Making Robots With The Arduino

Part 4 – Getting Feedback With Sensors

By Gordon McComb

Robots need information about the world around them, or they just stumble about looking stupid. Just like us humans, a robot uses senses to know when it's run into something; when it's light or dark; when it's too hot or too cold; when it's about to fall over; or when it's found the way to the cheese at the center of a maze.

Senses require sensors. In the practice of robotics, the basic senses make do with the most basic of sensors: mechanical switches for detecting contact with objects, and photosensitive resistors and transistors for detecting the presence (or absence) of light. A robot can perform a remarkable amount of work with just the sense of touch and the gift of simple sight.

In this month's installment, you'll learn about interfacing switches and photosensors to the Arduino, along with how to use the information these sensors provide to interactively command a robot's motors. These are the fundamental building blocks of most any autonomous robot you build. Once you learn how to use these sensors to do your bidding, you can apply them in dozens of ways, for all kinds of robotic chores.

Arduino Robotics: What We've Covered So Far

This article builds upon previous installments in this series, which is all about the construction and use of the *ArdBot* (see **Figure 1**) – an inexpensive two-wheeled differentially-steered robot based on the popular Arduino Uno and compatible microcontrollers. If you'd like to follow along, be sure to check out the previous three episodes, so you're familiar with the plot and characters.

Part 1 introduced the ArdBot project, the Arduino, and basic programming fundamentals of this powerful controller.

Part 2 detailed the construction of the ArdBot, using common materials such as plastic or aircraft grade plywood.

Part 3 covered the Arduino in more depth, and examined the ins and outs of programming R/C servo motors with the Arduino.

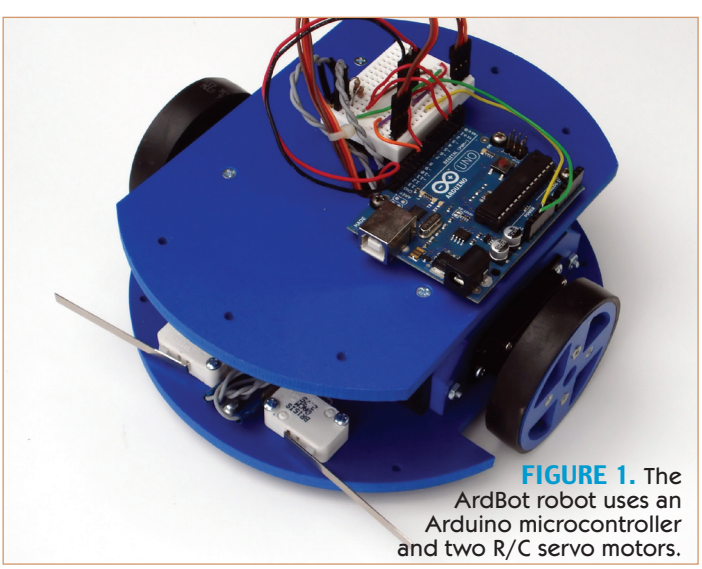


FIGURE 1. The ArdBot robot uses an Arduino microcontroller and two R/C servo motors.

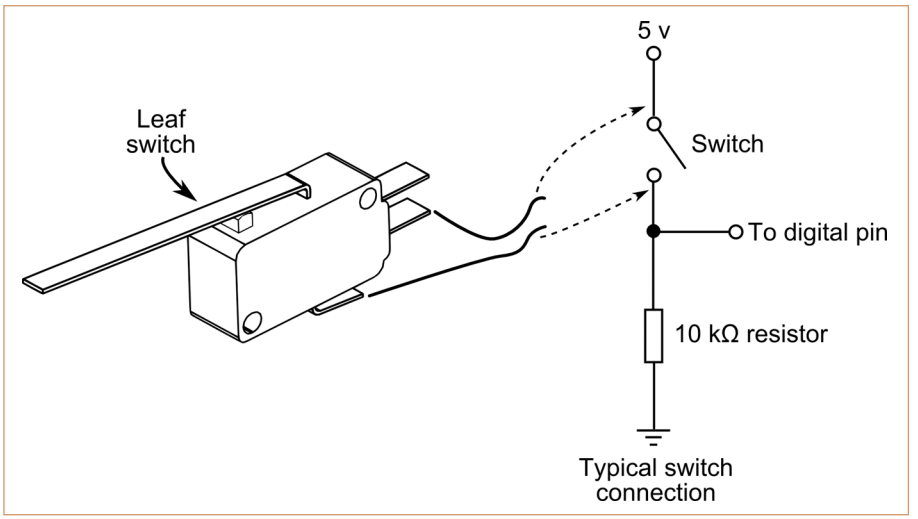


FIGURE 2. The leaf switch acts as a kind of cat's whiskers. It's connected to a digital I/O pin using a pull-down resistor.

What's covered here applies to most any robot that uses the Arduino microcontroller, and that runs on two motors and rolls on wheels or tracks. You're free to adapt the techniques and programming code to whatever bots you're constructing. The ArdBot is an expandable platform, but it's also a concept that represents the typical desktop-size robot.

The subject of sensors is pretty involved. There's no way to cover all the interesting things in just one article.

So, next month you'll learn about other kinds of inexpensive sensors you can use with your ArdBot (or other robot).

Getting in Touch With Your Robot

Sensors — whether in humans or in robots — are designed to produce a reaction. What that reaction is depends on the nature of the sensation. Type and quantity matter. We interpret the feeling of a soft summer breeze as a good sensation. Increase the amount of air pressure to hurricane force and decrease the temperature to something below freezing, and suddenly the same senses produce a highly negative reaction.

Touch — also called *tactile feedback* — is a primitive reactive sense. The robot determines its environment by making physical contact; this contact is registered through a variety of touch sensors. What happens when contact is made is entirely determined by the programming you apply within your robot.

Most often, a collision with an object is a cause for alarm. So, the reaction of the robot is to stop what it's doing, and back away from the condition. In other cases, contact can mean your robot has found its home base, or that it's located an enemy bot and is about to pound the living batteries out of it.

The lowly mechanical switch is the most common — and most simple — form of tactile (touch) feedback mechanism. Just about any momentary, spring-loaded switch will do. When the robot makes contact, the switch closes, completing a circuit.

I like to use *leaf* (or *lever*) switches (see **Figure 2**) because they function a lot like a cat's whiskers. These things are sometimes referred to as a microswitch — after a popular brand name — but I'll call them leaf switches to avoid confusion. Regardless of make or model, most are easy to mount, and come with plastic or metal strips of different lengths that enhance the sensitivity of the switch.

You can enlarge the contact area of the leaf by gluing or soldering bigger pieces of plastic or metal to it. For example, you can cut up some stiff music wire (available at hardware stores) or a cheap wire clothes hanger, and bend it to some fancy shape.

Making Reusable Sensor Components

The more you experiment with robotics, the more you'll want to build a drawer-full of reusable parts that easily plug into your projects. Sensors especially.

With just a bit of wire, some heat shrink tubing, and a length of snap-off male header pins you can build modular sensors that can be shared between projects. The pins easily plug into a solderless breadboard. **Figure A** shows a photoresistor attached to an eight inch length of wire which is terminated into a three-prong male header (only two pins are used; the third is cut off).

First, start by cutting some 22 or 24 gauge insulated stranded conductor wire to the desired length. Don't be stingy with the wire, but don't make it so long the extra gets in the way. Give yourself an additional inch or two so you can twist the leads together to make a nice pigtail.

Strip about 1/4" insulation off both ends, and use your soldering pencil to pre-tin the wire. Do the same for the leads on the photocell and the header pins. Exercise care when soldering to the photocell leads, as excessive heat can damage the component. After tinning is complete, carefully tack-solder the wires to the leads or pins.

I like to use heat shrink tubing to finish off the soldered ends. The tubing makes for a more professional look, plus it helps prevent short circuits. When applied properly, it acts as a strain relief to help keep the wires from pulling apart from their joints. Buy a small assorted package of tubing, and use the smallest diameter for the best fit.

Header pins have a 0.100" spacing which is a fairly tight space for all but the most seasoned solder pros. So, you'll probably want to snap off a set of three or four pins, and remove the center pins to make extra room for your solder joints.

FIGURE A. Use male header pins to prepare connectors for easy interchange with your various projects. The connectors plug into a solderless breadboard.

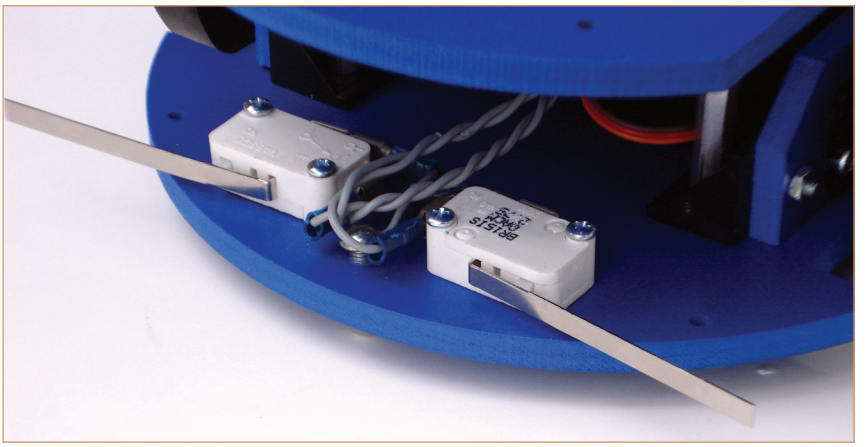


FIGURE 3. A pair of leaf switches on the ArdBot. You can attach things to the leaf of each switch to enlarge its contact area.

Solder the end(s) to the leaf. Or, you can use thin pieces of wood, plastic, or metal. Just be sure the weight of the extension doesn't accidentally activate the switch. You don't want false alarms.

The switch may be directly connected to a motor or, more commonly, it may be connected to a microcontroller. A typical wiring diagram for the switch is shown in **Figure 2**. The 10 kΩ pull-down resistor is there to provide a consistent digital LOW (0 volts) output for the switch when there is no contact. When contact is made, the switch closes, and the output of the switch goes HIGH — usually five volts, as shown here.

Using Leaf Switches as Bumpers

Two standard leaf switches mounted to the front of your ArdBot let it detect when it's hit something. With the switches situated to the sides, your bot can determine if the object is on the left or on the right, and then steer around it.

Figure 3 shows a pair of leaf switches mounted like bumpers to the front of the ArdBot. Switches like these are available at many online electronics outlets, and are common as surplus. I bought these at All Electronics (a *SERVO Magazine* advertiser) for $1.60 a pop.

I haven't augmented the switches with a larger contact area, as I'm more interested in demonstrating the concepts involved. Use your creativity in enhancing the switches to provide the level of sense detection you want. For example, right off you can see that the robot is "blind" to small objects directly between the switches. You can deal with this either by enlarging the contact area, or (my choice) using another form of "sense" to avoid collision in the first place.

To mount each switch, find two suitable holes in the base of your robot, or drill new ones. Most leaf switches have three connections: common, normally open (NO), and normally closed (NC). Wire the *common* and *NO* connections. If space is tight, break off the NC connection to make room.

Figure 4 shows the diagram for connecting the two switches to digital pins D2 and D3 of the Arduino. **Figure 5** shows the same circuit, but in breadboard view. Use the upper half of the ArdBot's 170 tie point solderless breadboard. The bottom half is already in use by the servo wiring for the ArdBot (see Part 2 of this series).

On my prototype, I made connectors for the switches by soldering the two wires to pins of a breakaway male header. With these, you break off the number of pins you

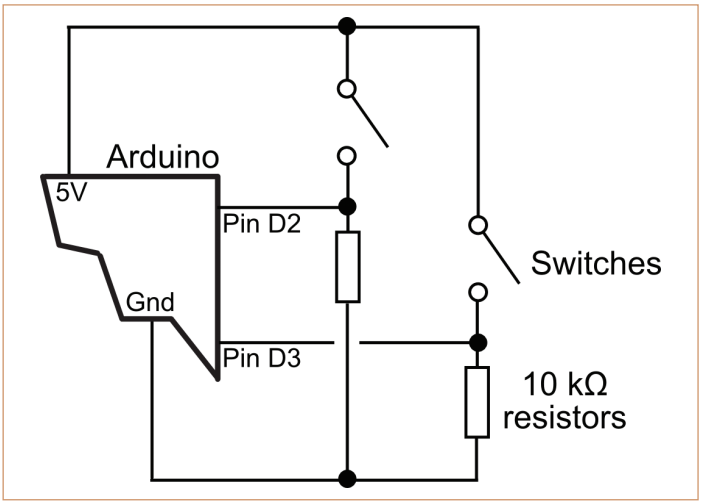


FIGURE 4. Schematic view of connecting two bumper switches to the Arduino microcontroller.

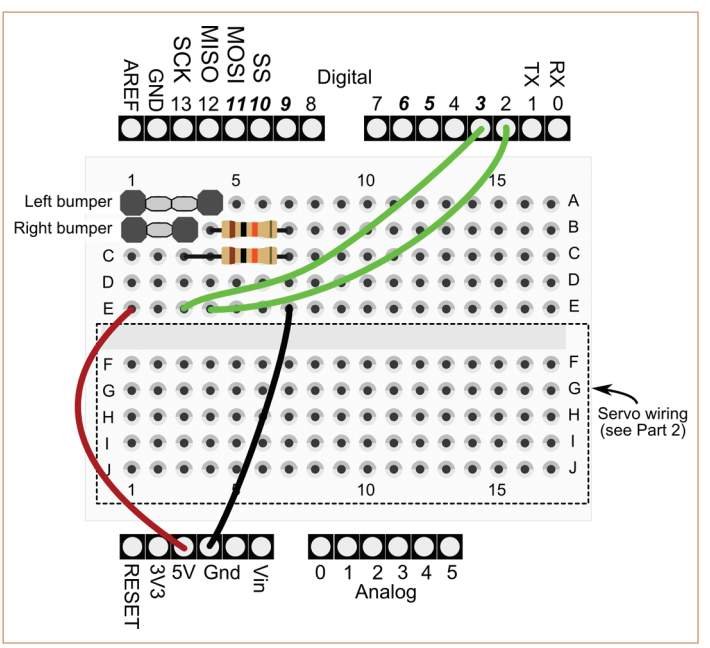


FIGURE 5. Breadboard view of connecting the bumper switches. Note that the bottom half of the solderless breadboard is already in use, wired for the two servo motors. (See Part 2 of this series for details.)

```
/*
 ArdBot bumper switch demo
 Requires Arduino IDE version 0017
  or later (0019 or later preferred)
*/

#include <Servo.h>

const int ledPin =  13;  // Built-in LED
const int bumpLeft = 2;  // Left bumper pin 2
const int bumpRight = 3; // Left bumper pin 3
int pbLeft = 0;           // Var for left bump
int pbRight = 0;          // Var for left bump
Servo servoLeft;          // Define left servo
Servo servoRight;         // Define right servo

void setup() {
  servoLeft.attach(10);  // Left servo pin D10
  servoRight.attach(9);  // Right servo pin D9
  // Set pin modes
  pinMode(bumpLeft, INPUT);
  pinMode(bumpRight, INPUT);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  forward();                // Start forward
  // Test bumper switches
  pbLeft = digitalRead(bumpLeft);
  pbRight = digitalRead(bumpRight);

  // Show LED indicator
  showLED();

  // If left bumper hit
  if (pbLeft == HIGH) {
   reverse();
   delay(500);
   turnRight();
   delay(1500);
  }

  // If right bumper hit
  if (pbRight == HIGH) {
   reverse();
   delay(500);
   turnLeft();
   delay(1500);
  }
}

// Motion routines
void forward() {
  servoLeft.write(180);
  servoRight.write(0);
}

void reverse() {
  servoLeft.write(0);
  servoRight.write(180);
}

void turnRight() {
  servoLeft.write(180);
  servoRight.write(180);
}

void turnLeft() {
  servoLeft.write(0);
  servoRight.write(0);
}

void stopRobot() {
  servoLeft.write(90);
  servoRight.write(90);
}

void showLED() {
  // Show LED if a bumper is hit
  if (pbRight == HIGH || pbLeft == HIGH) {
    // Turn LED on
    digitalWrite(ledPin, HIGH);
  }
  else {
    // Turn LED off
    digitalWrite(ledPin, LOW);
  }
}
```

Listing 1 – bumper.pde.

want to use. I cut one connector to three pins wide, removing the middle pin; I soldered the wires from the switch to the outer two pins. For the other connector, I cut it to four pins wide, removing the middle two pins. You can see in **Figure 5** how the two connectors plug into the breadboard.

Important! Make sure all the wires and other components are firmly seated into their breadboard tie-point sockets. Loose connections are the second most common cause of problems when using a solderless breadboard — the most common is plugging the wires into the wrong tie points!

Listing 1 shows the demo program *bumper.pde*. The ArdBot sets off going forward until one of its front bumper switches makes contact with an object. The moment the switch closes, the robot quickly reverses direction, then turns in the opposite direction of the obstacle. Time delays are specified in milliseconds. The robot backs up for 500 milliseconds (half a second). It then turns — actually spins — to the right or left for 1,500 milliseconds (1.5 seconds).

You can experiment with other delay settings, depending on how fast your robot travels. With faster servo motors, you can use a shorter delay. The idea is to spin the robot about one-quarter to one-half turn, so it moves away from the obstacle.

Note that "left" and "right" (and "front" and "back") are somewhat objective in a robot like the ArdBot. Either end can be the front, so left and right is relative. In my prototype, I put the two leaf switches on the end that had more mounting space available. That end became the "front." The coding in *bumper.pde* reflects this design choice. If your robot seems to behave opposite to what it should, swap the values in the motion routines (forward, reverse, etc.). See Part 3 of this series for more details on what the servo commands do, and how they work.

Understanding the bumper.pde Sketch

As with all Arduino sketches, *bumper.pde* has three principle parts: declaration, setup() function, and loop() function.

The *declaration* area at the top of the sketch sets up the variables used throughout the program. It also prepares two objects of the servo class. As you read in Part 3 of *Making Robots with the Arduino*, the servo class is provided as a library that comes with the Arduino programming tools. You use it to control one or more R/C servos. The declaration also defines the two leaf switches as connected to digital pins D2 and D3, and that we'll be using the Arduino's built-in LED (internally connected to pin D13) as a visual indicator.

In the *setup()* function, the servos are defined as connected to digital pins D9 and D10. The pins used for the LED and two switches are set as outputs and input, respectively.

The main body of the sketch is the *loop()* function

which repeats indefinitely. It begins by activating the two servos to move the robot forward. The sketch then uses the *digitalRead* statement to store the current state of the two switches. The instantaneous value of the switches is kept in a pair of variables (*pbLeft* and *pbRight* — the *pb* for pushbutton). These variables are used elsewhere.

Of main interest in the loop() function are the two *if* statements. Here's the one that tests the left leaf switch:

```
if (pbLeft == HIGH)
 reverse();
 delay(500);
 turnRight();
 delay(1500);
}
```

pbLeft == HIGH checks to see if the contents of the *pbLeft* variable (set earlier based on the state of the left leaf switch) is HIGH. If it is, then the left switch is closed, and the robot has made contact with something. If it's LOW, then the switch is open, and the robot continues on its way.

B*umper.pde* also includes a number of user-defined functions. Most — like *forward()* and *reverse()* — relate to driving the servo motors. Another function, *showLED()*, toggles the LED on pin D13 of the Arduino on or off, depending on whether a switch is closed. Use this as a visual indicator that the programming code is working as it should.

Switch Triggers Using Polling or Interrupts

The programming in *bumper.pde* relies on what's known as *polling*: The sketch repeatedly checks the status of the two switches. If a switch is closed, its value goes from LOW to HIGH; when HIGH, the robot is commanded to steer to a new heading. The switches are checked — polled — many times each second.

Polling is an acceptable method when the sketch is relatively simple, and the demands on the Arduino are light. For code that is more processing intensive, there is a remote chance the controller will miss when a leaf switch has closed. It'll be busy doing something else in between polls and be unaware anything has happened.

In truth, you can have a fairly involved sketch and it will still detect 99 percent of all switch closures. The reason: The switch will likely be closed for what are very long periods of time to a microcontroller. For a microcontroller running at 16 MHz, even a brief 100 millisecond (one-tenth second) contact is like a lifetime, and so in all likelihood the switch closure will be registered.

Still, if you absolutely must ensure that even the most fleeting contact is registered, you might want to consider using *hardware interrupts* rather than polling. With an interrupt, special code is run if — and only when — a specific external event occurs. Because the main program

```
/*
 ArdBot interrupt bumper demo
 Requires Arduino IDE version 0017
  or later (0019 or later preferred)
*/

#include <Servo.h>

const int ledPin =  13;
const int bumpLeft = 2;
const int bumpRight = 3;
int pbLeft = 0;
int pbRight = 0;
Servo servoLeft;
Servo servoRight;

void setup() {
  servoLeft.attach(10);
  servoRight.attach(9);
  // Set pin modes
  pinMode(bumpLeft, INPUT);
  pinMode(bumpRight, INPUT);
  pinMode(ledPin, OUTPUT);

  // Set up interrupts
  attachInterrupt(0, hitLeft, RISING);
  attachInterrupt(1, hitRight, RISING);

}

void loop() {
  forward();     // Start forward
  showLED();     // Show LED indicator

  // If left bumper hit
  if (pbLeft == HIGH) {
   reverse();
   delay(500);
   turnRight();
   delay(1500);
   pbLeft = LOW;
  }

  // If right bumper hit
  if (pbRight == HIGH) {
   reverse();
   delay(500);
   turnLeft();
   delay(1500);
   pbRight = LOW;
  }
}

// Motion routines
void forward() {
  servoLeft.write(180);
  servoRight.write(0);
}

void reverse() {
  servoLeft.write(0);
  servoRight.write(180);
}

void turnRight() {
  servoLeft.write(180);
  servoRight.write(180);
}

void turnLeft() {
  servoLeft.write(0);
  servoRight.write(0);
}

void stopRobot() {
  servoLeft.write(90);
  servoRight.write(90);
}

void showLED() {
  // Show LED if a bumper is hit
  if (pbRight == HIGH || pbLeft == HIGH) {
    digitalWrite(ledPin, HIGH);
  }
  else {
    digitalWrite(ledPin, LOW);
  }
}

// Interrupt handlers
void hitLeft() {
  pbLeft = HIGH;
}

void hitRight() {
  pbRight = HIGH;
}
```

Listing 2 – interrupt.pde.

loop() doesn't have to continually check the state of the pins, it frees up the controller to do other things. Reaction time of an interrupt is measured in microsecond timing, even if the Arduino is busy doing something else. (Actually, this is not always true, depending on how other hardware on the controller is being used. But any additional delay is usually minimal.)

The Arduino Uno supports two hardware interrupts (the Arduino Mega supports six) internally connected within the Arduino to digital pins D2 and D3. These are the pins that the leaf switches are already connected to, so the only change needed is in the software.

See **Listing 2** for *interrupt.pde*. Here, the *bumper.pde* sketch has been revised to "listen" to a state change on both of the hardware interrupts with the statements:

```
attachInterrupt(0, hitLeft, RISING);
attachInterrupt(1, hitRight, RISING);
```

Note that the interrupts are referred to as *0* and *1*. These correspond to pins D2 and D3, respectively. The labels *hitLeft* and *hitRight* are the functions that are called when the interrupt is triggered. Finally, *RISING* is a built-in constant that tells the Arduino to trigger the interrupt on a LOW-to-HIGH signal transition. This type of transition occurs when the switch closes.

Both *hitLeft* and *hitRight* set their corresponding "pb" variable to HIGH. The program then immediately exits the interrupt handler. The next time the Arduino repeats its *loop()*, it notices that a pushbutton is HIGH and performs the needed obstacle avoidance maneuver. (Note also that the pushbutton value is manually set back to LOW, in anticipation of the next bump.)

You might be asking why the code to control the servos isn't in the interrupt handlers. The reason is this: The *delay* statement — which is used to steer the robot around an obstacle — is disabled while in an interrupt. In any case, it's usually best not to place time-intensive functionality within interrupt handlers.

To Let Bounce or Debounce?

In a perfect world, mechanical switches would produce clean, reliable digital signals for our microcontrollers. We don't live in a perfect world; instead, we must suffer something called *switch bounce*. Instead of a nice LOW-to-HIGH digital pulse when a switch closes, what we get are five, 10, maybe even dozens of irregular glitches, all caused as the metal contacts in the switch settle into position. All this happens very quickly; usually in just a few milliseconds.

For some applications, it's absolutely necessary to *debounce* the output of a switch. In debouncing, all the glitches are removed, providing the microcontroller with that single sweet pulse we want. You can debounce with some extra hardware: a capacitor and resistor create an RC network that acts to delay the rise and fall of the switch signal — that effectively removes the glitches. You can also do it in software, typically using delays so that the microcontroller ignores all but the first signal transition.

Both the *bumper.pde* and *interrupt.pde* examples don't directly use switch debounce. Software delays are already built into the code, plus R/C servos are slow creatures and don't react fast enough for bounce to be a problem. Servos are commanded in 20 millisecond "frames" — that is, their operation is updated once every 20 milliseconds. Turns out that's about the worst-case duration of bounce glitches from most switches. So, even with a switch bouncing along merrily, it has little or no effect on the operation of the servos.

Should you need to debounce your switch inputs, there's a separate class library you can download and use with the Arduino. The library is called *Bounce*, and it's available from the main Arduino language reference pages. There's also a *Debounce* code example that comes with the Arduino programming IDE.

Mounting Alternatives, More Switches

A few quick notes before moving on. So far, I've talked about the two switches in the front of the robot, situated right and left. Feel free to put switches anywhere you want. You might instead have a front and back switch, or a bunch of switches all around the periphery of the robot.

If you use more than two switches, you'll have to rely on polling, as there are only two pins that support hardware interrupts (when using the Arduino Uno). If you use more than four or five switches, you may want to use a

Sources

If you'd like to build the ArdBot, be sure to start with the November '10 issue of *SERVO Magazine* for Part 1 of this series. Also check out the following sources for parts:

Example code and more:

Arduino
www.arduino.cc

Fritzing
www.fritzing.org

Prefabricated ArdBot body pieces with all construction hardware:

Budget Robotics
www.budgetrobotics.com

Partial list of Arduino resellers:

AdaFruit
www.adafruit.com

HVW Tech
www.hvwtech.com

Jameco
www.jameco.com

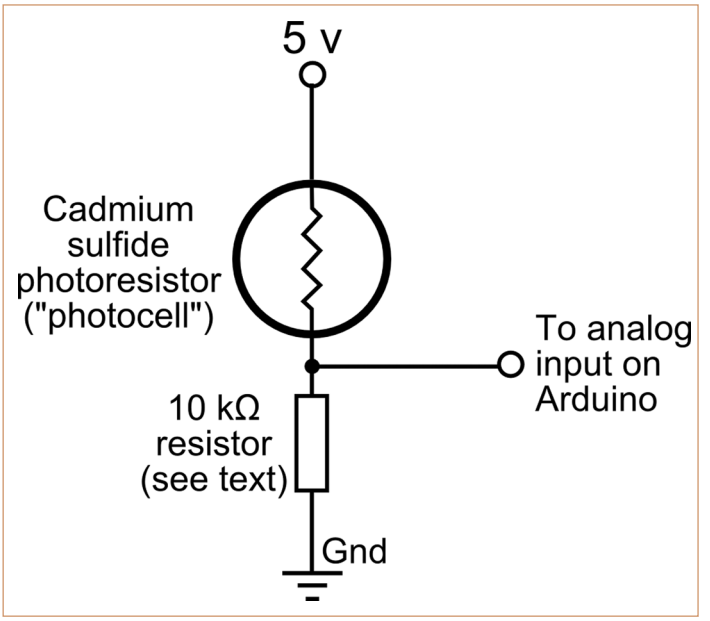
Pololu
www.pololu.com

Robotshop
www.robotshop.com

Solarbotics
www.solarbotics.com

Sparkfun
www.sparkfun.com

FIGURE 6. By connecting another resistor in series with a photoresistor, the output is converted to a varying voltage. In this particular arrangement, the voltage increases under stronger light.



parallel-to-serial (*PISO*) shift register chip, such as the 74HC165 or CD4021. These are integrated circuits that take eight parallel inputs and provide a serial data output that can be read by the Arduino. Assuming eight switches, the PISO reduces the number of required I/O pins from eight to three. There's a code example for how to do this at *arduino.cc/en/Tutorial/ShiftIn*.

Let There Be Light (and let your ArdBot see it!)

Next to tactile feedback, reacting to light is the most common robotic sense. In lieu of actual vision, the robot uses electronic components such as photoresistors and phototransistors that are sensitive to light. Your bot may react to the simple absence or presence of light, or it may be able to measure the brightness, color, or other qualitative aspect of the light.

Photoresistors — also called photocells, light dependent resistors, or CdS (for cadmium sulfide) cells — are perhaps the easiest to use as simple light sensors. The photocell is a resistor whose value changes depending on the amount of light that strikes its sensing surface. In darkness, the photocell has a high resistance, typically in the neighborhood of 100 kΩ to over one megohm, depending on the component. The resistance falls as more light strikes the cell. In high brightness, the resistance may be as low as 1 kΩ to 10 kΩ.

The exact dark/light resistance values differ depending on the component, and even among photocells of the same make and model. Typical value tolerance is 10 to 20 percent. You can purchase photocells new, but they're common finds in the surplus market. Get a variety pack, and use your multimeter to "grade" each one. Cracks and other injury spell doom to a photocell; air and moisture degrade the sensing surface, rendering it useless. Toss any that don't react properly to a nearby desk lamp.

Being a resistor, you can convert the output of a photocell to a varying voltage merely by connecting another resistor to it in series, as shown in **Figure 6**. The value of the series resistor depends on the dark/light resistance range of the photocell, and how you want to use it. The cells I used had a dark resistance of about 40 kΩ and a light resistance of 30 ohms. In average room brightness, the cells had 10 kΩ resistors, so I selected a 10 kΩ series resistor.

The voltage at the point between the photocell and series resistor is a ratio of the two resistance values. With the two resistances equal, the divided voltage between them is one-half of the supply voltage; in the case of five volts in average room light, the output voltage is 2.5 volts.

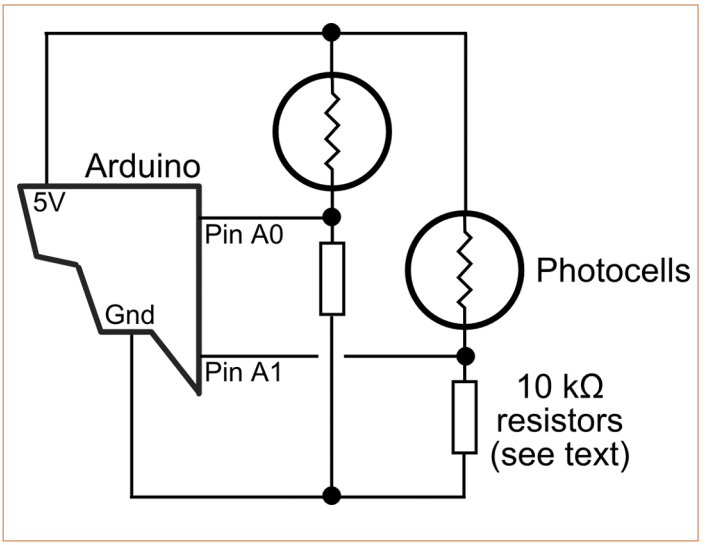


FIGURE 7. Schematic view of connecting two photocells to the Arduino microcontroller.

Listing 3 – simplecds.pde.

```
/*
 ArdBot CdS cell demo
*/

int cds = 0;

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

void loop() {
  // Read analog pin A0 and display value
  //  on Serial Monitor window
  cds = analogRead(A0);
  Serial.println(cds, DEC);
  delay (200);
}
```

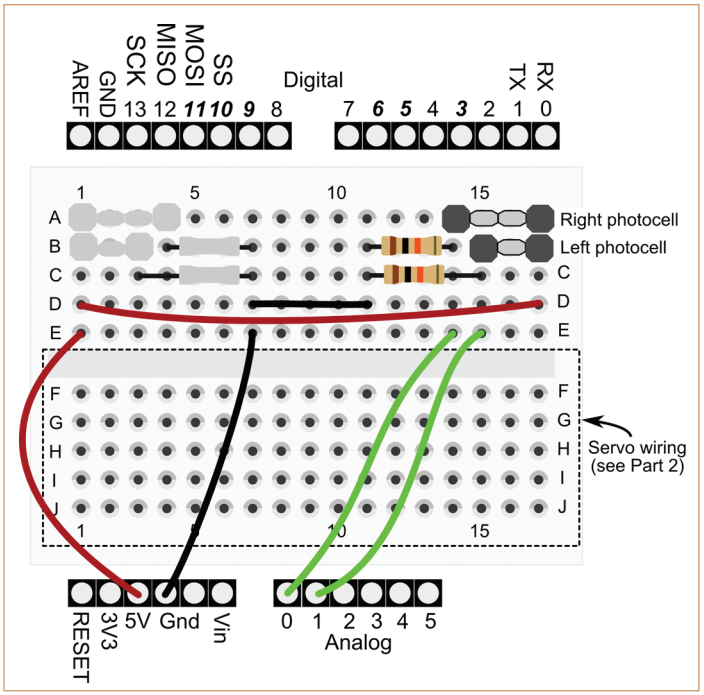


FIGURE 8. Breadboard view of connecting the photocells.

The voltage decreases in darkness and increases as more light strikes the photocell.

You will need to experiment with the series resistor to determine its best value, based on the specific photocells you use. You might want to try a 50 kΩ or 100 kΩ potentiometer in place of a fixed resistor, allowing you to fine-tune the series resistance as needed.

Listing 3 shows *simplecds.pde*, a basic sketch that tests the operation of the photocell. Wire the photocell as shown in **Figure 6** and connect the output to analog pin A0 of your Arduino. Compile and upload the sketch, then open the serial monitor window. You'll see a series of numbers; they correspond to the output voltage of the sensor converted to a 10-bit (0 to 1023) numeric value. You should get a low number when all light to the photocell is blocked, and a higher number under full illumination.

Steering Your Robot With a Flashlight

By using two photocells mounted on each side of your ArdBot, you can literally steer it by flashlight. Under just room light, the robot is set to stop, waiting for your command. Aim the flashlight so that light falls more or less equally on both photocells, and the robot will move forward. When the light levels aren't equal, the robot will turn toward the photocell that has more light falling on it.

Refer to **Figure 7** for a schematic of the two-photocell setup. **Figure 8** shows the same circuit but in breadboard view. For my prototype, I made small mounts for the photocells using scrap PVC plastic, then attached the mounts to the top deck of the ArdBot with metal brackets. The photocells I used measured 0.29" x 0.25" (elliptical shape). I drilled holes just slightly smaller, then used a rat-tail file to enlarge the holes so that the cell just fit inside. On my prototype, the cells are held in just by friction, but on yours you can use hot-melt glue or other adhesive that when set leaves no moisture for a possible short circuit.

(Bear in mind light can strike the photosensitive surface of the cell from the rear. You may want to add a layer or two of black tape to prevent light spoilage.)

Figure 9 shows my ArdBot with the two photocell "eyes" attached to the front. I've bent the brackets back a bit so that the cells point slightly upward.

Refer to **Listing 4** for *lightsteer.pde*. It uses the current values of the photocells to make quick steering adjustments to the left or to the right. In the declarations area, the code:

```
const int ambient = 600;
const int threshold = 800;
```

sets two comparison values used elsewhere in the sketch. *You will need to experiment with these values depending on the room environment and photocell characteristics!* These values worked for me; you can start with them, but expect to try other values as you fine-tune the performance of the steering.

The *ambient* value sets the upper level of just the ambient (natural) light in the

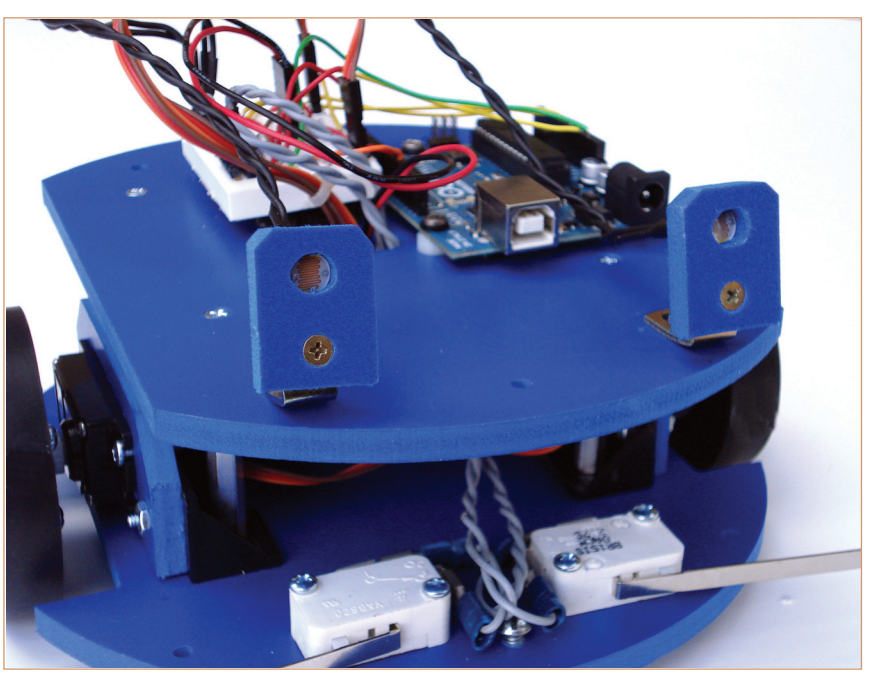


FIGURE 9. Mount the photocells on the top of the ArdBot — toward the left and right sides — to make eyes for following the flashlight beam.

room. This is the amount of light that hits the photocells under normal lighting conditions. For me, the ambient light value was about 520 to 530, so I made it a little higher (600) for extra headroom.

The *threshold* value sets the lower level of the light beamed from your flashlight. I set the value at 800 based on using a nine-LED flashlight (with fresh batteries), 1-2 feet away from the robot. For best results, use a bright flashlight near the robot — the farther away you get, the less light that falls on the photocells.

More Light Tricks

Just by switching around some of the code you can have your robot run away from you rather than try to follow you. Or, by placing colored gel filters over the photocells and using a color LED flashlight (blue and green are popular), your robot can more readily discriminate between light to follow and light to ignore.

You might also add one or two "room light" sensors that aren't in the direct line-of-sight of the flashlight beam. These could be used to set the ambient light level of the room.

Small lenses over the photocells help to focus the flashlight beam, improving steering performance and helping the bot reject any light not directly to the front.

These are just some of the things you can do to give your ArdBot (or other Arduino-powered bot) the gift of sight. Feel free to experiment. Photocells and other light sensitive components are inexpensive, and changing the code in an Arduino sketch is absolutely free.

Coming Up ...

In our next installment, you'll discover even more senses you can provide, including ultrasonic sound for measuring distances to objects, and infrared light to determine the proximity of nasty things that are in the way.

You'll also read about ways for your robot to scan the room to soak up its environment, rather than just see life through a narrow tunnel in front of it. Exciting stuff (at least for your ArdBot), so don't miss it! SV

Gordon McComb can be reached at arduino@robotoid.com.

Listing 4 – lightsteer.pde.

```
/*
 ArdBot steering by light demo
 Requires Arduino IDE version 0017
  or later (0019 or later preferred)
*/

#include <Servo.h>

// CdS cell reference values
//  (you need to experiment)
const int ambient = 600;
const int threshold = 800;

int lightLeft = 0;
int lightRight = 0;

Servo servoLeft;
Servo servoRight;

void setup() {
  servoLeft.attach(10);
  servoRight.attach(9);
}

void loop() {
  // Read light sensors connected to
  //  analog pins A0 and A1
  lightLeft = analogRead(A1);
  lightRight = analogRead(A0);

  // Stop robot if below ambient
  if (lightRight < ambient || lightLeft < ambient) {
    stopRobot();
  } else {
     forward();
    // Steer to right if right CdS below threshold
    if (lightRight < threshold) {
      turnLeft();
      delay (250);
    }
    // Steer to left if left CdS below threshold
    forward();
    if (lightLeft < threshold) {
      turnRight();
      delay (250);
    }
  }
}

// Motion routines
void forward() {
  servoLeft.write(180);
  servoRight.write(0);
}

void reverse() {
  servoLeft.write(0);
  servoRight.write(180);
}

void turnRight() {
  servoLeft.write(180);
  servoRight.write(180);
}

void turnLeft() {
  servoLeft.write(0);
  servoRight.write(0);
}

void stopRobot() {
  servoLeft.write(90);
  servoRight.write(90);
}
```