the frequency table above, add a `define` statement to your code. Include a number to show which column (i.e. octave) the tone is in. Use two separate `define` statements for each sharp and flat note, substituting the letter S for the sharp and the letter F for flat. For example, the first seven `define` statements would appear as follows:

```
#define C2 65
#define C2S 69
#define D2F 69
#define D2 73
#define D2S 78
#define E2F 78
#define E2 82
```

(NOTE: Be sure to insert your *define* statements right after your *include* statements.)

3. In your main function (for "Mary Had a Little Lamb"), replace the frequency numbers with the name of each note. For example, the first four notes would appear as follows:

```
beep (B5, 200) ;
beep (A5, 200) ;
beep (G5, 200) ;
beep (A5, 200) ;
```

4. Compile and test your new code.

## Challenge 2: Use Functions to Make About Note Durations

A musician can choose to perform a song slowly or quickly, and this decision affects the exact duration of every note in the song. Therefore, it is impossible to create a fixed matrix of exact note durations. However, the relationship between the duration of the longer notes and the duration of the shorter notes in a song never changes, no matter how quickly or slowly the song is performed. This information can be captured in *define* statements to streamline song writing.

In Western music, the longest note is called a *whole note*. Other note durations are related to one another by factors of two, as shown below.

**1 Whole Note = 2 Half Notes**

**1 Half Note = 2 Quarter Notes**

**1 Quarter Note = 2 Eighth Notes**

**1 Eighth Note = 2 Sixteenth Notes**

**1 Sixteenth Note = 2 Thirty-Second Notes**

There are also dotted notes, each of which is 50% longer than the next shortest note. (You will learn why they are called dotted notes in the next section.) These are shown below.

**1 Dotted Whole Note = 3 Half Notes**

**1 Dotted Half Note = 3 Quarter Notes**

**1 Dotted Quarter Note = 3 Eighth Notes**

**1 Dotted Eighth Note = 3 Sixteenth Notes**

**1 Dotted Sixteenth  
Note = 3 Thirty-Second Notes**

## **Challenge 2: Use Functions to Make Using Define Statements for Note Durations**

Since note durations have fixed relationships to one another, you can use *define* statements to specify the exact duration (in milliseconds) of one type of note, and then define the others with respect to the first.

**1. Using the Save As command, rename your code file new\_song3.c.**

**2. Add a define statement to your code for each type of note, starting with the 16th note, as shown below:**

```
#define sixtnote 50      //Defines 16th note as 50 milliseconds
#define eighnote 2*sixtnote
#define edotnote 3*sixtnote
#define quarnote 2*eighnote
#define qdotnote 3*eighnote
#define halfnote 2*quarnote
#define hdotnote 3*quarnote
#define wholnote 2*halfnote
#define wdotnote 3*halfnote
```



3. In your main function (for the song "Mary Had a Little Lamb"), replace the duration numbers with the name of each note. For example, the first four notes would appear as follows:

```
beep (B5,quarnote);  
beep (A5,quarnote);  
beep (G5,quarnote);  
beep (A5,quarnote);
```

4. Compile and test your new code.

### Unit 3: Musical Notation and RTTTL

By this point, you have all the tools that you need to produce some more sophisticated melodies, like the ones that you hear whenever someone's cell phone rings. Fortunately, there is an almost limitless supply of melodies available on the Internet. Two common formats for melodies are musical notation--a set of symbols used by composers and musicians to represent the frequency and duration of the notes in a song--and Ring Tone Text Transfer Language (RTTTL)--a simple text-based format developed by Nokia to represent cell phone ring tone melodies.

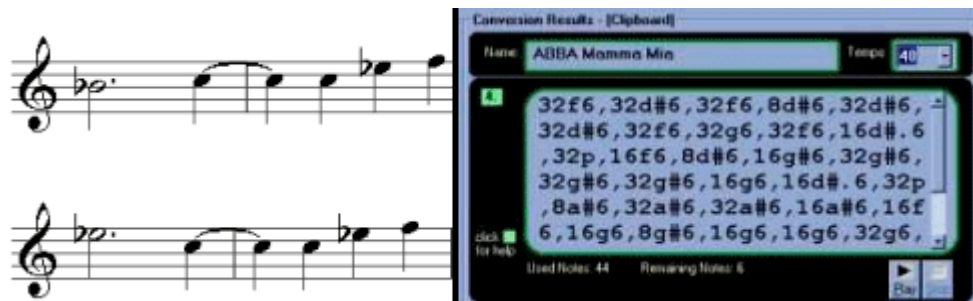


Figure 1. Musical notation (left) and RTTTL (right).

Unit 3 has two challenges. In Challenge 1, you will learn to read musical notation and program any melody you want. In Challenge 2, you will learn to understand RTTTL and use RTTTL code to program the XBoard--creating the exact same ring tones that people download for their cell phones!

#### Challenge 1: Translate Musical Notation into C



All musicians learn to read musical notation, no matter what instrument they play. If you can learn to read musical notation, you can program the XBoard to play virtually any melody you can find. In this challenge, you will learn how to translate musical notation--

readily available on the Internet for most popular and classical music--into C code.

## Challenge 1: Translate Musical Notation into C About the Staff Lines

In musical notation, the notes in a song are represented by symbols, which appear on horizontal lines, called *staff lines*. Note symbols are placed either on or between the lines, as shown in Figure 2. A note's vertical position indicates its frequency, with higher frequency notes appearing higher on the staff lines. For example, the note G4 appears higher on the staff lines than the note E4.

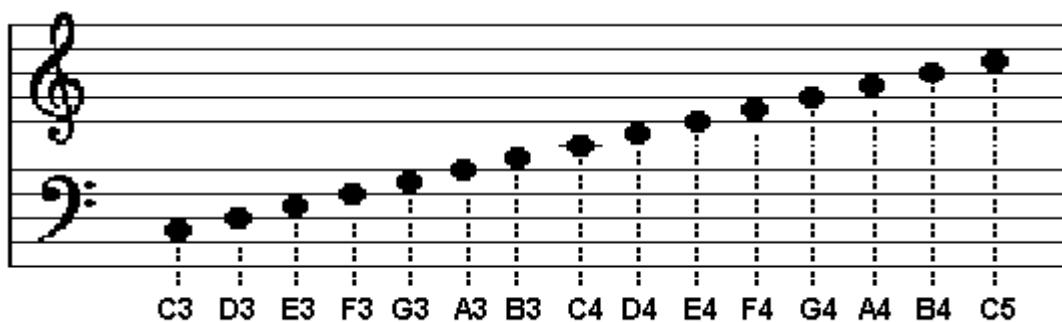


Figure 2. Staff lines with musical notation.

In sheet music, sharps and flats are indicated by special markings that are placed just to the left of the note symbol, shown in Figure 3. The symbol for a sharp looks like the # character, and the symbol for a flat looks like a lowercase **b**.

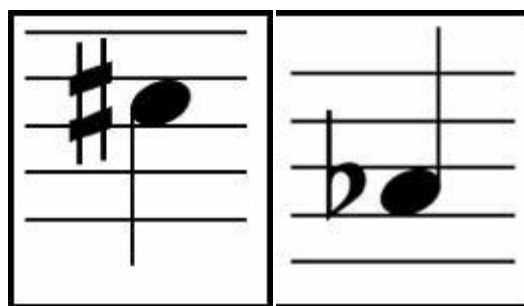


Figure 3. C sharp (left) and A flat (right).

When you see a symbol for a sharp, you have to make the note one half tone higher. For example, in Figure 3, C (C5) would become C sharp (C5S). When you see a symbol for a flat, you have to make the note one half tone lower. In Figure 3, A (A4) would become A flat or (A4F).

## Challenge 1: Translate Musical Notation into C About Note Shapes

In musical notation, the shape of a note indicates its duration, as shown in Figure 4.











ITEM	NOTE	REST
Whole note/rest		
Half note/rest		
Quarter note/rest		
Eighth note/rest		
Sixteenth note/rest		

Figure 4. Standard note and rest notations.

A whole note is represented by a small oval shape. A half note is a slightly smaller oval shape with a short vertical line rising on the right side. A quarter note looks like a half note with the oval filled in. Eighth notes and the sixteenth notes are identical to quarter notes, except for one or two small "flags" attached to the vertical line.

A rest is a point in a song where no note is played. Rests can be introduced in your C code with the `delay_ms` command, which will pause the song for a period measured in milliseconds. You can use your note duration define statements to control the duration of the rest. For example, an eighth note rest would be represented as follows:

```
delay_ms(eighnote); //This inserts a rest for the duration of an  
eighth note
```

## Challenge 1: Translate Musical Notation into C Translating Sheet Music

There are a few more quirks in sheet music that can be a little confusing. For one thing, notes are often grouped together according to the rhythm of the song. For example, in the song shown in Figure 5, the notes are in groups of four, each representing a beat in the song. The grouping changes the look of each note slightly. All of the notes in Figure 5 are sixteenth notes.

Another important thing to remember is that, once a note is marked with a sharp or flat symbol, it remains sharp or flat until the end of the measure, which is marked with a thin vertical line. A measure in a song is a group of beats. The song shown in Figure 5 has four beats to the measure.

Every note in this song is a 16<sup>th</sup> note.

Notes appear in groups of four, corresponding to the beats of the song.

This thin vertical line marks the end of a measure, or group of beats. This song has four beats to the measure.

The first note is A4, sixteenth note.

The second note is E4, sixteenth note.

This note is marked with a sharp symbol, so it is G#4, sixteenth note.

Since this note was made sharp earlier in the measure, it stays sharp, even though the # symbol is gone.

**Figure 5. Sample musical notation.**

### **Challenge 1: Translate Musical Notation into C Printing a Music Reference Sheet**

In the remainder of this unit, you will be asked to program the microcontroller to produce songs, using only musical notation as your guide. In working on these challenges, it may be useful to have printed reference sheet with a summary of what you have just learned about musical notation.

1. Click [here](#) to open the reference sheet in PDF format. You will need the [Acrobat Reader](#) from Adobe to view the PDF.
2. Select the Print option in the Acrobat Reader to print this page.

### **Challenge 1: Translate Musical Notation into C Making a Hip Hop Song**

1. Using the Save As command, rename your code file new\_song4.c.
2. Translate the following sheet music into C code. All notes are 16th notes. When two notes are tied together (by a horizontal parentheses), they should be treated as one eighth note.

The song starts with a 16<sup>th</sup> note rest: `delay_ms(sixtnote)`

Two 16<sup>th</sup> notes tied like this equal an eighth note: `beep(E5, eighthnote)`.

Remember—The # symbol means sharp: `beep(G4S, sixtnote)`.

This is tied to the next line: `beep(C5, eighthnote)`.

Once a note is marked as a sharp or flat once in a measure, it stays sharp or flat for the remainder of the measure. Therefore, this is a G sharp (G4S).

3. Compile and test your new code.

### Challenge 1: Translate Musical Notation into C Making a TV Theme Song

1. Program the XBoard to play the following melody:

This first note is a dotted quarter note C: `beep(C4, qdotnote)`.

Two eighth notes equal one quarter note: `beep(F4S, quarnote)`.

This note is A3.

This note is F3S.

This note is B3F.

2. Compile and test your new code.

## Challenge 1: Translate Musical Notation into C Making a Movie Theme Song

1. Program the XBoard to play the following melodies, placing the second line of notes after the first:

The image shows two musical staves. The top staff is in bass clef, 5/4 time, and contains a melody of eighth and quarter notes. A callout points to the first note: "This first note is: beep(A2,qdotnote)." The bottom staff is in treble clef, 5/4 time, and contains a melody of eighth, quarter, and whole notes. Four callouts explain musical symbols: "This first note is an eighth note C: beep(C5,eighnote).", "This symbol means the note is flat: beep(E4F,wholnote).", "This is a quarter note rest: delay\_ms (quarnote).", and "This is a whole note rest: delay\_ms (wholnote)."

2. Compile and test your new code.

## Challenge 1: Translate Musical Notation into C Making Another Theme Song

1. Program the XBoard to play the following melodies, placing the notes on the second line after the notes on the first line:

The image shows a single musical staff in treble clef, 5/4 time, with a melody of eighth and sixteenth notes. Three callouts explain musical symbols: "This is an eighth note E: beep(E4,eighnote).", "This is a 16<sup>th</sup> note F sharp: beep(F4S,sixtnote).", and "Once a note is marked sharp, it stays sharp until the end of the measure. Every F in this melody should be an F sharp."

This is an eighth note E: beep(E4,eighnote).

This symbol means that the note is not sharp anymore.

This is a dotted half note: beep(B5,hdotnote).

This is a whole note C: beep(C4,wholenote).

## 2. Compile and test your new code.

### Challenge 1: Translate Musical Notation into C Completing a Song

#### 1. Program the XBoard to play the following six notes:

```
beep(A5,quarnote);
beep(A5,quarnote);
beep(B5,halfnote);
beep(A5,halfnote);
beep(D6,halfnote);
beep(C6S,wholnote);
```

## 2. Compile and test your new code.

### Challenge 2: Translate RTTTL into C



Nokia developed RTTTL to allow customers to easily download new ring tones for their cell phones. Like musical notation, RTTTL format ring tones are readily available on the Internet. In Challenge 2, you will learn how to translate RTTTL into C code that can be played on your XBoard.

### Challenge 2: Translate RTTTL into C About RTTTL

In many ways, RTTTL is very similar to the C code that you have written to encode melodies on the microcontroller. For every note in a melody, RTTTL stores frequency and duration information. Note frequencies are represented by the letters A to G, coupled with numbers representing the different octaves, just as you have done in your C code. In RTTTL, lowercase letters are used instead of uppercase letters, and sharp notes are marked with the "#" symbol, instead of



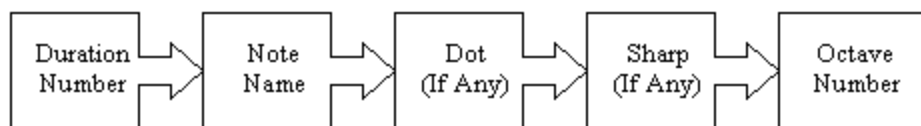
the letter "S." Also, when a sharp is indicated, the "#" symbol appears before the octave number, not after. The table below shows these differences.

Note	Frequency	C Code	RTTTL
<b>G</b>	784	G5	g5
<b>F sharp/G flat</b>	740	F5S	f#5
<b>F</b>	698	F5	f5
<b>E</b>	659	E5	e5
<b>D sharp/E flat</b>	622	D5S	d#5
<b>D</b>	587	D5	d5
<b>C sharp/D flat</b>	554	C5S	c#5
<b>C</b>	523	C5	c5

Note durations in RTTTL are represented by numbers, with 1 corresponding to a whole note, 2 corresponding to a half note, 4 corresponding to a quarter note, and so on. To represent a dotted note, RTTTL adds a "." to the note symbol. The table below shows these differences.

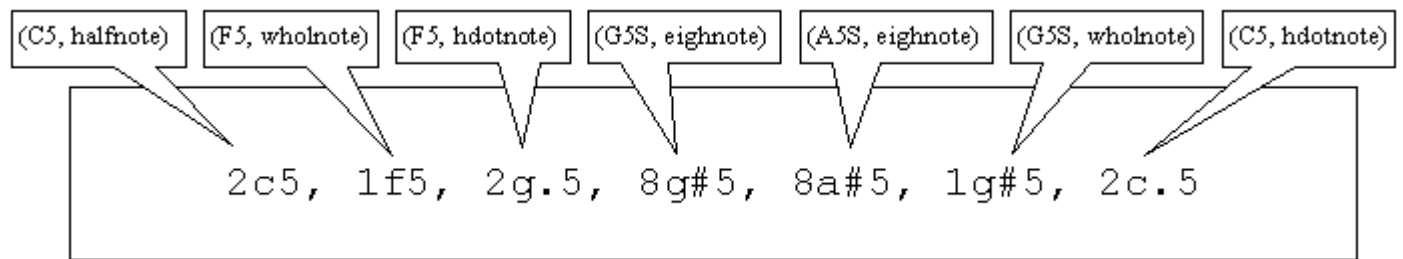
Note	C Code	RTTTL
<b>Dotted Whole Note</b>	wdotnote	1 .
<b>Whole Note</b>	wholnote	1
<b>Dotted Half Note</b>	hdotnote	2 .
<b>Half Note</b>	halfnote	2
<b>Dotted Quarter Note</b>	qdotnote	4 .
<b>Quarter Note</b>	quarnote	4
<b>Dotted Eighth Note</b>	edotnote	8 .
<b>Eighth Note</b>	eighnote	8
<b>Dotted Sixteenth Note</b>	sdotnote	16 .
<b>Sixteenth Note</b>	sixtnote	16
<b>Thirty-Second Note</b>	thirnote	32

In RTTTL, the elements of the duration and frequency information appear in a specific sequence, but they are not separated by commas (as they are in your C code). Figure 5 shows the RTTTL sequence.



**Figure 5. RTTTL data sequence.**

Commas are used to separate the notes in a melody. Figure 6 shows a simple seven-note melody in RTTTL format.



**Figure 6. Simple melody in RTTTL format.**

Rests, or pauses, in RTTTL melodies are represented with the letter "p". For example, a half note rest would be represented with 2p, a quarter note rest would be represented with 4p, and so on.

## Challenge 2: Translate RTTTL into C

### Translating RTTTL

1. Program the microcontroller to play the seven notes show in Figure 6.
2. Complete the melody, using the RTTTL shown below:

2c5, 1f5, 2g.5, 8g#5, 8a#5, 1g#5, 2c.5, 4c5, 2f.5, 4g5, 4g#5, 4c5, 8g#.5, 8c.5, 8c6, 1a#.5, 2c5, 2f.5, 4g5, 4g#.5, 8f5, 4c.6, 8g#5, 1f6, 2f5, 8g#.5, 8g.5, 8f5, 2c6, 8c.6, 8g#.5, 8f5, 2c5, 8c.5, 8c.5, 8c5, 2f5, 8f.5, 8f.5, 8f5, 2f5

3. Compile and test your new code.

## Challenge 2: Translate RTTTL into C

### About RTTTL Default Settings

Translating RTTTL songs into C code is simple, once you understand the differences in the way note frequencies and durations are represented. However, the designers of RTTTL often use a memory-saving measure that makes it slightly more complicated to translate RTTTL ring tones into C: they define default settings for note durations and octaves at the start of each melody. That way, if any note in a melody is the default duration or in the default octave, the duration and/or octave number do not need to be set for that note. Figure 7 shows how this works.

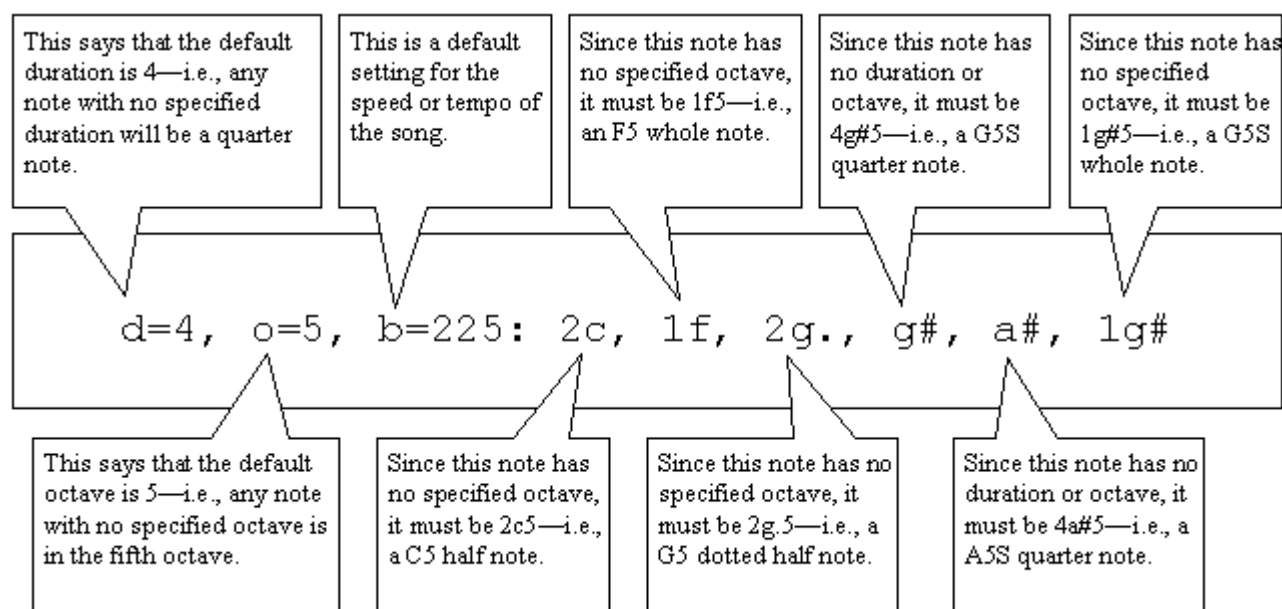


Figure 7. A simple RTTTL melody with default settings.

## Challenge 2: Translate RTTTL into C

### Translating RTTTL with Defaults

1. Translate the following RTTTL ring tone into C. (REMEMBER: The "p" represents a rest, or pause, in the music):

```
d=4,o=5,b=125: c, c, g, g, 8a, 8b, 8c6, 8a, g, p, f, f, e,
                e, d, d, c
```

2. Compile and test your new code.

## Unit 1: Making Lights and Sounds

Tabletop and handheld electronic games have been around since the 1970s. In the early days, most games were pretty simple; no color LCD screens, 3D graphics, or multiplayer networking! One of the most popular games challenged players to watch a sequence of lights and tones, and then reproduce that sequence by pressing lighted buttons. With each turn, the sequence grew longer and faster. The game, called Simon, was a huge hit, and it is still being sold today. Figure 1 shows the original Simon, together with an updated version, called Simon2.



Figure 1. Original Simon (left) and Simon2 (right).

In Unit 1, you will take the first steps toward building a Simon-like memory game on your XBoard. Unit 1 has three challenges. In Challenge 1, you will add four light emitting diodes (LEDs), four buttons, and a speaker to the board. In Challenge 2, you will learn to light the LEDs and produce tones from the speaker. And in Challenge 3, you will program the buttons to control the LEDs and the speaker.

### Challenge 1: Add the Hardware



Challenge 1 is a hardware task--adding the LEDs, buttons, and speaker to the breadboard. Since you will need to have easy access to the buttons in order to play the finished game, the arrangement of these components is very important. Extra jump wires are used so that the connections to the chip do not interfere with game play.

#### Challenge 1: Add the Hardware Collecting Your Components

In order to complete this unit, you will need the following components (shown in Figure 2):

Part	Quantity	Description
A	4	LEDs (1 yellow, 1 red, 1 orange, and 1 green)
B	4	Button switches
C	4	Resistors (1,000 Ohm)
D	4	Resistors (10,000 Ohm)
E	14	Jump wires (8 white, 3 orange, and 2 yellow)
F	1	Piezo-electric speaker
G	1	Bent connector (two-prong)
H	8	Flexible jump wires (not shown)

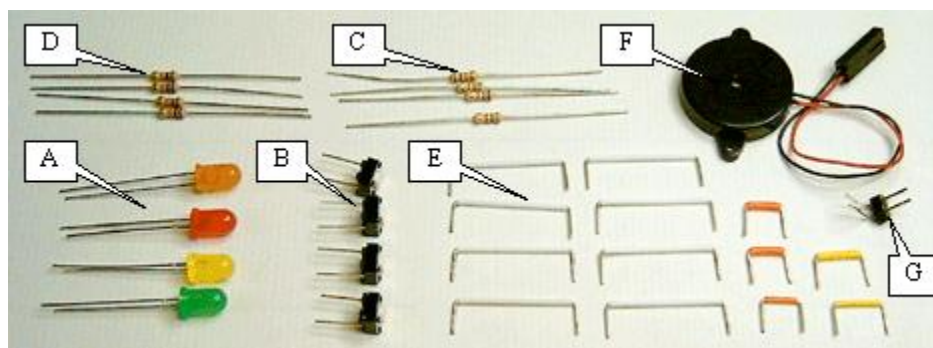


Figure 2. Memory game components.

## Challenge 1: Add the Hardware

### Removing the LCD

The components for this project take up quite a lot of space on the right side of the breadboard. To make room, it is helpful to remove the LCD and some of the jump wires underneath.

1. Remove the LCD from the board.
2. Remove the orange wires connecting holes J28 and J30 to ground, the orange wire connecting hole J24 to hole J27, and the yellow wire connecting hole J29 to power, as shown in Figure 3.

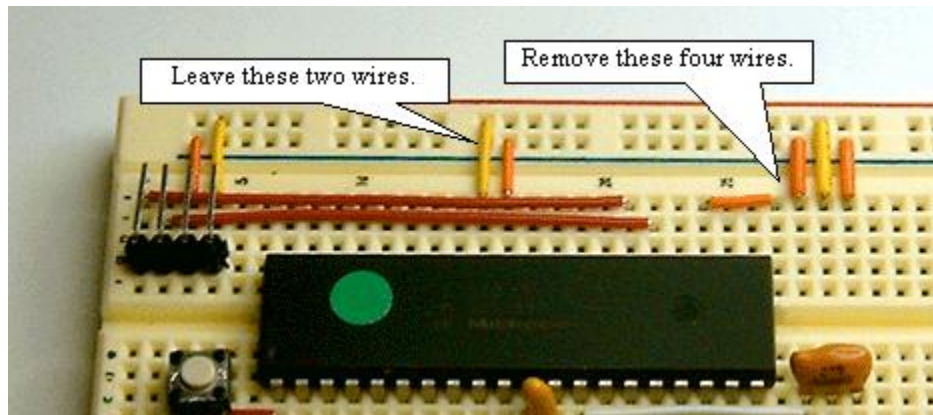


Figure 3. Removing the jump wires under the LCD.

3. Be sure to leave in place the jump wires connecting hole J15 to power and hole J16 to ground.

### Challenge 1: Add the Hardware About LEDs

Light emitting diodes (LEDs) are like tiny light bulbs that emit light when electricity is supplied. Unlike a conventional incandescent light bulb, every LED has a specific *polarity*, meaning one of its two pins is designed to connect to power and the other pin is designed to connect to ground. A tiny flat segment on the round rim of the LED's circular base marks the ground pin. The power pin is typically longer than the ground pin, but this is an unreliable indicator, since the pins may be trimmed during installation. Figure 4 shows these distinguishing features.

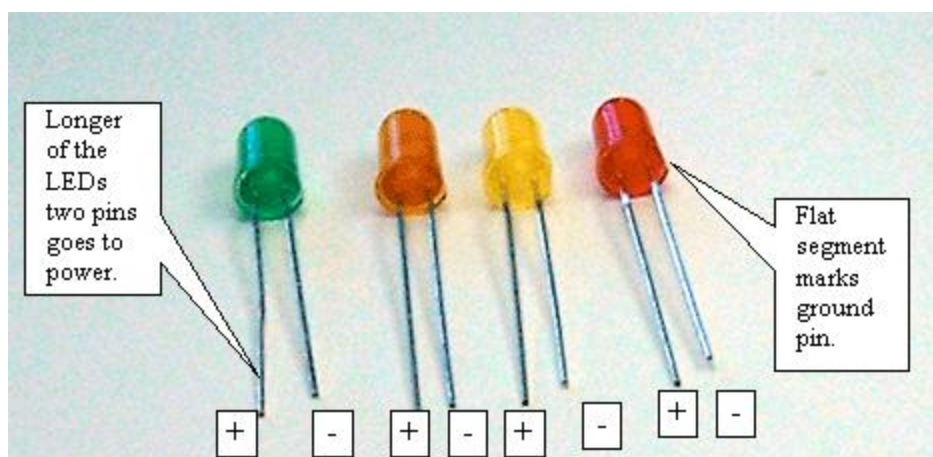


Figure 4. Power pin and ground pin on LEDs.

### Challenge 1: Add the Hardware Adding the LEDs



The game requires four LEDs: one yellow, one red, one orange, and one green. These are inserted into the board with the power pin on the right and the ground pin on the left. Each LED is connected to power by a 1,000 Ohm resistor.

1. Insert the four LEDs into the board, as shown in Figure 5. The yellow LED should go in holes I48 and I49; the red LED should go in holes I44 and I45; the orange LED should go in holes B44 and B45; and the green LED should go in holes B48 and B49. (Note: Be sure the flat edge of each LED is facing toward the microcontroller. You may have to trim the pins on each LED to get it to sit securely on the board.)

2. Using four 1,000 Ohm resistors, connect holes J49, J45, A45, and A49 to power. (Note: You may need to trim the resistors to get them to fit neatly.)

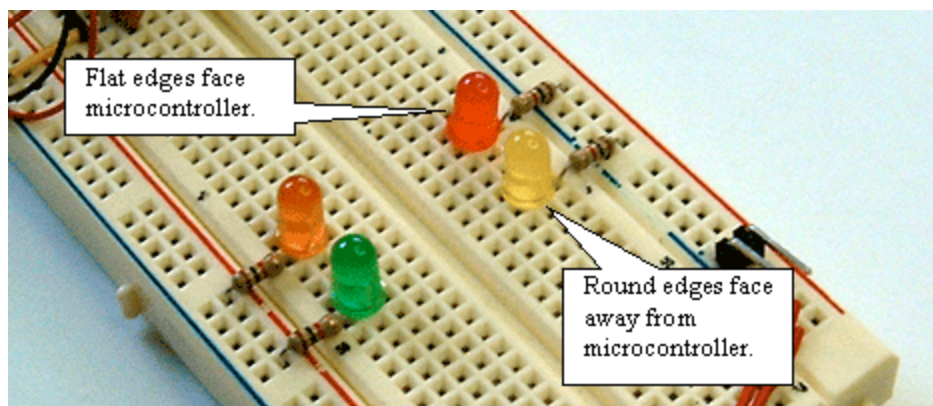


Figure 5. Adding the LEDs.

### Challenge 1: Add the Hardware Adding the Buttons

In the original Simon game, the buttons themselves light up. In your XBoard, each LED will have a button next to it. Pressing that button will light the LED.

1. Insert four buttons into the board, as shown in Figure 6. The button for the yellow LED should go in holes I51 and I53; the button for red LED should go in holes I40 and I42; the button for the orange LED should go in holes B40 and B42; and the button for the green LED should go in holes B51 and B53.

2. Using four 10,000-Ohm resistors, connect hole J53, J42, A42, and A53 to power. (Note: You may need to trim the resistors to get them to fit neatly.)

3. Using orange jump wires, connect holes J40 and J51 to ground.



4. Using yellow jump wires, connect holes A40 and A51 to ground.

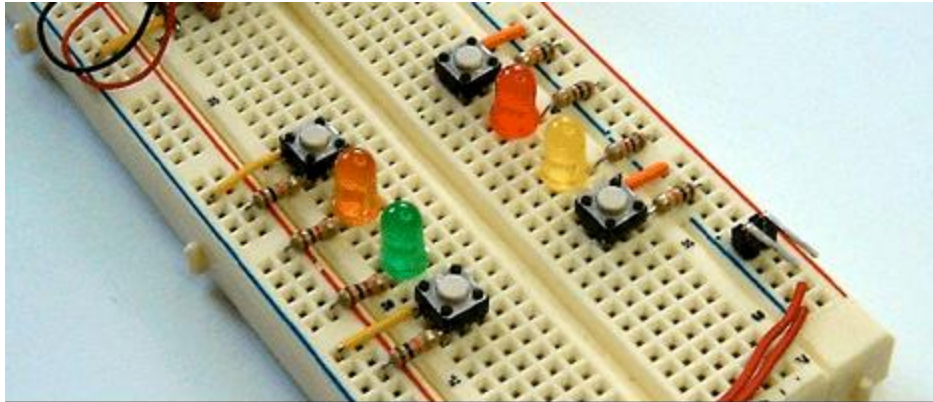


Figure 6. Adding the buttons.

### **Challenge 1: Add the Hardware** **Adding the Speaker**

The red lead on the speaker connects to Port D2 on the microcontroller, and the dark lead connects to ground, via an orange jump wire and a two-prong bent connector.

1. Using an orange jump wire, connect hole J27 to ground.
2. Insert the two-prong bent connector into holes I26 and I27.
3. Connect the speaker to the bent connector, making sure to align the black lead with the jump wire connecting to ground, as shown in Figure 7.

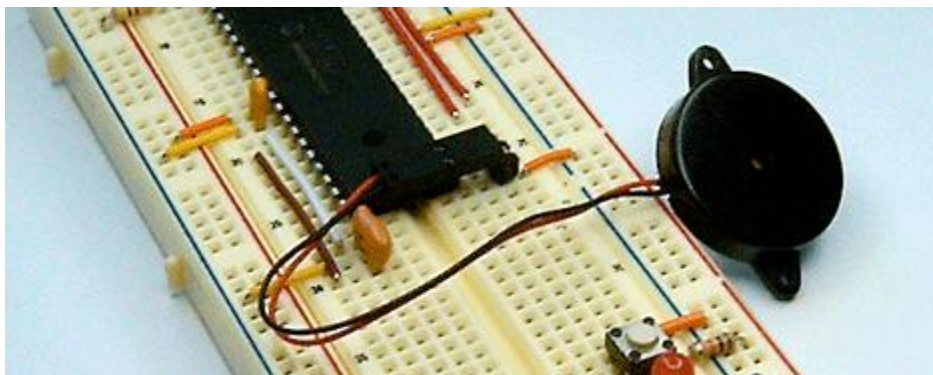


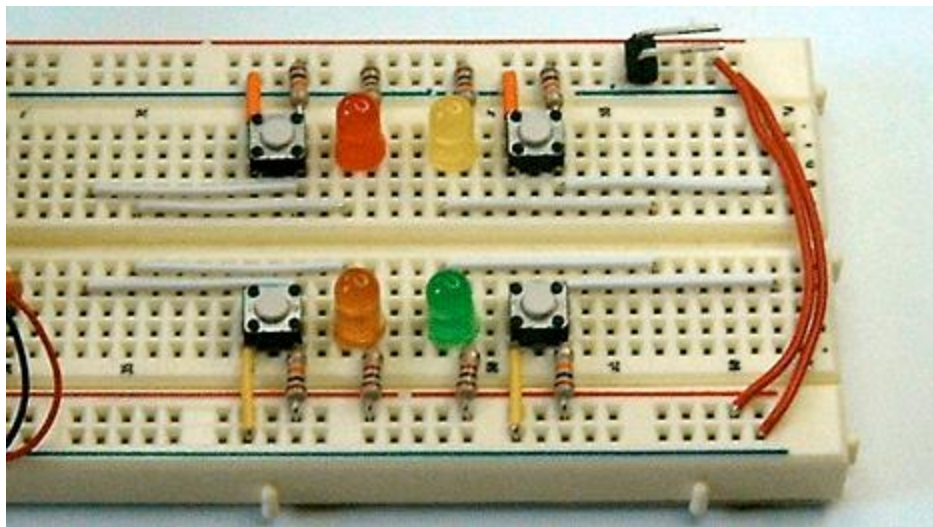
Figure 7. Installing the speaker.

### **Challenge 1: Add the Hardware** **Adding Extra Jump Wires**

At this stage, you could connect your LEDs and buttons directly to the microcontroller, using flexible jump wires. But the resulting tangle of jump wires might get in a player's way during a game. Using the white jump wires, you can clear some space around the LEDs and buttons, so that the buttons will be easier for players to reach.

**1. For the LEDs, use white jump wires to connect hole F48 to hole F57, hole F44 to hole F35, hole E44 to hole E35, and hole E48 to hole E57.**

**2. For the buttons, use white jump wires to connect hole G53 to hole G62, hole G42 to hole G33, hole D42 to hole D33, and hole D53 to hole D62.**



**Figure 8. Adding the white jump wires.**

### **Challenge 1: Add the Hardware Making Connections to the Chip**

The last step is to connect the ends of your white jump wires to the microcontroller, using flexible jump wires. The LEDs will connect to Ports B0 to B3, and the buttons will connect to Ports B4 to B7. Be careful in this step not to mix up the jump wires, as sorting them out later may be difficult.

**1. For the LEDs, use flexible jump wires to connect hole H57 (yellow LED) to Port B0, hole H35 (red LED) to Port B1, hole C35 (orange LED) to Port B2, and hole C57 (green LED) to Port B3.**

**2. For the buttons, use flexible jump wires to connect hole H62 (yellow LED button) to Port B4, hole H33 (red LED button) to Port B5, hole C33 (orange LED button) to Port B6, and hole C62 (green LED button) to Port B7.**

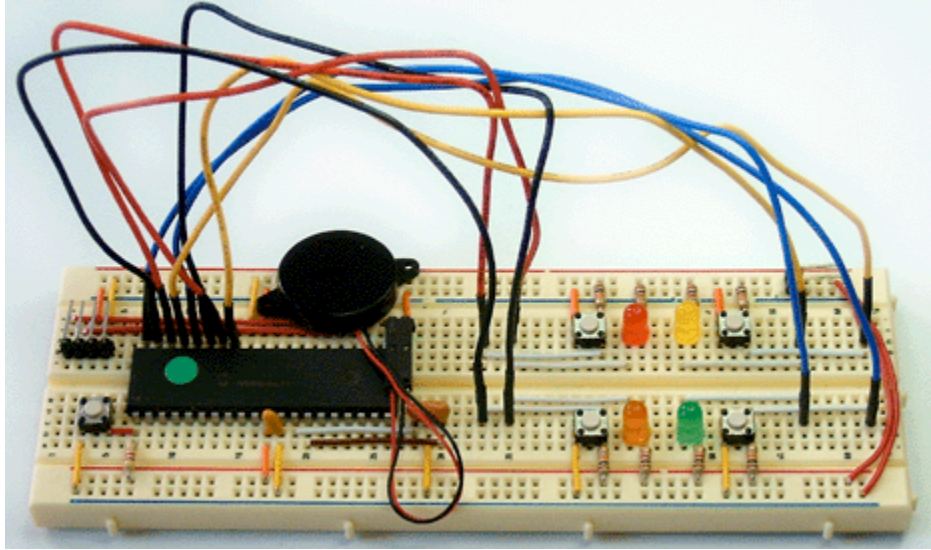


Figure 9. Connecting the LEDs and buttons to the microcontroller.

## Challenge 2: Control the LEDs and Speaker



With your hardware all installed, you are ready to start writing code to control the various input and output devices on your board. You will learn how to turn the LEDs on and off, produce sounds from the speaker, and collect input from the buttons.

### Challenge 2: Control the LEDs and Speaker Turning an LED On and Off

Each of the LEDs is connected to power by a 1,000 Ohm resistor and to the microcontroller by a flexible jump wire. To light the LED, you must send a value of 0 to pin connected to the ground side of the LED. This lowers the voltage at the LED's ground pin, causing the LED to light. To turn off the LED, you simply send a value of 1 to the same pin. This equalizes the voltage on either side of the LED, stopping the flow of current.

1. Create a new code file and save it as `one_led.c`.
2. Enter the following code:

```
#include "mxapi.h"
void main(void)
{
    TRISB0 = 0; //Set up Port B0 as an output
    RB0 = 0; //Set Port B0 low
    delay_ms(1000); //Wait 1000 milliseconds
    RB0 = 1; //Set Port B0 high
    end(); //End program
}
```

### 3. Compile and test your new code.

## Challenge 2: Control the LEDs and Speaker Light All Four LEDs

Lighting the rest of the LEDs is simple. Just set up Ports B1, B2, and B3 as output pins, and then use the code in the previous step to light each LED.

#### 1. Rename your code file it as four\_leds.c.

#### 2. Enter the following code:

```
#include "mxapi.h"
void main(void)
{
    TRISB0 = 0;
    TRISB1 = 0;
    TRISB2 = 0;
    TRISB3 = 0;
    RB0 = 0;
    RB1 = 0;
    RB2 = 0;
    RB3 = 0;
    delay_ms(1000);
    RB0 = 1;
    RB1 = 0;
    delay_ms(1000);
    RB1 = 1;
    RB2 = 0;
    delay_ms(1000);
    RB2 = 1;
    RB3 = 0;
    delay_ms(1000);
    RB3 = 1;
    end();
}
```

### 3. Compile and test your new code.

## Challenge 2: Control the LEDs and Speaker Using Define Statements

With four LEDs, it can be difficult to keep track of which LED is connected to which microcontroller pin. Carefully worded define statements can help avoid any confusion.

1. Rename your code file it as led\_defines.c.

2. Between your include statements and your main function, enter the following code:

```
#include "mxapi.h"
#define Y_LED RB0
#define R_LED RB1
#define O_LED RB2
#define G_LED RB3
```

3. Change the statements turning your LEDs on and off to make use of your new define statements.

4. Compile and test your new code.

## Challenge 2: Control the LEDs and Speaker About the Tone Function

If you have already completed the Music Synthesizer project, you have written a function, called beep, to make sounds from the speaker. The code library also includes a function, called tone, which is equivalent in many respects to the beep function. You pass two values to the tone function. The first indicates the desired frequency, in Hertz, of the tone produced, and the second indicates its duration, in milliseconds. To make life easier, the code library also includes a number of pre-defined note frequencies, as shown in Figure 10.

Note	Octave 3		Octave 4		Octave 5	
B	B3	247	B4	494	B5	988
A sharp/B flat	A3S	233	A4S	466	A5S	932
A	A3	220	A4	440	A5	880
G sharp/A flat	G3S	208	G4S	415	G5S	831
G	G3	196	G4	392	G5	784
F sharp/G flat	F3S	185	F4S	370	F5S	740
F	F3	175	F4	349	F5	698
E	E3	165	E4	330	E5	659
D sharp/E flat	D3S	156	D4S	311	D5S	622
D	D3	147	D4	294	D5	587
C sharp/D flat	C3S	139	C4S	277	C5S	554
C	C3	131	C4	262	C5	523

Figure 10. Pre-defined note frequencies.

In addition, the code library has a number of pre-defined note durations, as shown in Figure 11.



Duration	Define	Duration
32 <sup>nd</sup> note	thirnote	40 ms
16 <sup>th</sup> note	sixtnote	80 ms
8 <sup>th</sup> note	eighnote	160 ms
Quarter note	quarnote	320 ms
Half note	halfnote	640 ms
Whole note	wholnote	1280 ms

Figure 11. Pre-defined note durations.

Therefore, when calling the tone function, you can refer to notes by name, as shown in Figure 12. The first argument passed to the function indicates the frequency of each note (B5 = 988 Hz, A5 = 880 Hz, and G5 = 784 Hz), and the second argument indicates the duration (halfnote = 640 ms).

```
tone(B5, halfnote);
tone(A5, halfnote);
tone(G5, halfnote);
tone(A5, halfnote);
```

Figure 12. Calling the tone function.

## Challenge 2: Control the LEDs and Speaker Combining Lights and Sounds

Now that you know how to produce sounds from the speaker, you can associate a tone with each LED, just like in the original Simon game.

1. Rename your code file `tone_light.c`.
2. In your main function, replace each of the delay statements with statements calling the tone function, as shown below:

```
Y_LED = 0;
tone(C5, halfnote);
Y_LED = 1;
R_LED = 0;
tone(D5, halfnote);
R_LED = 1;
O_LED = 0;
tone(E5, halfnote);
O_LED = 1;
G_LED = 0;
tone(F5, halfnote);
G_LED = 1;
```

### 3. Compile and test your new code.

## Challenge 2: Control the LEDs and Speaker

### Write a Function to Control the LEDs and Speaker

Since the memory game involves frequently lighting the LEDs and producing tones, it is useful at this stage to write a function that performs these actions each time it is called. The function will be called `light`, and you will pass a variable to it, called `color`. The value of `color` will determine which LED lights up (0 for yellow, 1 for red, 2 for orange, and 3 for green).

#### 1. Rename your code file `light_function.c`.

#### 2. Above your main function, write a function called `light`, as shown below:

```
#include "mxapi.h"
void light(char color)
{
    if(color==0)
    {
        Y_LED=0;
        tone(C5, halfnote);
        Y_LED=1;
    }
    if(color==1)
    {
        R_LED=0;
        tone(D5, halfnote);
        R_LED=1;
    }
    if(color==2)
    {
        O_LED=0;
        tone(E5, halfnote);
        O_LED=1;
    }
    if(color==3)
    {
        G_LED=0;
        tone(F5, halfnote);
        G_LED=1;
    }
}
```

#### 3. Now, in your main function, call your `light` function, as shown below:



```

void main(void)
{
    TRISB0 = 0;
    TRISB1 = 0;
    TRISB2 = 0;
    TRISB3 = 0;
    light(0);
    light(1);
    light(2);
    light(3);
    end();
}

```

#### 4. Compile and test your new code.

### Challenge 3: Use the Buttons



In this challenge, you will program the microcontroller to call the light function whenever a button is pressed.

### Challenge 3: Use the Buttons About the Buttons

The buttons are connected to Ports B4 to B7 on the microcontroller. As with the reset button, the voltage at the pin connecting the button to the microcontroller is held high (5 volts) by the 10,000 Ohm resistor. Whenever one of the buttons is pressed, a connection to ground is made, and the voltage at the pin immediately goes low (0 volts). Figure 13 shows the values at each button in the pressed and unpressed states.

Button	Pin	Not Pressed	Pressed
Yellow LED	Port B4	1	0
Red LED	Port B5	1	0
Orange LED	Port B6	1	0
Green LED	Port B7	1	0

Figure 13. Values at Ports B4 to B7 when buttons are pressed and not pressed.

### Challenge 3: Use the Buttons Linking the Buttons and the LEDs

You want the LED next to each button to light whenever that button is pressed. To do this, you need a series of if"else if" statements that call the light function whenever the voltage at the pin connected to the button goes low.

### 1. Rename your code file button\_check.c.

### 2. To simplify coding, add the following define statements to your code:

```
#define Y_BUTTON RB4
#define R_BUTTON RB5
#define O_BUTTON RB6
#define G_BUTTON RB7
```

### 3. In your main function, add the following TRIS statements to set up the button pins as inputs:

```
TRISB4 = 1;
TRISB5 = 1;
TRISB6 = 1;
TRISB7 = 1;
```

### 4. After the TRIS statements, replace the contents of your main function with a while"1 loop that contains a series of if" statements to monitor the buttons, as follows:

```
while(1==1)
{
    if(Y_BUTTON==0)
    {
        light(0);
    }
    else if(R_BUTTON==0)
    {
        light(1);
    }
    else if(O_BUTTON==0)
    {
        light(2);
    }
    else if(G_BUTTON==0)
    {
        light(3);
    }
    delay_ms(100);
}
```

### 5. Compile and test your new code.

## Unit 2: Programming the Game Play

Now that you know how to control the LEDs, the speaker, and the buttons, it's time to start programming the game play. You want your game to generate a random sequence of lights and sounds for the player to duplicate. The sequence should expand with each turn, starting with one light/tone, then two, then three, and so on. Notably, in each turn, only the last light/tone is random; the others are the same as in the previous turn. The animation in Figure 1 shows one such sequence: green, then green-blue, then green-blue-yellow, then green-blue-yellow-red.



Figure 1. An expanding sequence of lights.

After each sequence is presented, the player must correctly duplicate the sequence by pressing the corresponding buttons. If one incorrect button is pressed, the player loses!

### Challenge 1: Generate a Random Array



In this challenge, you will program the microcontroller to produce the random sequence of lights and tones needed for game play. The sequence will be stored in an array.

### Challenge 1: Generate a Random Array About Arrays

An array is an ordered list of two or more values that is assigned a specific name. To introduce an array into your code, you need to declare what types of values the array will store (e.g., char or int values), give the array a name, and indicate how many values are stored in the array. Figure 2 shows how to set up the array for this project. This array, called pattern, and can store 11 char values.

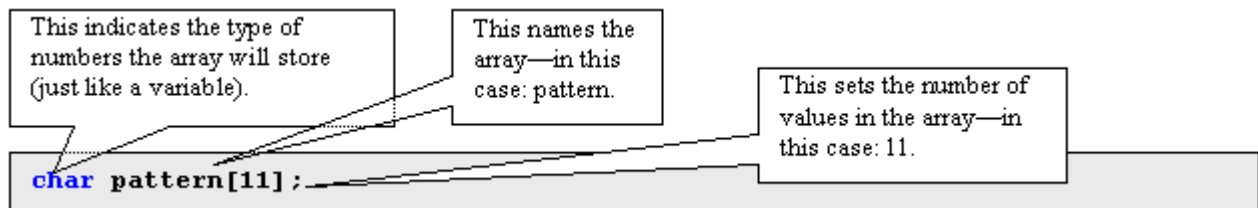


Figure 2. Setting up the pattern array.

When you first declare your array, all of its values are set to 0 by default. If you want, you can specify different values in the statement declaring the array, as shown in Figure 3.

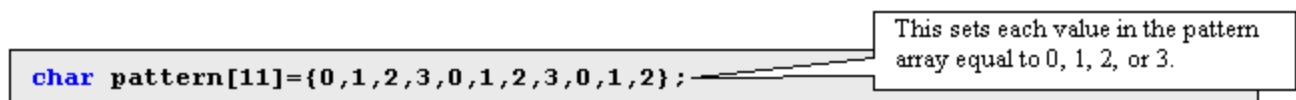


Figure 3. Setting a specific value in the pattern array.

You can refer to any individual value in your array by specifying its position.

**Note:** The first position in an array is position 0, not 1. You can specify the position with a number or with a variable, as shown in Figure 4.

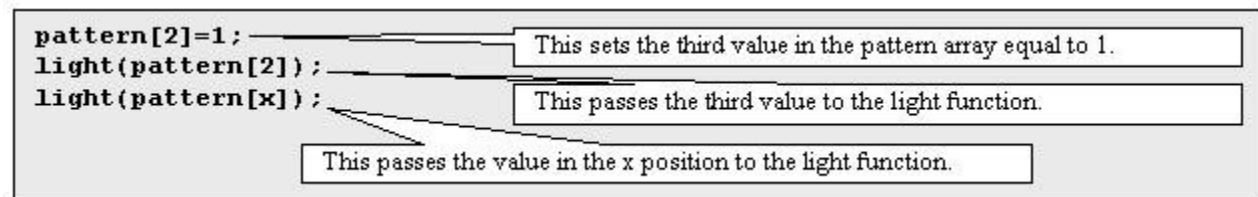


Figure 4. Referring to individual values in the pattern array.

## Challenge 1: Generate a Random Array

### Passing Values from an Array to the Light Function

In this step, you will declare your array and then use it to pass values to the light function, generating an expanding sequence of lights and tones. To do this, you will also need two new variables, called *x* and *count*.

1. Rename your code file `game_array.c`.

2. Between your include statements and your define statements, add two new lines of code: one to declare your *x* variable and the other to declare your pattern array and set each of its values, as follows:

```
char pattern[11]={0,1,2,3,0,1,2,3,0,1,2};
int x;
```

**3. At the top of your main function, add a statement to declare a variable called count, as shown below:**

```
int count;
```

**4. In your main function, above your while... loop, create a for... loop, based on the count variable, that repeats 10 times, as shown below. The count for... loop should contain a second for... loop, based on the x variable, that passes each value in the pattern array to the light function.**

```
for (count=1; count<11; count++)
{
    for (x=0; x<count; x++)
    {
        light(pattern[x]);
    }
}
```

**5. Compile and test your new code.**

## **Challenge 1: Generate a Random Array Requiring the Player to Respond**

In the original Simon game, the player has to respond each time a new sequence of lights and tones is displayed. You can make this happen by enclosing your while... loop with a for... loop based on the x variable.

**1. Rename your code file player\_respond.c.**

**2. Enclose your while... loop inside a new for... loop based on the x variable. Change the arguments of the while... loop to prevent the code from moving forward until a button is pressed, as shown below:**

```
for (count=1; count<11; count++)
{
    for (x=0; x<count; x++)
    {
        light(pattern[x]);
    }
    for (x=0; x<count; x++)
    {
        while (Y_BUTTON!=0 && R_BUTTON!=0 && O_BUTTON!=0 &&
G_BUTTON!=0);
    }
}
```

```

    if (Y_BUTTON==0)
    {
        light(0);
    }
    else if (R_BUTTON==0)
    {
        light(1);
    }
    else if (O_BUTTON==0)
    {
        light(2);
    }
    else if (G_BUTTON==0)
    {
        light(3);
    }
    delay_ms(100);
}
}
}

```

### 3. Compile and test your new code.

#### Challenge 1: Generate a Random Array About Timer0

As written, your code always generates the same sequence of lights and tones every time. This would not make for a very challenging game! In the next step, you will write code to assign a random value, between 0 and 3, to each position in the pattern array. To do this, you will make use of one of the microcontroller's internal timers. This timer, called Timer0 or just TMR0, changes in value so rapidly that at any given instant, its value is essentially random.

TMR0 increases in value by 1 every microsecond until it reaches its maximum value of 255 and then it resets to 0. Therefore, at any given instant, there is about a 25-percent chance that TMR0's value is between 0 and 63, or between 64 and 128, or between 129 and 192, or between 193 and 255.

#### Challenge 1: Generate a Random Array Making the Array Random

Now that you understand how TMR0 works, you can use this timer to assign values to the pattern array, creating a random sequence of lights/tones each time you play the game.

### 1. Rename your code file `random_array.c`.

### 2. Add the highlighted code, initializing timer0 before the start of your count for "i" loop, and then setting the value of `pattern[x]` to 0, 1, 2, or 3 each time through the loop, based on TMR0:

```
tmr0_init(DIV_256);           //Initialize timer0
for(count=1; count<11; count++)
{
    if(TMR0>0&&TMR0<=63)      //Timer value in first quarter
    {pattern[x]=0;}           //Set element x of array to yellow
    else if(TMR0>63&&TMR0<=128) //Timer value in second quarter
    {pattern[x]=1;}           //Set element x of array to red
    else if(TMR0>128&&TMR0<=192) //Timer value in third quarter
    {pattern[x]=2;}           //Set element x of array to orange
    else                       //Timer value in fourth quarter
    {pattern[x]=3;}           //Set element x of array to green
}
```

### 3. Compile and test your new code.

## Challenge 2: Signal Wins and Losses



With your pattern array, the rudiments of the memory game are taking place. Each time you turn the board on, an expanding sequence of lights and tones is displayed, and the player must respond by pressing buttons on the board. In this challenge, you will program the game to evaluate each button pressed to make sure that the player is following the correct sequence. Any wrong button press will trigger a "losing routine," signaling that the game is lost. If a player makes it all the way to the end, a "victory routine" will be performed, signaling that the game has been won.

## Challenge 2: Signal Wins and Losses

### Checking for Correct Responses

Since the correct pattern for each game is stored in the pattern array, checking whether the correct button has been pressed is easy. You just have to compare the value of the color variable passed to the light function with the corresponding value of the pattern array.

### 1. Rename your code file `check_value.c`.



2. At the top of your light function, add the following code:

```
void light(char color)
{
    if(color!=pattern[x])
    {
        end();
    }
}
```

3. Compile and test your new code.

## Challenge 2: Signal Wins and Losses Programming a Losing Routine

You can use any routine you want to signal an incorrect button was pressed. The code presented below will light all four LEDs and make a low buzzing sound.

1. Rename your code file `losing_routine.c`.

2. Add the following code to the first if... statement in your light function:

```
void light(char color)
{
    if(color!=pattern[x])
    {
        Y_LED=0;
        R_LED=0;
        O_LED=0;
        G_LED=0;
        tone(50,wholnote);
        Y_LED=1;
        R_LED=1;
        O_LED=1;
        G_LED=1;
        end();
    }
}
```

3. Compile and test your new code.

## Challenge 2: Signal Wins and Losses Adding a Victory Routine

With your losing routine, the player knows when the game is lost, but what about when the player wins? To signal the end of the game, you need to program a victory routine. Again, you can have any routine you want. The code shown

below will flash all four LEDs and play a short melody if a player makes it to the end of the game.

**1. Rename your code file victory\_routine.c.**

**2. At the bottom of your main function, after the for... loops, add the following code:**

```
delay_ms(1000);
Y_LED=0;
R_LED=0;
O_LED=0;
G_LED=0;
tone(C5,sixtnote);
tone(D5,sixtnote);
tone(E5,sixtnote);
tone(G5,eighnote);
tone(E5,sixtnote);
tone(G5,halfnote);
tone(C6,wholnote);
Y_LED=1;
R_LED=1;
O_LED=1;
G_LED=1;
end();
```

**3. Compile and test your new code.**

### Challenge 3: Add Advanced Features



Congratulations! You have built a functioning electronic memory game, much like the original Simon. In this challenge, you will add two advanced features to make the game more exciting to play. First, you will make the pace of play accelerate as the game goes on, so that each sequence will be presented a little more quickly than the last. And then, you will build in a time limit, so that players must press the correct sequence within a certain time frame or automatically lose!

### Challenge 3: Add Advanced Features Accelerating Game Play

With the code you have written so far, each LED is lit for a period equal to the duration of one half note (640 ms). To make the game speed up as it goes along, you need to add a variable to represent the duration of each tone, and then make the variable decrease in value as the game goes on.

1. Rename your code file `speed_play.c`.

2. Near the top of your code file, where the rest of your global variables are declared, declare a new integer variable called `length`, as shown below:

```
int length;
```

3. In your `light` function, replace all of the halfnote tone durations with your new `length` variable, as shown below:

```
if (color==0)
{
    Y_LED=0;
    tone(C5,length);
    Y_LED=1;
```

4. In your `main` function, add a line of code at the top of your `count for...` loop that will make the `length` variable decrease with each turn.

```
for(count=1; count<11; count++)
{
    length=640/count+100;
```

5. Compile and test your new code.

### Challenge 3: Add Advanced Features Building in a Time Limit

One final step will make your memory game just like the original: making sure players don't take too much time to repeat each sequence. Otherwise, a player could cheat by writing down the sequence each time! To build in the time limit, you will need to add some code to the `while...` loop that waits for a button to be pressed each time. Using `TMR0`, you can automatically end the game if a player takes too long.

1. Rename your code file `timelimit.c`.

2. Near the top of your code file, where the rest of your global variables are declared, declare a new integer variable called `check`, as shown below:

```
int check;
```

**3. Remove the semicolon at the end of your while... statement, and add the following code inside your while... loop to end the game if too much time goes by:**

```
while (Y_BUTTON!=0&&R_BUTTON!=0&&O_BUTTON!=0&&G_BUTTON!=0)
{
    if (T0IF==1)
    {
        check=check+1;
        T0IF=0;
    }
    if (check>30)
    {
        light(5);
    }
}
check=0;
```

**4. Compile and test your new code.**

## Unit 1: Programming the Keypad

These days, almost every cell phone features a text-messaging capability, which allows the user to enter a short text message and then send the message to another cell phone customer. Text messages are sent using wireless radio frequency communication, through cellular telephone networks.



**Figure 1. Text-messaging device.**

In this project, you will create a device that uses infrared light to transmit text messages wirelessly from one breadboard to another. You will connect a keypad, an infrared transmitter, and an infrared receiver to the board. You will program the microcontroller to decipher input from the keypad, display the input on the LCD, send this data via the infrared transmitter, and receive text messages from other boards. Figure 1 shows a text-messaging device.

Text Messenger Unit 1 has three challenges. In Challenge 1, you will add a keypad to the breadboard and learn about its connections to the microcontroller. In Challenge 2, you will write code to "scan" the keypad--checking one key at a time to determine whether it has been pressed. In Challenge 3, you will refine your code to enable text entry.

### Challenge 1: Install the Keypad



In this challenge, you will add a 4x4 keypad to the breadboard, using an eight-strand flat-flex cable to connect the keypad to the microcontroller. You will also learn about the keypad's matrix layout and examine how each key is linked to the microcontroller.

### Challenge 1: Install the Keypad Collecting Your Components

In order to complete Unit 1, you will need the following components (shown in Figure 2):

Part	Quantity	Description
A	1	Keypad
B	1	Flat-flex cable (eight-strand)
C	1	Straight connector (eight-pin)



Figure 2. Text Messenger components.

### Challenge 1: Install the Keypad Adding the Keypad

The keypad is connected to the breadboard by an eight-strand flat-flex cable. This cable links to an eight-pin straight connector, which plugs into the board, adjacent to Ports B0 through B7.

**1. Insert the eight-pin straight connector into the XBoard, as shown in Figure 3. *NOTE: Be sure to align the pins with Ports B0 to B7, as shown.***

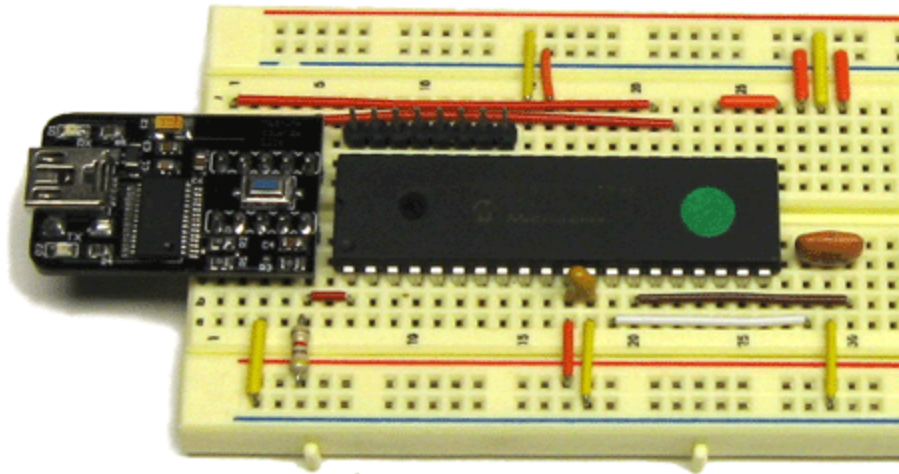


Figure 3. Inserting eight-pin straight connector.

2. Connect the flat-flex cable to the pins on the back of the keypad, as shown in Figure 4.

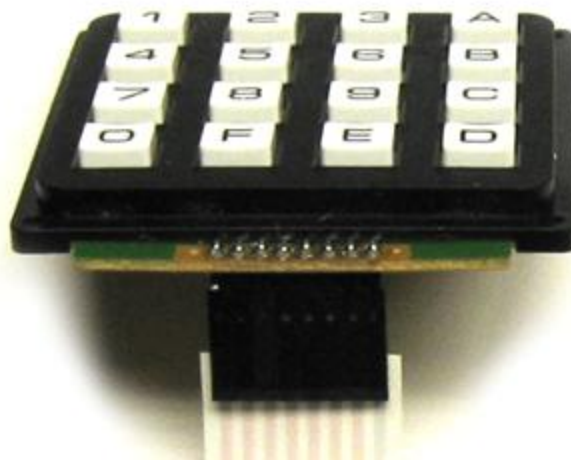


Figure 4. Connecting flat-flex cable to the back of the keypad.

3. Connect the loose end of the flat-flex cable to the eight-pin straight connector. Orient the cable so that you can read text on the LCD when you are pressing buttons on the keypad.

### **Challenge 1: Install the Keypad**

#### **Understanding the Keypad**

Take a close look at the keypad. The keys are arranged in a 4x4 matrix, with 16 keys in total. Inside the flat-flex cable, there are eight wires connecting the keypad to pins on the microcontroller—one for each column in the matrix (these



connect to Ports B4 to B7) and one for each row (these connect to Ports B0 to B3). Figure 5 shows how the keys are arranged, and how each column and row is connected to the microcontroller.

	B7	B6	B5	B4
B3	1	2	3	A
B2	4	5	6	B
B1	7	8	9	C
B0	O	F	E	D

**Figure 5. Keypad connections.**

There are letters and numbers on each key, but these letters and numbers mean nothing. Each key is just a switch. When you press a key, you make a connection between the key's column and the key's row. For instance, when you press the 1 key, you connect Port B3 to Port B7. When you press the D key, you connect Port B0 to Port B4.

## **Challenge 2: Scan the Keypad**



Your second challenge in this project is to write code that gathers input from the keypad. To do this, your code will "scan" the keypad--checking one key at a time to see whether it has been pressed.

### **Challenge 2: Scan the Keypad Understanding the Scanning Code**

In order to scan the keypad, you will set up Ports B0 to B3 as outputs, and set up Ports B4 to B7 as inputs. Initially, all eight pins will be set high (i.e. equal to 1). Then, one pin at a time, you will set Ports B0, B1, B2, and B3 low (i.e. equal to 0), while simultaneously checking the value at Ports B4, B5, B6, and B7. Whenever a key is pressed in a row that has been set low, that key's column will go low as well.

Figure 6 shows how the key scanning code works. The ports on the left are all outputs, while the ports on the right are all inputs. Notice that, for each key, there is a unique pair of ports that have a value of 0, while every other port has a value of 1. For example, in row one, Port B3 (an output) has been set low, and the 1 key has been pressed. As a result, Port B7 (an input) goes low as well.

Output Pins					Key		Input Pins			
B3	B2	B1	B0				B7	B6	B5	B4
0	1	1	1	→	1	→	0	1	1	1
0	1	1	1	→	2	→	1	0	1	1
0	1	1	1	→	3	→	1	1	0	1
0	1	1	1	→	A	→	1	1	1	0
1	0	1	1	→	4	→	0	1	1	1
1	0	1	1	→	5	→	1	0	1	1
1	0	1	1	→	6	→	1	1	0	1
1	0	1	1	→	B	→	1	1	1	0
1	1	0	1	→	7	→	0	1	1	1
1	1	0	1	→	8	→	1	0	1	1
1	1	0	1	→	9	→	1	1	0	1
1	1	0	1	→	C	→	1	1	1	0
1	1	1	0	→	O	→	0	1	1	1
1	1	1	0	→	F	→	1	0	1	1
1	1	1	0	→	E	→	1	1	0	1
1	1	1	0	→	D	→	1	1	1	0

Figure 6. Port values for each key.

## Challenge 2: Scan the Keypad

### Scanning One Row

As a first step, you will write code to scan one row of the keypad--the row with the following four keys: 1, 2, 3, and A. These keys are all connected to Port B3. In your code, you will set Port B3 low (i.e. equal to 0) and then continuously check the value at Ports B4 through B7. If any of these ports goes low, you will know that a key has been pressed. Each time this happens, you will change the value of a variable called key, and display the value of key on the LCD.

1. In the Programming Window, create a new file, and save it as one\_row.c.

2. Type or paste the following code into the window:

```
#include "mxapi.h"

void main(void)
{
    char key=0; //Declare a variable key
    RBPU=0;     //Set pull-up resistors on Port B
    lcd_init(); //Initialize the LCD

    TRISB3=0;   //Set Port B3 as an output
    TRISB2=0;   //Set Port B2 as an output
    TRISB1=0;   //Set Port B1 as an output
```

```

TRISB0=0;          //Set Port B0 as an output

TRISB7=1;          //Set Port B7 as an input
TRISB6=1;          //Set Port B6 as an input
TRISB5=1;          //Set Port B5 as an input
TRISB4=1;          //Set Port B4 as an input

RB3=1;             //Set Port B3 high
RB2=1;             //Set Port B2 high
RB1=1;             //Set Port B1 high
RB0=1;             //Set Port B0 high

while(1==1)        //Set up while loop
{
    RB3=0;
    if(RB7==0) //If 1 key is pressed
    {
        key=1;
    }
    else if(RB6==0) //If 2 key is pressed
    {
        key=2;
    }
    else if(RB5==0) //If 3 key is pressed
    {
        key=3;
    }
    else if(RB4==0) //If A key is pressed
    {
        key=4;
    }
    RB3=1;

    lcd_decimal(key); //Display value of key
    delay_ms(400);
    lcd_instruction(CLEAR);
}
}

```

### 3. Compile and test your new code.

## Challenge 2: Scan the Keypad Scanning All Four Rows

Now that you know how to scan one row of keys, scanning more rows is straightforward. Just set the port for each row (first Port B3, then B2, then B1, then B0) low, and check the value at Ports B4 to B7 (using an *if...* or an *else if...*

statement for each port). Whenever each of these ports goes low, change the value of the key variable.

**1. Using the Save As command, rename your code file more\_columns.c.**

**2. Edit your code file as follows:**

```
#include "mxapi.h"

void main(void)
{
    char key=0;    //Declare a variable key
    RBPU=0;        //Set pull-up resistors on Port B
    lcd_init();    //Initialize the LCD

    TRISB3=0;      //Set Port B3 as an output
    TRISB2=0;      //Set Port B2 as an output
    TRISB1=0;      //Set Port B1 as an output
    TRISB0=0;      //Set Port B0 as an output

    TRISB7=1;      //Set Port B7 as an input
    TRISB6=1;      //Set Port B6 as an input
    TRISB5=1;      //Set Port B5 as an input
    TRISB4=1;      //Set Port B4 as an input

    RB3=1;         //Set Port B3 high
    RB2=1;         //Set Port B2 high
    RB1=1;         //Set Port B1 high
    RB0=1;         //Set Port B0 high

    while(1==1)    //Set up while loop
    {
        RB3=0;
        if(RB7==0) //If 1 key is pressed
        {
            key=1;
        }
        else if(RB6==0) //If 2 key is pressed
        {
            key=2;
        }
        else if(RB5==0) //If 3 key is pressed
        {
            key=3;
        }
        else if(RB4==0) //If A key is pressed
        {
            key=4;
        }
    }
}
```

```

    }
    RB3=1;
    RB2=0;
    if(RB7==0) //If 4 key is pressed
    {
        key=5;
    }
    else if(RB6==0) //If 5 key is pressed
    {
        key=6;
    }
    else if(RB5==0) //If 6 key is pressed
    {
        key=7;
    }
    else if(RB4==0) //If B key is pressed
    {
        key=8;
    }
    RB2=1;
    RB1=0;
    if(RB7==0) //If 7 key is pressed
    {
        key=9;
    }
    else if(RB6==0) //If 8 key is pressed
    {
        key=10;
    }
    else if(RB5==0) //If 9 key is pressed
    {
        key=11;
    }
    else if(RB4==0) //If C key is pressed
    {
        key=12;
    }
    RB1=1;
    RB0=0;
    if(RB7==0) //If O key is pressed
    {
        key=13;
    }
    else if(RB6==0) //If F key is pressed
    {
        key=14;
    }
    else if(RB5==0) //If E key is pressed
    {

```

```

        key=15;
    }
    else if(RB4==0) //If D key is pressed
    {
        key=16;
    }
    RB0=1;

    lcd_decimal(key); //Display value of key
    delay_ms(400);
    lcd_instruction(CLEAR);
}
}

```

**3. Compile and test your new code.** If your keypad code is working properly, you now have a main function that changes the value of the variable called `key`, depending on what key you press. The value of `key` should range from 1 to 16, as shown in Figure 7.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure 7. Value of `key` variable for each key on the keypad.

## Challenge 2: Scan the Keypad

### Converting Your Keypad Code into a Library Function

You will need your keypad scanning code for the next step in this project--using the keypad to enter text. However, since the keypad scanning code is quite long, it makes sense at this stage to save this code as a separate file, which can then be inserted into your main code file with an include statement. This process--known as creating a *library function*--is often used in C programming to keep code files from getting too long. Library functions also have the advantage of being reusable--once they are created, you can use them any time you want.

**1. Using the Save As command, save your code file as `mxkeyscan.c`.**

**2. Modify your code file, as shown below:**

```

int keyscan(void)
{
    char key=0;    //Declare a variable key

```

```

RBPB=0;          //Set pull-up resistors on Port B

TRISB3=0;        //Set Port B3 as an output
TRISB2=0;        //Set Port B2 as an output
TRISB1=0;        //Set Port B1 as an output
TRISB0=0;        //Set Port B0 as an output

TRISB7=1;        //Set Port B7 as an input
TRISB6=1;        //Set Port B6 as an input
TRISB5=1;        //Set Port B5 as an input
TRISB4=1;        //Set Port B4 as an input

```

```

RB3=0;
  if(RB7==0) //If 1 key is pressed
  {
    key=1;
  }
  else if(RB6==0) //If 2 key is pressed
  {
    key=2;
  }
  else if(RB5==0) //If 3 key is pressed
  {
    key=3;
  }
  else if(RB4==0) //If A key is pressed
  {
    key=4;
  }
RB3=1;
RB2=0;
  if(RB7==0) //If 4 key is pressed
  {
    key=5;
  }
  else if(RB6==0) //If 5 key is pressed
  {
    key=6;
  }
  else if(RB5==0) //If 6 key is pressed
  {
    key=7;
  }
  else if(RB4==0) //If B key is pressed
  {
    key=8;
  }
RB2=1;

```



```

RB1=0;
if(RB7==0) //If 7 key is pressed
{
    key=9;
}
else if(RB6==0) //If 8 key is pressed
{
    key=10;
}
else if(RB5==0) //If 9 key is pressed
{
    key=11;
}
else if(RB4==0) //If C key is pressed
{
    key=12;
}
RB1=1;
RB0=0;
if(RB7==0) //If O key is pressed
{
    key=13;
}
else if(RB6==0) //If F key is pressed
{
    key=14;
}
else if(RB5==0) //If E key is pressed
{
    key=15;
}
else if(RB4==0) //If D key is pressed
{
    key=16;
}
RB0=1;

return key; //Return the value of key
}

```

Note that, if you try to compile this file, you will get errors, because it doesn't have a main function. You don't need to compile it. Just be sure to save it in the same folder that your main code file will be in, using the name "mxkeyscan.c".

## Challenge 2: Scan the Keypad

### Calling the Key Scanning Function

So far, mxkeyscan.c is the first library file that you have written, but you have been using library files all along. The file that you include at the top of every Machine Science code file--mxapi.h--is also a library file. Including this file allows you to use certain functions, like delay\_ms() and end(), in your code. From now on, to use the keyscan function in your code, you will have to include mxkeyscan.c at the top of your code file.

By including mxkeyscan.c in your code file, you can then call the keyscan function whenever you need it. The keyscan function returns the value of the variable called key, which ranges from 1 to 16, depending on which key is pressed. In this section, you will write code to call the keyscan function, and display the value returned by the function.

**1. In the Programming Window, start a new code file and save it as call\_keyscan.c.**

**2. Enter the following code into the window:**

```
#include "mxapi.h"
#include "mxkeyscan.c"
void main(void)
{
    int keynumber; //Declare a variable called keynumber
    lcd_init(); //Initialize the LCD
    while(1==1)
    {
        keynumber=keyscan(); //Set keynumber equal to the value
        returned by the keyscan function
        lcd_decimal(keynumber);
        delay_ms(400);
        lcd_instruction(CLEAR);
    }
}
```

**3. Compile and test your new code. (Note: When no key is pressed, the LCD will display a 0. This is because, when no key is pressed, the keyscan function returns a 0.)**

### Challenge 3: Enable Text Entry



As a next step, you will program the microcontroller to convert the input gathered from the keypad into real text messages--starting first with individual letters, then multiple letters. By the end of this challenge, you will be able to key in strings of text, which will be saved as arrays on the microcontroller.

### Challenge 3: Enable Text Entry

#### About ANSI Codes

In order to enter text messages, you will need to convert keypad input into letters and punctuation marks. Fortunately, the microcontroller has a built-in ability to translate numerical values into characters, using ANSI codes. The ANSI character codes were developed by the American National Standards Institute (ANSI), a non-profit organization that helps U.S. companies adhere to standards for the design of their products and systems. Figure 8 shows the ANSI codes for various characters.

ANSI	Character	ANSI	Character	ANSI	Character	ANSI	Character
65	A	77	M	89	Y	33	!
66	B	78	N	90	Z	34	"
67	C	79	O	48	0	35	#
68	D	80	P	49	1	36	\$
69	E	81	Q	50	2	37	%
70	F	82	R	51	3	38	&
71	G	83	S	52	4	39	'
72	H	84	T	53	5	44	,
73	I	85	U	54	6	45	-
74	J	86	V	55	7	46	.
75	K	87	W	56	8	47	/
76	L	88	X	57	9	63	?

Figure 7. ANSI codes for letters, numbers, and punctuation marks.

### Challenge 3: Enable Text Entry

#### Converting Keypad Input into Letters

Using an `lcd_character` statement, you can display the character equivalent of whatever ANSI code you pass to it. For example, `lcd_character(65)` displays an A, `lcd_character(66)` displays a B, and so on. In this section, you will introduce a variable called `ansi`, and set `ansi` equal to the value of `keynumber` plus 64. That way, if `keynumber` equals 1, `ansi` will equal 65 (A); if `keynumber` equals 2, `ansi` will equal 66 (B), and so on.

#### 1. Rename your code file `ansi_codes.c`.

## 2. Modify your code file, as follows:

```
#include "mxapi.h"
#include "mxkeyscan.c"

void main(void)
{
    int keynumber;
    int ansi;           //Declares new variable called ansi
    lcd_init();
    while(1==1)
    {
        keynumber=keyscan();
        ansi=keynumber+64;
        lcd_character(ansi);
        delay_ms(400);
        lcd_instruction(CLEAR);
    }
}
```

**3. Compile and test your new code.** (Note: When no key is pressed, the LCD will display the @ symbol. This is because, when no key is pressed, keynumber is equal to 0, and the ANSI code for the @ symbol is 64.)

### Challenge 3: Enable Text Entry Entering More Characters

So far, your code assigns one character to each key on the keypad, which limits you to only 16 characters--not enough for a real text message. To get around this problem, you need to assign multiple characters to each key, and then program the keypad to scroll through those letters with each key press. (Since most cell phones have only a numeric keypad, cell phone users employ a similar method for entering contact information and text messages.)

In this section, you will assign four letters to each key. Each time you press the key, you will scroll through those four letters. The 1 key will scroll through A, B, C, and D, the 2 key will scroll through E, F, G, and H, and so on. To do this, you will need to introduce another variable, called rotate, which will increase in value every time a key is pressed, cycling from 0 to 3 and then back to 0.

In order to convert keynumber to ansi, you will use the following formula:

$$\text{ansi} = (\text{keynumber} * 4) + 61 + \text{rotate}$$

This may look confusing, but it is a relatively straightforward equation. Figure 8 shows how it works for the first three keys in row one--the 1 key, the 2 key, and the 3 key.

Key	rotate	keynumber	*4=	+61=	+rotate=	Character
1	0	1	4	65	65	A
1	1	1	4	65	66	B
1	2	1	4	65	67	C
1	3	1	4	65	68	D
2	0	2	8	69	69	E
2	1	2	8	69	70	F
2	2	2	8	69	71	G
2	3	2	8	69	72	H
3	0	3	12	73	73	I
3	1	3	12	73	74	J
3	2	3	12	73	75	K
3	3	3	12	73	76	L

**Figure 8. Converting keynumber to ansi for scrolling text.**

**1. Rename your code file more\_characters.c.**

**2. Modify your code file as follows:**

```
#include "mxapi.h"
#include "mxkeyscan.c"

void main(void)
{
    int keynumber;
    int ansi;
    int rotate=0;
    lcd_init();
    while(1==1)
    {
        keynumber=keyscan();
        if(keynumber!=0)
        {
            ansi=(keynumber*4)+61+rotate;
            lcd_character(ansi);
            delay_ms(400);
            lcd_instruction(GOTO_LINE1+0);
        }
    }
}
```

```

        rotate=rotate+1;
        if (rotate==4)
        {
            rotate=0;
        }
    }
}

```

### 3. Compile and test your new code.

## Challenge 3: Enable Text Entry Building Strings of Characters

Now you can enter any character you want, but you can only enter and display one character at a time. To make text messages, you will want to build a string of characters. To do this, you will assign a specific function to one key--each time you press this key, it will shift the character entry position one space to the right on the LCD. Essentially, this will be your "Enter" key.

In the code below, you will program the O key to serve as the Enter key. You will declare another variable, called position. The position variable will be passed to an *lcd\_instruction* (GOTO) statement. Position will increase by 1 every time you press the O key. That way, the cursor will be repositioned before the next character is entered.

### 1. Rename your code file character\_string.c.

### 2. Modify your main function, as follows:

```

#include "mxapi.h"
#include "mxkeyscan.c"

void main(void)
{
    int keynumber;
    int ansi;
    int rotate=0;
    int position=0;
    lcd_init();
    while (1==1)
    {
        keynumber=keyscan();
        if (keynumber>0&&keynumber<13)
        {
            ansi=(keynumber*4)+61+rotate;

```

```

        lcd_instruction(GOTO_LINE1+position);
        lcd_character(ansi);
        delay_ms(400);
        rotate=rotate+1;
        if(rotate==4)
        {
            rotate=0;
        }
    }
    else if(keynumber==13)
    {
        position=position+1;
        rotate=0;
        delay_ms(400);
    }
}

```

### 3. Compile and test your new code.



#### PROGRAMMING NOTE

The double ampersands (&&) used in `character_string.c` represent a logical AND operator. The condition `(keynumber>0&&keynumber<13)` will be true only if `keynumber` is greater than 0 AND `keynumber` is less than 13.

## Challenge 3: Enable Text Entry Adding a Clear Key

The code you have written so far allows you to build and display strings of text, adding one character at a time from left to right. If you happen to make a mistake, it would be useful to back up and clear the last character you entered. In this section, you will program the F key to perform this function--shifting the cursor to the left and clearing the previous character. This will essentially become your "Clear" key. To clear characters you will use an `lcd_character` statement with an ANSI code of 32, which is a space character.

### 1. Rename your code file as `clear_key.c`.

### 2. Add another *else if...* statement to your code, assigning a specific routine to the F key, as follows:

```

else if(keynumber==14)
{
    lcd_instruction(GOTO_LINE1+position);

```



```

    lcd_character(32);
    position=position-1;
    delay_ms(400);
}

```

### 3. Compile and test your new code.

## Challenge 3: Enable Text Entry Creating an Array to Store Text Input

Using the code you have written so far, you can enter virtually any text you want. However, the data is not actually stored anywhere--it's simply displayed on the LCD. In order to transmit a text message, you will have to store the data, using an array.

An array is an ordered list of two or more values that is assigned a specific name. Figure 10 shows how to set up your array. The array is called message, and it has 16 values. Initially, all 16 values are set to 32 (the ANSI code for a blank character).

```

char message[16]={32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32};

```

**Figure 10. Setting up the message array.**

Figure 11 shows the procedure for changing a specific value in the message array.

```

message[position]=ansi;

```

**Figure 11. Setting a specific value in the message array.**

In the code below, you will set up the message array. Every time the Enter key is pressed, you will set the corresponding value in the array equal to ansi. Every time the Clear key is pressed, you will set the corresponding value in the array equal to 32. Finally, you will program the E key to display the entire message array on the second line of the LCD.

### 1. Rename your code file message\_array.c.

### 2. Modify your code file, as follows:

```

#include "mxapi.h"
#include "mxkeyscan.c"

char message[16]={32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32};

```

```

void main(void)
{
    int keynumber;
    int ansi;
    int rotate=0;
    int position=0;
    lcd_init();
    while(1==1)
    {
        keynumber=keyscan();
        if(keynumber>0&&keynumber<13)
        {
            ansi=(keynumber*4)+61+rotate;
            lcd_instruction(GOTO_LINE1+position);
            lcd_character(ansi);
            delay_ms(400);
            rotate=rotate+1;
            if(rotate==4)
            {
                rotate=0;
            }
        }
        else if(keynumber==13)
        {
            message[position]=ansi;
            position=position+1;
            rotate=0;
            delay_ms(400);
        }
        else if(keynumber==14)
        {
            lcd_instruction(GOTO_LINE1+position);
            lcd_character(32);
            message[position]=32;
            position=position-1;
            delay_ms(400);
        }
        else if(keynumber==15)
        {
            lcd_instruction(GOTO_LINE2+0);
            lcd_text(message);
            delay_ms(400);
        }
    }
}

```

**3. Compile and test your new code. You should be able to key in a message, one character at a time, and then press the D key to display the entire message on the LCD's second line.**

### Challenge 3: Enable Text Entry Extra Activities

If you have finished with all of the tasks in this chapter, try tackling some of the additional activities described below.

1. Program an unused key to display other useful characters, such as lowercase letters, numbers, or punctuation marks. (Hint: You will need to add one or more *else if...* statements for these keys.)
2. Program an unused key to serve as a Reset key, which clears the whole LCD and erases the stored message. (Hint: You will need to add another *else if...* statement for this key. Inside this statement, you will need separate statements to clear the LCD, reset the values of the message array to 32, and set all of your variables back to 0.)
3. Using pieces of tape or adhesive-backed label paper, affix labels to each key indicating its group of letters or its function.

### Unit 2: Decoding Infrared Signals

Using your keypad code, you can enter short messages and display them on the LCD, and with your message array, you can store a sequence of characters on the microcontroller. In order to build a working text messenger, you need to program your the microcontroller to transmit this data, and to receive text messages from other microcontrollers.

Cell phones, pagers, and text-messaging devices have antennas that emit and receive radio frequency signals, communicating with towers that telecommunication companies have established along roadways and in densely populated areas. These signals can pass through walls and have a range of several miles. In contrast, your text messenger will be communicate using carefully timed pulses of infrared light, which require "line of sight" proximity.

In this unit, you will connect an infrared receiver and program the microcontroller to decipher infrared signals. In the next unit, you will connect an infrared transmitter to the XBoard and program it to send messages. Figure 1 shows the infrared transmitted and receiver used in this project.

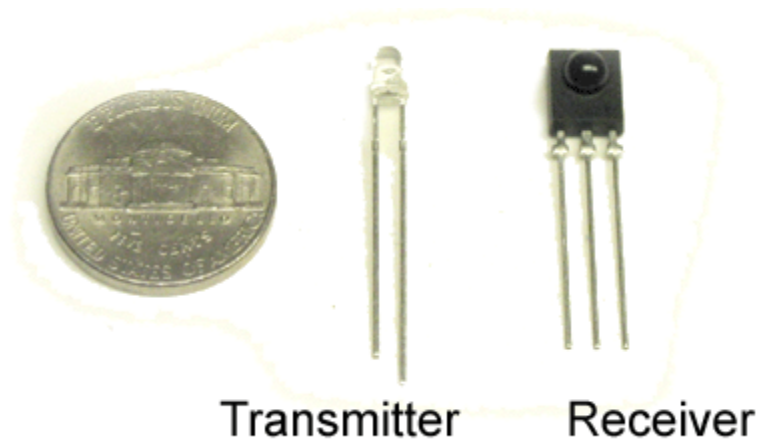


Figure 1. Infrared transmitter (left) and receiver (right).

### Challenge 1: Install the Infrared Receiver



Challenge 1 is a simple hardware task--adding the infrared receiver to the XBoard. The installation of the receiver is very simple. It has three prongs that connect to power, ground, and the microcontroller.

### Challenge 1: Install the Infrared Receiver Collecting Your Components

In order to complete Unit 2, you will need the following components (shown in Figure 2):

Part	Quantity	Description
A	1	Infrared receiver
B	2	Jump wires (1 short yellow and 1 short orange)
C	1	Flexible jump wire



Figure 2. Text Messenger Unit 2 components.

## Challenge 1: Install the Infrared Receiver

### Installing the Infrared Receiver

The infrared receiver has three prongs, which connect to power, to ground, and to Port D0 on the microcontroller.

1. Make sure the power switch on your battery pack is in the OFF position.
2. Insert a short yellow jump wire, a short orange jump wire, a flexible blue jump wire, and the infrared receiver into the XBoard, as shown in the video to the right and in Figure 3. **NOTE: Make sure that the small shiny dome on the infrared receiver is oriented exactly as shown.**
3. Connect the loose end of the flexible blue jump wire to Port D0 on the microcontroller. (If you need help finding Port D0, refer to the [PIC Microcontroller Quick Reference document](#).)

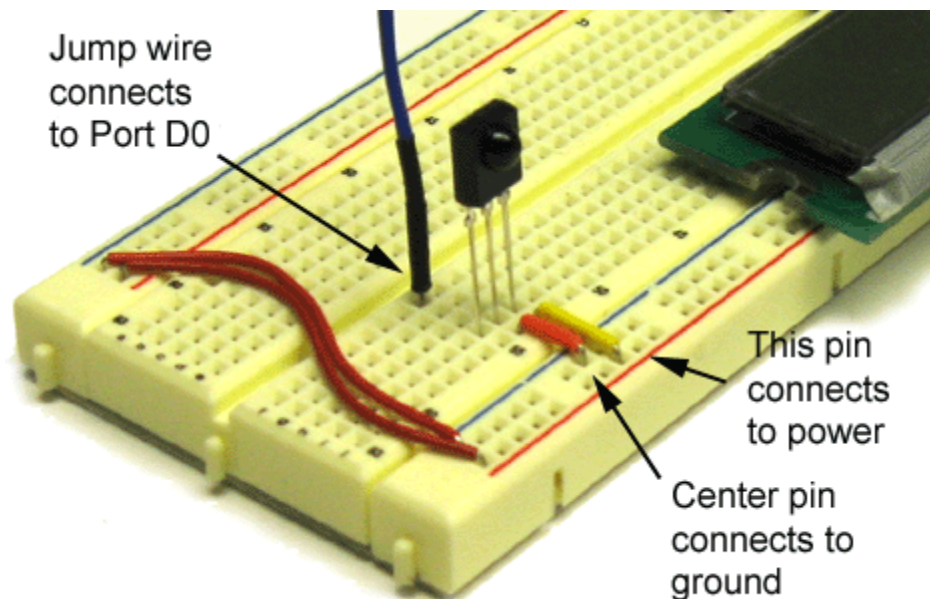


Figure 3. Infrared receiver properly installed.

## Challenge 2: Understand Infrared Communication



Your second challenge is to understand the basics of infrared communication. This includes learning about the infrared signals sent out by remote control devices, as well as the binary number system (a system of counting used by computers).

## Challenge 2: Understand Infrared Communication About Infrared Signals

Many electronic devices use infrared signals to communicate. For example, any time you push a button on a remote control for a TV or stereo, the remote control sends out a coded, infrared signal, as shown in Figure 4. The device being controlled has an infrared receiver, which relays the signal from the remote to a microcontroller. The microcontroller decodes the signal and then initiates tasks, such as adjusting the volume on the TV, switching the VCR to fast-forward, or flipping the channel on the cable box from CNN to HBO.

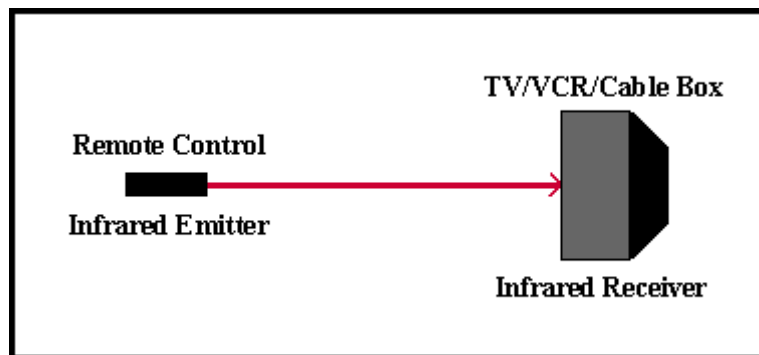


Figure 4. Infrared remote control.

The signal from the remote control is not a continuous beam of infrared light--it is a series of intermittent short and long pulses. The receiver translates these short and long pulses into periods of high voltage (5 volts) and low voltage (0 volts), which the microcontroller interprets as 1s and 0s. Figure 5 shows how this works.

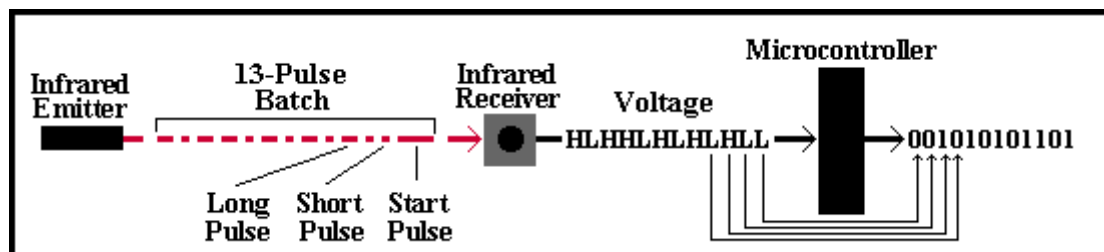


Figure 5. Translating infrared pulses into 1s and 0s.

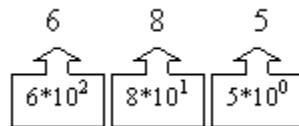
The pulses from the infrared emitter are sent in batches, with 13 pulses in each batch. The first pulse is a "start" pulse, which signals that a batch is starting. Each of the next 12 pulses is either long or short, with long pulses representing 1s and short pulses representing 0s. If you look closely at Figure 5, you will notice that, after the start pulse, the infrared receiver sends a high voltage signal for every long pulse it receives and a low voltage signal for every long pulse it receives. The microcontroller then interprets these signals as 1s and 0s, respectively, and arranges them into 12-digit strings, such as 001010101101.

Notice that the order of the 1s and 0s is the reverse of the order of the infrared pulses.

## Challenge 2: Understand Infrared Communication About Binary Numbers

To the microcontroller, every string of 1s and 0s has a specific meaning: it represents a *binary* number. Binary is a system of counting used by computers. Binary counting differs from our usual way of counting in one important respect--binary uses only two digits: 1 and 0. The number system we commonly use is called the *decimal* system, and it is based on 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. We can use these 10 digits to represent any number we want--from single-digit numbers, like 3 and 7, to multi-digit numbers, like 685.

Each digit in a multi-digit decimal number has a different value, depending on its place in the number. For example, in the number 685, the digit 6 is in the hundreds place, meaning it has a value of 600. The digit 8 is in the tens place, and has a value of 80. The 5 is in the ones place, and has a value of 5. This is represented schematically below:

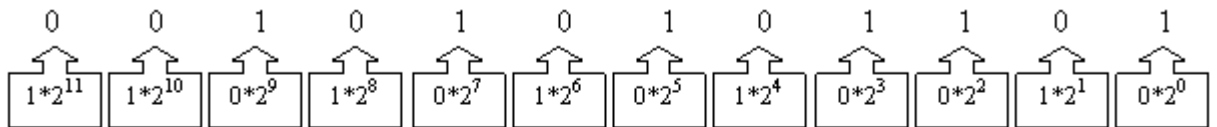


$$\begin{aligned} 6 \cdot 100 + 8 \cdot 10 + 5 \cdot 1 &= \\ 600 + 80 + 5 &= \\ 685 \end{aligned}$$

Seems obvious, doesn't it? You are so used to seeing decimal numbers, you probably don't realize that you are performing a calculation each time you see one, but you are!

Likewise, each digit in a binary number has a different value, depending on its place. The value of each digit is based on a power of 2, rather than a power of 10. For example, consider the first string of 1s and 0s sent to the microcontroller from the infrared receiver in the previous section--001010101101. The value of this binary number can be calculated, as follows:





$$\begin{aligned}
 &0*2048 + 0*1024 + 1*512 + 0*256 + 1*128 + 0*64 + 1*32 + 0*16 + 1*8 + 1*4 + 0*2 + 1*1 = \\
 &\quad 0 + 0 + 512 + 0 + 128 + 0 + 32 + 0 + 8 + 4 + 0 + 1 = \\
 &\quad \quad \quad 685
 \end{aligned}$$

As it turns out, the decimal number 685 and the binary number 001010101101 have the same value. This many seem confusing, but the important thing to remember is that any number can be represented in binary, using just 1s and 0s, and any binary number can be easily converted into a decimal number, using the process shown above.

### Challenge 3: Decipher Signals from the Remote Control



In Challenge 3, you will program the microcontroller to receive and interpret the coded signals sent by the remote control. First, you will write code to detect any signal. Next, you will write code to detect the specific start pulse that signals the beginning of a transmission from the remote. By the end of this challenge, you will have written a library function that returns a number whenever the receiver gets a transmission from the remote control.

### Challenge 3: Decipher Signals from the Remote Control Setting Up the Remote Control

In order to use the remote control, you must first install batteries in the device and program it with the correct TV code. The procedure to do this is different for every remote. Below are the instructions to program the Radio Shack 3-in-1 Remote Control, the Aifa URC4, and the Philips Universal Remote.



Figure 6. Installing remote batteries.  
The Radio Shack 3-in-1 is shown at the top, and the Aifa URC4 is shown below.

***For the Radio Shack 3-in-1 Remote Control, use the following procedure:***

1. Insert two AAA batteries in the remote control, as shown above in Figure 6. ***NOTE: Batteries are not included with Remote Expansion Pack.***
2. Press and hold the CODE SEARCH button on the remote control until the red indicator stays on.
3. Press the TV button. The red indicator light should blink and then stay lit.
4. Press the 4 button, then the 1 button, then the 4 button. The red light should go off. If it stays lit, repeat steps 2, 3, and 4.

***For the Aifa URC4 Remote, use the following procedure:***

1. Insert two AAA batteries in the remote control, as shown above in Figure 5. ***Batteries are not included with Remote Expansion Pack.***
2. Press the green SET button on the remote control and hold it while pressing the TV button.

3. When the red indicator light comes on, release both buttons. The red light should stay on.

4. Press the 1 button, then the 2 button, then the 8 button. The red light should go off. If it stays lit, repeat steps 2, 3, and 4.

***For the Philips Universal Remote, use the following procedure:***

1. Insert two AAA batteries in the remote control, as shown above in Figure 5. *Batteries are not included with Remote Expansion Pack.*

2. Press the CODE SEARCH button and hold it down until the red LED stays lit.

3. Press the 0 button, then the 4 button, then the 1 button, then the 4 button. The red light should go off. If it stays lit, repeat steps 1 and 2.

### **Challenge 3: Decipher Signals from the Remote Control**

#### **Receiving a Signal from the Remote**

As a first step, you will write code to indicate on the LCD whenever the infrared receiver detects a signal from the remote control. This is relatively straightforward, because any signal from the remote control will cause Port D0 to go low (i.e. have a value of 0).

1. Start a new code file in the Programming Portal and name it `got_signal.c`.

2. Enter the following code:

```
#include "mxapi.h"
void main(void)
{
    lcd_init();
    TRISD0=1;        //Sets up Port D0 as input
    while(RD0==1);    //Waits here while D0 is high
    lcd_text("Got signal!"); //Displays "Got signal!"
    end();
}
```

3. Compile and test your new code. Point the remote at your XBoard and press a button--your LCD should display the message "Got signal!" Press the reset button on the XBoard, and try again.

#### **PROGRAMMING NOTE**



In certain rare instances, the infrared receiver may pick up stray signals from sources other than the remote control, causing the LCD to display the "Got signal!" message spontaneously. If you notice unexpected "Got signal!" messages, try testing your device in another room.

### Challenge 3: Decipher Signals from the Remote Control Measuring the Start Pulse

The remote control sends infrared signals in batches of 13 pulses, beginning with a "start" pulse. Each start pulse lasts about 2.5 milliseconds. Whenever you press a button, the voltage at Port D0 goes low, remains low for about 2.5 milliseconds, and then goes high again. Using one of the microcontroller's internal timers, you can precisely measure the length of the start pulse.

1. Using the **Save As** command, rename your code file `measure_pulse.c`.
2. Modify your code file, as follows:

```
#include "mxapi.h"
void main(void)
{
    int pulse_length; //Declare a variable
    lcd_init();        //Initialize the LCD
    tmr0_init(DIV_64); //Initialize the internal timer
    TRISD0=1;          //Set Port D0 as an input
    while(RD0==1);      //Wait here while D0 is high
    TMR0=0;             //Reset the internal timer to 0
    while(RD0==0);      //Wait here while D0 is low
    pulse_length=TMR0;  //Record timer value
    lcd_decimal(pulse_length); //Display timer value
    end();
}
```

3. Compile and test your new code.

#### PROGRAMMING NOTE



The PICF16877 microcontroller has three internal timers--TMR0, TMR1, and TMR2. TMR0 and TMR2 constantly cycle from 0 to 255, and TMR1 constantly cycles from 0 to 65535. You can use these timers to precisely measure the duration of certain

events or to perform a task for a precise period of time.

### Challenge 3: Decipher Signals from the Remote Control Timing the Start Pulse in Microseconds

The DIV\_64 in the line initializing timer 0 means that TMR0 increases in value by 1 approximately every 64 microseconds. Therefore, by multiplying TMR0's value by 64, you can calculate the duration of the start pulse in microseconds. The following code example performs this calculation and displays the result on the LCD.

1. Using the Save As command, rename your code file `time_pulse.c`.

2. Modify your main function, as follows:

```
#include "mxapi.h"
void main(void)
{
    int pulse_length;
    lcd_init();
    tmr0_init(DIV_64);
    TRISD0=1;
    while(RD0==1);
    TMR0=0;
    while(RD0==0);
    pulse_length=TMR0;
    pulse_length=pulse_length*64;
    lcd_decimal(pulse_length);
    lcd_text("  microsecs");
    end();
}
```

3. Compile and test your new code.

### Challenge 3: Decipher Signals from the Remote Control Detecting a Start Pulse

As you can see, there is some variation in the length of the start pulse, but the value is usually quite close to 2500 microseconds. Using this information, you can program the microcontroller to recognize a start pulse. This will be important later on, because a start pulse will signal the beginning of a transmission of 0s and 1s.

1. Using the Save As command, rename your code file start\_pulse.c.

2. Modify your main function, as follows:

```
#include "mxapi.h"
void main(void)
{
    int pulse_length;
    lcd_init();
    tmr0_init(DIV_64);
    TRISD0=1;
    while(RD0==1);
    TMR0=0;
    while(RD0==0);
    pulse_length=TMR0;
    pulse_length=pulse_length*64;

    if((pulse_length>2300)&&(pulse_length<2800))
    {
        lcd_text("Got start pulse!");
    }
    end();
}
```

3. Compile and test your new code. Press reset and try different buttons on the remote.

### Challenge 3: Decipher Signals from the Remote Control Detecting Ones and Zeros

Every start pulse is followed by a series of long and short pulses, representing 1s and 0s, respectively. Each long pulse lasts approximately 1,300 microseconds, and each short pulse lasts about 700 microseconds. In this section, you will write code to detect whether the first pulse that arrives after the start pulse is long or short, and display either "One" or "Zero" on the LCD.

1. Using the Save As command, rename your code file other\_pulses.c.

2. Modify your main function, as follows:

```
#include "mxapi.h"
void main(void)
{
    int pulse_length;
    lcd_init();
    tmr0_init(DIV_64);
```

```

TRISD0=1;
while(RD0==1);
TMR0=0;
while(RD0==0);
pulse_length=TMR0;
pulse_length=pulse_length*64;

if((pulse_length>2300)&&(pulse_length<2800))
{
    while(RD0==1);
    TMR0=0;
    while(RD0==0);
    pulse_length=TMR0;
    pulse_length=pulse_length*64;
    lcd_text("Start plus ");
    if((pulse_length>500)&&(pulse_length<1000))
    {
        lcd_text("Zero");
    }
    if((pulse_length>1100)&&(pulse_length<1800))
    {
        lcd_text("One");
    }
}
end();
}

```

**3. Compile and test your new code.** Press reset and try different buttons on the remote. Some will produce "plus Zero" messages, while others will produce "plus One" messages.

### Challenge 3: Decipher Signals from the Remote Control Making Strings of Ones and Zeros

Now that you can detect start pulses and pulses representing 1s and 0s, you need to build strings of 1s and 0s that the microcontroller can store as binary numbers. Fortunately, there is an easy way to do this in C. To start, you need to declare a variable called `ir_code` to store your binary number. Your variable will be a 12-digit number, with an initial value of 000000000000. Then, with the following code, you can "shift" all 12 digits of the binary value of `ir_code` one place to the left and insert a new digit:

```

ir_code=(ir_code<<1)+1;
ir_code=(ir_code<<1)+0;

```

**1. Using the Save As command, rename your code file `binary_string.c`.**



## 2. Modify your code file, as follows:

```
#include "mxapi.h"

void main(void)
{
    int pulse_length;
    int ir_code;
    int i;
    lcd_init();
    tmr0_init(DIV_64);
    TRISD0=1;
    while(RD0==1);
    TMR0=0;
    while(RD0==0);
    pulse_length=TMR0;
    pulse_length=pulse_length*64;

    if((pulse_length>2300)&&(pulse_length<2800))
    {
        for(i=0; i<12; i++)
        {
            while(RD0==1);
            TMR0=0;
            while(RD0==0);
            pulse_length=TMR0;
            pulse_length=pulse_length*64;

            if((pulse_length>500)&&(pulse_length<1000))
            {
                ir_code=(ir_code<<1)+0;
            }
            if((pulse_length>1100)&&(pulse_length<1800))
            {
                ir_code=(ir_code<<1)+1;
            }
        }
    }
    lcd_digits(ir_code, BASE_2, LEADING_ZEROS, 12);
    end();
}
```

**3. Compile and test your new code. Whenever you press a button on the remote, your LCD should display a different 12-digit binary number on the LCD. NOTE: You will need to press the reset button between each trial.**

## Challenge 3: Decipher Signals from the Remote Control Converting Your Infrared Receiving Code into a Library Function

You will need your infrared receiving code for the next step in this project--programming the XBoard to receive text messages from other XBoards. Since the infrared receiving code is quite long, you should make it into a library file, just as you did with your keypad scanning code in the previous section.

1. Using the **Save As** command, save your code file as `mxread_ir.c`.
2. At the top of your code file, delete the include statements, the line initializing the LCD, and the line initializing the timer, and rename your main function "read\_ir," as shown below. Note that there is an "int" declaration before the function name, indicating that the read\_ir function will return an integer value. At the bottom of your code file, add a *return* statement, delete the *lcd\_digits* and *end* statements. The completed library function is shown below:

```
int read_ir(void)
{
    int pulse_length;
    int ir_code;
    int i;
    TRISD0=1;
    tmr0_init(DIV_64);
    while(RD0==1);
    TMR0=0;
    while(RD0==0);
    pulse_length=TMR0;
    pulse_length=pulse_length*64;

    if((pulse_length>2300)&&(pulse_length<2800))
    {
        for(i=0; i<8; i++)
        {
            while(RD0==1);
            TMR0=0;
            while(RD0==0);
            pulse_length=TMR0;
            pulse_length=pulse_length*64;

            if((pulse_length>500)&&(pulse_length<1000))
            {
                ir_code=(ir_code<<1)+0;
            }
        }
    }
}
```

```

        if ( (pulse_length>1100) && (pulse_length<1800) )
        {
            ir_code=(ir_code<<1)+1;
        }
    }
    return(ir_code);
}

```

**NOTE:** The library file will not compile on its own. It works only within the context of a code file that calls the `read_ir` function. In the next section, you will write a code file and include this library file.

### Challenge 3: Decipher Signals from the Remote Control Calling the Library Function

By including `mxread_ir.c` in your code file, you can then call the `read_ir` function whenever you need it. The `read_ir` function returns the value of the variable called `ir_code`.

1. In the Programming Window, start a new code file called `call_read_ir.c`.
2. Enter the following code into the Programming Window:

```

#include "mxapi.h"
#include "mxread_ir.c"
void main(void)
{
    int ir_code=0;
    lcd_init();
    while(1==1)
    {
        ir_code=read_ir();
        lcd_instruction(GOTO_LINE1+0);
        lcd_digits(ir_code, BASE_2, LEADING_ZEROS, 12);
    }
    end();
}

```

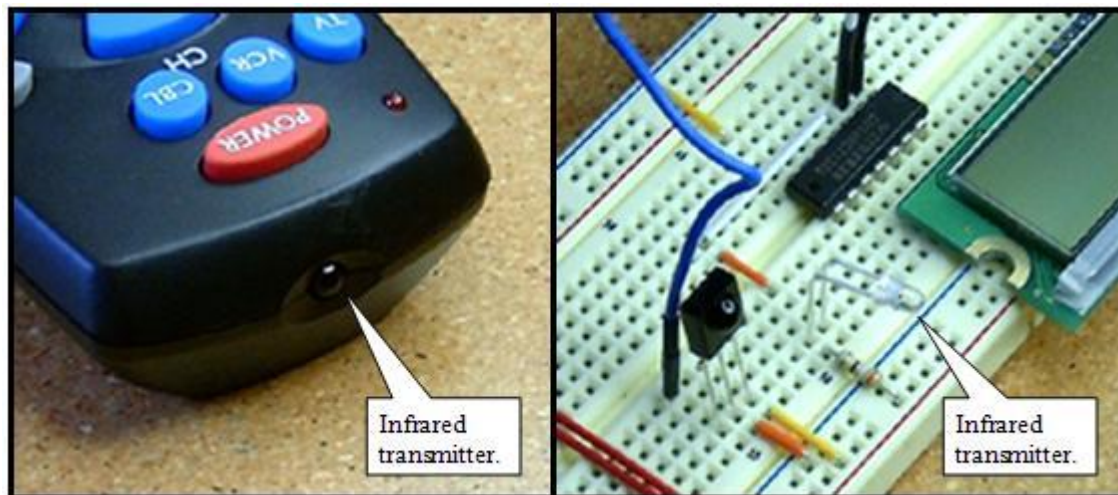
3. Compile and test your new code.

## Unit 3: Sending and Receiving Text

By now, you have completed two important steps toward building a working text messenger. In Unit 1, you added a keypad and programmed the microcontroller

to decipher input from the keypad. In Unit 2, you connected an infrared receiver to your board, and learned how to gather and interpret signals from an infrared remote control. You have created two library files-- **mxkeyscan.c** and **mxir\_read.c**--that enable text entry and infrared signal reception.

In this unit, you will connect an infrared transmitter to the breadboard and program the microcontroller to send messages from one board to another. The infrared transmitter is similar to the one found on the front end of the remote control. Figure 1 shows the remote control's transmitter, next to the breadboard with its infrared transmitter attached.



**Figure 1. Remote control transmitter (left) and breadboard infrared transmitter (right).**

### **Challenge 1: Install the Infrared Transmitter**



To begin, you must add an infrared transmitter to the breadboard. Later, you will program the transmitter to send infrared signals, just as signals are emitted by the infrared transmitter at the end of the remote control. The installation of the transmitter is slightly more involved than the receiver, requiring a resistor and a small integrated circuit.

### **Challenge 1: Install the Infrared Transmitter Collecting Your Components**

In order to complete Unit 3, you will need the following components (shown in Figure 2):

Part	Quantity	Description
------	----------	-------------

A	1	Infrared transmitter
B	1	Integrated circuit (Texas Instruments 74HCT132E)
C	4	Jump wires (2 short yellow, 1 short orange, and 1 white)
D	1	Resistor (390 Ohm)
E	2	Flexible jump wires (1 black and 1 white)

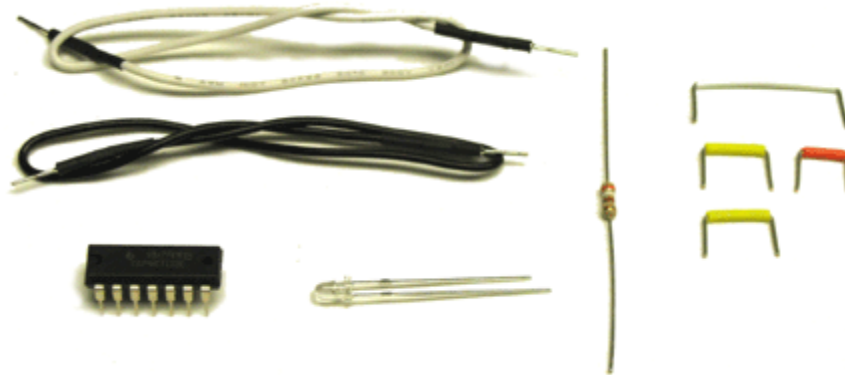


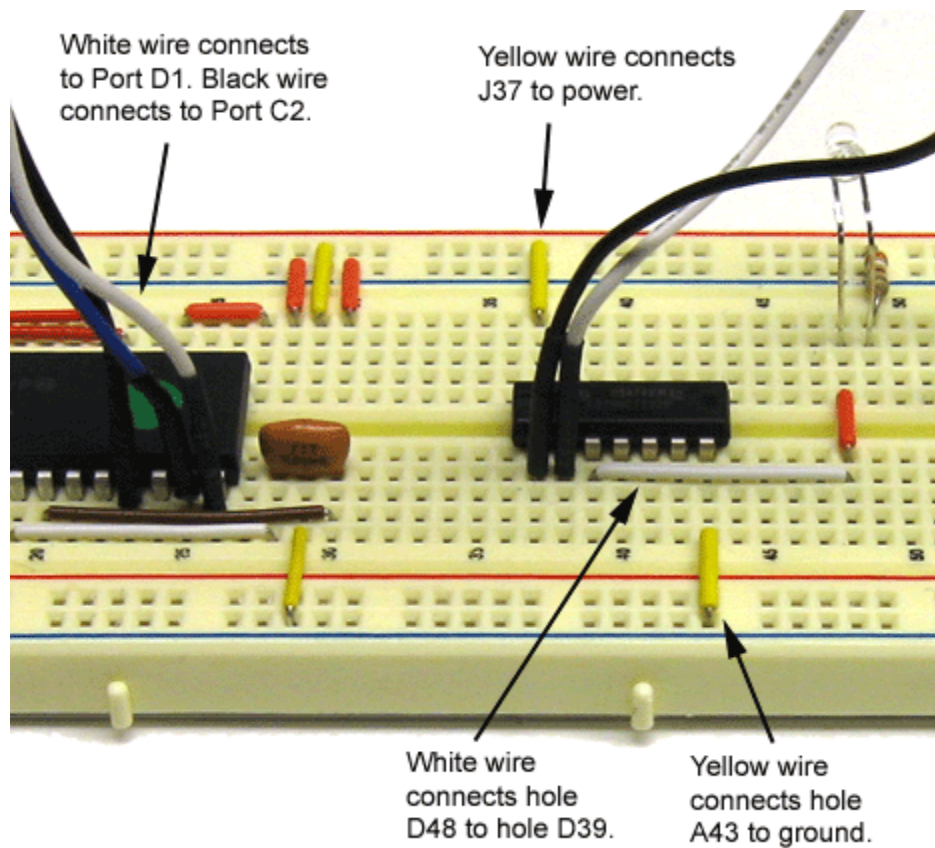
Figure 2. Text Messenger Unit 3 components.

### Challenge 1: Install the Infrared Transmitter

#### Installing the Infrared Transmitter

The infrared transmitter should be installed on the breadboard between the infrared receiver and the LCD. An integrated circuit, which looks like a miniature version of the microcontroller, is needed to control the output from the transmitter.

1. Make sure the power switch on your battery pack is in the OFF position.
2. Remove the LCD screen from the XBoard. (NOTE: Take note of the position of the LCD's pins, so that you can reinstall it in step 4.)
3. Insert the integrated circuit, a yellow jump wire, a white jump wire, an orange jump wire, and two flexible jump wires into the breadboard, as shown in the video and in Figure 3.



4. Return the LCD to its original position.

5. Look closely at the infrared transmitter. Find the flat edge along rim of the diode, and the shorter of the two pins. Carefully bend a 90-degree angle in the two pins, keeping the transmitter oriented as shown in Figure 4.

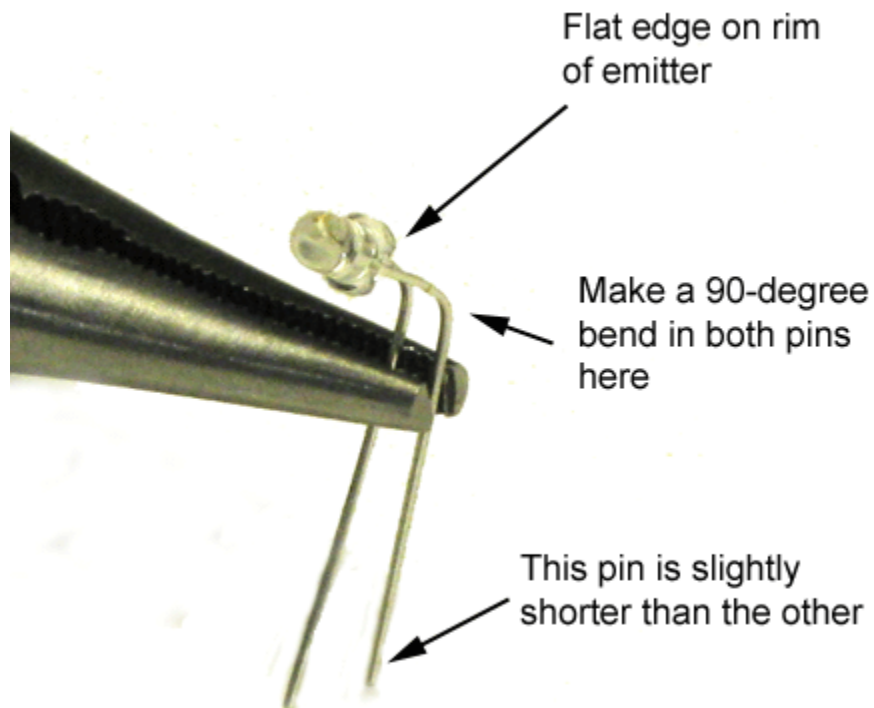


Figure 4. Bending the infrared transmitter.

6. Insert the infrared transmitter and a 390-Ohm resistor into the XBoard, as shown in Figure 5. **NOTE:** *The transmitter should face in the same direction as the infrared receiver. The transmitter should go in holes I48 and I49, and the resistor should connect hole J49 to power.*



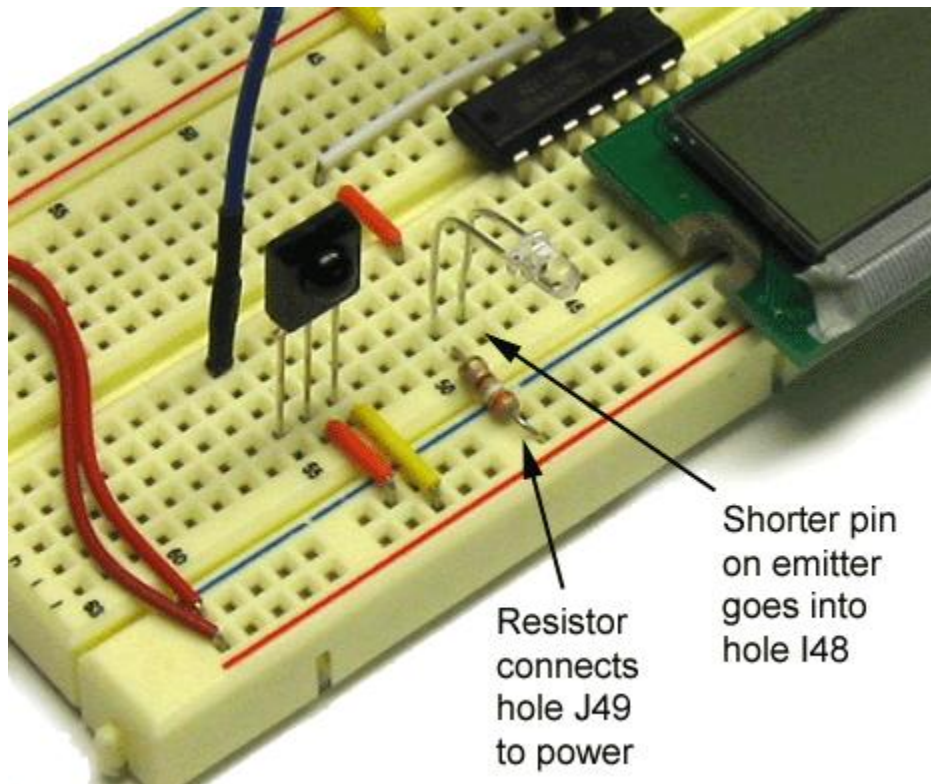


Figure 5. Installing the infrared transmitter.

## Challenge 2: Send Infrared Signals



In this challenge, you will program the microcontroller to send out infrared signals from the transmitter. In order to check whether your transmitter is working, you will need to have two boards--one board to send signals (the SENDER board) and another board to receive signals (the RECEIVER board). For this reason, you *must have a partner* for this challenge. You and your partner will take turns sending and receiving signals to verify that both boards are working properly. Before long, you will be able to send whole text messages back and forth.

## Challenge 2: Send Infrared Signals Finding a Partner

1. Find a partner to work with on this challenge.
2. Position your board so that the infrared receiver and transmitter on one board are aligned with the corresponding components on the other board, as shown in Figure 6. **NOTE:** You may have to reposition one or both boards in order to program and reprogram them. Always position them as shown in Figure 6 when testing your code.

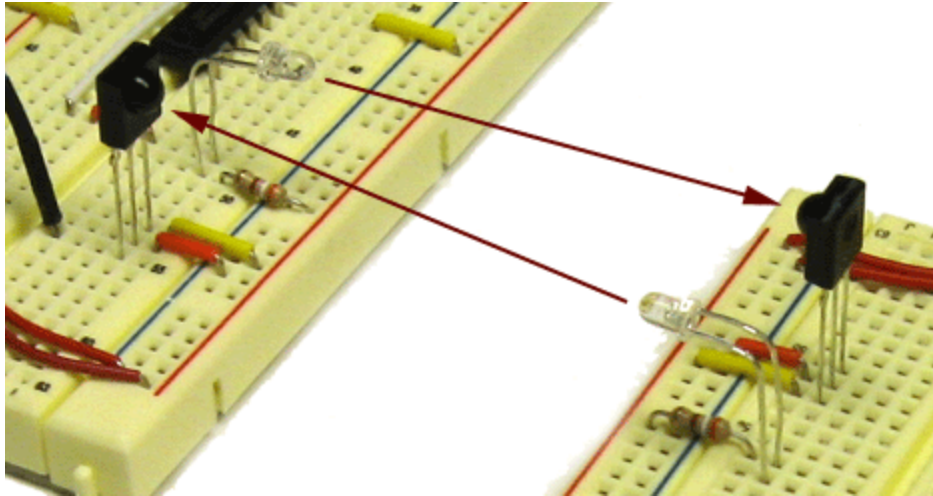


Figure 6. Two XBoards facing each other.

## Challenge 2: Send Infrared Signals

### Sending a Signal

As a first step, you will program the SENDER board to transmit an infrared signal.

1. Decide which board will be the RECEIVER and which will be the SENDER.
2. Program the RECEIVER board with the `got_signal.c` code that you wrote in Unit 2.
3. On the computer hooked up to the SENDER board, launch the Programming Portal, and open the `message_array.c` file that you wrote in Unit 1.
4. Using the Save As command, rename `message_array.c` code file `send_signal.c`.
5. Modify your code file, as follows:

```
#include "mxapi.h"
#include "mxkeyscan.c"

char message[16]={32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32};
void main(void)
{
    int keynumber;
    int ansi;
    int rotate=0;
```

```

int position=0;
lcd_init();
TRISD1=0;
RD1=0;
pwm_init(38000,50,50);

while(1==1)
{
    keynumber=keyscan();
    if(keynumber>0&&keynumber<13)
    {
        ansi=(keynumber*4)+61+rotate;
        lcd_instruction(GOTO_LINE1+position);
        lcd_character(ansi);
        delay_ms(400);
        rotate=rotate+1;
        if(rotate==4)
        {
            rotate=0;
        }
    }
    else if(keynumber==13)
    {
        message[position]=ansi;
        position=position+1;
        rotate=0;
        delay_ms(400);
    }
    else if(keynumber==14)
    {
        lcd_instruction(GOTO_LINE1+position);
        lcd_character(32);
        message[position]=32;
        position=position-1;
        delay_ms(400);
    }
    else if(keynumber==15)
    {
        lcd_instruction(GOTO_LINE2+0);
        lcd_text(message);
        RD1=1;
        delay_us(500);
        RD1=0;
        delay_ms(400);
    }
}
}

```

6. Compile and test your new code. Remember to position the two XBoards as shown in Figure 6. Use the reset button to reset the code on the RECEIVER board and make sure the RECEIVER is consistently receiving signals from the SENDER.

7. Reprogram the boards so that the RECEIVER becomes the SENDER, and vice versa, and make sure both boards are working properly.

## Challenge 2: Send Infrared Signals

### Sending a Start Pulse

Once you have sent your first signal from one board to another, you are ready to send a start pulse. Remember that every transmission from the remote control began with a start pulse, lasting approximately 2500 microseconds.

1. Decide which board will be the RECEIVER and which will be the SENDER.

2. Program the RECEIVER with the start\_pulse.c code that you wrote in Unit 2.

3. On the computer hooked up to the SENDER board, load your send\_signal.c file and rename it send\_start\_pulse.c.

4. In the section of the file that controls the E (send) key, make the following modifications:

```
else if(keynumber==15)
{
    lcd_instruction(GOTO_LINE2+0);
    lcd_text(message);
    RD1=1;
    delay_us(2500);
    RD1=0;
    delay_ms(400);
}
```

5. Compile and test your new code. Remember to orient the two boards as shown in Figure 6. Use the reset button to reset the code on the RECEIVER board and make sure the RECEIVER is consistently receiving start pulses from the SENDER.

6. Reprogram the boards so that the RECEIVER becomes the SENDER, and vice versa, and make sure both boards are working properly.

## Challenge 2: Send Infrared Signals Sending Ones and Zeros

1. Decide which board will be the RECEIVER and which will be the SENDER.
2. Program the RECEIVER with the 'other\_pulses.c' code that you wrote in Unit 2.
3. On the computer hooked up to the SENDER board, load your 'send\_start\_pulse.c' code file and rename it 'send\_other\_pulse.c'.
4. In the section of the file that controls the E (send) key, make the following additions:

```
else if(keynumber==15)
{
    lcd_instruction(GOTO_LINE2+0);
    lcd_text(message);
    RD1=1;
    delay_us(2500);
    RD1=0;
    delay_us(500);
    RD1=1;
    delay_us(750);
    RD1=0;
    delay_ms(400);
}
```

5. Compile and test your new code. Remember to orient the two XBoards as shown in Figure 6. Use the reset button to reset the code on the RECEIVER board and make sure the RECEIVER is consistently receiving other pulses from the SENDER.
6. Reprogram the boards so that the RECEIVER is the SENDER, and vice versa, and make sure both boards are working properly.
7. Modify the SENDER code so that it sends the pulse for a "One" instead of a "Zero" after the start pulse.

## Challenge 2: Send Infrared Signals Sending Binary Numbers

Each text character that you transmit from the SENDER board to the RECEIVER board will be represented by a string of 1s and 0s--in other words, a binary number. Unlike the remote control, which sent 12-digit strings of 1s and 0s after each start pulse, your infrared transmitter will use only 8 digits to send each character. In order to do this efficiently, you will write a function, called pulse, with a loop that sends each of the eight digits one at a time.

**1. Decide which board will be the RECEIVER and which will be the SENDER**

.

**2. On the computer hooked up to the RECEIVER, load the mxread\_ir.c file that you wrote in Unit 2.**

**3. Modify the *for...* loop in the mxread\_ir.c file as follows:**

```
for(i=0; i<8; i++)
{
    while(RD0==1);
    TMR0=0;
    while(RD0==0);
    pulse_length=TMR0;
    pulse_length=pulse_length*16;

    if((pulse_length>500)&&(pulse_length<1000))
    {
        ir_code=(ir_code<<1)+0;
    }
    if((pulse_length>1100)&&(pulse_length<1800))
    {
        ir_code=(ir_code<<1)+1;
    }
}
```

**4. Save your new mxread\_ir.c file.**

**5. On the computer hooked up to the RECEIVER, load the call\_read\_ir.c code file that you wrote in Unit 2. Rename the file receive\_binary.c.**

**6. Modify your receive\_binary.c code file as follows:**

```
#include "mxapi.h"
#include "mxread_ir.c"
void main(void)
{
    int ir_code=0;
    lcd_init();
```

```

    ir_code=read_ir();
    lcd_digits(ir_code, BASE_2, LEADING_ZEROS, 8);
    end();
}

```

**7. Compile your code and program the RECEIVER board.**

**8. On the computer hooked up to the SENDER board, load your send\_other\_pulses.c file and rename it send\_binary.c.**

**9. Near the top of the file, right after the statement declaring your message array, add a new function called pulse:**

```

void pulse(void)
{
    int i=0;
    RD1=1;
    delay_us(2500);
    RD1=0;
    delay_us(500);

    for(i=0; i<8; i++)
    {
        RD1=1;
        delay_us(1300);
        RD1=0;
        delay_us(500);
    }
}

```

**10. In the section of the code file that controls the E (send) key, make the following additions:**

```

else if(keynumber==15)
{
    lcd_instruction(GOTO_LINE2+0);
    lcd_text(message);
    pulse();
    delay_ms(400);
}

```

**11. Compile and test your new code. Remember to position the two XBoards as shown in Figure 6. Use the reset button to reset the code on the RECEIVER board and make sure the RECEIVER is consistently receiving other pulses from the SENDER.**



12. Reprogram the boards so that the RECEIVER is the SENDER, and vice versa, and make sure both boards are working properly.

## Challenge 2: Send Infrared Signals About AND-ing Binary Numbers

In the next step, you will pass a binary number to your pulse function, which will then send out a long pulse for each 1 in the number and a short pulse for each 0. To do this, your function will perform an operation called "ANDing" to determine the value of each digit in the number. "ANDing" is a mathematical operation, like adding or multiplying. The symbol for "ANDing" is the ampersand (&), just as the symbol for adding is the plus (+) and the symbol for multiplying is the asterisk (\*).

When you AND two binary numbers together, the microcontroller lines up all the digits in the two numbers, compares each digit, and produces a result. If both digits are 1s, the result is a 1. If both digits are 0s, or if one digit is a 1 and the other is a 0, the result is a 0. This may sound confusing, but it is actually very simple. Below, some one-digit binary numbers are ANDed:

	0 &	1 &	0 &	1 &
	<u>0</u>	<u>0</u>	<u>1</u>	<u>1</u>
RESULT →	0	0	0	1

For multi-digit binary numbers, each digit is ANDed one at a time:

	00001111 &	11111111 &	11110000 &	01010101 &
	<u>10000000</u>	<u>10000000</u>	<u>10000000</u>	<u>10000000</u>
RESULT →	00000000	10000000	10000000	00000000

Note that, if the left-most digit in the top number is a 0, the result of ANDing it with 10000000 is 0. If the left-most digit in the top number is a 1, the result is not a 0. Using this information, you can determine the value of any digit in a binary number.

## Challenge 2: Send Infrared Signals Passing a Value to the Pulse Function

Rather than sending all 1s or all 0s, you can pass a specific binary number to the pulse function.

1. Decide which board will be the RECEIVER and which will be the SENDER.

2. On the computer hooked up to the SENDER, load your 'send\_binary.c' file and rename it 'pass\_value.c'.

3. Modify your pulse function, as follows:

```
void pulse(int data_value)
{
    int i=0;
    RD1=1;
    delay_us(2500);
    RD1=0;
    delay_us(500);

    for(i=0; i<8; i++)
    {
        RD1=1;
        if((data_value & 0b10000000) !=0)
        {
            delay_us(1300);
        }
        else
        {
            delay_us(700);
        }
        RD1=0;
        delay_us(500);
        data_value = data_value << 1;
    }
}
```

4. In the section of the code file that controls the E (send) key, make the following additions:

```
else if(keynumber==15)
{
    lcd_instruction(GOTO_LINE2+0);
    lcd_text(message);
    pulse(0b11110001);
    delay_ms(400);
}
```

5. Compile and test your new code. Remember to orient the two XBoards as shown in Figure 6. Use the reset button on the RECEIVER a few times and make sure the RECEIVER is consistently receiving the right binary number from the SENDER.

6. Reprogram the boards so that the RECEIVER is the SENDER, and vice versa, and make sure both boards are working properly.

### Challenge 3: Send Text



Now that you can send binary numbers from one XBoard to another, you are ready to start sending text. Remember that every text character has an associated ANSI code. If you send the ANSI code for the character, your RECEIVER board can easily display the corresponding character.

### Challenge 3: Send Text Sending Characters

To send text characters from one XBoard to another, all you need to do is program the SENDER board to transmit the ANSI character code associated with that character. You can display text character on the RECEIVER board's LCD, using an *lcd\_character* statement.

1. Decide which board will be the RECEIVER and which will be the SENDER.

2. On the computer hooked up to the RECEIVER, load the *receive\_binary.c* file that you created during the previous challenge. Rename your file *receive\_character.c*.

3. Modify your *receive\_character.c* code file as follows:

```
#include "mxapi.h"
#include "mxread_ir.c"
void main(void)
{
    int ir_code=0;
    lcd_init();
    ir_code=read_ir();
    lcd_character(ir_code);
    end();
}
```

4. Compile your code and program the RECEIVER board.

5. On the computer hooked up to the SENDER, load your *pass\_value.c* file. Rename this file *send\_character.c*.

**6. In the section of the code file that controls the D (send) key, make the following modifications:**

```
else if (keynumber==15)
{
    lcd_instruction(GOTO_LINE2+0);
    lcd_text(message);
    pulse(65);
    delay_ms(400);
}
```

**7. Compile and test your new code. Use the reset button on the RECEIVER a few times and make sure the RECEIVER is consistently receiving the right character.**

**8. Reprogram the boards so that the RECEIVER is the SENDER, and vice versa, and make sure both boards are working properly.**

### **Challenge 3: Send Text Sending Text Messages**

In order to send a full message, you need to pass the values of your message array to the pulse function one at a time. These will then be received and displayed by the RECEIVER board.

**1. Decide which board will be the RECEIVER and which will be the SENDER.**

**2. On the computer hooked up to the RECEIVER, load the receive\_character.c file. Rename your file receive\_message.c.**

**3. Modify the main function in receive\_message.c, as follows:**

```
void main(void)
{
    int ir_code=0;
    tmr0_init(DIV_64);
    lcd_init();
    while(1==1)
    {
        ir_code=read_ir();
        lcd_character(ir_code);
    }
    end();
}
```

4. On the computer hooked up to the SENDER, load the send\_character.c file. Rename your file send\_message.c.

5. Modify your code, as shown below:

```
#include "mxapi.h"
#include "mxkeyscan.c"
char message[16]={32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32};
void pulse(int data_value)
{
    int i=0;
    RD1=1;
    delay_us(2500);
    RD1=0;
    delay_us(500);

    for(i=0; i<8; i++)
    {
        RD1=1;
        if((data_value & 0b10000000) !=0)
        {
            delay_us(1300);
        }
        else
        {
            delay_us(700);
        }
        RD1=0;
        delay_us(500);
        data_value = data_value << 1;
    }
}
void main(void)
{
    int keynumber;
    int ansi;
    int rotate=0;
    int position=0;
    int m=0;
    lcd_init();
    TRISD1=0;
    RD1=0;
    pwm_init(38000,50,50);
    while(1==1)
    {
        keynumber=keyscan();
        if(keynumber>0&&keynumber<13)
        {
            ansi=(keynumber*4)+61+rotate;
```

```

        lcd_instruction(GOTO_LINE1+position);
        lcd_character(ansi);
        delay_ms(400);
        rotate=rotate+1;
        if(rotate==4)
        {
            rotate=0;
        }
    }
    else if(keynumber==13)
    {
        message[position]=ansi;
        position=position+1;
        rotate=0;
        delay_ms(400);
    }
    else if(keynumber==14)
    {
        lcd_instruction(GOTO_LINE1+position);
        lcd_character(32);
        message[position]=32;
        position=position-1;
        delay_ms(400);
    }
    else if(keynumber==15)
    {
        lcd_instruction(GOTO_LINE2+0);
        lcd_text(message);
        for(m=0; m<16; m++)
        {
            pulse(message[m]);
        }
        delay_ms(400);
    }
}
}
}

```

**6. Compile and test your new code. Remember to position the two XBoards as shown in Figure 6. You should be able to key in and send any message you want (16 characters or less). You can use the reset button on the RECEIVER to receive new messages.**

**8. Reprogram the boards so that the RECEIVER is the SENDER, and vice versa, and make sure both boards are working properly.**

## Building and Controlling the XBot

Mobile robots are becoming increasingly common. In factories and hospitals, mobile robots are often used to make deliveries or perform dangerous jobs, and mobile robots are even being used in the home. Figure 1 shows a basic robot with an aluminum chassis and two servo motors.



**Figure 1. Mobile robot.**

In this project, you will create your own mobile robot, which we refer to as the XBot. Mobile Robot Unit 1 has three challenges. Challenge 1 is the hardware portion of the project--building the XBot. In Challenge 2, you will program the XBot to execute some basic maneuvers. In Challenge 3, you will write a function to make robot navigation much easier.

### Challenge 1: Build the XBot



The Mobile Robot project begins with a hardware challenge--mounting the Arduino and breadboard on an aluminum chassis, adding motors and wheels, and making electrical connections. By the end of this challenge, you will have your own XBot, ready for programming!



## Challenge 1: Build the XBot Collecting Your Components

In order to build the XBot, you will need the components listed below and shown in Figure 2. NOTE: The components contained in some older kits may differ slightly.

Part	Quantity	Description
A	1	Anodized aluminum chassis
B	2	Servo motors
C	2	Plastic disc wheels (with O-ring tires)
D	4	Aluminum standoffs (1.25" 4-40)
E	1	Cotter pin (3.5")
F	1	Plastic ball (1.5" diameter)
G	1	Press-on velcro strip (6") or rubberband
H	14	Machine screws, round head slotted (3/8" 4-40)
I	14	Machine screw nuts (4-40)
J	6	Jump wires (2 long red, 2 short yellow, 2 short orange)
K	1	Bent six-prong connector



Figure 2. XBot components.

## Challenge 1: Build the XBot Installing the Servo Motors

The XBot is propelled by two servo motors--each secured with four machine screws.

1. Insert the servos into the chassis, as shown in Figure 3. **NOTE: Be sure to orient the *servo motors* exactly as shown, with the wires protruding near the two round holes in the chassis.**
2. Secure each servo with four machine screws, attaching the nuts on the inside of the chassis.

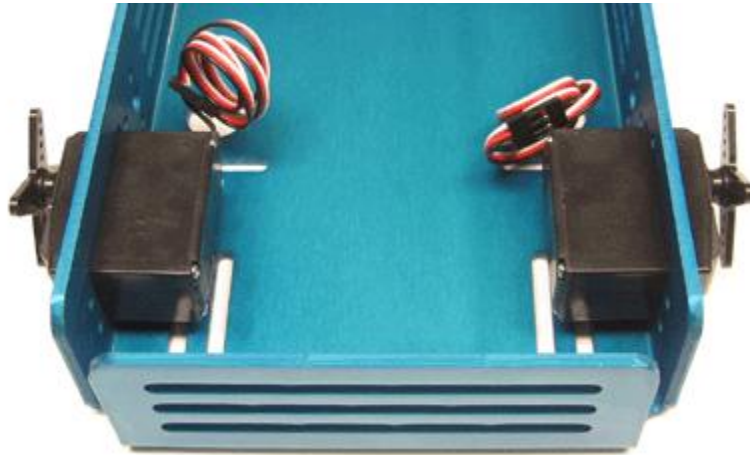


Figure 3. Servo motors correctly positioned in chassis.

### **Challenge 1: Build the XBot Attaching the Wheels**

1. If your servo motors have a four-pointed plastic star on each axle, remove this piece and save the black metal screw, as shown in Figure 4.
2. Fit a rubber O-ring tire over each wheel. **NOTE: Some wheels may already have tires in place.**
3. Push the two wheels firmly onto the servo motor axles.
4. Secure the wheels using the black metal screws from each axle, as shown in Figure 4.



**Figure 4. Removing the plastic star (left) and attaching the wheel (right).**

### **Challenge 1: Build the XBot Adding the Front Caster**

The front of the robot is supported by a *caster*--a plastic ball attached to the underside of the robot chassis by two aluminum rods, called *standoffs*. The plastic ball is suspended between the standoffs by a cotter pin, which serves as an axle. The ball rolls when the robot is traveling forward or reverse and slides when the robot turns. There are two different ways to mount the caster, depending on the contents of your kit.

- 1. If your kit has 1.5" 6-32 standoffs with holes drilled through the ends, insert the cotter pin through standoffs and the plastic ball, as shown in Figure 5 (left).**
- 2. If your kit has only 1.25" 4-40 standoffs, mount the cotter pin and the ball to the standoffs with two 4-40 machine screw nuts, as shown in Figure 5 (right).**



Figure 5. Two ways to mount the cotter pin and plastic ball.

3. Insert the standoffs into the two round holes at the front of the robot, as shown in Figure 6, and secure each end. If you have 1.5" 6-32 standoffs, use two 6-32 machine screw nuts to secure the standoffs. If you have 1.25" 4-40 standoffs, use two 3/8" 4-40 machine screws to secure the standoffs. *NOTE: In either case, be careful not to overtighten the standoffs .*



Figure 6. Caster secured to robot chassis.

## Challenge 1: Build the XBot Mounting the Battery Pack

The XBot is powered by a **battery pack**. Click [here](#) if you need help. The battery pack mounts on the underside of the robot chassis, and is held in place by two velcro strips. In order to connect to the breadboard, the battery pack must be oriented as shown in Figure 8. If Velcro is not available, create a mount of your own design.

1. **Disconnect the battery pack from the breadboard, if it is connected.**
2. **Using scissors, cut two 1.5-inch strips of double-sided velcro tape.**
3. **Keeping the two sides of the velcro together, peel the backing from one side of each strip and press the two strips onto the battery pack, as shown in Figure**

7.

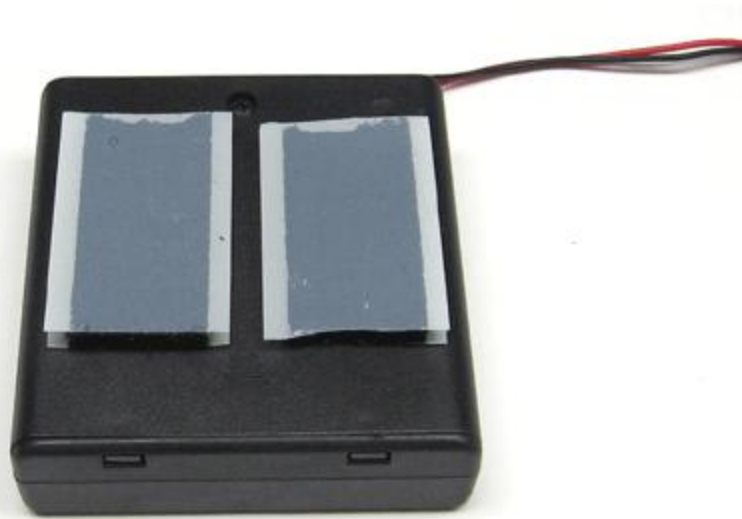


Figure 7. Velcro strips attached to the battery pack.

4. Peel the remaining backing from the velcro strips, and press the battery pack onto the underside of the robot chassis, as shown in Figure 8. Be careful to orient the battery pack so that the switch and wire are positioned exactly as shown.

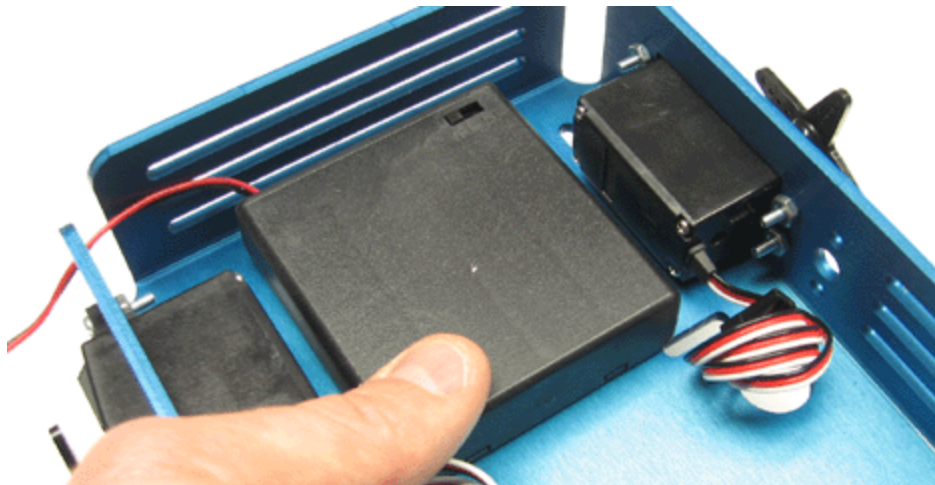
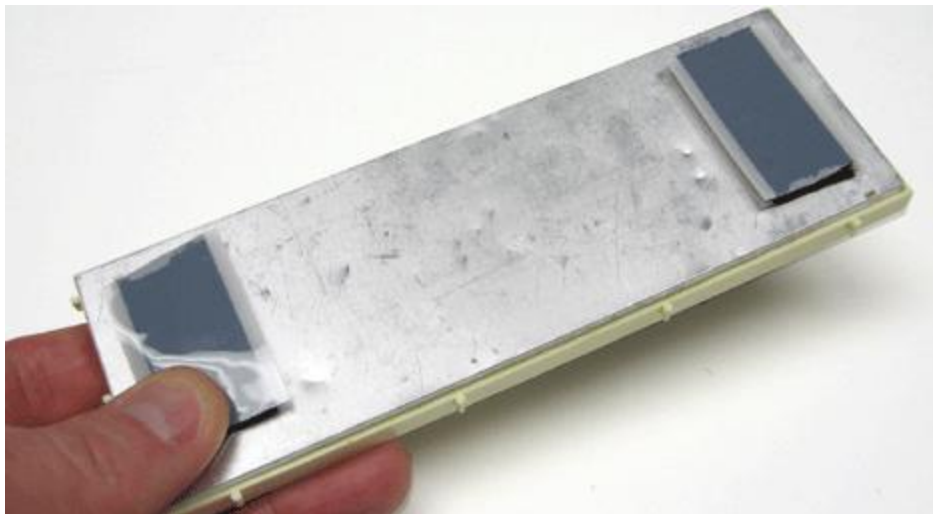


Figure 8. Mounting the battery pack.

### **Challenge 1: Build the XBot Securing the Atmega and Breadboard**

The Atmega and breadboard are secured to the top of the robot chassis with Velcro strips. If Velcro is unavailable, use a thick rubber band. As with the battery pack, the orientation of the Atmega and breadboard are critical to ensure that all of the necessary electrical connections can be made in the next step.

- 1. Using scissors, cut two 1.5-inch strips of double-sided Velcro tape.**
- 2. Keeping the two sides of the Velcro together, peel the backing from one side of each strip and press the two strips onto the underside of the breadboard, as shown in Figure 9.**



**Figure 9. Velcro strips attached to the underside of the XBoard.**

- 3. Peel the backing from the velcro, and press the breadboard onto the robot chassis, as shown in Figure 10. Align the board's front edge with the holes where the standoffs protrude through the chassis.**



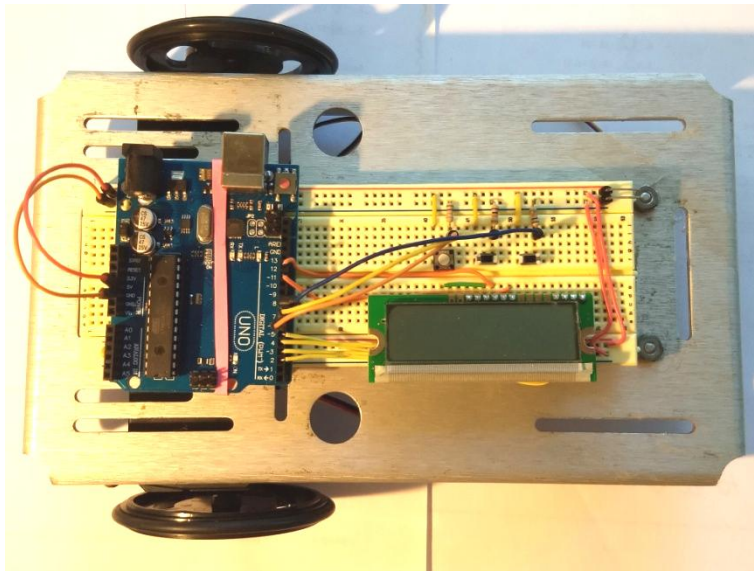


Figure 10. Positioning the breadboard.

## Challenge 1: Build the XBot Making Electrical Connections

In order to finish the XBot, a few additional jump wires and a bent six-prong connector must be added to the breadboard. In addition, the servo motor leads must be plugged into the breadboard, and the battery pack must be reconnected.

1. Remove the LCD and its connecting wires from the breadboard. Remove all switches and LED circuits, so that just the Arduino, its power and ground connections, and the capacitor remain.
2. Insert two short yellow jump wires, two short orange jump wires, and two long red jump wires into the XBoard, as shown in Figure 11. The short yellow jump wires should connect holes J31 and J34 to power. The short orange jump wires should connect holes J30 and J33 to ground. The long flexible jump wires should connect port 9 to hole J32, and port 10 to J35.

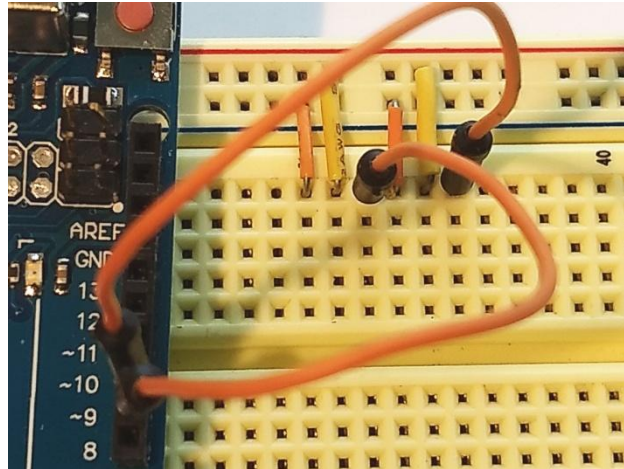


Figure 11. Yellow and orange jump wires.

3. Insert a bent six-prong connector into the XBoard, as shown in Figure 12.

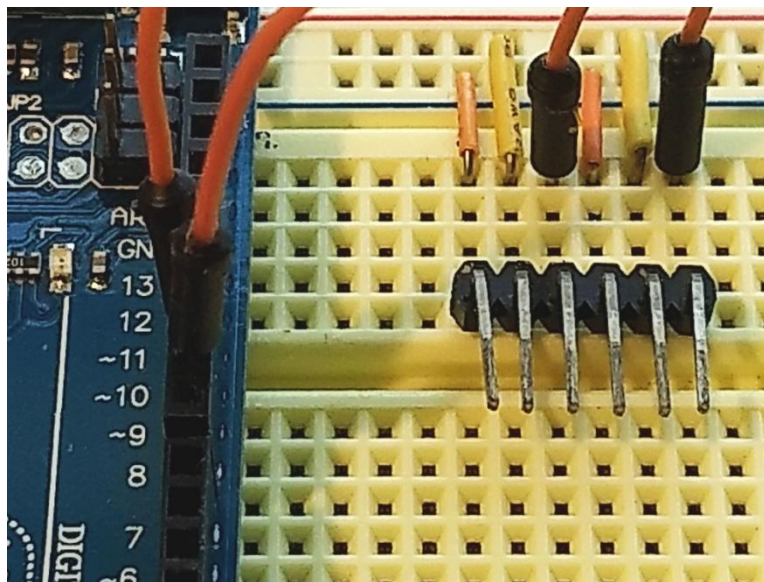


Figure 12. Adding six-prong bent connector.

4. Thread the wire harnesses from the servo motors through the round hole in the robot chassis.

5. Plug the wire harnesses from the servo motors into the bent six-prong connector, as shown in Figure 13. **NOTE: Be sure to align the white, red, and black wires from left to right, as shown. (Black to ground, red to power white to the Arduino.)** Also, be sure to plug the wires from the left motor onto the left side of the connector, and the wires from the right motor onto the right side of the connector.



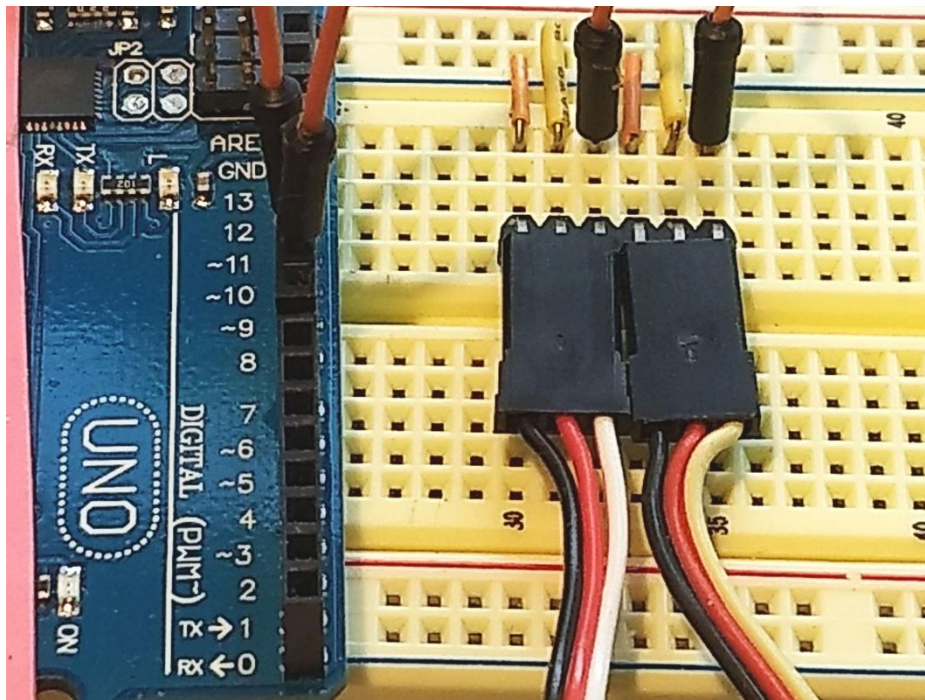


Figure 13. Connecting servo motor wire harnesses.

## Challenge 1: Build the XBot

### Connecting a Battery Power Pack

1. Make sure the switch on your battery pack is in the OFF position.
2. Move the capacitor from the rail next to J3, and move it so it connects the blue and red rail next to J61, as shown in Figure 14.



Figure 14. The 1uf capacitor attached to the rail near J61.

3. Disconnect the two-prong battery connector that you used in the 7 segment LED project, and move it to the rail next to A3. Face it away from the Arduino. As shown in Figure 15.

4. Reconnect the wire harness from the battery pack to the breadboard, as shown in Figure 15. NOTE: The two-prong connector can be positioned anywhere on the XBoard, provided that the black lead aligns with ground and the red lead aligns with power.

**Be sure the red wire from the battery pack connects with the red rail on the breadboard, and the black wire from the battery pack connects with the blue rail on the breadboard. Getting these wires backwards could destroy your Arduino.**

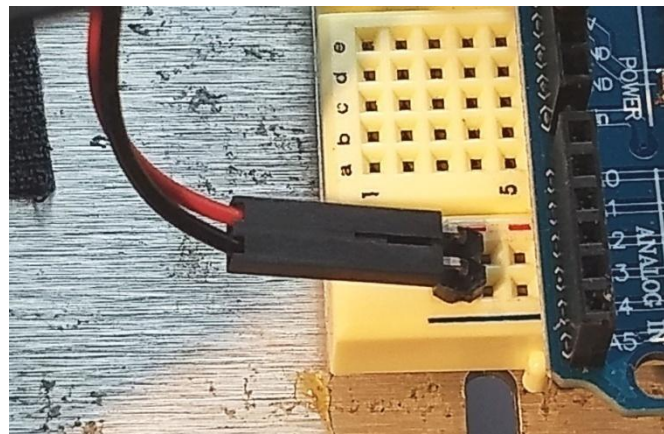


Figure 15. Connecting a battery pack to the XBoard.

## Protecting Your Arduino with a Schottkey Diode

Both your USB port and Arduino are designed to provide 100 milliamps of current. This amount of electricity is easily enough to run LEDs and sensors, but is not enough to run all but the smallest motors. To fix this problem, and to allow your robot to run without a USB wire attached, you are going to attach a battery power supply.

To prevent motors from drawing too much current from your Arduino, when you program it, you are going to attach an electronic component called a **diode**, that only lets electricity go one way. Diodes have a white band on the side that will provide power (+5v). Schottkey diodes are particularly fast and only lower the voltage by a very small amount.



5. Move the jumper cable that connects the 5v power port from the red rail and insert it into hole H4. Connect a Schottkey diode from the red power rail to hole I4 facing the positive end toward hole I4, as shown in Figure 16.

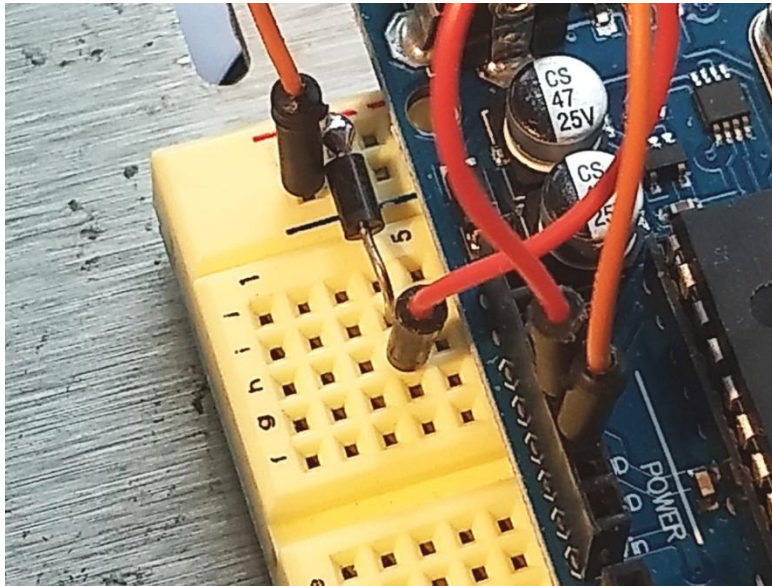


Figure 16. Connecting power through a Schottkey diode.

Reconnect the wire harness from the battery pack to the breadboard, as shown in Figure 15. **NOTE:** The two-prong connector can be positioned anywhere on the XBoard, provided that the black lead aligns with ground and the red lead aligns with power.

## Challenge 1: Build the XBot Final Hardware Check



Before moving on to the next challenge, take a few minutes to check your hardware set-up. If possible, ask a teacher or mentor to examine your robot. The following checklist will help you ensure that your robot is ready for programming.

- **Servo motors:** Both servo motors should be mounted to the chassis with four machine screws. The wires from each servo should pass up through the round holes in the chassis.
- **Wheels:** A plastic wheel with an O-ring tire should be mounted on each servo motor axle, and held in place with a small black metal screw.
- **Front caster:** The front caster should be held securely in place with two aluminum standoffs. The caster should turn freely on the cotter pin.

- **Battery pack:** The battery pack should be positioned on the underside of the robot chassis and held in place with two velcro strips, or attached to the back of the Arduino securely by your invention.
- **Breadboard:** The breadboard should be secured with Velcro or a rubber band to the top of the robot chassis, with the LCD close to the left servo motor.
- **Jump wires:** All jump wires should be positioned as shown in Figure 11.
- **Motor connections:** Looking at the robot with the microcontroller on the left, the wire harness from the left servo motor should be on the left, and the wire harness from the right servo motor should be on the right. In each harness, the colored wires should be ordered white, red, black, from left to right.
- **Battery leads:** As always, the red lead from the battery pack should be attached to H4, and the black lead should align with a blue line hole.
- **Diode:** A schottkey diode should connect I4 to the power rail with the red line.

## Challenge 2: Make the XBot Move



The defining feature of a mobile robot is the ability to move, and mobile robots are very good at moving in controlled ways. In Challenge 2, you will learn how to control the XBot's motors, execute forward and reverse movements, make right and left turns, and navigate fixed patterns.

### Challenge 2: Make the XBot Move About Servo Motors

The XBot is propelled by two hobby servo motors. **Servo motors**<sup>4</sup> are commonly used in the steering systems of radio-controlled vehicles, such as boats and planes. Servo motors are usually designed to rotate in precise increments between 0 and 180 degrees--a feature that makes them useful for controlling the angle of rudders in radio-controlled boats

---

<sup>4</sup> To fully understand how the servo works, you need to take a look under the hood. Inside there is a pretty simple set-up: a small [DC motor](#), [potentiometer](#), and a control circuit. The motor is attached by gears to the control wheel. As the motor rotates, the potentiometer's resistance changes, so the control circuit can precisely regulate how much movement there is and in which direction.

When the shaft of the motor is at the desired position, [power](#) supplied to the motor is stopped. If not, the motor is turned in the appropriate direction. The desired position is sent via electrical pulses through the [signal wire](#). The motor's speed is proportional to the difference between its actual position and desired position. So if the motor is near the desired position, it will turn slowly, otherwise it will turn fast. This is called **proportional control**. This means the motor will only run as hard as necessary to accomplish the task at hand, a very efficient little guy. <https://www.jameco.com/jameco/workshop/howitworks/how-servo-motors-work.html>



and ailerons on airplane wings. The servo motors on your robot have been modified to turn a full 360 degrees, so that they can drive your robot's wheels. Figure 17 shows a typical hobby servo motor.



Figure 17. Hobby servo motor.

## Challenge 2: Make the XBot Move Controlling Servo Motors

Unlike conventional electric motors, which run continuously when connected to a power source, servo motors are controlled with very short electrical pulses--each lasting **between 1,000 and 2,000 microseconds**. Each pulse causes the servo motor to rotate a few degrees. Servo motors can be made to rotate continuously by sending them a rapid series of pulses, each separated by a pause of 20,000 microseconds (20 milliseconds). Figure 18 illustrates one pulse series that could be used to drive a servo motor.

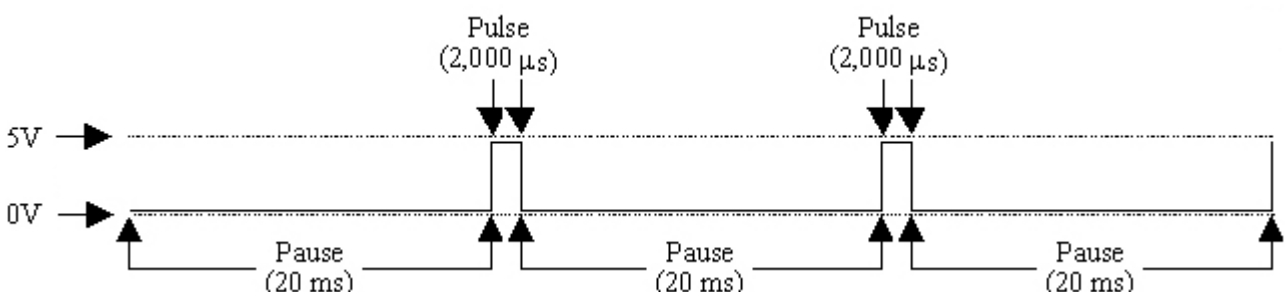


Figure 18. Servo motor pulse series.

In the pulse series shown in Figure 18, the servo motor rotates a few degrees with every pulse. Although there are pauses between the pulses, the pauses are short (only 20 milliseconds), so the rotation of the servo is essentially continuous.

## **Challenge 2: Make the XBot Move About Speed and Direction**

The length of the pulse determines the speed and direction of the servo motor's rotation. Pulses of 2,000 microseconds cause the motor to turn full speed in one direction; pulses of 1,000 microseconds cause the motor to turn full speed in the opposite direction. No matter what the pulse length, the pause between the pulses is always 20 milliseconds. Figure 19 illustrates the relationship between pulse length and rotation.

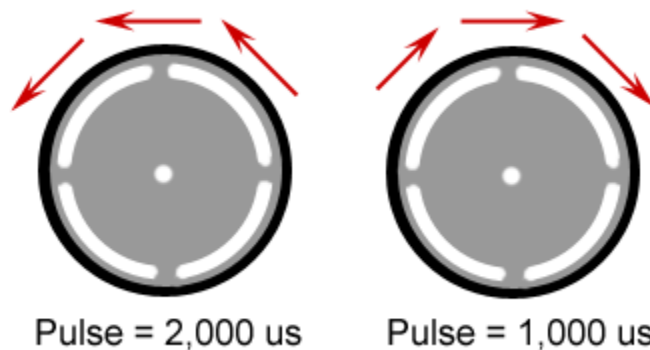


Figure 19. Relationship between pulse duration and wheel rotation.

## **Challenge 2: Make the XBot Move Sending a Single Pulse**

Your first programming task is to send a single pulse to the robot's left servo motor. Recall that the left servo motor is connected to Port 9 on the microcontroller. In order to send a pulse to the left servo motor, your code should set up Port 9 as an output, using a `pinMode()` statement. Then, your code should raise the voltage at Port 9 to 5 volts (i.e. `digitalWrite(9,1)`) delay for 2,000 microseconds using a `delayMicroseconds()` command, and then lower the voltage (i.e. `digitalWrite(9,0)`)

1. Open the Programming Portal and start a new code file. Save it as `one_pulse`.
2. Enter the following code into the Editor window:

```
// Program 11.1Pulse
```

```
// servo is connected to port 9

void setup() {
  pinMode(9, OUTPUT); //set up port 0 to output 5 volts
  digitalWrite(9,1); // instruct servo to turn
  delayMicroseconds(2000);
  digitalWrite(9,0);
}

void loop() { }
```

3. Program the microcontroller, and watch the left servo. It should turn a few degrees. *NOTE: A few degrees is a very small turn!*

4. Press the reset button on the USB programming board a few times. The motor will turn each time.

## Challenge 2: Make the XBot Move Sending a Series of Pulses

The next step is to send a series of pulses, causing the left servo motor to rotate continuously. This is accomplished by creating a *while...* loop that sends a pulse to the motor each time through. As you learned in Getting Started, *while...* loops execute a piece of code while a certain condition is true. To create an infinite loop, you need to tie the *while...* loop to a condition that is always true--for example, `while(1==1)`.

1. Using the Save As command, rename your code file `pulse_series`. If you don't have a code file open, just enter the code shown below and save it as `pulse_series`.

2. Modify the main function code in your code file, as follows. *NOTE: You still need your include statements at the top of your code file.*

```
// Program 11.2Pulse
// servo is connected to port 9

void setup() {
  pinMode(9, OUTPUT); //set up port 0 to output 5 volts
  while (1==1){
    digitalWrite(9,1); // instruct servo to turn
    delayMicroseconds(2000);
    digitalWrite(9,0);
```



```
delay(20);  
} // while  
} // setup  
  
void loop() { }
```

3. Program the microcontroller with this new code. The motor should now run continuously.

4. Turn the battery pack OFF before moving to the next step.

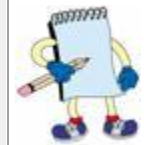
### Challenge 2: Make the XBot Move Changing the Direction of Rotation

By adjusting the duration of the pulse, you can change the direction of the servo motor's rotation. The duration of the pulse is determined by the value inside the *delay\_us* statement in your code. Remember: all pulses must be greater than or equal to 1,000 microseconds and less than or equal to 2,000 microseconds. Never send a pulse that is shorter than 1,000 microseconds or longer than 2,000 microseconds.

1. Change the duration of the pulse to 1,000 microseconds and reprogram the microcontroller.

2. Change the pulse back to 2,000 microseconds and reprogram the microcontroller.

#### PROGRAMMING NOTE



Describing the direction of a wheel's rotation is tricky, because it depends on how you are looking at the robot. In the Mobile Robot project guides, *forward* means *turning in the direction that makes the robot travel forward* and *reverse* means *turning in the direction that makes the robot travel in reverse*. With the robot upright, forward is *clockwise* for the right servo and *counterclockwise* for the left servo.

### Challenge 2: Make the XBot Move Printing a Robot Data Sheet

In the remainder of this unit, you will be asked to record important data about the performance of your robot. By recording this information on paper, you will be creating a data sheet for your robot, which will be a useful reference for future activities involving the robot.

1. Click [here](#) to open a printable version of this page in PDF format. You will need the [Acrobat Reader](#) from Adobe to view the PDF.

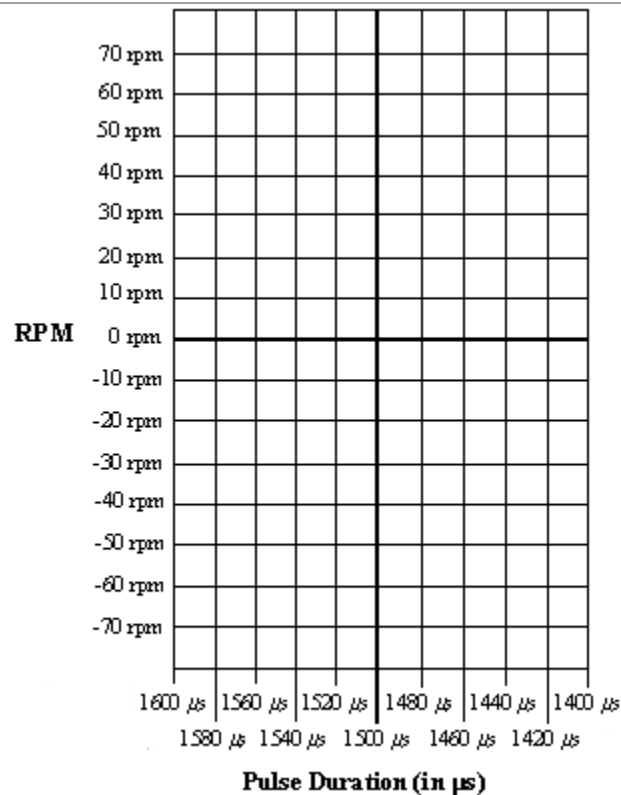
2. Select the Print option to print this page.

CENTER POINT		
Value	Left Motor Pulse Duration	Right Motor Pulse Duration
Longest Pulse with No Rotation		
Shortest Pulse with No Rotation		
True Center Point		

FORWARD AND REVERSE		
Motion	Left Motor Pulse Duration	Right Motor Pulse Duration
Full Speed Forward		
Full Speed Reverse		
Complete Stop		

#### REVOLUTIONS PER MINUTE (RPM)



#### STRAIGHT MOTION

Motion	Left Motor Pulse Duration	Right Motor Pulse Duration
Straight Forward		
Straight Reverse		

LEFT AND RIGHT TURNS		
Motion	Left Motor Pulse Duration	Right Motor Pulse Duration
Left Turn		
Right Turn		

SPECIFIED DISTANCE	
Value	Value
No. of Pulses for Complete Rotation	
Wheel Diameter	
Wheel Circumference	
Distance Per Pulse Ratio	

PRECISE TURNS	
Value	Value
Distance Between Wheels	
Turning Circumference	
25% of Turning Circumference	
No. of Pulses for a 90-Degree Turn	
No. of Pulses for a 180-Degree Turn	

## Challenge 2: Make the XBot Move Finding the Center Point

Every servo motor has a *center point*-the pulse duration at which the motor does not turn at all. This pulse duration is always between 1,000 and 2,000 microseconds, and usually somewhere around 1,500 microseconds. A narrow range of values produce no motion in the motor. The motor's "true" center point is the *average* of the high and low values. The average of any two numbers is equal to their sum, divided by 2, as shown in the following formula.

$$\text{average of value 1 and value 2} = (\text{value 1} + \text{value 2}) / 2$$

Using this formula, you can calculate the true center point for the left servo motor.

1. Start with a pulse of 1,500 microseconds and adjust this value until the motor stops.
2. Find the highest value and the lowest value that produce no motion in the motor. Record these values, because you will need them for the exercises below.
3. Calculate the average of the high and low values. Record your result.

## **Challenge 2: Make the XBot Move Controlling Both Motors**

You can run both of your robot's motors at the same time by adding code to control the right motor. Remember that the right motor is connected to Port B0.

1. Using the **Save As** command, rename your code file `both_motors`. If you don't have a code file open, just enter the code shown below and save it as `both_motors`.
2. Modify your main function to get the right motor running. *NOTE: Before downloading this code file, you may want to elevate your robot chassis slightly, so that the wheels can turn freely without moving the robot.*

```
// Program 11.3 Both
// Left servo port 9, Right servo port 10

void setup() {
  pinMode(9, OUTPUT); //set up port 0 to output 5 volts
  while (1==1){

    digitalWrite(9,1); // Program Left Motor
    delayMicroseconds(2000);
    digitalWrite(9,0);

    digitalWrite(10,1); // Program Right Motor
    delayMicroseconds(2000);
    digitalWrite(10,0);

    delay(20); //Run
  } // while
} // setup

void loop() { }
```

3. Determine the true center point for the right motor, using the same procedure you used in the previous section. Record all your values on the Robot Data Sheet.

## **Challenge 2: Make the XBot Move Making Forward and Reverse Movements**

With both motors turning, you can program the robot to travel forward and in reverse. Remember that the two motors are facing in opposite directions.

1. Adjust the pulse durations on each motor to make the robot travel forward. Record these values in the Robot Data Sheet.
2. Adjust the pulse durations on each motor to make the robot travel in reverse. Record these values in the Robot Data Sheet.
3. Adjust the pulse durations for each motor to make the robot stop. Record these values in Robot Data Sheet.

## **Challenge 2: Make the XBot Move Calculating Motor Speed**

The speed of any motor--whether a car engine or a hobby servo motor--can be measured in revolutions per minute, or RPMs. RPMs indicate how many times the motor makes a complete rotation in one minute. A servo motor's RPMs depend on the pulse lengths in your code. The motors have high RPMs at pulse lengths close to 1,000 and 2,000 microseconds, and low RPMs at pulse lengths near the center point.

1. Using tape, make a small mark on the outer edge of one of your robot's wheels. This will make it easier to monitor the wheel's position.
2. Program the motor to turn full speed in one direction (with a pulse duration of either 2,000 microseconds or 1,000 microseconds).
3. Count how many times the wheel rotates in one minute. This is the wheel's RPM at full speed. *NOTE: It may be easier to work with a partner on this step, since it can be tricky to watch the clock and count rotations at the same time.*

## Challenge 2: Make the XBot Move Adjusting Motors for Straight Motion

Chances are, your robot is not traveling in a perfectly straight line in either the forward or the reverse direction. Using what you know about pulse duration and RPM, you can straighten the motion of your robot.

1. With your robot programmed to travel full speed forward, detach the programming cable and place the robot on the floor. *NOTE: For the most accurate results, find a level floor with a hard surface, rather than rug or carpet.*
2. Determine whether your robot is veering slightly to the right or slightly to the left.
3. If your robot is veering to the left, reduce the speed of the right servo motor. If your robot is veering to the right, reduce the speed of the left servo motor. *REMEMBER: Most variation in motor speed occurs near the center point.*
4. Once you have achieved reasonably straight motion in the forward direction, reprogram your robot to travel in reverse, and adjust the pulses to produce straight motion in the reverse direction. Record all values in the Robot Data Sheet.

**Field Goal**  
Show your work  
to the instructor  
for a grade.

## Challenge 2: Make the XBot Move Making Left and Right Turns

By turning the wheels in opposite directions, you can make your robot execute turns. Although it is also possible to make turns by stopping one wheel and rotating the other, you should make turns using both wheels. This way, your robot turns in place, which will allow more precise navigation later on.

1. Change your code so that your robot turns to the right. This will require driving the left wheel forward and the right wheel in reverse. Record these values.
2. Change your code so that your robot turns to the left. This will require driving the right wheel forward and the left wheel in reverse. Record these values.

## Challenge 2: Make the XBot Move Executing Finite Movements and Turns

By now, you can make your robot travel forward and backward in a straight line, turn right, and turn left, but these motions are all produced by infinite loops. For precise navigation, you will need to execute finite movements and turns. This will allow you to make several different movements within a single code file. For finite movements, you will introduce a finite *for...* loop into your code. As you learned in Getting Started, a *for...* loop repeats a section of code a specified number of times, represented by a variable.

1. Using the **Save As** command, rename your code file **fixed\_motion.c**.
2. Enter the following code into the Editor window:

```
// Program 11.4 Fixed Motion
// Left servo port 9, Right servo port 10
int i;

void setup() {
  pinMode(9, OUTPUT); //set up port 0 to output 5 volts
  for(i=0;i<200;i++){

    digitalWrite(9,1); // Program Left Motor
    delayMicroseconds(2000);
    digitalWrite(9,0);

    digitalWrite(10,1); // Program Right Motor
    delayMicroseconds(2000);
    digitalWrite(10,0);

    delay(20); //Run
  } // for
} // setup

void loop() { }
```

3. Reprogram the microcontroller and observe the robot's movement. The robot should move forward for approximately 5 seconds and then stop.

## Challenge 2: Make the XBot Move Controlling the Duration of Movements and Turns



Controlling the duration of movements is critical to making the robot do what you want. For example, you could have the robot move forward for five seconds, turn exactly 90 degrees to the left, and then continue on. The duration of the robot's motion is determined by two factors: 1) the total duration of the pulses and pauses in each loop, and 2) and the total number of times that the loop repeats. For example, in **fixed\_motion.c**, there are two pulses--a 2,000-microsecond (2-ms) pulse to the left servo and a 1,000-microsecond (1-ms) pulse to the right servo--and a 20-ms pause in each loop. This loop repeats 200 times. The total duration of the motion can be calculated as follows:

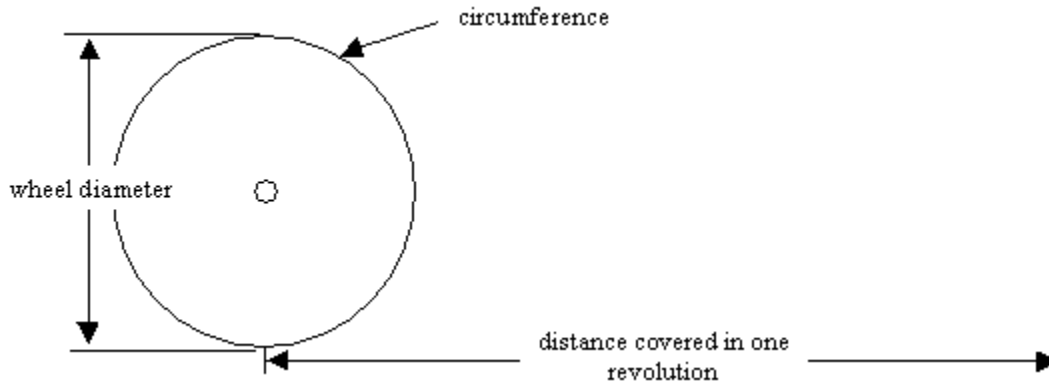
$$\begin{aligned}\text{duration of motion} &= (\text{total duration of pulses and pauses}) * (\text{number of times loop repeats}) \\ &= (2 \text{ ms} + 1 \text{ ms} + 20 \text{ ms}) * 200 \\ &= (23 \text{ ms}) * 200 \\ &= 4600 \text{ ms or } 4.6 \text{ seconds}\end{aligned}$$

By adjusting the loop variable ("i"), you can change the duration of the robot's motion.

1. Change the loop variable to produce a forward motion lasting 10 seconds.
2. Reprogram the robot to turn in place for 5 seconds.

## **Challenge 2: Make the XBot Move Traveling a Specific Distance**

In this step, you will program your robot to travel a specific distance: 24 inches (61centimeters). Using some simple geometry, you can calculate exactly how many pulses are needed to cover this distance. Each time the robot's wheels make one complete turn, the robot travels a distance equal to the circumference of the wheel, as shown in Figure 20. To make the robot travel a specific distance, you must first determine how many pulses are needed for the wheel to rotate a full 360 degrees, calculate the wheel's circumference, and then determine how far the robot travels with every pulse that you send to the servo motors. This last figure, your robot's distance-per-pulse ratio, will enable you to precisely control forward and reverse movements.



**Figure 20. Wheel diameter, circumference, and distance traveled.**

1. Put a small piece of tape on the outer edge of one of your robot's wheels. This will make it easier to monitor the wheel's position.
2. Using trial and error, program the microcontroller as many times as necessary to determine the number of pulses needed for a complete 360-degree rotation of the wheel. Record your final result.
3. With a ruler, measure the diameter of one of your robot's wheels. You can make this measurement in inches or centimeters, but be sure to record your units, along with your result.
4. Using the following formula, calculate the wheel's circumference. Remember that pi is equal to approximately 3.14. Record your result.

$$\text{wheel circumference} = \text{wheel diameter} * \pi$$

5. Now that you know the wheel's circumference and the number of pulses needed for a complete rotation of the wheel, you can calculate the distance that your robot travels with every pulse, or its distance-per-pulse ratio. Using the following formula, calculate the distance per-pulse-ratio, and record your result.

$$\text{distance-per-pulse ratio} = \text{wheel circumference} / \text{pulses needed for a 360-degree wheel rotation}$$

6. Finally, using the following formula, determine the number of pulses needed for your robot to cover 24 inches (61 centimeters).

$$\text{number of pulses} = \text{distance} / \text{distance-per-pulse ratio}$$

7. Using the result of your calculation, program the robot to travel exactly 24 inches (61 centimeters).

## Challenge 2: Make the XBot Move Making a 90-Degree Turn

Your next task is to program the robot to make a 90-degree right turn. As with any right turn, the left wheel will be moving forward, and the right wheel will be moving in reverse. The number of pulses needed for a 90-degree turn can be calculated using the distance-per-pulse ratio. In order to make this calculation, you must first calculate your robot's *turning circumference*, which is the distance that the wheels travel over the floor when the robot makes a complete 360-degree turn. The turning circumference is determined by the distance between the wheels, which is the diameter of the turning circumference. In a 90-degree turn, each wheel covers one-quarter of the robot's turning circumference, as shown in Figure 21.

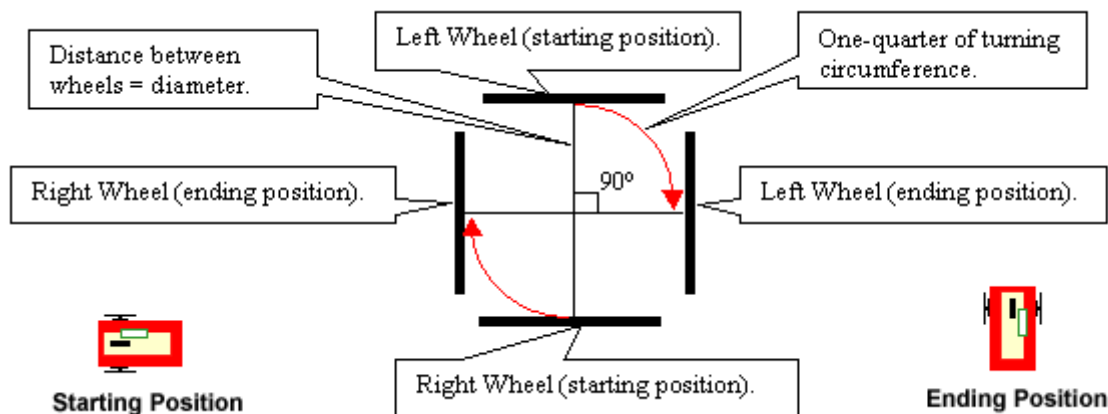


Figure 21. Dimensions of the 90-degree turn.

1. With a ruler, measure the distance between your robot's wheels. This is the diameter for the turning circumference. You can use either inches or centimeters, but be sure to record your units, along with the measurement.

2. Using the following formula, calculate your robot's turning circumference. Remember that pi is equal to approximately 3.14. Record your result.

$$\text{turning circumference} = \text{distance between wheels} * \pi$$

3. Using the following formula, calculate the number of pulses needed to make a 90-degree right turn, and record your result.

number of pulses =  $(0.25 * \text{turning circumference}) / \text{distance-per-pulse ratio}$

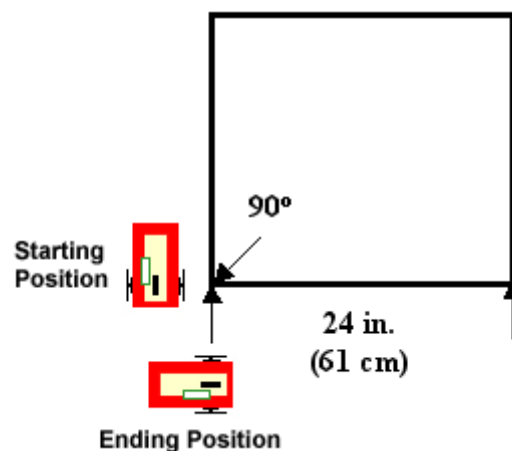
4. Program your robot to make the 90-degree turn.

5. Next try a 180-degree turn.

## Challenge 2: Make the XBot Move Navigating Fixed Patterns

By combining straight line movements with turns, you can program your robot to navigate a fixed pattern on the floor. You will have to create a separate loop for each required motion.

1. Program your robot to trace the following figure on the floor. Try to complete the figure *without using trial and error*.



2. Program your robot to trace the following figure on the floor. Again, try to complete the figure *without using trial and error*.

## Challenge 3: Write a Move Function



By now, you have learned to execute forward and reverse movements, make right and left turns, and navigate simple patterns using sequences of finite loops. This approach requires many lines of code per movement or turn, and it would quickly become cumbersome if you wanted to make the robot navigate a complex pattern involving many turns and movements. Fortunately, there is a simpler way to program the XBot's movements, using functions. In this challenge, you will write a function, called `move`, that will greatly simplify robot navigation.

### Challenge 3: Write a Move Function

#### About the Move Function

In C, functions allow you to execute a section of code repeatedly, without having to repeat the code over and over. Instead, you simply define the function once, and then you can execute the function whenever you need it, with a single line of code. (Executing a function is usually referred to as "calling" the function.) A useful feature of functions is that you can pass one or more variables to the function, to control how the function operates.

Your move function will cause the XBot to execute a turn or a movement every time you call the function. You will pass three variables to the move function--the first variable determines the number of pulses sent to the servo, the second variable determines the pulse duration to the left servo, and the third variable determines the pulse duration to the right servo. So, for example, by typing the following code, you would send a total of 150 pulses to each servo--with a pulse length of 2,000 microseconds to the left servo and a pulse length of 1,000 microseconds to the right servo.

```
move(150, 2000, 1000)
```

This would cause the robot to move at full speed for approximately three complete revolutions of the wheel, covering about 25 inches. With the following code, you would send a total of 20 2,000-microsecond pulses to the left servo and 20 2,000-microsecond pulses to the right servo.

```
move(20, 2000, 2000)
```

This would cause the XBot to rotate in place for a total of 20 pulses--roughly a 90-degree turn.

### Challenge 3: Write a Move Function

#### Setting Up the Move Function

Recall that every C function has three parts:

- \* A *return value*. Since the move function has no return value, you will put "void" here as a placeholder. In later units, you will use functions with return values.
- \* A *function name*, which can be anything you like--in this case, your function will be called "move."
- \* *Arguments*, which are the variables that you pass to the function. The move function will have **three variables--"count", "left\_pulse", and "right\_pulse."**

**1. Using the Save As command, rename your file move\_functions.**

2. Set up the move function at the bottom of your code file, and the program at the top:

```
// Program 11.5 Move Function
// Left servo port 9, Right servo port 10
int i;

void setup() {
  pinMode(9, OUTPUT); //set up port 0 to output 5 volts
  pinMode(10, OUTPUT); //set up port 0 to output 5 volts

  move(100,2000,1000); //Call the move function
  move(200,2000,2000); //Call the move function

  }//setup

void loop() { }

//----- MOVE Function -----
void move(int count, int leftPulse, int rightPulse){
  for(i=0;i<=count;i++){

    digitalWrite(9,1); // Program Left Motor
    delayMicroseconds(leftPulse);
    digitalWrite(9,0);

    digitalWrite(10,1); // Program Right Motor
    delayMicroseconds(rightPulse);
    digitalWrite(10,0);

    delay(20); //Run
  } // for
}
```

Every time you want to make the robot move, you can just call the move function, as follows:

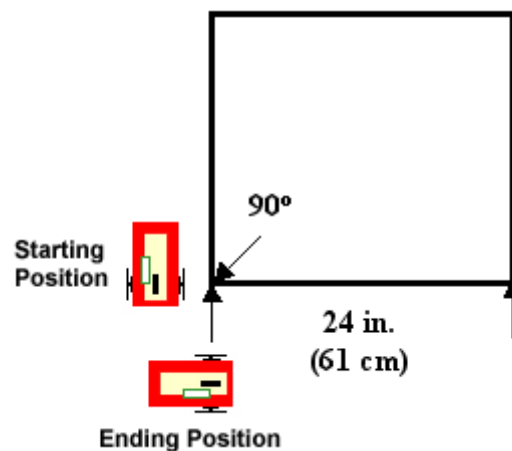
```
move( #Times, Left motor pulse, Right motor pulse);
```

4. Reprogram the microcontroller and observe the XBot's movements.

### Challenge 3: Write a Move Function Using the Move Function

With your move function defined, navigating fixed patterns becomes much easier. In this step, you will program the robot to retrace the figures from Challenge 2. For each movement, you need only one extra line of code!

1. Using your move function, program your robot to trace the following figure on the floor. Try to complete the figure *without using trial and error*.



2. Program your robot to trace the following figure on the floor. Again, try to complete the figure *without using trial and error*.

**Box**  
Show your work  
to the instructor  
for a grade.

### Challenge 3: Write a Functions to Simplify Your Code

Here is an example of a function that drives the robot forward.

```
// Program 11.7 Forward  
  
// Left servo port 9, Right servo port 10  
  
int i;
```



```

void setup() {
  pinMode(9, OUTPUT); //set up port 0 to output 5 volts
  pinMode(10, OUTPUT); //set up port 0 to output 5 volts

  forward(100);

  }//setup

void loop() { }

//----- MOVE Function -----
void move(int count, int leftPulse, int rightPulse){
  for(i=0;i<=count;i++){

    digitalWrite(9,1); // Program Left Motor
    delayMicroseconds(leftPulse);
    digitalWrite(9,0);

    digitalWrite(10,1); // Program Right Motor
    delayMicroseconds(rightPulse);
    digitalWrite(10,0);

    delay(20); //Run
  } // for
} // move

//----- Forward Function -----
void forward(int count){
  int leftPulse = 2000; // change these to straighten robot tracking
  int rightPulse=1000;

  for(i=0;i<=count;i++){

    digitalWrite(9,1); // Program Left Motor
    delayMicroseconds(leftPulse);
    digitalWrite(9,0);

    digitalWrite(10,1); // Program Right Motor
    delayMicroseconds(rightPulse);
    digitalWrite(10,0);

    delay(20); //Run
  }
}

```

```
} // for  
} // forward
```

Write the following functions: reverse, left90 and right90, then program a solution for the Zig Zag puzzle.

**Box**  
Show your work  
to the instructor  
for a grade.

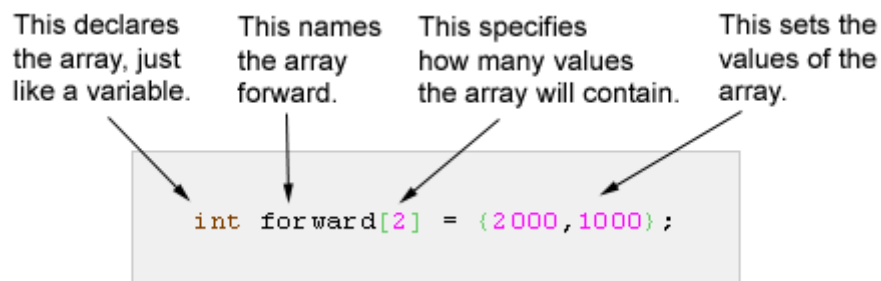
An **array** is an ordered list of two or more values that is assigned a specific name. By adding arrays to your code, you can make the move function even simpler to use. For example, you can create an array to store the pulse lengths required for forward motion and call it "forward." Likewise, you could create an array called "spin\_right" to store the pulse lengths that make the XBot spin right. Then, in order to move the XBot forward for 50 pulses and make it turn right for 25 pulses, you could use the following code:

```
move(50, forward);  
move(25, spin_right);
```

To set up an array, you need to:

- \* *Declare the types of values in the array.* Just as when you declare a variable, you must specify what types of values the array will store.
- \* *Name the array.* In the examples given above, the arrays were named "forward" and "spin\_right," but you can use any name you want.
- \* *Define the size of the array.* The arrays used with the move function must store two values--the pulse width for the left servo and the pulse width for the right servo--so they have a size of two.
- \* *List the values in the array.* These are the pulse width values that will make the robot execute the desired movement or turn.

For example, to set up an array called "forward," with a size of two, and the values 2,000 and 1,000, you would use the following code:



```
int forward[2] = {2000, 1000};
```

### Challenge 3: Write a Move Function

#### Putting Arrays to Work

In this step, you will set up four arrays, called forward, spin\_right, reverse, and spin\_left.

1. Using the Save As command, rename your code file move\_arrays.c.
2. Modify your code, setting up four arrays, as follows. *HINT: The lines setting up your arrays should go between your include statements and the start of your move function. For the forward and reverse arrays, use whatever values produce straight motion in your robot.*

```
// Program 11.6 Array
// Left servo port 9, Right servo port 10

int forward[2]   = {2000,1000};
int spin_right[2] = {2000,2000};
int reverse[2]   = {1000,2000};
int spin_left[2]  = {1000,1000};

int i;
```

3. Next, modify your move function, so that it passes a variable called count for the cycles, and a variable called pulsewidth for the values of the array.

```
//----- MOVE Function -----
void move(int count, int pulse[ ]){
    for(i=0;i<=count;i++){

        digitalWrite(9,1); // --- Program Left Motor
        delayMicroseconds(pulse[0]);
        digitalWrite(9,0);

        digitalWrite(10,1); // --- Program Right Motor
        delayMicroseconds(pulse[1]);
        digitalWrite(10,0);

        delay(20); //Run
    } // for
} // move
```

4. Now, in your setup function, to execute a move, you can use your arrays.

```

void setup() {

  pinMode(9, OUTPUT); //set up port 0 to output 5 volts
  pinMode(10, OUTPUT); //set up port 0 to output 5 volts

  move(10, spin_right);
  move(70, forward);
  move(70, reverse);
  move(30, spin_left);

} //setup

void loop() { }

```

5. Change the arrays in your move statements to make the robot move in different ways.

### Challenge 3: Write a Move Function Writing More Arrays

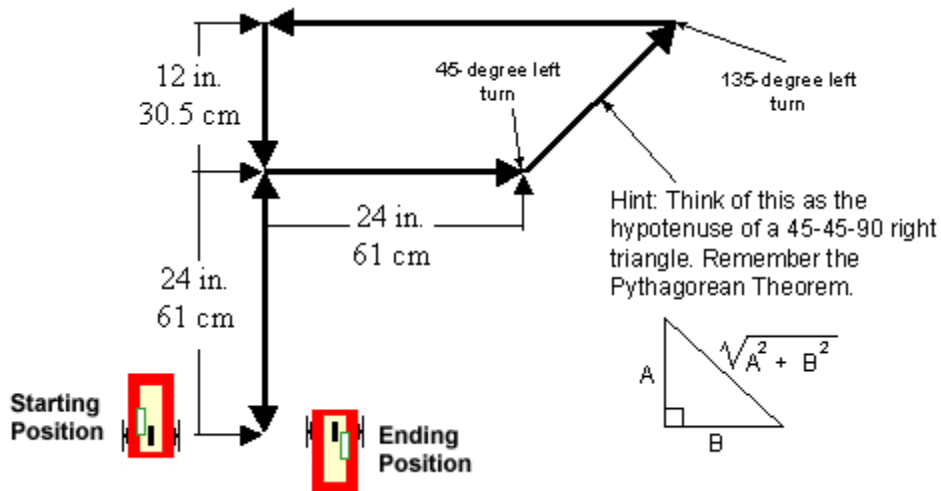
**Slalom**  
Show your work  
to the instructor  
for a grade.

Moving forward, spinning right, spinning left, and moving in reverse are just a few of the many movements that you can make with your robot. Why stop there? Using arrays, you can store the pulse lengths needed for a wide array of movements, including the following:

- Slow forward motion. Both wheels turn forward slowly. (Suggested name: slow\_forward).
- Slow reverse motion. Both wheels turn backwards slowly. (Suggested name: slow\_reverse).
- A bend to the right. The left wheel turns forward and the right wheel stops or turns more slowly. (Suggested name: bend\_right).
- A bend to the left. The right wheel turns forward and the left wheel stops or turns more slowly. (Suggested name: bend\_left).

### Challenge 3: Write a Move Function A Navigational Challenge

1. Calculate the number of pulses needed for all of the movements shown in the course shown below. *NOTE: Except where marked, all turns are 90 degrees.*



2. Using your arrays, write code to make your robot navigate the course, beginning and ending at the same point.

### Challenge 3: Write a Move Function

#### More Challenges

If you have finished with all of the tasks in this unit, try tackling some of the additional challenges in this section.

1. Imagine that you have to parallel park your XBot on the street. Set up obstacles on the floor to simulate a parking spot on the street, and program your XBot to drive up to the spot, and park without touching other cars. Use Figure 22 as a reference.

**Parallel Park**  
Show your work to the instructor for a grade.

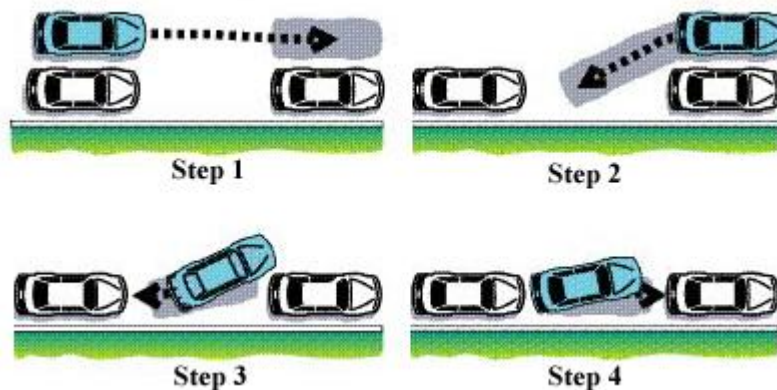


Figure 22. The basics of parallel parking.

2. By adjusting your pulse durations, program the XBot to trace a circle on the floor.

3. Look closely at the Roomba navigation pattern shown in Figure 23. The pattern starts out with a spiral motion, traveling in expanding circles. Program the XBot to trace a similar spiral pattern.

### **Spiral**

Show your work to the instructor for a grade.

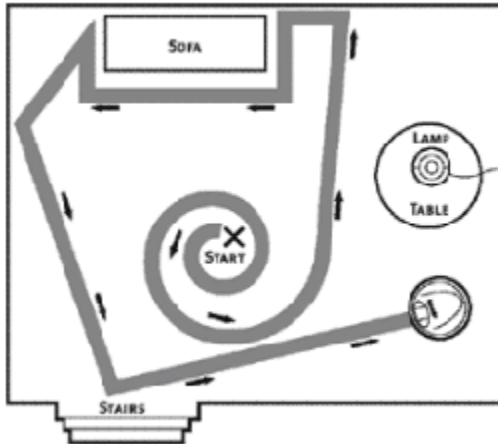
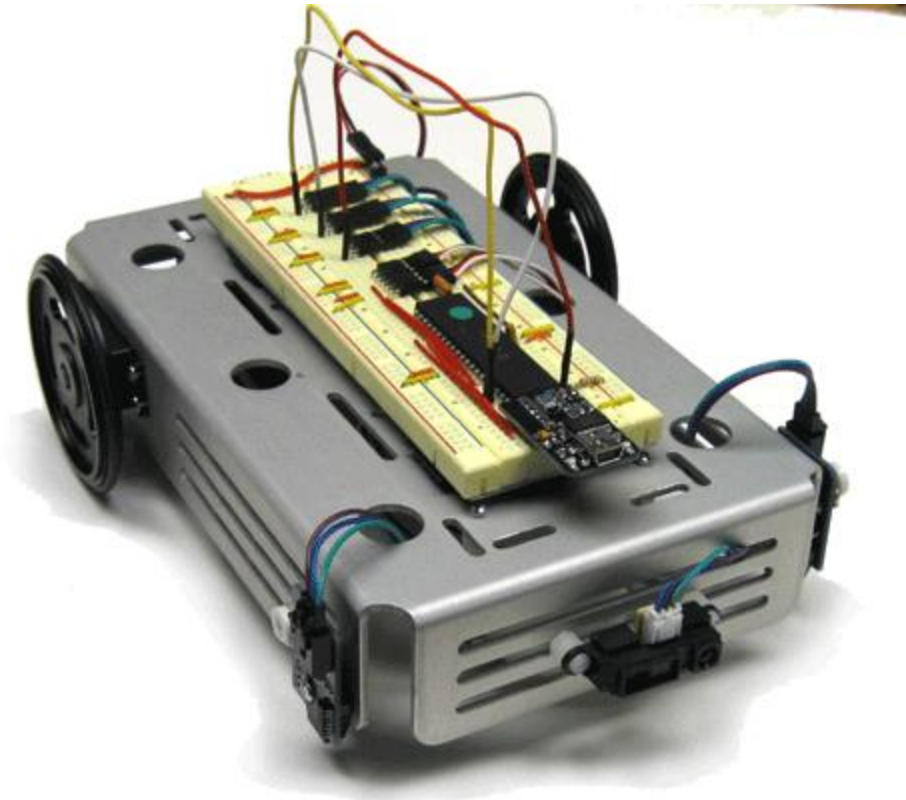


Figure 23. Roomba navigation pattern.

## Sensor Robot

In order to move around without becoming stuck or damaged, a mobile robot needs to monitor and respond to its environment. This ability is made possible by sensors, which send signals to the microcontroller, triggering certain navigational responses. Figure 1 shows a robot equipped with sensors.



**Figure 1. Sensor robot.**

The sensor robot project has two challenges. In Challenge 1, you will add two types of sensors to the robot--a front sensor that will keep the robot from bumping into walls and objects in its path, and two down-facing sensors that will keep the robot from falling down stairs or off tables. In Challenge 2, you will program the robot to execute an avoidance maneuver in response to each sensor.<sup>5</sup>

### Challenge 1: Install the Sensors

---

<sup>5</sup> There are a couple of variants for the IR sensor. There are IR sensors with three or four pins. The version of the four pins allows you to digital values as well as analog values. IR sensors with three pins usually are only digital.





In this challenge, you will add two types of sensors to the robot--a front sensor that will keep the robot from bumping into walls and objects in its path, and two down-facing sensors that will keep the robot from falling down stairs or off tables. Both sensors use infrared light to detect the presence or absence of objects and surfaces.<sup>6</sup>

## Challenge 1: Install the Sensors

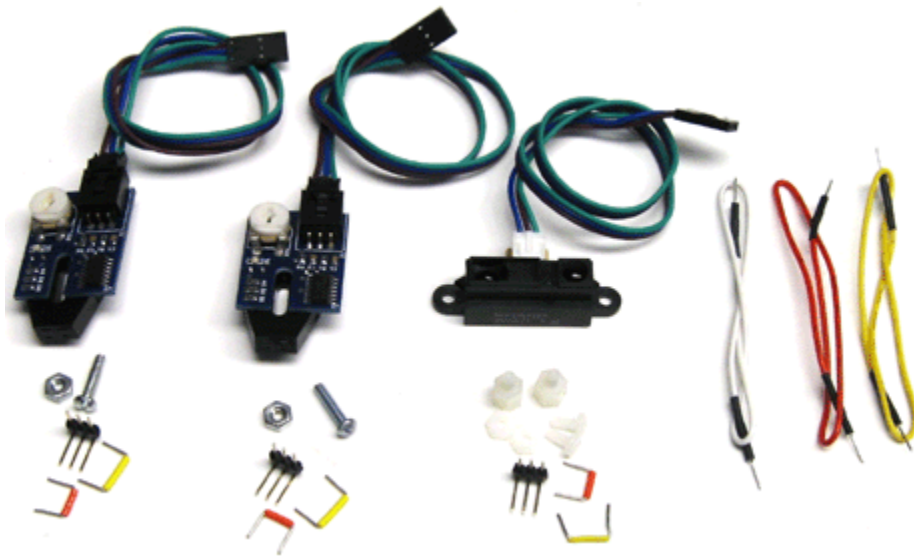
### Collecting Your Components

In order to add sensors to your robot, you will need the following components (shown in Figure 2):

Part	Quantity	Description
A	1	Sharp infrared sensor (front sensor)
B	2	Infrared reflectance sensors (down-facing sensors)
C	2	Machine screws, round head slotted (1/2" 4-40)
D	2	Machine screw nuts (4-40)
E	2	Nylon standoffs (3/8" 4-40)
F	2	Nylon screws (4-40)
G	2	Nylon nuts (4-40)
H	3	Bent connectors (three-prong)
I	6	Jump wires (3 short yellow, 3 short orange)
J	3	Flexible jump wires (1 white, 1 red, 1 yellow)

---

<sup>6</sup> An IR LED and a Photo diode are used in a combination for proximity and color detection. An IR LED (transmitter) emits IR light, that light gets reflected by the object, the reflected light is received by an IR receiver (Photo Diode). Amount of reflection and reception varies with the distance. . This difference causes to change in input voltage through IR input. This variation in input voltage is used for proximity detection.  
<https://roboindia.com/tutorials/digital-analog-ir-pair-arduino/>



NOTE: Some of the components shown in Figure 2, including the jump wires and bent connectors, are contained in the Breadboard Starter Kit. Older Sensor Expansion Packs may contain different sensor equipment. If the sensors in your kit do not match those shown in Figure 2, you will have options that you can click that will take you to instructions for your equipment.

## Challenge 1: Install the Sensors

### Identifying Your Front Sensor

Before moving ahead, carefully inspect the code printed on the bottom of the front sensor, referring to Figure 3. If you have the newer analog sensors, you may proceed with the instructions in this tutorial.

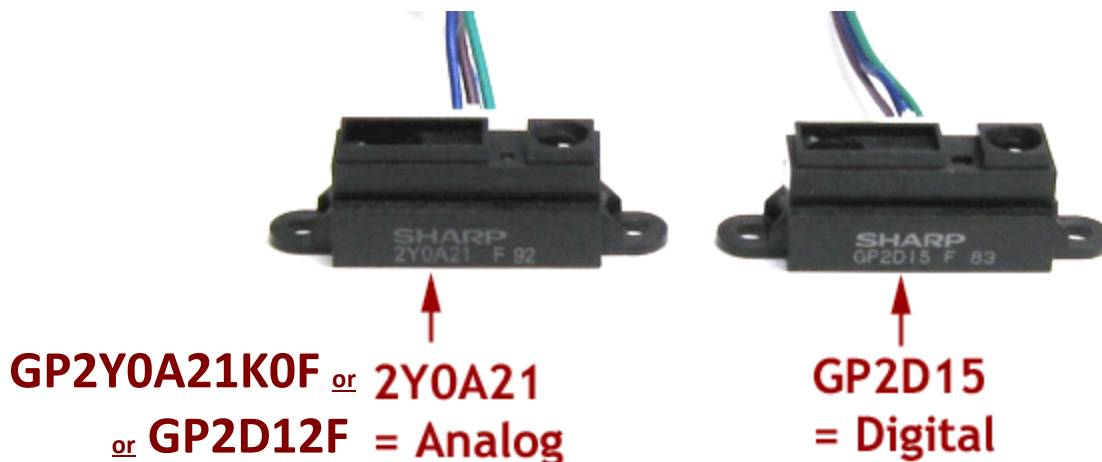


Figure 3. Identifying analog and digital front sensors.



### IMPORTANT!

Before proceeding with the instructions in this tutorial, you must identify the type of sensors included in your kit. Failure to do so will result in the sensors malfunctioning, and it could cause permanent damage to the sensor hardware.

## Connecting the Front Sensor (Digital Version)

If you have the analog sensor, hold Ctrl and Click [HERE](#) to go to the analog version.

The front-facing sensor is mounted to the front of the chassis. It is best to position the sensor on the lowest slot on the chassis, so that the sensor doesn't miss small obstacles in the robot's path.

**1. Using two ½" 4-40 machine screws and two 4-40 machine screw nuts, secure the sensor to the lowest slot on the front of the chassis, as shown in Figure 3.**

***NOTE: Make sure that the colored wires protrude from the top of the sensor, as shown.***

**2. Check to make sure that the sensor is directly centered in the slot before tightening the screws.**

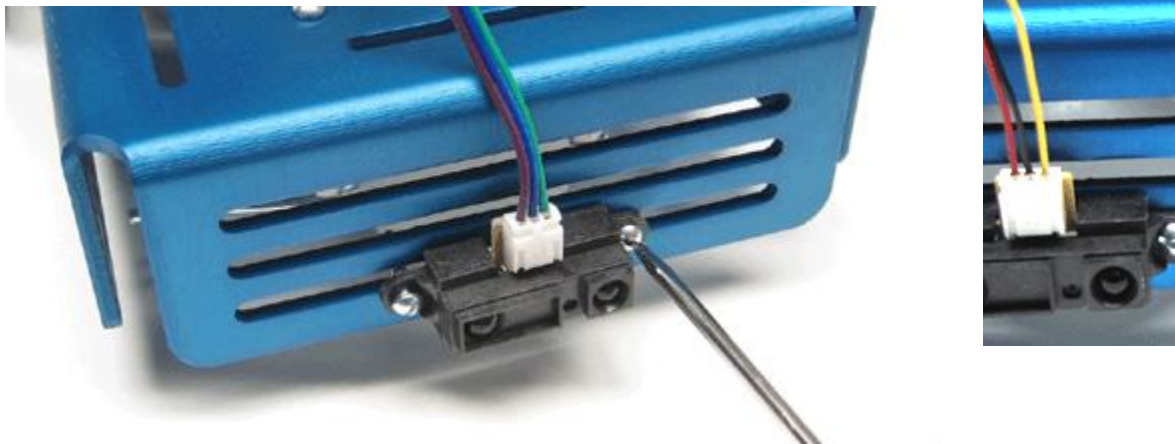


Figure 4. Securing front-facing sensor to chassis.

**3. Insert a flexible red jump wire, a 10,000-Ohm resistor, a three-prong bent connector, a short yellow jump wire, and a short orange jump wire into the XBoard, as shown in Figure 5.**

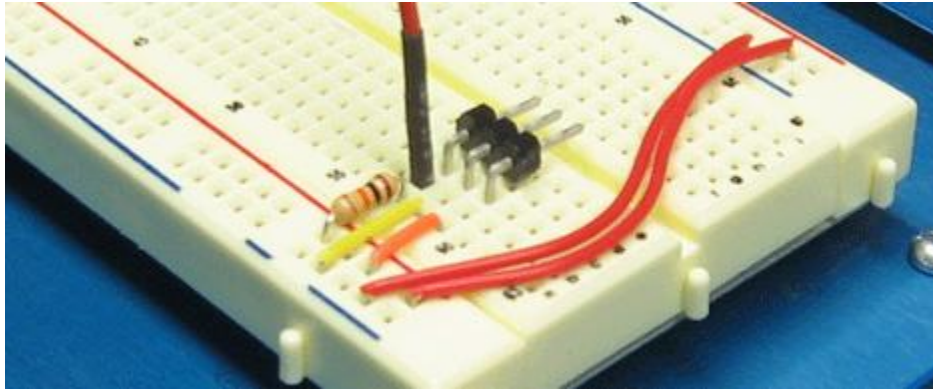


Figure 5. Wires for front-facing sensor.

4. Thread the wire harness from the front-facing sensor through the front panel of the chassis and up through a slot in the top of the chassis. *NOTE: The wire can be routed however you want, provided it stays clear of the front caster and reaches the three-prong connector.*

5. Connect the wire harness to the three-prong connector, as shown in Figures 6 and 7. *NOTE: Be sure to align the colored wires exactly as shown.*

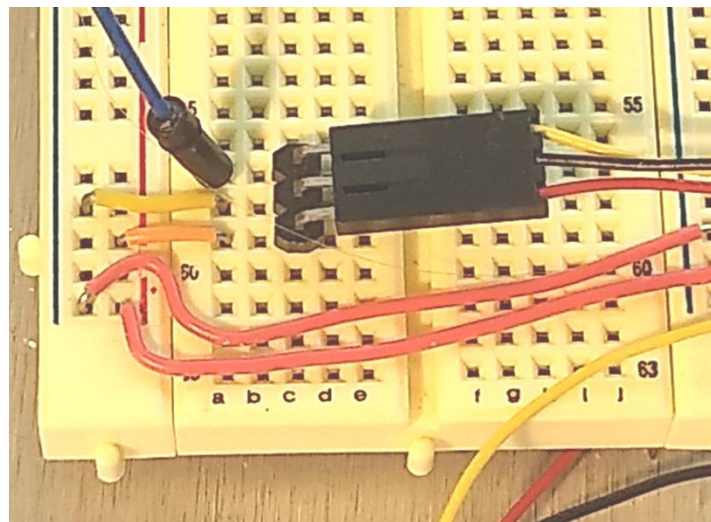


Figure 6. Connecting the older-style sensor leads to the bent three-prong connector. **Red** goes to power; **black** to ground; **yellow** to resistor.

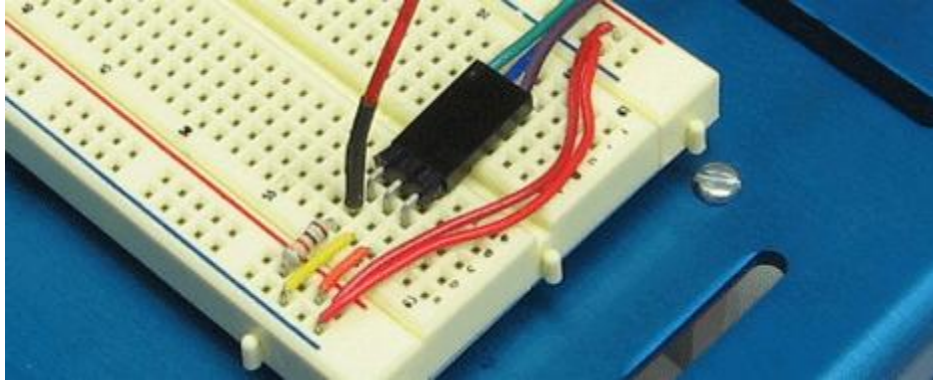


Figure 7. Connecting the newer-style sensor leads to the bent three-prong connector. **Purple=Red(+5v), Blue=Black (gnd), Green=Yellow (Signal)**

**5. (Digital Version)** Connect the free end of the flexible jump wire to Port A4 on the microcontroller.

### **Challenge 1: Install the Sensors**

#### **Installing the Front-Facing Sensor**

The front-facing sensor is mounted to the slots on the front of the chassis.

**1. Using nylon standoffs, screws, and nuts, secure the sensor to the front of the chassis, as shown in Figure 8. NOTE: Make sure that the standoffs insulate the sensor from the chassis.**

**2. Check to make sure that the sensor is directly centered in the slot before tightening the standoffs and screws.**



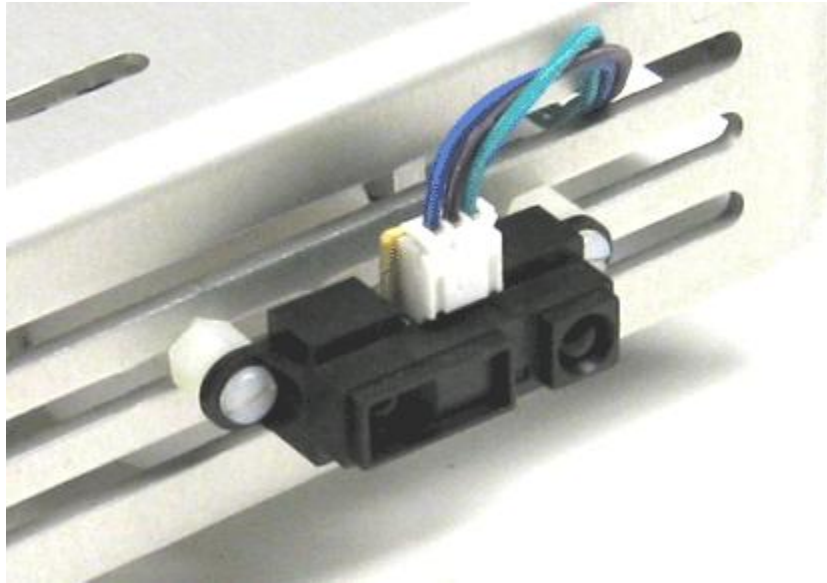
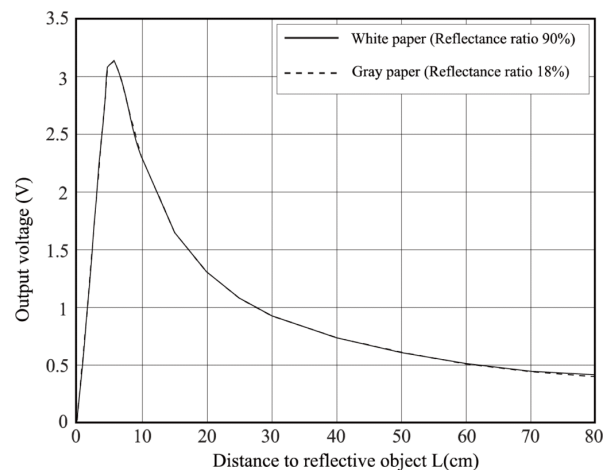


Figure 8. Securing front-facing sensor to chassis.

## Challenge 1: Install the Sensors

### Attaching the Analog Front Sensor

An IR (Infrared) distance sensor uses a beam of infrared light to reflect off an object to measure its distance. The distance is calculated using triangulation. Sharp<sup>7</sup> analog IR sensors, the kind we have in our lab, produces an analog output that varies from 3.1V at 10cm to 0.3V at 80cm. Because this sensor uses angles to calculate distance, it suffers from two limitations. The farther the object is, the less accurate the reading is, and objects can be too close for the triangulation to work. See the graph of voltages vs distance to the right.<sup>8</sup>



Attach the front sensor as shown in Figure 9 below.

The right most wire, **green** in this picture, goes to the microcontroller. The center wire, **blue** in this picture, should be grounded. It will be wired to the blue holes on your breadboard. The left most wire, **purple** in this picture will be attached by a wire to power, the holes with the red line next to them on your breadboard.

<sup>7</sup> Sharp Corporation ... is a Japanese multinational corporation that designs and manufactures electronic products, headquartered in Sakai-ku, Sakai, Osaka Prefecture. Wikipedia

<sup>8</sup> Graph from: <https://www.makerguides.com/sharp-gp2y0a21yk0f-ir-distance-sensor-arduino-tutorial/>

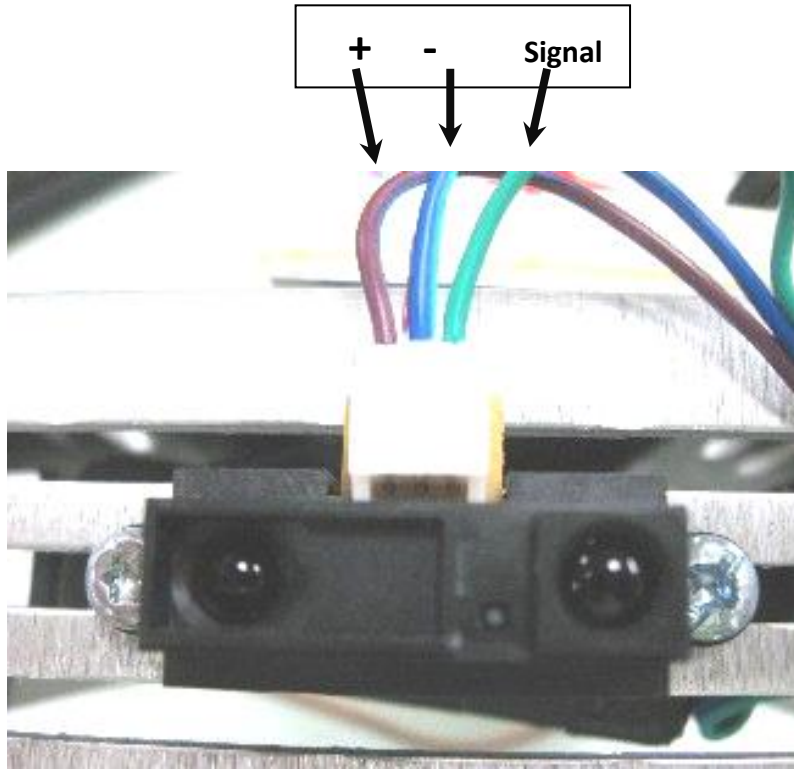


Figure 9. Front sensor wires.

### Wiring the Sensors (Analog Front Sensor)

Each sensor is connected to the breadboard by a three-strand cable. Before connecting the sensors, examine these cables. Each has a **green** or **yellow** (signal) strand, a **blue** or **black** (ground) strand, and a **purple** or **red** (power) strand. The order of these strands is *critical* to the proper functioning of the down sensors, although the color of the wires may be different.

**1. Connect the cables from the three sensors to the breadboard using three bent headers, three short orange jump wires, three short yellow jump wires, and three flexible jump wires, as shown in Figure 10.**

**2. Insert a flexible jump wire, a three-prong bent connector, a short yellow jump wire, and a short orange jump wire into the breadboard, as shown in the Figure 10 below.**



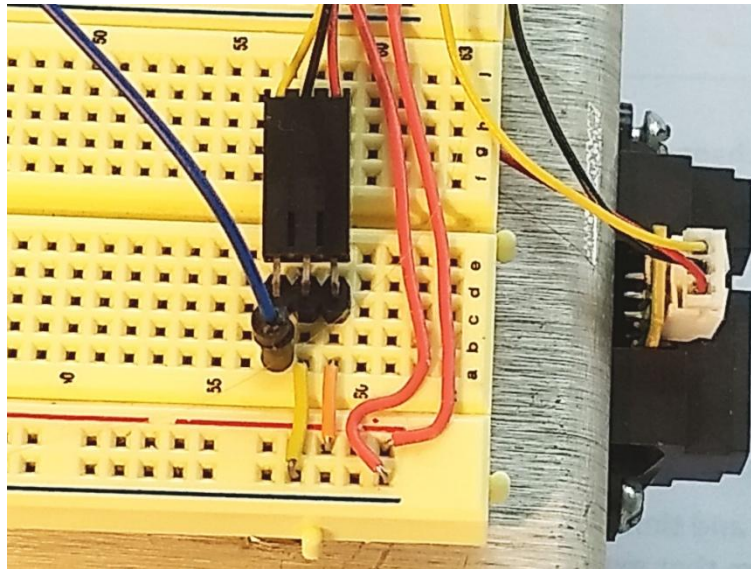


Figure 10. Breadboard set-up for front sensor with yellow, black and red wires.

## 2. Connect the free end of the flexible jump wire to ANALOG Port A4

*(connect the free end of the flexible jump wires from the downward sensors to **Port A1**, and to **Port A2**, if you have wired the down sensors.) See Figure 11.*

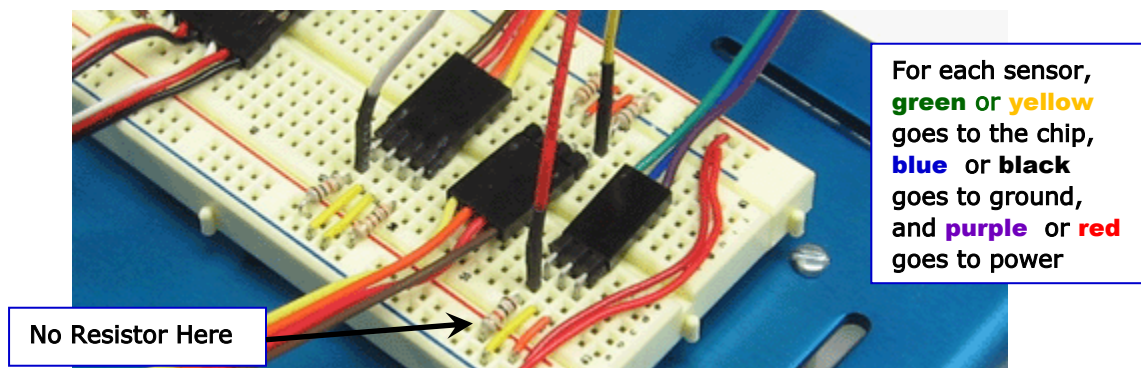


Figure 11. Connecting flexible jump wires to microcontroller.

## 3. Testing your analog front sensor

In the testing program, you will use Arduino's `Serial.begin()`, `analogRead()` and `Serial.print()` functions to monitor the sensor's readings from your computer.

1. Save your code as `sharpir`.

2. Type in the following program then run it.
3. While the program is running, run the serial monitor found in the tools tab.

```
//Sharp IR Sensor
int IR=0;

void setup() {
  Serial.begin(9600); // Set up Serial so computer will show values
  pinMode (A4,INPUT);
}
void loop() {

  IR = analogRead(A4);

  // Print the measured distance to the serial monitor:

  Serial.print(" IR reading -> ");
  Serial.println(IR);
  delay(100);
}
// To see the results, go to
// Tools,then Serial Monitor
```

4. In the real world, sensors vary in sensitivity, mountings are different, wires have resistance; to make programming easy, make a table of sensor values for objects at varying distances.

Distance	IR Reading
10 cm	
15 cm	
20 cm	
25 cm	

Distance	IR Reading
40 cm	
45 cm	
50 cm	
55 cm	

30 cm	
35 cm	

60 cm	
65 cm	

If the circuit worked, continue on to install downward facing sensors.

### Connecting the Down Sensors (Soldered Ribbon Cables)

If you have flat cables, [click here](#) for instructions.

If you have a WCS sensor, [click here](#) for instructions.

The down-facing sensors are attached to the front of the chassis with machine screws. It is important to orient each sensor properly, paying close attention to the position of the infrared emitter (the side labeled E) and the infrared receiver or sensor (the side labeled S).



Figure 12 Soldered Ribbon Cables

**1. Attach the left down-facing sensor to the front of the chassis, using two ½" 4-40 machine screws and nuts. Mount the sensors on the lowest slot, as shown in Figure 1. NOTE: Make sure that the letters E and S are upside down and facing you.**

**2. Repeat step 1 to attach the right down-facing sensor in the same slot on the other side of the front sensor.**

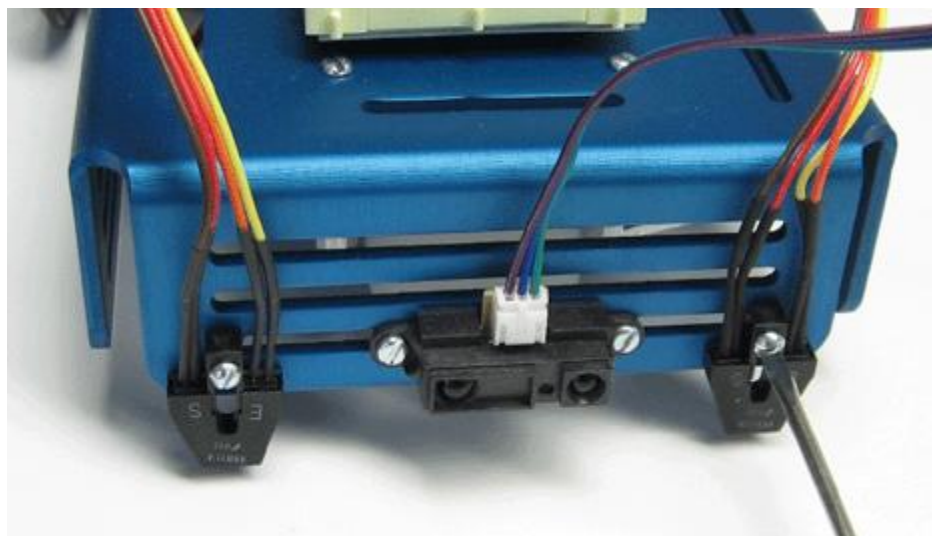
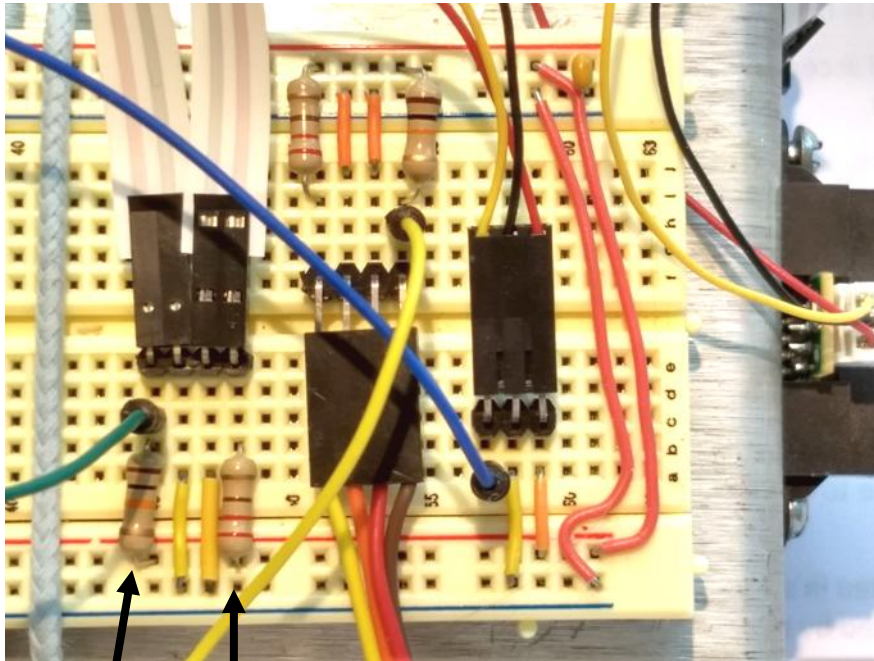


Figure 13. Securing the newer-style down-facing sensor to the chassis.

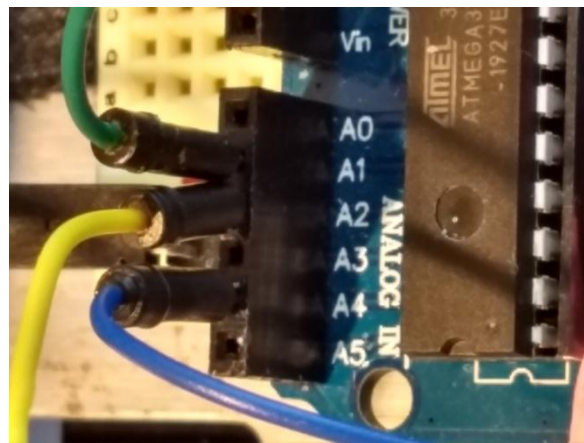
**3. Connect the colored ribbon cables to the breadboard as shown in Figure 14. NOTE: Be sure that the brown strand on each cable aligns with the flexible jump wire and the 10K-Ohm resistor.**



**10KΩ 220Ω Resistors** Be sure the 10KΩ and jumper to the Arduino are connected to the signal wire of the sensor.

**Figure 14. Down-facing sensor connections completed**

**3. Connect the free end of the flexible yellow jump wire to Port B6, and connect the free end of the flexible white jump wire to Port B5, as shown in Figure 15.**



**Figure 15. Connecting flexible jump wires to the Arduino.**



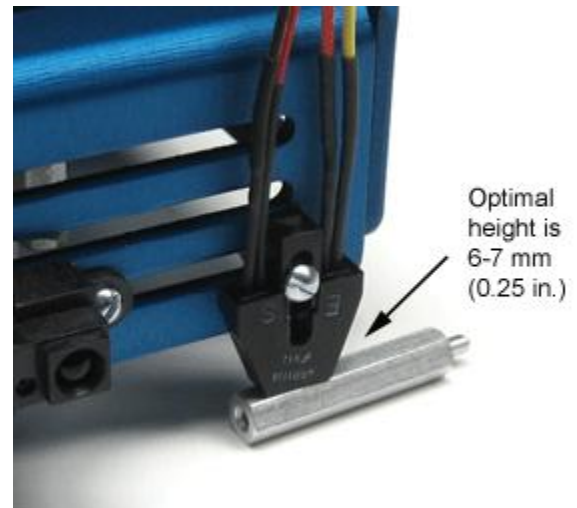
## Challenge 1: Install the Sensors

### Installing the Down-Facing Sensors

The down-facing sensors are attached to the chassis with machine screws.

**1. Attach the left down-facing sensor board to the front of the chassis, using two ½" 4-40 machine screws and nuts. Mount the sensor on the lowest slot, as shown in Figure 5.**

**2. Repeat step 1 to attach the right down-facing sensor in the same slot on the other side of the robot. Click [HERE](#) to continue.**



**Figure 16. Securing the down-facing sensor board.**

### Connecting the Down Sensors (Flat Flex Connections)

If you missed regular multicolored wires, [click here](#) for instructions. If you have a WCS sensor, [click here](#) for instructions.

The down-facing sensors are attached to the front of the chassis with machine screws. It is important to orient each sensor properly, paying close attention to the position of the infrared emitter (the side labeled E) and the infrared receiver or sensor (the side labeled S).

**1. Attach the left down-facing sensor to the front of the chassis, using two ½" 4-40 machine screws and nuts. Mount the sensors on the lowest slot, as shown in Figure 18. NOTE: *Make sure that the letters E and S are upside down and facing you.***

**2. Repeat step 1 to attach the right down-facing sensor in the same slot on the other side of the front sensor.**



**Figure 17 Flat Flex Connections**



Figure 18. Sensor being mounted onto the chassis.

**3. Connect the flat-flex cables to each sensor, as shown in Figure 19.**  
**NOTE: For each sensor, orient the two flat-flex cables so that the dark strands are facing towards the mounting screw and the red strands are facing away from the mounting screw.**

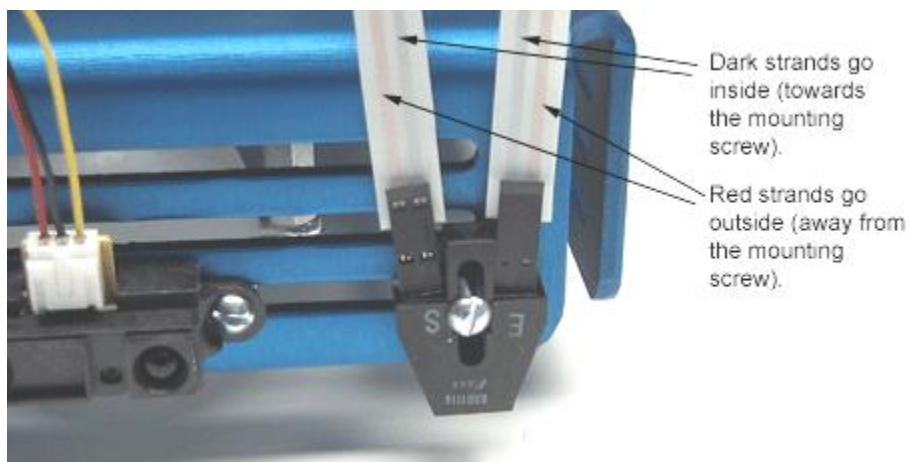


Figure 19. Two-color wires connected to down-facing sensors.

**4. Insert two short yellow jump wires, a flexible white jump wire, a 10,000-Ohm resistor, and a 220-Ohm resistor into the XBoard, as shown in Figure 20.**

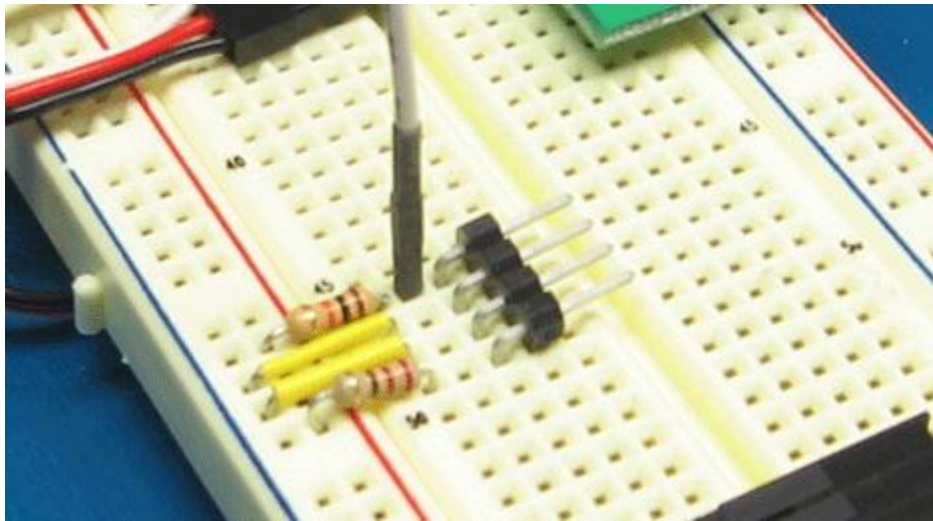


Figure 20. Jump wires and resistors for down-facing sensor.

5. Connect the flat-flex cables from the left down-facing sensor to the four-prong connector, as shown in Figure 21. *NOTE: Make sure that the dark strands align with the yellow jump wires, the red strand from the emitter aligns with the 220-Ohm resistor, and the red strand from the receiver aligns with the 10,000-Ohm resistor.*

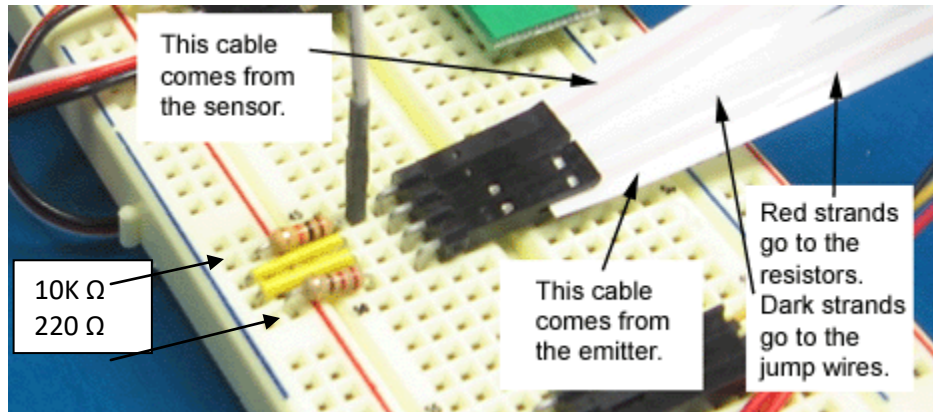


Figure 21. Plugging cables from sensor into the four-prong connector.

6. In the space on the breadboard between the left down-facing sensor connection and the front sensor connection, repeat steps 3 and 4 to connect the right down-facing sensor. Instead of yellow jump wires, use short orange jump wires to make your connections to ground. Also, to make it easier to keep your sensors straight, use a flexible yellow jump wire, instead of a white one. When you have finished, your XBoard should look like the one shown in Figure 22.



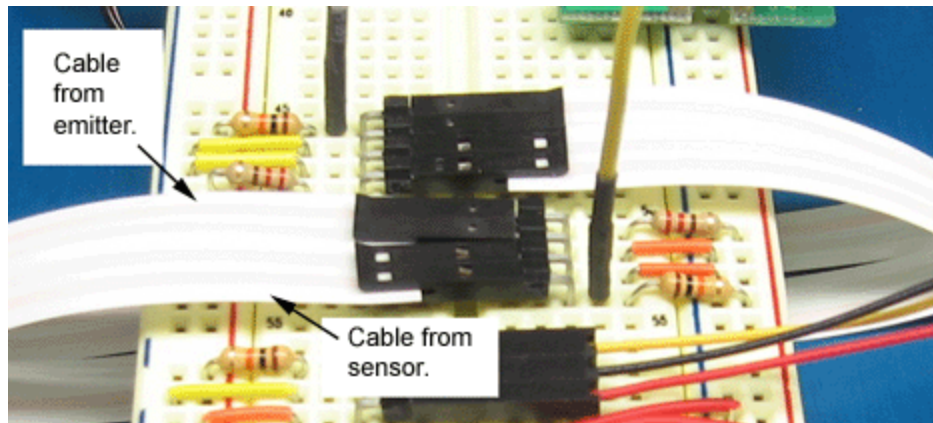


Figure 22. Older style down-facing sensor connections completed.

7. Connect the free end of the flexible jump wires to Port A2, connect the other flexible jump wire to Port A1, as shown in Figure 23.

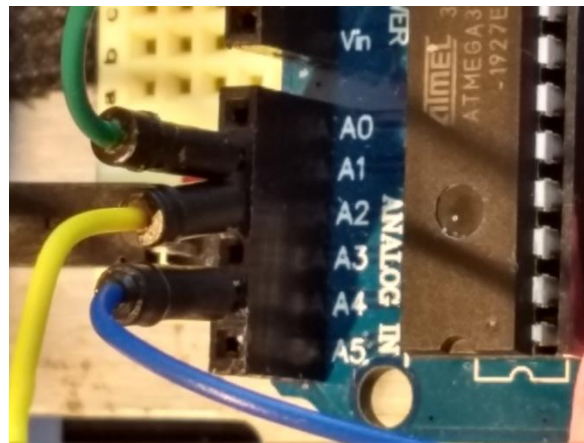


Figure 23. Connecting flexible jump wires to the Arduino

Hold down CTRL and click [HERE](#) to continue.

### Connecting the down sensors (WCS sensor, the LTH1550-01)

If you have multicolored wires, [click here](#) for instructions.

If you have flat cables, [click here](#) for instructions.

The down-facing sensors are attached to the front of the chassis by attaching it to a card and attaching the card to your robot with machine screws. It is important to orient each sensor properly. The **red** wire needs to be attached to a **220  $\Omega$**  resistor that goes to power. The **blue** wire goes to the ground. The **green** wire needs to be



connected to the microcontroller and a **10K  $\Omega$**  resistor that is connected to power.

Figure 24. LTH1550

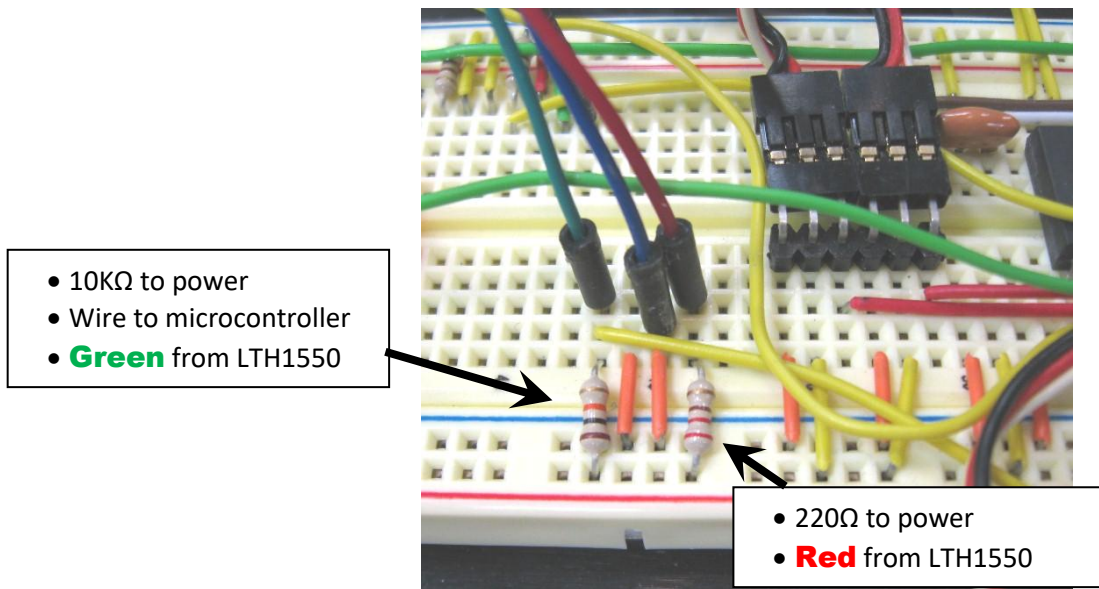


Figure 25. Connecting flexible jump wires from the LTH1550

Attach a jump wire to the green signal wire and attach the other end to port **A2** of the microcontroller, as shown in figure 26.

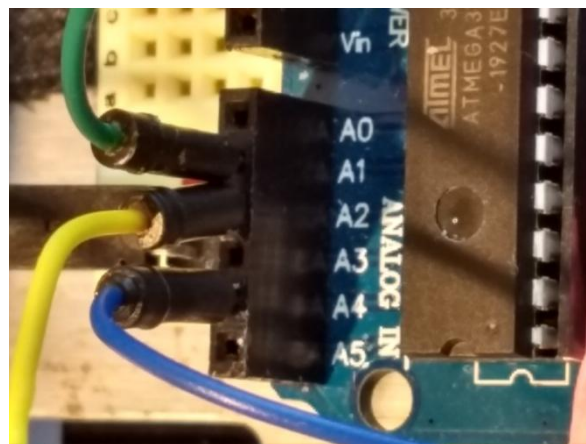


Figure 26. Connecting flexible jump wires to the Arduino

Repeat the process for the sensor on the other side, and port **A1**.

**Continue reading below:**

## Challenge 2: Avoid Obstacles and Hazards



With the sensors mounted and connected, you are ready to write code to make the robot respond to input from the sensors. Your code for this challenge will incorporate your move function and the arrays.

You will program the front sensor to keep the robot from bumping into walls and other obstacles, and program the down-facing sensors to keep the robot from falling down stairs or off tables. By modifying your code, you can also program the down-facing sensors to make the robot track lines on the floor.

## **Challenge 2: Avoid Obstacles and Hazards About Infrared Sensors**

The sensors that you will use in this unit are *infrared sensors*. Each has an infrared emitter that transmits a beam of infrared light and an infrared receiver that detects when the beam bounces back to the sensor from an object. The infrared light emitted from the sensor is a beam of electromagnetic radiation, much like a flashlight beam, except that it is invisible to the eye. Figure 27 shows how this infrared detection works.

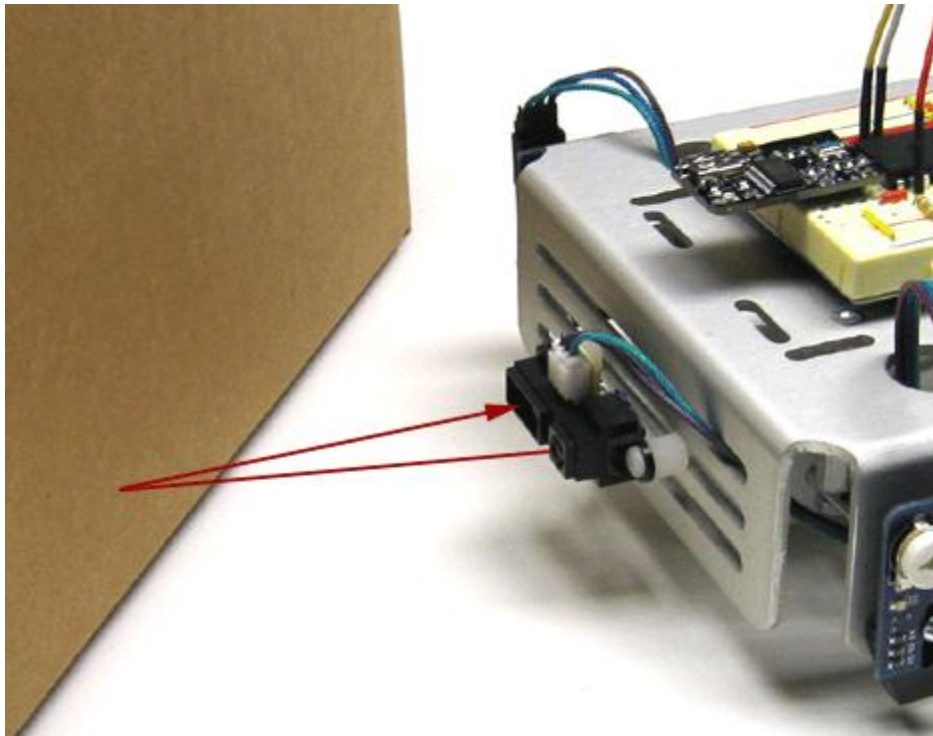


Figure 27. Infrared object detection.

## **Challenge 2: Avoid Obstacles and Hazards Using the Analog Front Sensor**

For instructions on how to install your robot's sensors, click [HERE](#).

The front sensor can detect objects at a distance of 5 to 150 centimeters (approximately 2 to 60 inches). Depending on the object's distance from the sensor, the sensor returns a value ranging from about 900 (the object is very close) to about 30 (object is at the maximum detectable distance from the sensor). When an object is extremely close to the sensor, the sensor cannot detect it properly. Sensors that return values that are continuous over time, as opposed to a discrete off/on, are called analog sensors.

**1. Save your code file as stop.**

**2. Set up your move function, so the robot can move. Then, modify the code to make the robot respond to the front sensor:**

```
// Program 12.3 Stop  
// Left servo port 9, Right servo port 10, A4 IR sensor  
int i;  
  
void setup() {  
  pinMode(9, OUTPUT);           //set up port 0 to output 5 volts  
  pinMode(10, OUTPUT);         //set up port 0 to output 5 volts  
  pinMode (A4,INPUT);  
  
  while (analogRead(A4)<250)    // move while no reflection  
  { move(5,2000,1000); }      //move forward for 5 cycles  
  // ----- no movement instruction here so robot stops  
  
  }//setup  
  
void loop() { }  
  
//----- MOVE Function -----  
void move(int count, int leftPulse, int rightPulse){  
  for(i=0;i<=count;i++){  
  
    digitalWrite(9,1); // Program Left Motor  
    delayMicroseconds(leftPulse);  
    digitalWrite(9,0);  
  
    digitalWrite(10,1); // Program Right Motor  
    delayMicroseconds(rightPulse);  
    digitalWrite(10,0);  
  
    delay(20); //Run  
  } // for
```

```
// move
```

2. Compile and test your new code. The robot should move in the forward direction when there is no object in front of the sensor. When you place your hand or another object within the sensor's sight range (10 to 24 centimeters), the wheels should stop moving.

3. Try adjusting the sensor value that makes the robot stop. Larger values will make the robot stop closer to an object; smaller values will make it stop farther from an object.

## Challenge 2: Avoid Obstacles and Hazards Using Define Statements

With many ports in use, it is easy to get confused about which motor or sensor is connected to which port. By adding **define statements** to your code, you can store this information, so that you are less likely to mix up your ports. A define statement allows you to give a specific name to a port. For example, you can call Port 9 "right\_motor," call Port 10 "left\_motor," and call Port A2 "left\_down," and Port A1 "right\_down." That way, when you want to send a pulse to a motor or read a value from a sensor, you can refer to the component by name without having to remember which port it is attached to.

1. Using the Save As command, rename your code file sensor\_defines.c.

2. Add define statements to your code to simplify programming, as follows:

```
#define left_motor    9
#define right_motor   10
#define left_down     A2
#define right_down    A1
#define front_sensor  A4
```

These statements should go between the comments at the top of your program and the **void setup();** command.

3. Replace the port references in your move function, as follows:

```
//----- MOVE Function -----
void move(int count, int pulse[]){
    for(i=0;i<=count;i++){

        digitalWrite(9,1); // Program Left Motor
```

```

delayMicroseconds(pulse[0]);
digitalWrite(9,0);

digitalWrite(10,1); // Program Right Motor
delayMicroseconds(pulse[1]);
digitalWrite(10,0);

delay(20); //Run
} // for

} // move

```

4. In your main function, replace the port references, as follows: <sup>9</sup>

```

// Program 12.3 Stop
#define left_motor 9
#define right_motor 10
#define left_down A2
#define right_down A1
#define front_sensor A4

int forward[2] = {2000,1000};
int spin_right[2] = {2000,2000};
int reverse[2] = {1000,2000};
int spin_left[2] = {1000,1000};

int i;

void setup() {
  pinMode(right_motor, OUTPUT);
  pinMode(left_motor, OUTPUT);
  pinMode (front_sensor,INPUT);

  while (analogRead(front_sensor)<250) // move while no reflection
  { move(5,forward); } //move forward for 5 cycles
  // no movement instruction here so robot stops

} //setup

void loop() { }

```

<sup>9</sup> Using define statements is good programming practice. If the program if the robot is rewired, just changing a number in the define statement at the top changes everything throughout the program.



## 5. Compile and test your new code.

### **Stop**

Show your work  
to the instructor  
for a grade.

### **Escape**

Show your work  
to the instructor  
for a grade.

## **Challenge 2: Avoid Obstacles and Hazards Using the Down Sensors**

Although you will need both down-facing sensors to keep the robot from falling down stairs or off a table, it is much simpler to program the sensors one at a time. In this section, you will write code to for the left down-facing sensor. Your code will be very similar to the code you used to program the front-facing sensor. You have to set up each down-facing sensor port as an input, and use an *if ... e/se* statement to make the robot's behavior hinge on the value sent by the sensor.

1. To make it easier to remember which sensor you are trying to program in this step, put a small piece of tape on the left down-facing sensor.
2. Rename your code file left.
3. Modify your main function to read from the left down-facing sensor, as follows:

### **// Program 12.4 Left Downward Sensor**

```
#define left_motor  9 // Change these if you wire to different ports
#define right_motor 10
#define left_down   A2
#define right_down  A1
#define front_sensor A4

int forward[2]  = {2000,1000};
int spin_right[2] = {2000,2000};
int reverse[2]   = {1000,2000};
int spin_left[2] = {1000,1000};

int i;
```



```

void setup() {
  pinMode(right_motor, OUTPUT);
  pinMode(left_motor, OUTPUT);
  pinMode (front_sensor,INPUT);
  pinMode (left_down, INPUT);
  pinMode (right_down, INPUT);

  while (1==1){ // loop forever
    if (analogRead(left_down)>300){ //Doesn't see the floor
      move(10,reverse);
      move(10,spin_right); }
    else {
      move(1,forward);}
  }//while

} //setup

void loop() { }

//----- MOVE Function -----
void move(int count, int pulse[ ]){
  for(i=0;i<=count;i++){

    digitalWrite(9,1); // --- Program Left Motor
    delayMicroseconds(pulse[0]);
    digitalWrite(9,0);

    digitalWrite(10,1); // --- Program Right Motor
    delayMicroseconds(pulse[1]);
    digitalWrite(10,0);

    delay(20); //Run
  } // for
} // move

```

#### 4. Compile and test your new code.



#### **IMPORTANT!**

Until you have both down-facing sensors properly programmed, there is a danger of your robot falling off the table. *BE SURE TO CATCH YOUR ROBOT BEFORE IT FALLS!*

If your robot keeps backing and turning, or does not respond when the sensor has no surface under it, you can adjust it by raising or lowering the number 300.<sup>10</sup>

## Challenge 2: Avoid Obstacles and Hazards Using Both Down Facing Sensors

In order to prevent the robot from falling off the table, you need to modify the language of your *if...else if...* statement to gather input from both sensors.

1. Rename your code file `both_sensors.c`.

2. Modify the *if ... else* statement in your main function, as follows:

### // Program 12.6 LeftRight Left and Right Downward Sensor

```
#define left_motor 9
#define right_motor 10
#define left_down A2
#define right_down A1
#define front_sensor A4

int forward[2] = {2000,1000};
int spin_right[2] = {2000,2000};
int reverse[2] = {1000,2000};
int spin_left[2] = {1000,1000};

int i;

void setup() {
  pinMode(right_motor, OUTPUT);
  pinMode(left_motor, OUTPUT);
  pinMode(front_sensor, INPUT);
  pinMode(left_down, INPUT);
  pinMode(right_down, INPUT);

  while (1==1){ // loop forever
```

<sup>10</sup> There are three things you can do to change the downward sensor's response to the surfaces it moves across. First, raising the number in the IF statement will make it more sensitive; lowering it will make it less sensitive. Second, you can also raise or lower the sensor itself. Lowering it makes it more sensitive, raising it makes it less sensitive, but if you raise it too much or lower it too much it will stop working. The ideal height is roughly one-eighth of an inch, or 4 to 5 millimeters. Third, you can also change the 220Ω resistor. Lowering it makes the sensor more sensitive, but do not go below 100Ωs.

```

if (analogRead(left_down)>300){ //L Doesn't see the floor
  move(10,reverse);
  move(10,spin_right); }

else{ // not left_down
  if (analogRead(right_down)>300){ //R Doesn't see the floor
    move(10,reverse);
    move(10,spin_left); }

  else { // not right_down and left_down
    move(1,forward);}

  }// else not left_down

} //while

} //setup

void loop() { }

//----- MOVE Function -----
void move(int count, int pulse[ ]){
  for(i=0;i<=count;i++){

    digitalWrite(9,1); // --- Program Left Motor
    delayMicroseconds(pulse[0]);
    digitalWrite(9,0);

    digitalWrite(10,1); // --- Program Right Motor
    delayMicroseconds(pulse[1]);
    digitalWrite(10,0);

    delay(20); //Run
  } // for
} // move

```

### 3. Compile and test your new code.

**LeftRight**

Show your work  
to the instructor  
for a grade.

## Challenge 2: Avoid Obstacles and Hazards Adjusting the Sensitivity

The sensitivity of some down sensors can be adjusted, using their on-board potentiometer, as shown in Figure 9. Turning the dial counterclockwise makes the device more sensitive, and turning it clockwise makes it less sensitive. The red LED glows to indicate when the device does not “see anything,” meaning it is sending a LOW signal to the microcontroller. The sensitivity adjustment can be important when programming the sensor to detect the difference between light and dark colors.

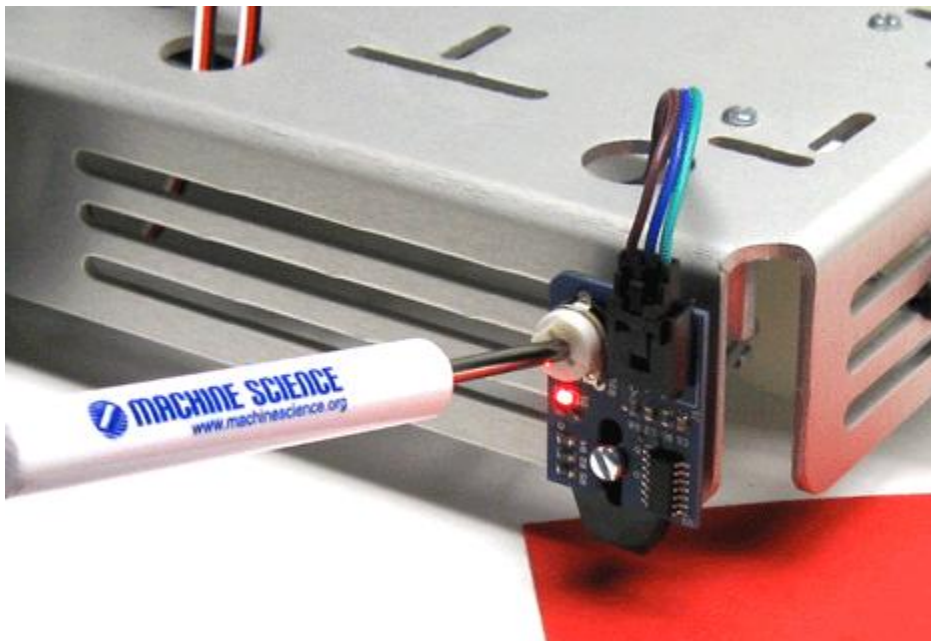


Figure 9. Adjusting the sensitivity of the down-facing sensor.

**Big S**

Show your work  
to the instructor  
for a grade.

## Challenge 2: Avoid Obstacles and Hazards Using All Three Sensors

By combining input from the front sensor and the down-facing sensors, you can program the robot to perform more sophisticated behaviors. For example, in this section, you will program the robot to track the line, but to stop if it encounters an object in its path.

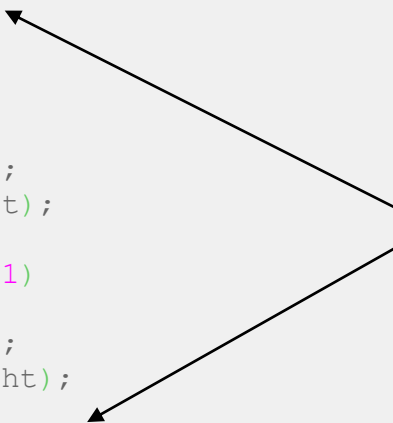
1. Rename your code file `all_sensors.c`.

2. Modify your code to incorporate input from the front sensor, as follows:

```
void main(void)
{
    left_motor_dir=0;
    right_motor_dir=0;
    left_down_dir=1;
    right_down_dir=1;
    adc_init(ALL_ANALOG);
    while(1==1)
    {
        if(right_down==1)
        {
            move(10,reverse);
            move(10,spin_left);
        }
        else if(left_down==1)
        {
            move(10,reverse);
            move(10,spin_right);
        }
        else if(adc_read(0)>250)
        {
            move(100,stop);
        }
        else
        {
            move(1,forward);
        }
    }
}
```

Analog Sensor

For Digital  
Replace these



3. Compile and test your new code.

## Challenge 2: Avoid Obstacles and Hazards Tracking a Line

The down-facing sensors have an interesting characteristic--they can't see dark colors very well. This is because infrared light does not reflect very well off of dark-colored surfaces. As a result, if you put a strip of black electrical tape down on the table, the robot will mistake it for the edge of the table. Using this feature, you can program the robot to track a line. Before starting this section, establish a continuous line on a white or light-colored, reflective surface, using black electrical tape. The line can have curves and bends, but it should not have any bend sharper than 90 degrees.

**1. Rename your code file `line_tracker.c`.**

**2. Modify your code to so that the robot tracks the line, making minor adjustments to its course whenever it detects the line. HINT: The direction of the robot's turning maneuver will be the opposite of the direction needed for table edge/stair avoidance. To avoid confusion, you may want to add define statements for `SEE_LINE` and `NOT_SEE_LINE`.**

**3. Compile and test your new code.**

## Remote Control Robot

Although mobile robots are capable of moving around and responding to their environment autonomously, it can be useful to control a robot's behavior directly, using a remote control. Figure 1 shows a mobile robot, with a remote control.

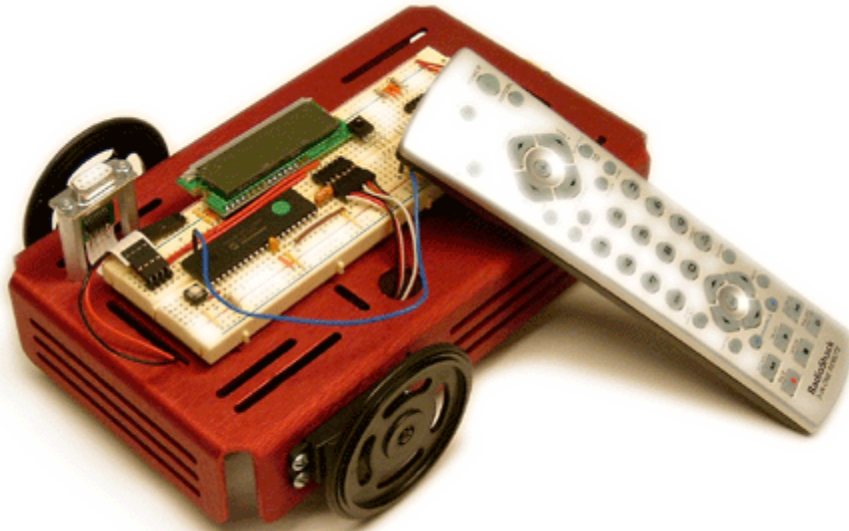


Figure 1. Remote control robot.

In this project, you will add a component to the XBot to receive signals from a remote control, and then program the microcontroller to decipher these signals. This will allow you to control the robot directly, executing forward and reverse motions and right and left turns. This project has two challenges--adding the infrared sensor and programming the remote control functionality.

### Challenge 1: Set Up the Receiver and the Remote



Mobile Robot Unit 3 begins with a hardware challenge--adding the infrared receiver to the XBoard to receive signals from the remote control. The installation of the receiver is very simple. It has three prongs that connect to power, ground, and the microcontroller. In this challenge, you will also set up the universal remote control for your robot.

### Challenge 1: Set Up the Receiver and the Remote Collecting Your Components

In order to complete this unit, you will need the following components (shown below in Figure 2):



Part	Quantity	Description
A	1	Infrared receiver
B	2	Jump wires (1 yellow and 1 orange)
C	1	Flexible jump wire
D	1	Programmable remote control



Figure 2. Remote control pack components.

### Challenge 1: Set Up the Receiver and the Remote Identifying Your Infrared Receiver

Remote Control Expansion Packs shipped before December 2010 have a Panasonic infrared receiver. Packs shipped after December 2010 have a Toshiba receiver. The two parts (shown in Figure 3) differ only slightly in appearance, but they are wired differently, so be sure to identify your receiver before proceeding to the next step.

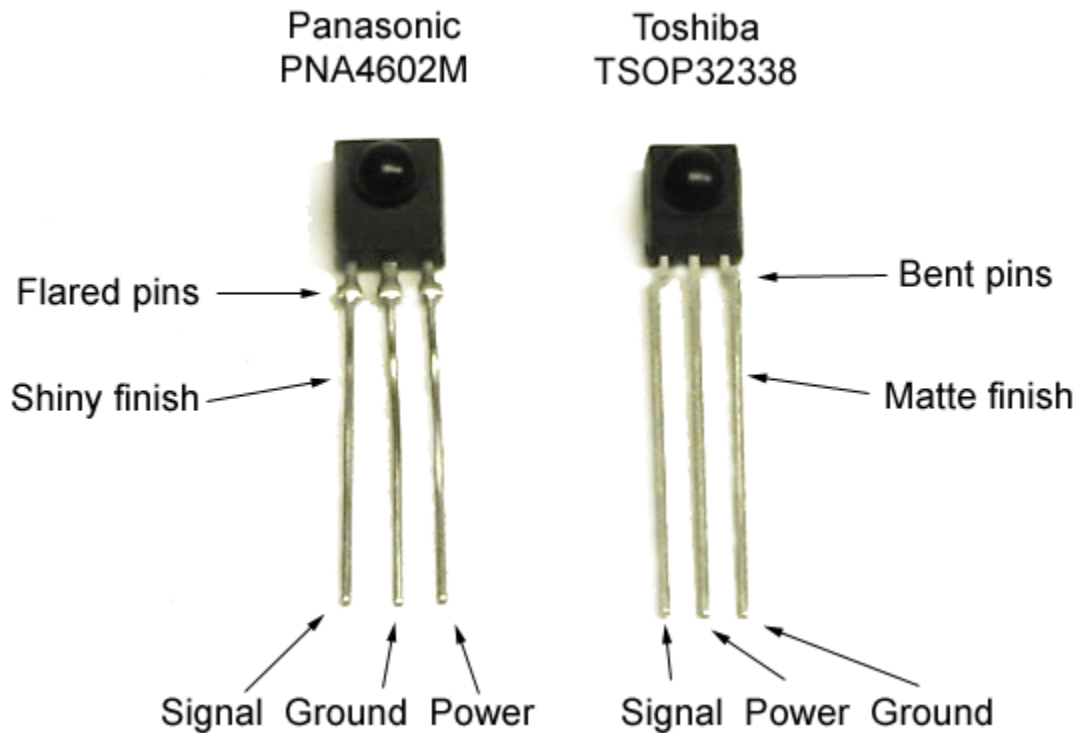


Figure 3. Panasonic (left) and Toshiba (right) infrared receivers.

### Challenge 1: Set Up the Receiver and the Remote

#### Installing the Infrared Receiver

1. Using pliers, gently bend a 90-degree angle in the three prongs on the infrared receiver, as shown in Figure 4.

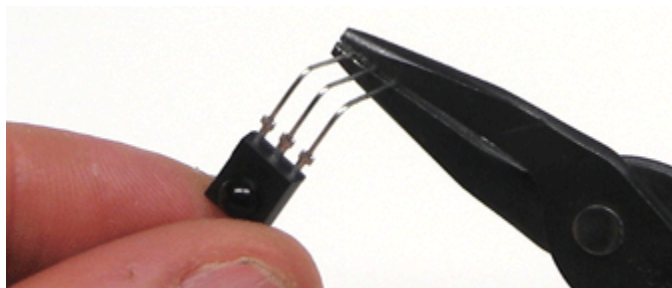


Figure 4. Bending the prongs of the infrared receiver.

2. Insert the infrared receiver, a short yellow jump wire, a short orange jump wire, and a long flexible blue jump wire into the XBoard, as shown in Figure 5. Note that *the polarity of the two types of receivers is reversed*.

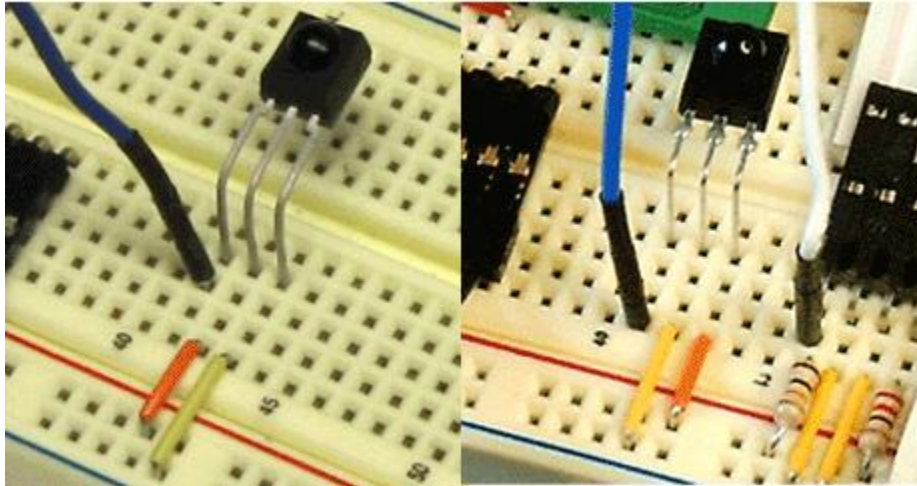


Figure 5. Installing the Toshiba (left) and Panasonic (right).

3. Connect the loose end of the flexible jump wire to Port B4 on the microcontroller.

### **Challenge 1: Set Up the Receiver and the Remote** **Setting Up the Remote**

In order to use the remote control, you must first install batteries in the device and program it with the correct TV code. The procedure to do this is different for every remote. Below are the instructions to program several remote controls that have been included in Machine Science robot kits.



Figure 6. Four different types of remotes, with Sony TV code.

***For the Radio Shack 3-in-1 Remote Control, use the following procedure:***

1. Press and hold the CODE SEARCH button on the remote control until the red indicator stays on.
2. Press the TV button. The red indicator light should blink and then stay lit.
3. Press the 4 button, then the 1 button, then the 4 button. The red light should go off. If it stays lit, repeat steps 2, 3, and 4.

***For the Aifa URC4 Remote, use the following procedure:***

1. Press the green SET button on the remote control and hold it while pressing the TV button.
2. When the red indicator light comes on, release both buttons. The red light should stay on.
3. Press the 1 button, then the 2 button, then the 8 button. The red light should go off. If it stays lit, repeat steps 2, 3, and 4.

***For the Philips Universal Remote, use the following procedure:***

1. Insert two AAA batteries in the remote control, as shown above in Figure 5. *Batteries are not included with Remote Expansion Pack.*

2. Press the CODE SEARCH button and hold it down until the red LED stays lit.

3. Press the 0 button, then the 4 button, then the 1 button, then the 4 button. The red light should go off. If it stays lit, repeat steps 1 and 2.

***For the RCU-4450 remote, use the following procedure:***

1. Press the CODE SEARCH button and hold it down until the red LED stays lit.

2. Press and release the TV button.

3. Press the 0 button, then the 0 button again, then the 2 button. The red light should go off. If it stays lit, repeat steps 1 to 3.

***For the Unmarked remote, use the following procedure:***

1. Press the CODE SEARCH button and hold it down until the red LED stays lit.

2. Press and release the TV button.

3. Press the 0 button, then the 0 button again, then the 2 button. The red light should go off. If it stays lit, repeat steps 1 to 3.

## **Challenge 2: Control the Robot**



In this challenge, you will write code to decipher signals from the remote control, and then program the XBot to respond to these signals. First, you will learn how infrared communication works, using a standard universal remote control (for TVs, VCRs, and cable boxes). Then, you will write code so that your XBot's infrared receiver can receive and decipher the remote control's coded signals.

### **Challenge 2: Control the Robot About Remote Controls**

The XBot's remote control is a standard universal remote for home electronic devices, such as TVs, VCRs, and cable boxes. (This type of remote can be

purchased at any electronics store for about \$10). When you push a button on the remote, it sends a coded, infrared signal to your TV, VCR, or cable box, as shown in Figure 7. These electronic devices have infrared receivers, which relay the signals from the remote to a microcontroller. The microcontroller decodes the signals and then initiates tasks, such as adjusting the volume on the TV, switching the VCR to fast-forward, or flipping the channel on the cable box from CNN to HBO.

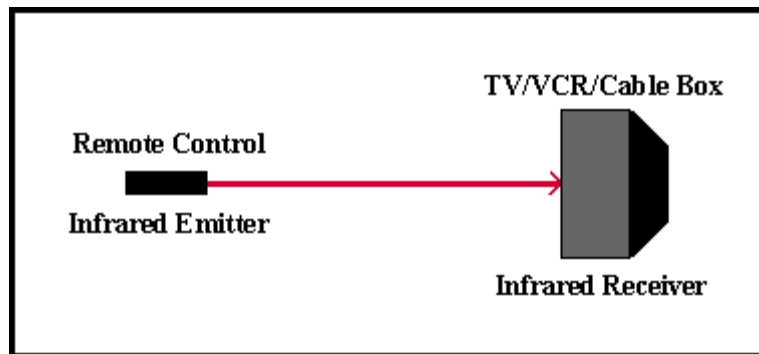


Figure 7. Infrared remote control.

Each of the major electronics companies has its own scheme for encoding the infrared signals sent out by their remotes. A universal remote can emulate the encoding scheme used by almost any remote control. When you set up the universal remote in the previous step, you were programming it to use Sony's infrared encoding scheme for TVs. This means that most Sony TV remote controls will control your XBot as well as the universal remote supplied in this kit.

## Challenge 2: Control the Robot About the Sony Encoding Scheme

In the Sony encoding scheme, the signal from the remote control is not a continuous beam of infrared light—it is a series of intermittent short and long pulses. The receiver translates these short and long pulses into periods of high voltage (5 volts) and low voltage (0 volts), which the microcontroller interprets as 1s and 0s. Figure 8 shows how this works.

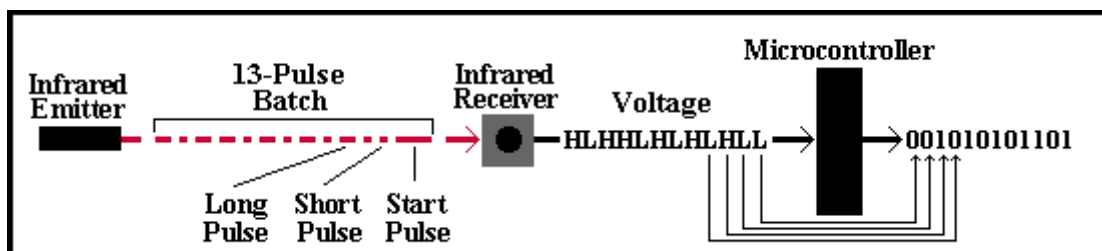


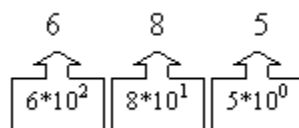
Figure 8. Translating infrared pulses into 1s and 0s.

The pulses from the infrared emitter are sent in batches, with 13 pulses in each batch. The first pulse is a "start" pulse, which signals that a batch is starting. Each of the next 12 pulses is either long or short, with long pulses representing 1s and short pulses representing 0s. If you look closely at Figure 8, you will notice that, after the start pulse, the infrared receiver sends a high voltage signal for every long pulse it receives and a low voltage signal for every long pulse it receives. The microcontroller then interprets these signals as 1s and 0s, respectively, and arranges them into 12-digit strings, such as 001010101101. Notice that the order of the 1s and 0s is the reverse of the order of the infrared pulses.

## Challenge 2: Control the Robot About Binary Numbers

To the microcontroller, every string of 1s and 0s has a specific meaning: it represents a *binary* number. Binary is a system of counting used by computers. Binary counting differs from our usual way of counting in one important respect--binary uses only two digits: 1 and 0. The number system we commonly use is called the *decimal* system, and it is based on 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. We can use these 10 digits to represent any number we want--from single-digit numbers, like 3 and 7, to multi-digit numbers, like 685.

Each digit in a multi-digit decimal number has a different value, depending on its place in the number. For example, in the number 685, the digit 6 is in the hundreds place, meaning it has a value of 600. The digit 8 is in the tens place, and has a value of 80. The 5 is in the ones place, and has a value of 5. This is represented schematically below:



$$\begin{aligned}
 6*100 + 8*10 + 5*1 &= \\
 600 + 80 + 5 &= \\
 685 &
 \end{aligned}$$

Seems obvious, doesn't it? You are so used to seeing decimal numbers, you probably don't realize that you are performing a calculation each time you see one, but you are!

Likewise, each digit in a binary number has a different value, depending on its place. The value of each digit is based on a power of 2, rather than a power of



10. For example, consider the first string of 1s and 0s sent to the microcontroller from the infrared receiver in the previous section--001010101101. The value of this binary number can be calculated, as follows:

0	0	1	0	1	0	1	0	1	1	0	1
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
1*2 <sup>11</sup>	1*2 <sup>10</sup>	0*2 <sup>9</sup>	1*2 <sup>8</sup>	0*2 <sup>7</sup>	1*2 <sup>6</sup>	0*2 <sup>5</sup>	1*2 <sup>4</sup>	0*2 <sup>3</sup>	0*2 <sup>2</sup>	1*2 <sup>1</sup>	0*2 <sup>0</sup>

$$\begin{aligned}
 &0*2048 + 0*1024 + 1*512 + 0*256 + 1*128 + 0*64 + 1*32 + 0*16 + 1*8 + 1*4 + 0*2 + 1*1 = \\
 &\quad 0 + 0 + 512 + 0 + 128 + 0 + 32 + 0 + 8 + 4 + 0 + 1 = \\
 &\quad \quad \quad 685
 \end{aligned}$$

As it turns out, the decimal number 685 and the binary number 001010101101 have the same value. This may seem confusing, but the important thing to remember is that any number can be represented in binary, using just 1s and 0s, and any binary number can be easily converted into a decimal number, using the process shown above.

## Challenge 2: Control the Robot Printing a Remote Data Sheet

In the next step, you will be asked to record important data about the signals sent from the remote control. By recording this information on paper, you will be creating a data sheet for your remote control, which will be a useful reference for future activities.

1. Click [here](#) to open a printable version of this page in PDF format. You will need the [Acrobat Reader](#) from Adobe to view the PDF.

2. Select the Print option to print this page.

REMOTE CONTROL	
Button	Number Displayed on LCD
UP ARROW (CHAN+)	
DOWN ARROW (CHAN-)	
LEFT ARROW (VOL-)	
RIGHT ARROW (VOL+)	
1	
2	
3	
4	

5	
6	
7	
8	
9	

## Challenge 2: Control the Robot Displaying Data from the Remote on the LCD

In effect, when you press a button on the remote control, you are sending a specific number to the microcontroller. The microcontroller's code determines what the device will do each time it receives this number. In the next section, you will program the robot to move in various ways, based on the number it receives. As a first step, you will program the microcontroller to convert the binary number it is receiving to decimal, and display the decimal number on the LCD.

**1. Launch the Programming Portal, create a new code file, and save it as remote\_lcd.c.**

**2. Enter the following code into the Editor window:**

```
#include "mxapi.h"
void main(void)
{
    int button; //Declare a variable, called button
    button=0;   //Set button equal to 0
    lcd_init(); //Initialize the LCD
    while(1==1)
    {
        button=remote(); //Set button equal to value from remote
        if(button!=0)
        {
            lcd_instruction(CLEAR); //Clear LCD
            lcd_decimal(button); //Display button value on LCD
            button=0; //Set button equal to 0
        }
    }
}
```

**3. Compile and test your new code.**

## Challenge 2: Control the Robot Controlling the XBot with the Remote

Now that you know how to decode the infrared signals being sent to the microcontroller, programming the XBot to respond to the remote control is relatively simple. You need to add an *if...else* statement to your code, which uses input from the remote control in a conditional statement. Remember that, in the previous unit, you used *if...else* statements to program the XBot to respond to input from infrared sensors. (For more information on *if... else* statements, see the *Quick Reference: Programming* document.)

**1. Using the Save As command, rename your code file `remote_forward.c`.**

**2. At the top of your code file, add your include statements, your define statements for the left and right motors, and the move function for your robot.**

**3. In your main function, add an *if...else* statement, as follows:**

```
void main(void)
{
    int button;
    button=0;
    right_motor_dir=0;
    left_motor_dir=0;

    while(1==1)
    {
        button=remote();
        if(button==144)
        {
            move(20,forward);
        }
        else
        {
            move(10,stop);
        }
    }
}
```

**4. Compile and test your new code.**

## **Challenge 2: Control the Robot Adding More Controls**

With three more conditional statements, you can program the XBot to respond to other directional commands, backing up, turning right, and turning left.

**1. Using the Save As command, rename your code file `remote_robot.c`.**

## 2. In your main function, add more conditional statements, as follows:

```
void main(void)
{
    int button;
    button=0;
    right_motor_dir=0;
    left_motor_dir=0;

    while(1==1)
    {
        button=remote();
        if(button==144)
        {
            move(20,forward);
        }
        else if(button==2192)
        {
            move(20,reverse);
        }
        else if(button==1168)
        {
            move(20,spin_right);
        }
        else if(button==3216)
        {
            move(20,spin_left);
        }
        else
        {
            move(10,stop);
        }
    }
}
```

## 3. Compile and test your new code.

## Using DC Motors

With a few modifications, your robot can be set up to be run on DC motors, instead of the servo motors included with the Mobile Robot Base Kit. In the field of hobby robotics, the term "DC" is often used to distinguish these motors from servo motors, but in fact, servo motors are just another type of DC motor, with internal circuitry that enables precise control of the motor's axle. The term DC motor applies to any motor that draws direct current from a battery, rather than the alternating current that drives the motors in household appliances, such as ceiling fans. Figure 1 shows a DC motor.



Figure 1. DC motor.

### Challenge 1: Mount the Motors



The first challenge is to mount the DC motors on your robot chassis. Each motor requires a custom mounting bracket, made of PVC foamboard, which affixes the motor to your stock or custom robot frame.

### Challenge 1: Mount the Motors

#### Collecting Your Components

In order to mount the DC motors, you will need the following components, shown in Figure 2:

Part	Quantity	Description
A	2	DC motors (BaneBot)
B	1	PVC foamboard (3mm sheet)
C	8	Machine screws (1/2" 4-40)
D	4	Machine screws (3/8" 4-40)
E	8	Machine screw nuts (4-40)



Figure 2. Components for mounting DC motors.

## **Challenge 1: Mount the Motors**

### **Mounting Motors on the Stock Frame**

Some chassis have pre-drilled holes that accept the BaneBot motors. These holes, located just forward of the servo mounting area, make it very easy to install the DC motors. Simply secure each motor in place, using 4-40 machine screws.

If your chassis does not have the mounting holes, you must create a mounting bracket, which attaches to the outside of the chassis, covering the rectangular openings for the servo motors. A printable template for this bracket can be found [here](#).

1. If you have already built a robot with the stock frame, remove the servo motors and their associated electronic components (the six-prong bent connector and red jump wires connecting the motors to Ports B0 and B3).
2. Cut two rectangular pieces of PVC foamboard, each measuring 1.5 inches by 2.5 inches.
3. Using the robot chassis or the printed template as a guide, mark the position of the four servo mounting screw holes on each rectangular piece with the tip of a screwdriver.
4. On each piece, drill holes for the 4-40 screws at each of the four marked positions and check to make sure these align with the holes on the

chassis.

5. Measuring from corner to corner, determine the precise center point of each mounting bracket and mark this position with the tip of a screwdriver.

6. In the center of each rectangle, drill a hole; at least 11/64-inch in diameter; to accommodate the axle of each DC motor.

7. On either side of the axle hole, carefully mark the position of the holes for the motor mounting screws and drill holes there for two 4-40 screws. Each bracket should look like the one shown in Figure 3.



Figure 3. Motor mounting bracket for stock frame.

8. Use 1/2-inch screws to secure each bracket to the outside of the chassis, and then use 3/8-inch screws to secure the motors to the brackets.

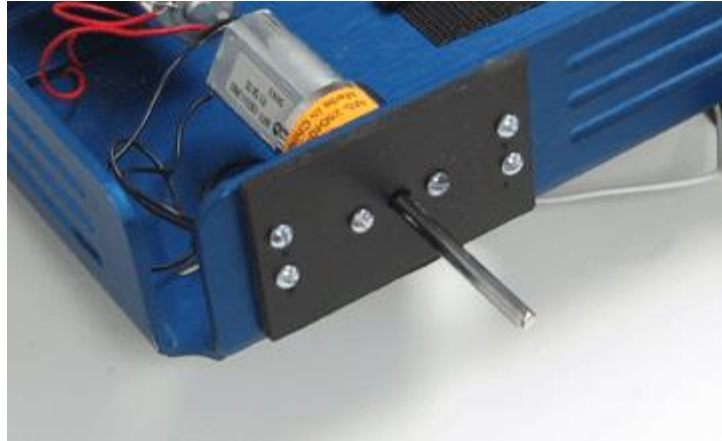


Figure 4. Motor mounted on stock frame.

### **Challenge 1: Mount the Motors**

#### **Mounting Motors on the Custom Frame**

The mounting bracket for the custom frame attaches to the main platform, using the same two angle brackets used to secure the servo motor bracket.



1. If you have already built a custom robot, remove the servo motor mounting brackets, the servo motors, and their associated electronic components (the six-prong bent connector and the red jump wires connecting the motors to Ports B0 and B3). Leave the angle brackets in place on the main platform.
2. Cut two rectangular pieces of PVC foam board, each measuring 1.5 inches by 2.5 inches.
3. Using the servo mounting brackets or the printed template as a guide, mark the position of the two angle bracket mounting screw holes on each rectangular piece.
4. On each piece, drill holes for the 4-40 screws at each of the two marked positions and check to make sure these align with the angle brackets.
5. Measuring from corner to corner, determine the precise center point of each mounting bracket and mark this position with the tip of a screwdriver.
6. In the center of each rectangle, drill a hole; at least 11/64-inch in diameter; to accommodate the axle of each DC motor.
7. On either side of the axle hole, carefully mark the position of the holes for the motor mounting screws and drill holes there for two 4-40 screws. Each bracket should look like the one shown in Figure 5.



Figure 5. Motor mounting bracket for custom frame.

8. Use two 1/2-inch screws to secure each bracket to the angle brackets on the main platform, and then use 3/8-inch screws to secure the motors to the mounting brackets, as shown in Figure 6.



Figure 6. Motor mounted on custom frame.

## Challenge 2: Wire the Motors



The next step is to establish electrical connections between the DC motors and the microcontroller. Since the DC motors can draw more current than can safely pass through the microcontroller, an additional chip, called an integrated circuit (IC) or "motor driver," is needed on the breadboard.

## Challenge 2: Wire the Motors Collecting Your Components

In order to wire the DC motors, you will need the following components, shown in Figure 7:

Part	Quantity	Description
A	1	Integrated circuit (SN754410NE motor driver)
B	1	Capacitor (100 uF)
C	4	Flexible insulated wire (8" lengths, stripped at both ends)
D	6	Jump wires (3 orange and 3 red)
E	6	Flexible jump wires



**Figure 7. Components for wiring the DC motors.**

## **Challenge 2: Wire the Motors About the IC and Capacitor**

The DC motors in the expansion pack can draw up to 1 amp of electric current; about 50 to 1,000 times the amount of current carried in the microcontroller's logic circuitry (1 to 20 milliamps). Consequently, an additional integrated circuit is needed to interface between the logic circuitry of the microcontroller and the DC motor circuitry. The IC looks like a smaller version of the microcontroller, with only 16 pins, instead of 40 pins.

Because the DC motors draw so much current, they can cause fluctuations in the voltage difference between power and ground. If left unchecked, these voltage fluctuations can disrupt the microcontroller's logic circuitry, which requires power to be 5 volts and ground to be 0 volts to function properly. By putting a capacitor from power to ground, these voltage irregularities are kept to a minimum. Remember that a capacitor is like a surge protector, storing current spikes and then releasing them to prevent voltage fluctuations.

## **Challenge 2: Wire the Motors Adding the IC and Capacitor**

Figure 8 shows a diagram of the 16 pins on the motor driver chip. You do not need to understand all of the pin labels right now. Just note that pin 1 is at the end of the chip marked by a small semi-circular notch. Like the microcontroller's capacitor, the capacitor needed for the DC motors has a specific polarity; one ground pin and one power pin. The ground pin is clearly marked with a minus (-) sign.

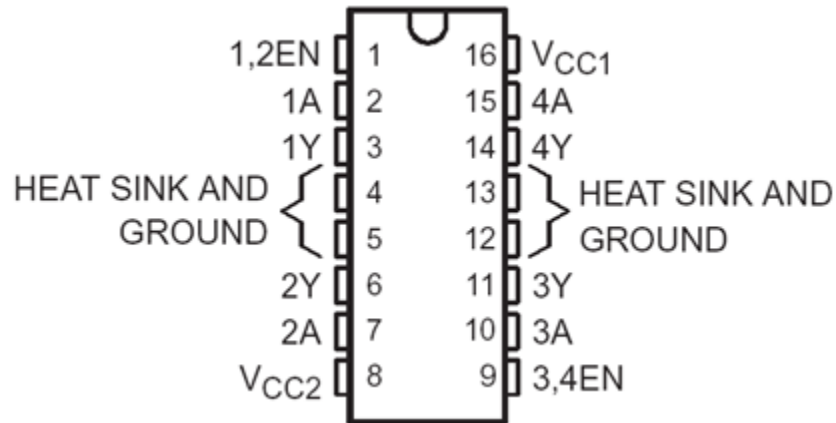


Figure 8. Pin map for the motor controller IC.

1. Insert the IC into the board, so that it straddles the center groove, just like the microcontroller. Pin 1 should go in hole E36.
2. Connect pins 4 and 5 to ground, using two yellow jump wires from A39 and A40 to ground.
3. Connect pin 8 to power, using an orange jump wire from hole A43 to power.
4. Connect pins 12 and 13 to ground, using two orange jump wires from holes J39 and J40 to ground.
5. Connect pin 16 to power, with a yellow jump wire from hole J36 to power, as shown in Figure 9.

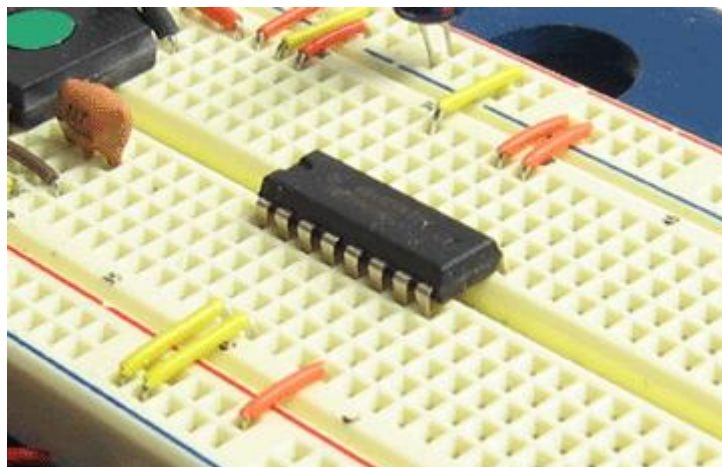


Figure 9. Motor driver IC with power and ground connections.

6. Add the capacitor, inserting the pin marked with a minus sign (-) into a

ground hole near hole J35, and the other pin into an adjacent power hole, as shown in Figure 10.

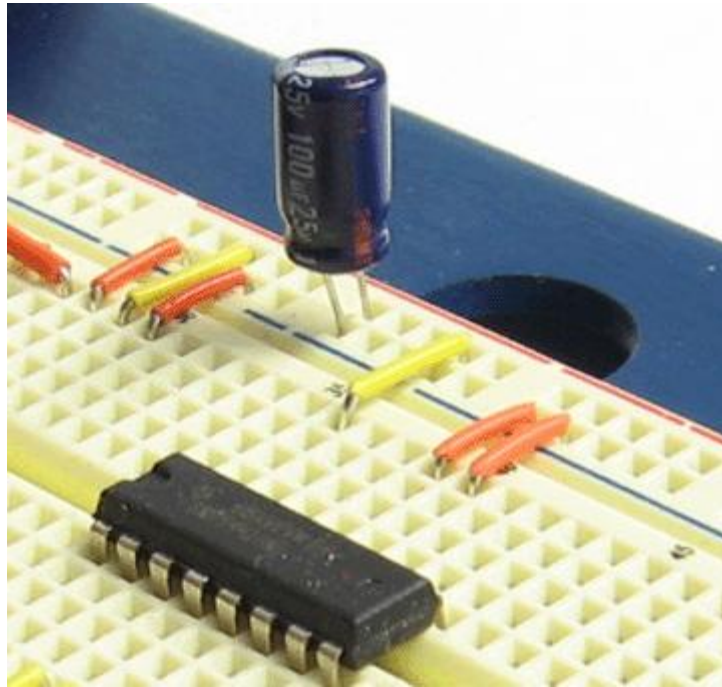


Figure 10. Capacitor in power and ground holes.

## **Challenge 2: Wire the Motors**

### **Connecting the Motors to the IC**

Flexible insulated wire is used to connect the motors to the motor controller IC. This type of wire is more rigid than the flexible jump wires included in the Starter Kit, and it is longer than the kit's pre-bent jump wires. The flexible insulated wire can be connected to the terminals on the motor with a gentle twist, or soldered in place for greater stability. In either case, be very careful not to put too much pressure on the motor terminals, as they may tear or break off.

**1. At each of the four motor terminals, insert a stripped end of the flexible insulated wire into the small hole, and gently twist the wire back on itself to secure it in place, as shown in Figure 11.**

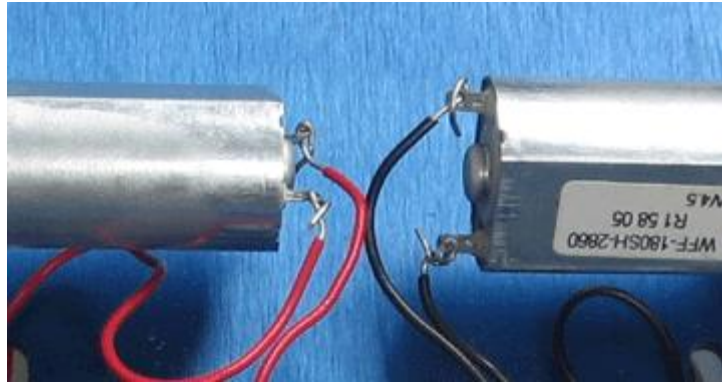


Figure 11. Wire leads connected to DC motor terminals.

2. Connect the wires from the right motor to pins 3 and 6 on the motor controller, by inserting the free ends into holes D38 and D41. 3. Connect the wires from the left motor to pins 11 and 14 on the motor controller, by inserting the free ends into holes G38 and G41.

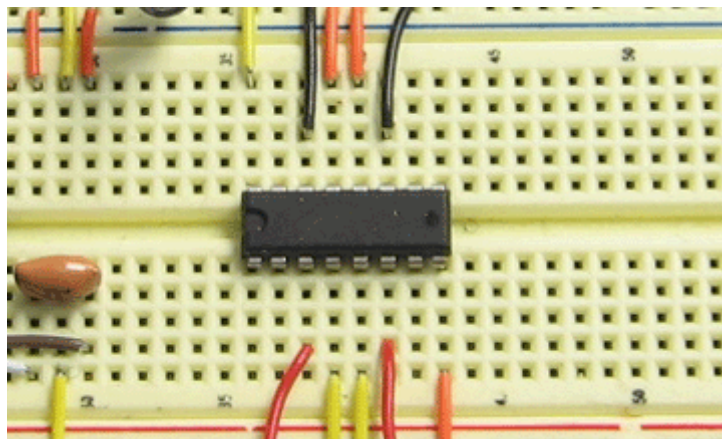


Figure 12. Motor leads inserted into board.

## Challenge 2: Wire the Motors

### Connecting the IC to the Microcontroller

The connections between the IC and the microcontroller enable you to control the activity, speed, and direction of each DC motor. These connections may be made with flexible jump wires, as shown in Figure 13, or - for greater stability - with carefully routed pre-bent jump wires.

1. Connect pin 1 to Port C2.
2. Connect pin 2 to Port D0.
3. Connect pin 7 to Port C3.



4. Connect pin 9 to Port C1.
5. Connect pin 10 to Port D2.
6. Connect pin 15 to Port D3.

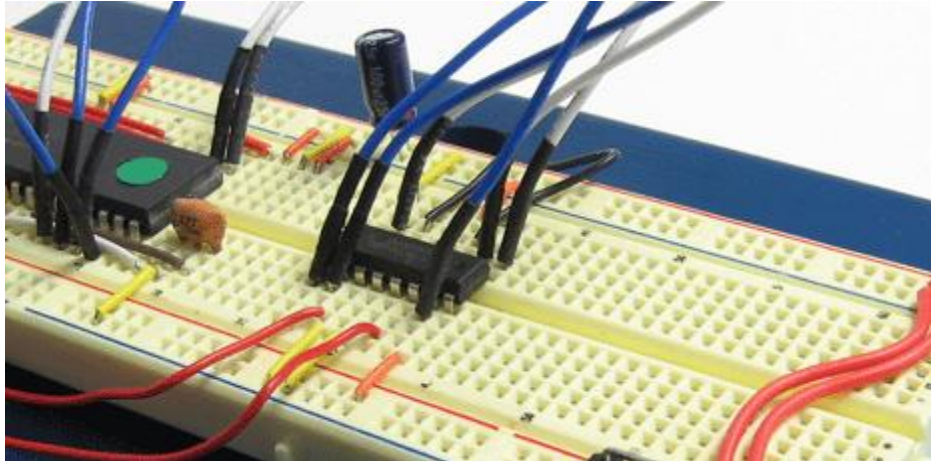


Figure 13. Motor driver connected to microcontroller.

### Challenge 3: Attach the Wheels



wheel.

Attaching wheels to the axles of the DC motors is an interesting challenge. Each axle is a smooth metal shaft, with a flat face on one side. In order to function properly, the wheel must be secured so that it won't slide off the axle and the axle won't turn freely inside the

### Challenge 3: Attach the Wheels Collecting Your Components

In order to attach wheels to the DC motors, you will need the following components, shown in Figure 14:

Part	Quantity	Description
A	1	PVC foamboard (6mm sheet)
B	10	Machine screws (5/8" 4-40)
C	10	Machine screws nuts (4-40)





Figure 14. Components for attaching wheels to motors.

### Challenge 3: Attach the Wheels About DC Motor Wheels

Wheels are typically attached to DC motor axles with hubs, which fit snugly on the axles and have threaded holes for securing the wheels. In some cases, the hubs have set screws, which prevent the hubs from sliding off the axle or slipping. Figure 15 shows two hubs with their set screws and wheel retaining screws.



### Challenge 3: Attach the Wheels Making the Hub

A simple version of a set-screw hub can be made with a square piece of PVC foamboard. The 6mm foamboard is thick enough that a narrow diameter hole can be drilled into its edge. This way, the set screw can be inserted and driven perpendicular to the axle.

1. Cut two square pieces of PVC foamboard, each measuring 1-inch by 1-inch.
2. Measuring from corner to corner, determine the precise center point of each square and mark this position with the tip of a screwdriver.

3. In the center of each square, drill a hole; at least 11/64-inch in diameter; to accommodate the axle of each DC motor, as shown in Figure 16.

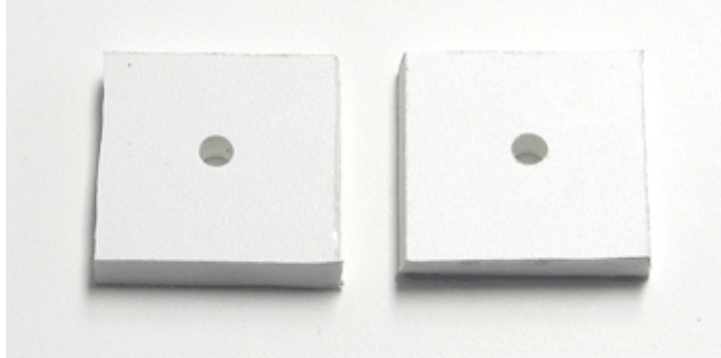


Figure 16. Square hubs with center holes drilled.

4. Using a 3/32-inch drill bit, drill a hole in the edge of the hub; perpendicular to the axle hole; so that the tip of the drill bit connects with the center hole, as shown in Figure 17.



Figure 17. Drilling the set screw hole.

5. Drive a 5/8-inch 4-40 machine screw into the new hole, checking to see that it enters the center hole, as shown in Figure 18.



Figure 18. Hub with set screw.

### Challenge 3: Attach the Wheels Attaching the Wheels

With your hub built, attaching the hubs to the wheels is relatively straightforward. You will have to tailor the following procedure to the specific style of wheel that you have. If you are working with wheels that press fit onto the servo motors, be careful not to damage any of this original linkage.

1. Drill two or more additional holes through each hub and secure the hubs to the wheels with 4-40 machine screws, as shown in Figure 19.



Figure 19. Wheel with hub attached.

2. Position the hub and wheel on the DC motor axle and snug up the set screw to hold the wheel in place, as shown in Figure 20.



Figure 20. Wheel attached to DC motor.

### Challenge 4: Control the Motors



Challenge 4 is a software challenge; writing code to drive the DC motors. If you are used to writing code for servo motors, driving DC motors takes some getting used to, but it is actually much simpler

than working with servo motors. With one or two lines of code, you can turn each motor on or off, reverse its direction, or cause it come to a rigid stop.

## Challenge 4: Control the Motors

### About the Motor Driver IC

The following table shows the port values associated with different behaviors for the right and left DC motor. To enable the motors, port C2 (left motor) must be set equal to 1, and port C1 must be set equal to 1 (right motor). The values of ports D0 and C3 control the motion of the left motor, while the values of ports D2 and D3 control the motion of the right motor. Note that, as with the servo motors, the definition of "forward" and "reverse" will depend on how the motor's orientation with respect to the chassis. If you want to invert these directions for a motor, simply exchange the position of the Drive A and Drive B wires (e.g. for the left motor, move the Drive A wire to port C3 and the Drive B wire to port D0).

LEFT MOTOR				RIGHT MOTOR			
Enable (Port C2)	Drive A (Port D0)	Drive B (Port C3)	Motor Action	Enable (Port C1)	Drive A (Port D2)	Drive B (Port D3)	Motor Action
1	0	0	Stop	1	0	0	Stop
1	1	0	Forward	1	1	0	Forward
1	0	1	Reverse	1	0	1	Reverse
1	1	1	Stop	1	1	1	Stop
0	-	-	Off	0	-	-	Off

## Challenge 4: Control the Motors

### Driving a Motor Directly

Unlike a servo motor, you can drive a DC motor directly from the batteries. To test this, try pulling the motor leads from the IC and inserting them directly into power and ground.

1. Remove the motor wires from one side of the IC.
2. Insert one of the wires into any power hole.
3. Insert the other wire into any ground hole.
4. Try reversing switching the wires, and observe the motor.

5. Return the wires to their original positions.

## Challenge 4: Control the Motors Driving One Motor

1. Create a new code file and save it as dcmotor.c.

2. Enter the following code:

```
#include "mxapi.h"
void main(void)
{
    TRISC2 = 0; //Set up Port C2 as an output
    TRISD0 = 0; //Set up Port D0 as an output
    TRISC3 = 0; //Set up Port C3 as an output
    RC2 = 1; //Set Port C2 high (enable left motor)
    RD0 = 1; //Set Port D0 high (forward motion)
    RC3 = 0; //Set Port C3 low (forward motion)
    end();    //End program
}
```

3. Compile and test your new code.

## Challenge 4: Control the Motors Reversing the Motor

1. Rename your code file dcmotorreverse.c.

2. Enter the following code:

```
#include "mxapi.h"
void main(void)
{
    TRISC2 = 0; //Set up Port C2 as an output
    TRISD0 = 0; //Set up Port D0 as an output
    TRISC3 = 0; //Set up Port C3 as an output
    RC2 = 1; //Set Port C2 high (enable left motor)
    RD0 = 1; //Set Port D0 low (reverse motion)
    RC3 = 0; //Set Port C3 high (reverse motion)
    end();    //End program
}
```

3. Compile and test your new code.

## Challenge 4: Control the Motors

### Driving Both Motors

1. Rename your code file `bothdcmotors.c`.

2. Enter the following code:

```
#include "mxapi.h"
void main(void)
{
    TRISC2 = 0;
    TRISD0 = 0;
    TRISC3 = 0;

    TRISC1 = 0; //Set up Port C1 as an output
    TRISD2 = 0; //Set up Port D2 as an output
    TRISD3 = 0; //Set up Port D3 as an output

    RC2 = 1;
    RD0 = 0;
    RC3 = 1;

    RC1 = 1; //Set Port C1 high (enable right motor)
    RD2 = 0; //Set Port D2 low (reverse motion)
    RD3 = 1; //Set Port D3 high (reverse motion)

    end();      //End program
}
```

3. Compile and test your new code.

## Challenge 4: Control the Motors

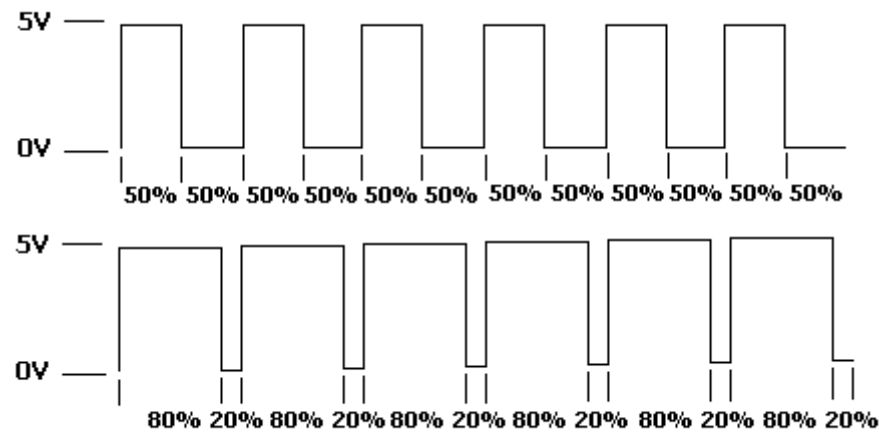
### Controlling the Motors' Speed

The speed of the DC motors can be controlled with the microcontroller's internal pulse width modulator (PWM). The PWM is built-in a feature of the microcontroller that generates a precisely timed series of high and low voltage pulses, like those used to drive the servo motors. The PWM outputs the pulses on two ports: Port C2 (channel 1) and Port C1 (channel 2). When these ports are connected to the enable pins on the motor controller, the PWM turns the motors on and off very rapidly, thereby modulating their speed.

A function called `pwm_init` is used to activate and control the PWM. The `pwm_init` function takes three arguments. The first argument sets the frequency of the pulse series; i.e. how many times per second the PWM cycles between high and

low voltage. The second two arguments set the "duty cycle" for each of the two channels; i.e. in each cycle of high and low voltage, what percentage of the time the voltage is high. The greater the percentage, the faster the motor's speed; the smaller the percentage, the slower the motor's speed.

Figure 21 shows a two different duty cycles at the same frequency; one with a duty cycle of 50% high and 50% low, and the other with a duty cycle of 80% high and 20% low.



**Figure 21. PWM at two different duty cycles with the same frequency.**

**1. Rename your code file pwminit.c.**

**2. In your main function, remove the lines of code that set up ports C1 and C2 as outputs, and the lines that set the value of these ports equal to 1, as shown below. Add a line to initialize the chip's pulse width modulator, at a frequency of 10,000 hertz, with a duty cycle of 100% on both channels.**

```
#include "mxapi.h"
void main(void)
{

    TRISD0 = 0;
    TRISC3 = 0;

    TRISD2 = 0;
    TRISD3 = 0;

    pwm_init(10000, 100, 100); //Initialize PWM at 10KH

    RD0 = 0;
    RC3 = 1;
```



```

RD2 = 0;
RD3 = 1;

end();
}

```

### 3. Compile and test your new code.

## Challenge 4: Control the Motors Changing the Duty Cycle

The `pwm_init` function sets up the initial duty cycle for each channel; this function should be used only once in your main function. If you want to change the speed of a motor later on, you can use a second function, called `pwm_duty`. The `pwm_duty` function takes two arguments: 1) the duty cycle for the motor being controlled, ranging from 0 (off) to 100 (full speed); and 2) a number indicating which motor you want to control, with 1 corresponding to motor 1 and 2 corresponding to motor 2.

#### 1. Rename your code file `pwm_duty.c.c`.

#### 2. In your main function, use the `pwm_duty` function to control the speed of the motors, as shown below:

```

#include "mxapi.h"
void main(void)
{
    TRISD0 = 0;
    TRISC3 = 0;
    TRISD2 = 0;
    TRISD3 = 0;

    pwm_init(10000, 100, 100);

    RD0 = 0;
    RC3 = 1;
    RD2 = 0;
    RD3 = 1;

    while(1==1)
    {
        pwm_duty(50,1);    //Set cycle 50-50
        pwm_duty(50,2);    //Set cycle 50-50
        delay_ms(1000);    //Wait 1000 ms
        pwm_duty(75,1);    //Set cycle 75-75
        pwm_duty(75,2);    //Set cycle 75-75
    }
}

```

```

    delay_ms(1000);        //Wait 1000 ms
    pwm_duty(100,1);       //Set cycle 100-100
    pwm_duty(100,2);       //Set cycle 100-100
    delay_ms(1000);        //Wait 1000 ms
}
}

```

### 3. Compile and test your new code.

## Challenge 4: Control the Motors Using a Function to Control the Motors

With one function, you can controls the speed and the direction of each motor. The function will extend the capability of `pwm_duty`, taking three arguments: 1) a number indicating which motor you want to control, with 1 corresponding to motor 1 and 2 corresponding to motor 2; 2) the duty cycle for the PWM for the motor being controlled, ranging from 0 (off) to 100 (full speed); and 3) a number indicating the direction of the motor, with 1 corresponding to forward and 0 corresponding to reverse. In the code below, note the use of the tilde (~), which inverts the value of any variable. For example, if `x=1`, then `~x = 0`; if `x=0`, then `~x` equals 1. In the motor function, the tilde ensures that drive A and drive B are always opposite values.

### 1. Rename your code file `motorfunction.c.c`.

### 2. Modify your code file, as follows:

```

#include "mxapi.h"

void motor(char motor, char speed, char direction)
{
    if(motor == 1)
    {
        pwm_duty(speed, motor);
        RD0 = direction;
        RC3 = ~direction;
    }
    else if(motor == 2)
    {
        pwm_duty(speed, motor);
        RD2 = direction;
        RD3 = ~direction;
    }
}

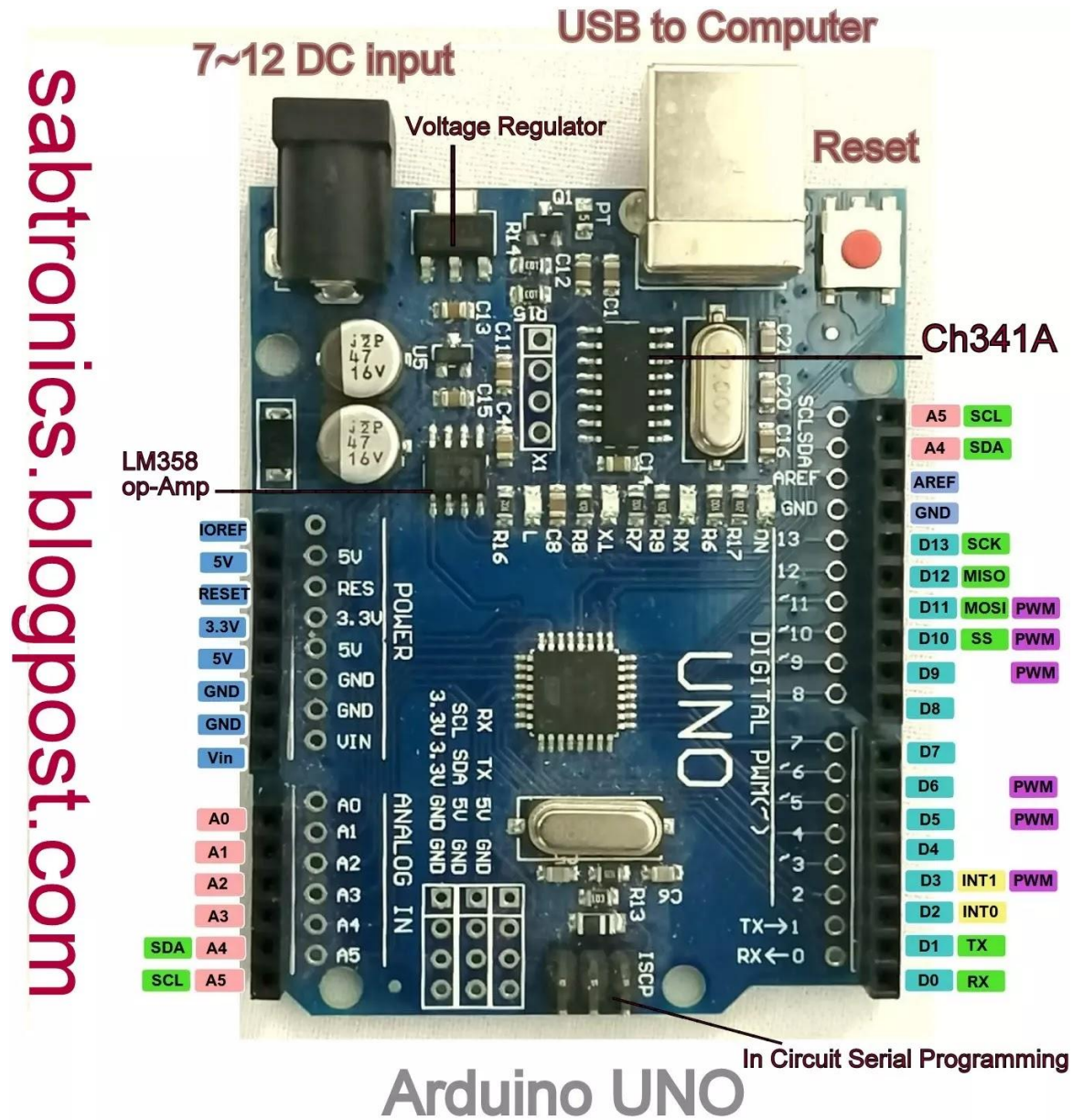
void main(void)

```

```
{  
    TRISD0 = 0;  
    TRISC3 = 0;  
    TRISD2 = 0;  
    TRISD3 = 0;  
  
    pwm_init(10000,100,100);  
    motor(1,100,0);  
    motor(2,100,1);  
    while(1==1);  
}
```

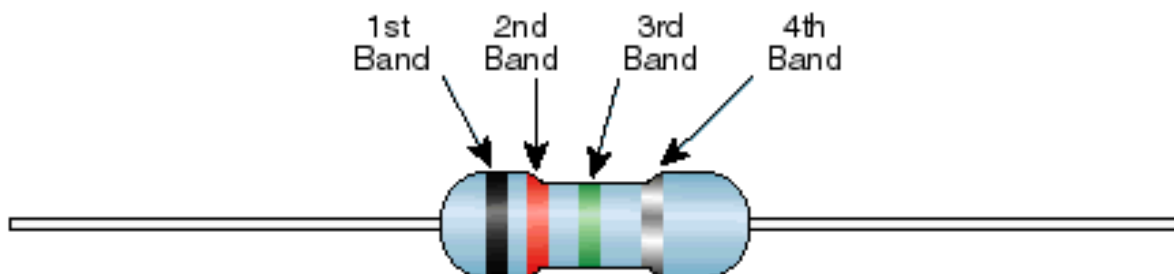
### 3. Compile and test your new code.

# Arduino Uno Reference Sheet



# Resistor Band Reference

**Standard EIA Color Code Table 4 Band:  $\pm 2\%$ ,  $\pm 5\%$ , and  $\pm 10\%$**



Color	1st Band (1st figure)	2nd Band (2nd figure)	3rd Band (multiplier)	4th Band (tolerance)
Black	0	0	$10^0$	
Brown	1	1	$10^1$	
Red	2	2	$10^2$	$\pm 2\%$
Orange	3	3	$10^3$	
Yellow	4	4	$10^4$	
Green	5	5	$10^5$	
Blue	6	6	$10^6$	
Violet	7	7	$10^7$	
Gray	8	8	$10^8$	
White	9	9	$10^9$	
Gold			$10^{-1}$	$\pm 5\%$
Silver			$10^{-2}$	$\pm 10\%$

Chart Provided By 