



Published on learn.parallax.com (<http://learn.parallax.com>)

[Home](#) > [Propeller C Learning System](#) > [Propeller C Tutorials](#) > ActivityBot

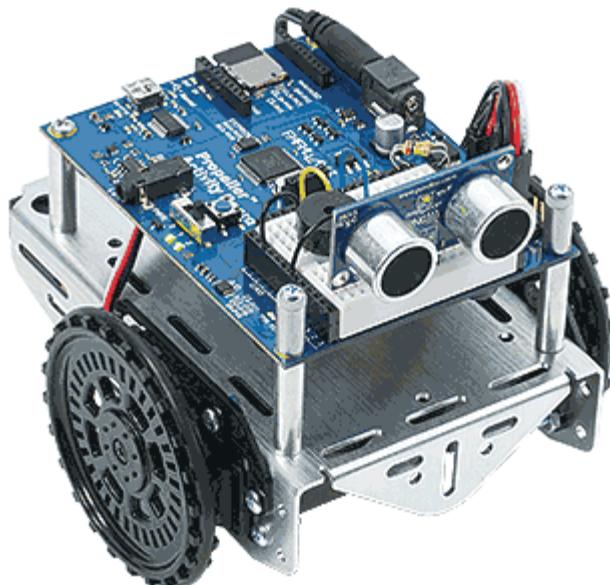
ActivityBot

This C language tutorial for the 8-core Propeller microcontroller and the ActivityBot Robot Kit is designed for independent learners age 14 and up, and is also suitable for a classroom robotics curriculum.

Meet the ActivityBot

This compact, zippy robot matches a multi-core Propeller microcontroller brain with great hardware:

- Versatile [Propeller Activity Board](#) [1] perched atop our classic, sturdy aluminum chassis
- Custom-made High Speed servos
- Optical encoders and wheels with secure O-ring tires ensure straight straightaways and consistent maneuvers
- SD card for datalogging and file storage
- Electronic components for building navigation systems using touch, visible light, infrared light, and ultrasonic sensors



What It's About

This tutorial will show you how to:

- Assemble and wire up your robot
- Set up the SimpleIDE programming software
- Get special C libraries just for your ActivityBot
- Write simple programs in C for the multi-core Propeller microcontroller
- Program your ActivityBot to navigate with encoders
- Build sensor-based navigation systems for your ActivityBot so it can navigate on its own

What's Included, What's Coming Soon

The ActivityBot menu at left is the Table of Contents for this tutorial. Activities that use the SD card for datalogging will follow. Expect new sections and projects that make use of optional accessories such as the speaker jack for playing WAV files, and the XBee module for RF communication.



ActivityBot Library Updates

We frequently add features and improvements to the ActivityBot library. The latest release date and a link to the ActivityBot Library are in the Stay Current! block on the top right.

Before You Start

You can start right here. All you need is your ActivityBot Robot Kit and five 1.5 V AA batteries. In fact, you may have already built your robot following the Mechanical Assembly Guide that comes in the box. If not, don't worry, assembly instructions are repeated in the links below. During this tutorial you will also be directed to other Propeller C Tutorials to get software, try example programs, and build some circuits.

After You Finish

Once you've mastered the basics, you can mod your 'bot with your newfound skills. The ActivityBot is ready for tinkering, with onboard circuitry to support these expansion kits:

- Veho Speaker and mounting bracket - play WAV files from the SD card via the board's audio jack
- XBee Wireless Pack - plug one RF module into the board's socket, and connect the other to your computer for wireless control
- PING))) Mounting Bracket Kit to elevate and rotate the PING))) Ultrasonic Distance Sensor for scanning the terrain and locating objects

Ready to get started with your ActivityBot? Just follow the links below.

Meet the Propeller Chip

New to microcontrollers? Want to know what "multicore" means?



- ✓ Read [Propeller Brains for Your Inventions](#) [2], then return here.

Welcome back!

If you want to know more about the Propeller chip, you can browse the [Propeller Q & A webhelp](#) [3]. If you are into datasheets, you can [find the Propeller P8X32A Datasheet here](#) [4].

Now, back to the ActivityBot!

- ✓ If your ActivityBot is still in pieces, go on to [Mechanical Assembly](#) [5].
- ✓ If your robot is already built, it's time to make or double-check the [Electrical Connections](#) [6].

Mechanical Assembly

If your Propeller ActivityBot came with a Mechanical Assembly Guide, you may have put yours together already.



- ✓ If your robot is already assembled, skip ahead to [Electrical Connections](#) [6].
- ✓ If not, follow the steps below to put your robot together.



Replacement Parts

Most of the pieces in the kit are available for purchase individually online from www.parallax.com [7]. Any piece can also be obtained by contacting sales@parallax.com [8] or calling 916-512-1024.

Check your Hardware

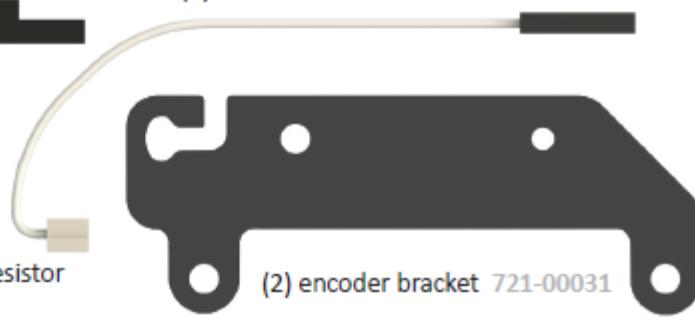
HARDWARE

(2) infrared sensor 350-00038



ActivityBot Encoder Kit 32501

(2) cable for encoder 805-28107



(2) 7/8" #4-40 panhead screw
710-00007



(1) 13/32" rubber grommet
700-00025



(10) 3/8" #4-40 panhead screw
700-00002



(10) 1/4" #4-40 panhead screw
700-00028



(3) 3/16" #4-40 flathead screw
700-00016



(1) cotter pin
700-00023



(4) 1" round #4-40 standoff
700-00060



(2) 1/2" round #4-40 spacer
713-00007



(3) #4 nylon washer
700-00015



(11) #4-40 nylon core locknut
700-00024



(10) #4-40 nut
700-00003



Made in USA
(1) Parallax combination wrench
700-10025



Parallax screwdriver 700-00064

You will have pieces left over. That's ok!



(2) ActivityBot encoder wheel 721-00021



(2) ActivityBot O-Ring tire 710-00200



(2) high speed servo 900-00025



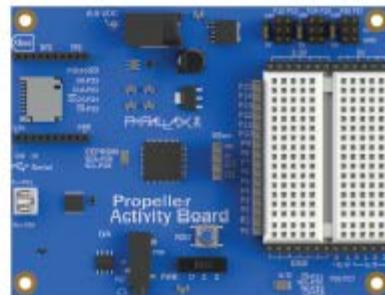
(1) 1" tail wheel ball
700-00009



(1) small robot chassis
700-00022



(1) 5 AA battery holder
753-00007



(1) Propeller Activity Board
32910

Additional parts (*not used in this assembly guide*)



(1 bag) electronic components
572-32500



(1) MicroSD Card
32319



(1) Ping))) Ultrasonic Distance Sensor
753-00007



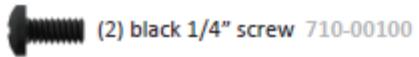
(1) USB A to
mini-B cable
805-00006

Most, but not all, parts are sold separately from the web store. Contact Parallax Sales directly for any part: sales@parallax.com, 888-512-1024 in U.S., or 916-624-8333.

Step 1 - Prepare your Encoders

STEP 1

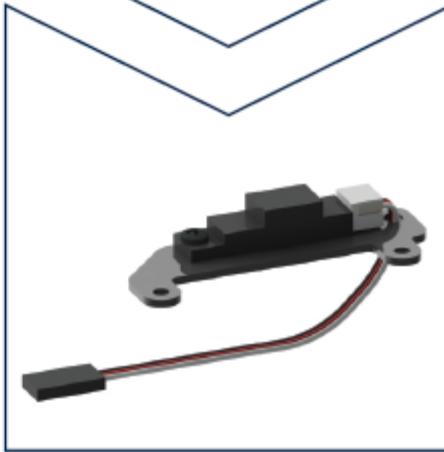
Prepare right and left encoders.



RIGHT



LEFT



Step 2 - Prepare the Tires

STEP 2

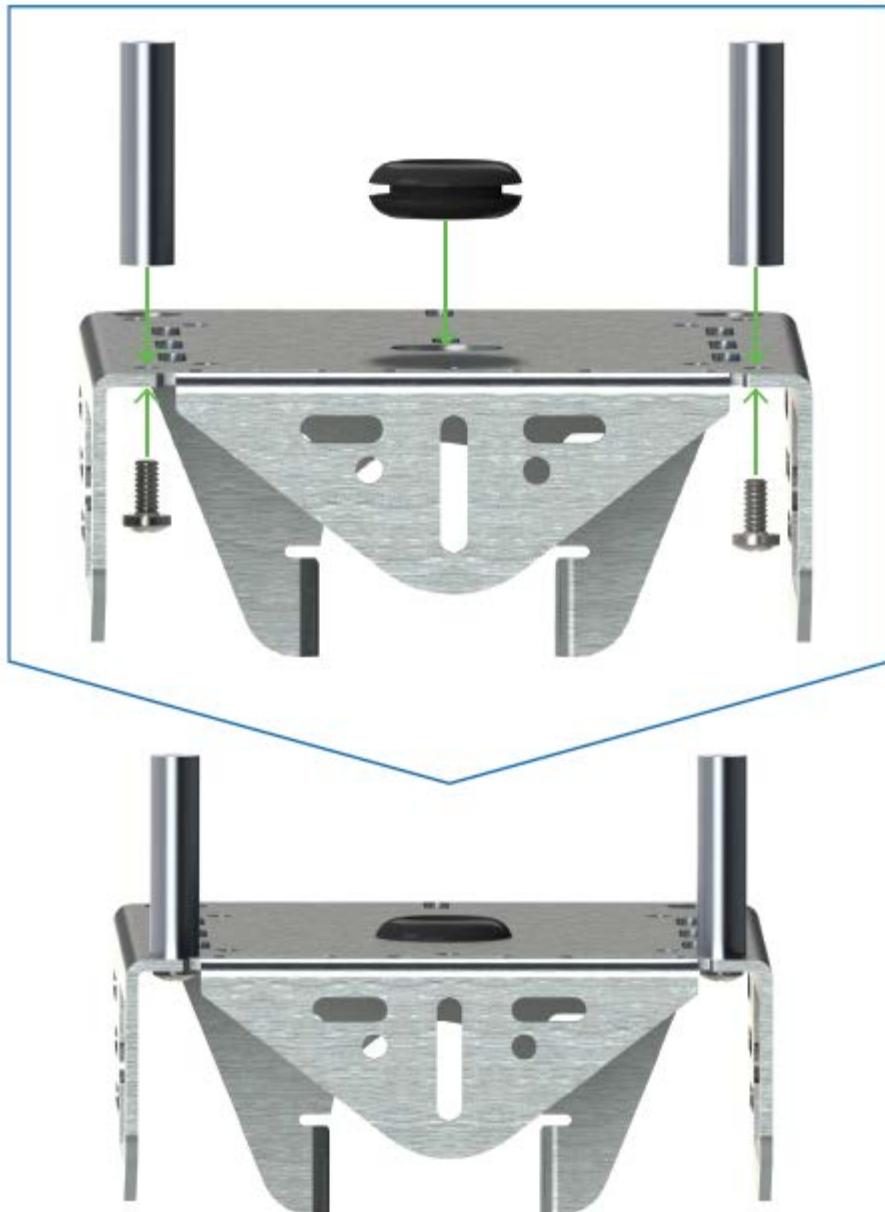
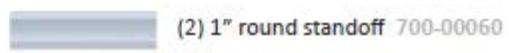
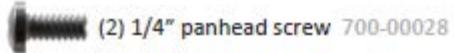
Push a tire onto each wheel. Repeat.



Step 3 - Prepare the Chassis

STEP 3

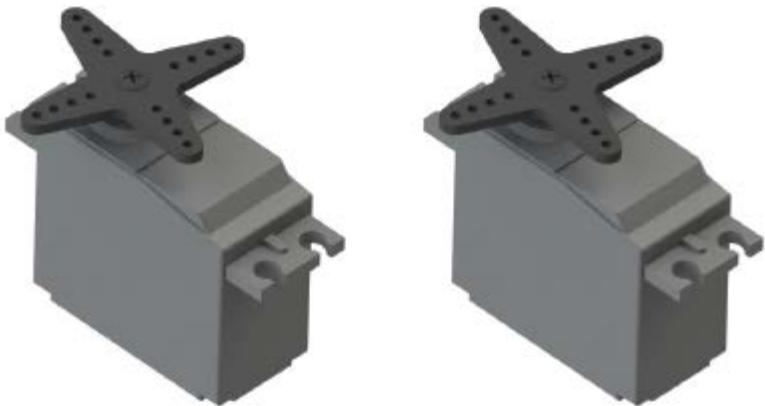
Prepare chassis.



Step 4 - Prepare the Servos

STEP 4

Remove servo horns. Save the screws!



Step 5 - Mount the Right Servo

STEP 5

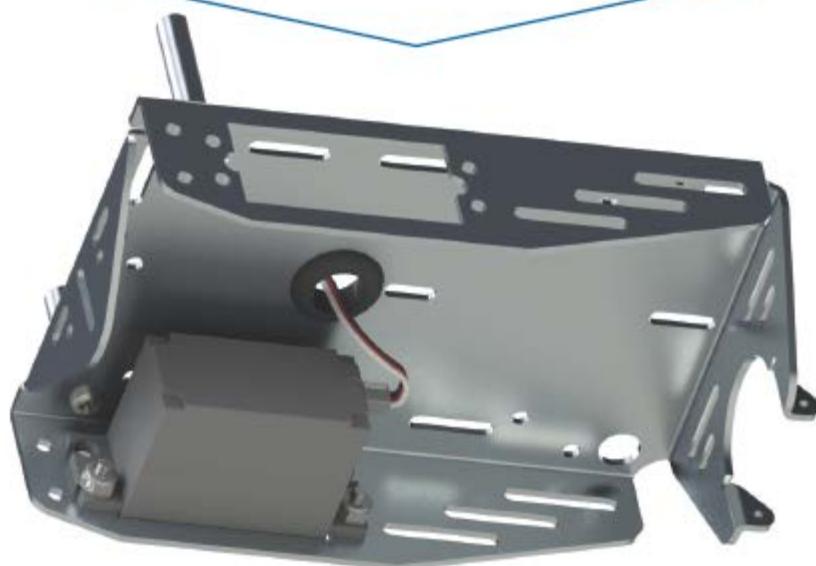
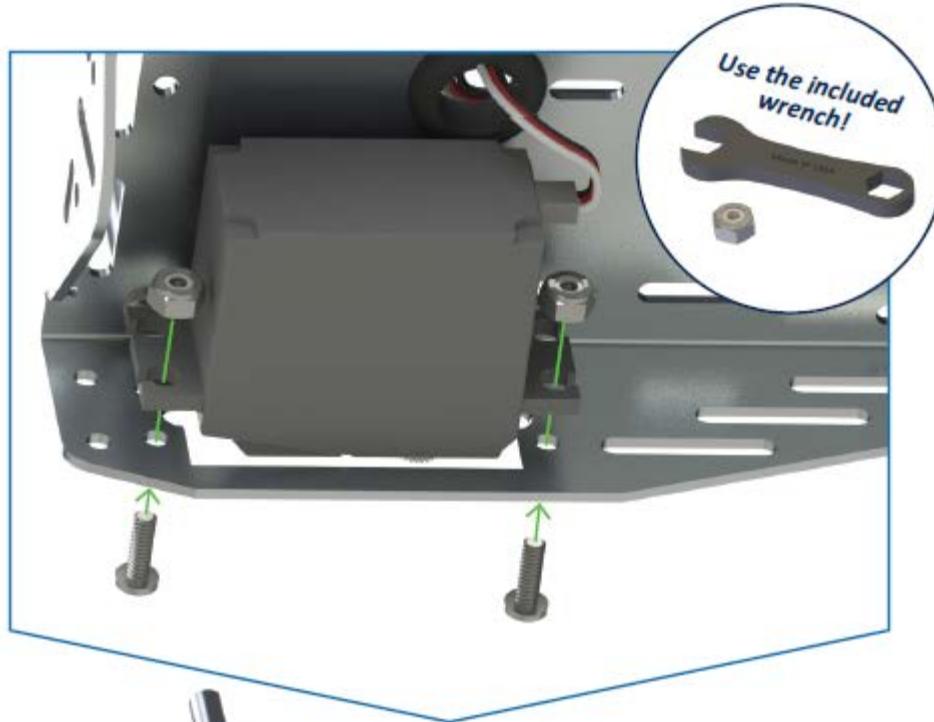
Mount right servo.



(2) 3/8" panhead screw 700-00002



(2) nylon core locknut 700-00024



Step 6 - Mount the Left Servo

STEP 6

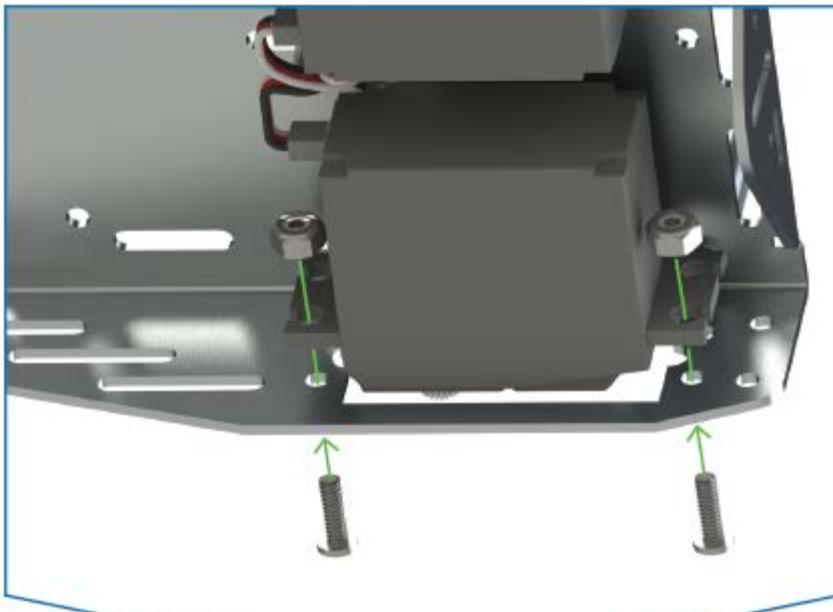
Mount left servo.



(2) 3/8" panhead screw 700-00002



(2) nylon core locknut 700-00024



Step 7 - Mount the Right Encoder

STEP 7

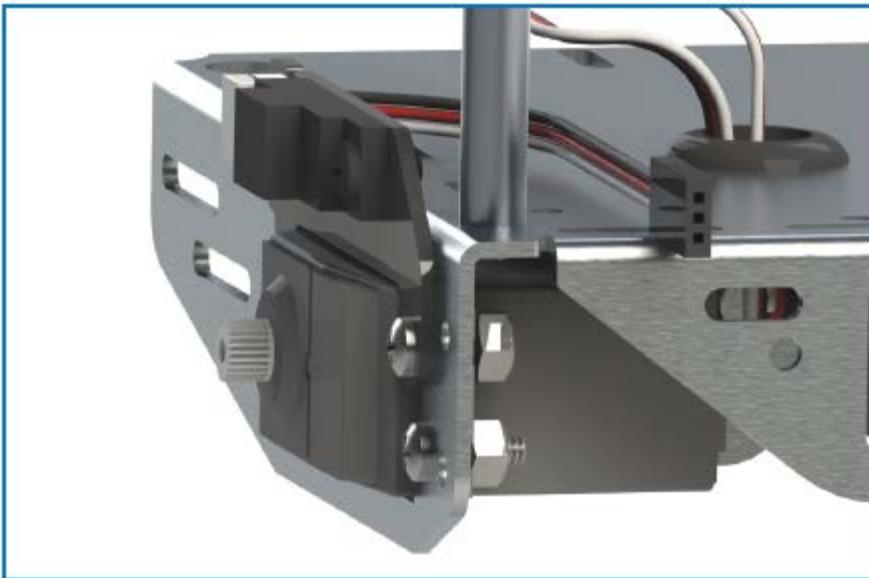
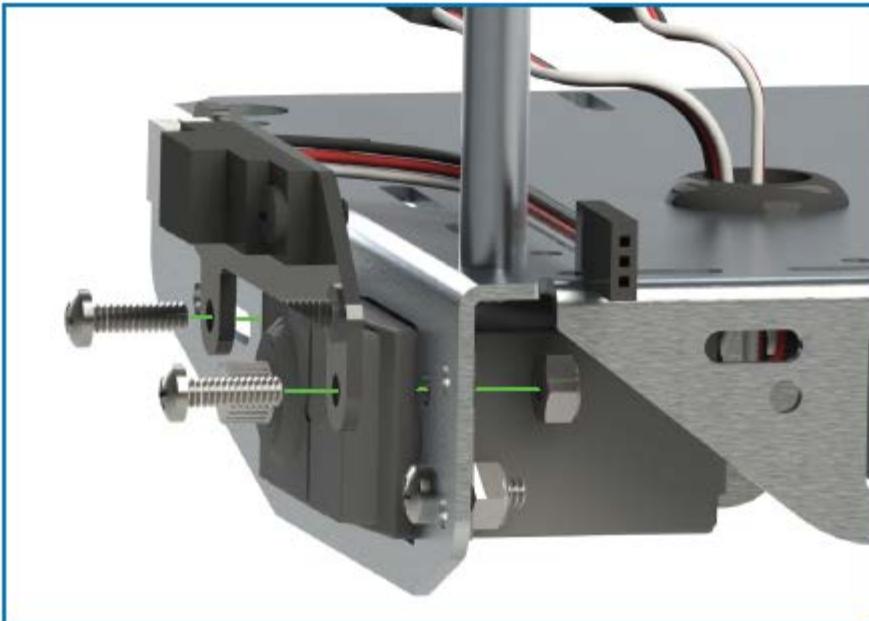
Mount right encoder.



(2) 3/8" panhead screw 700-00002



(2) nylon core locknut 700-00024



Step 8 - Mount the Left Encoder

STEP 8

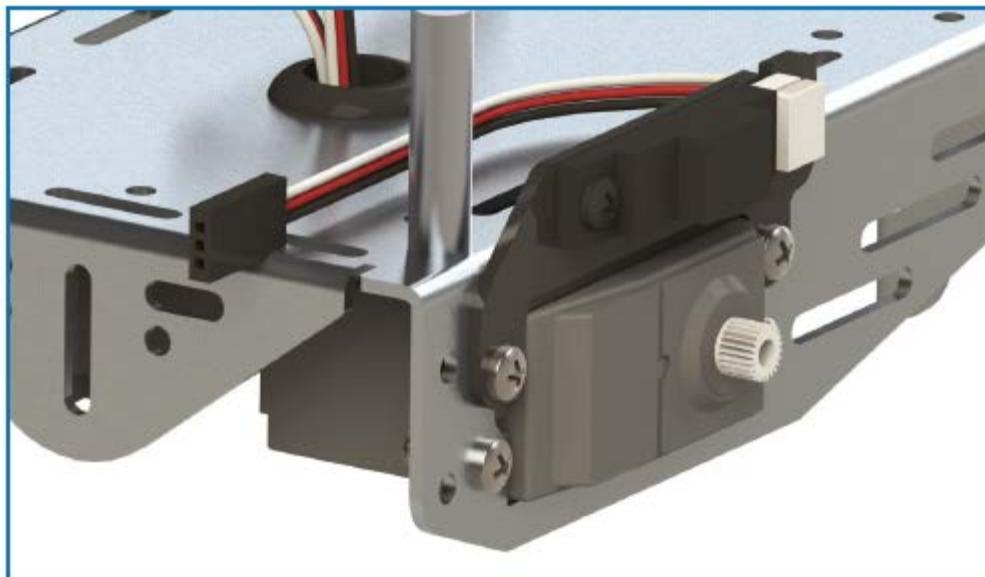
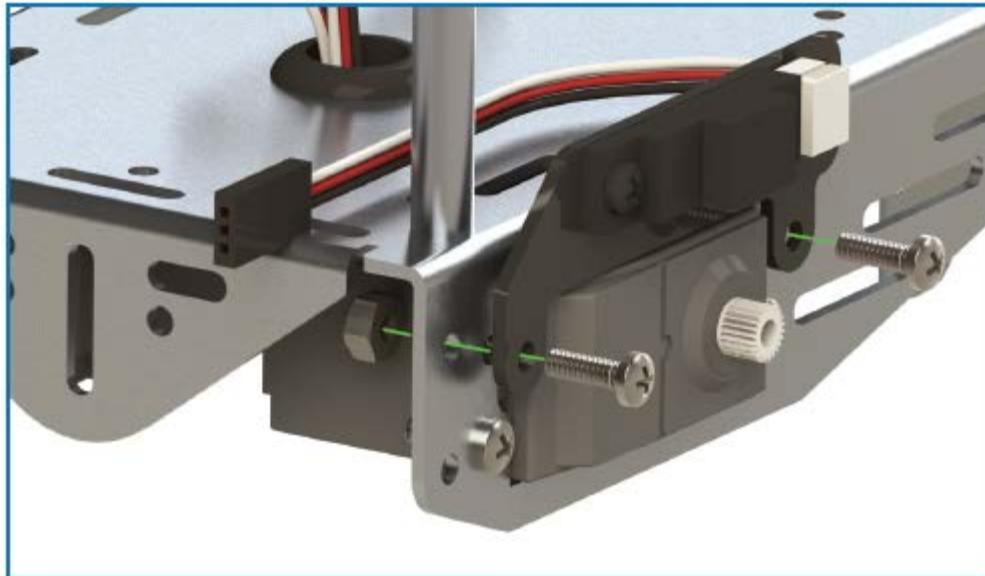
Mount left encoder.



(2) 3/8" panhead screw 700-00002



(2) nylon core locknut 700-00024



Step 9 - Mount the Battery Pack

STEP 9

Mount battery pack.



(2) 3/16" flathead screw 700-00016



(2) 1" round standoff 700-00060

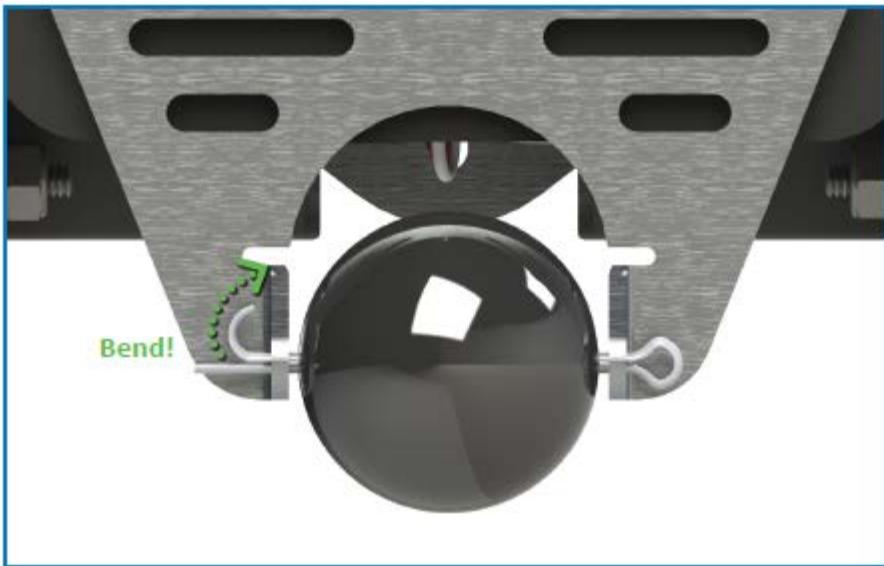


Step 10 - Mount the Tail Wheel

STEP 10

Mount tail wheel.

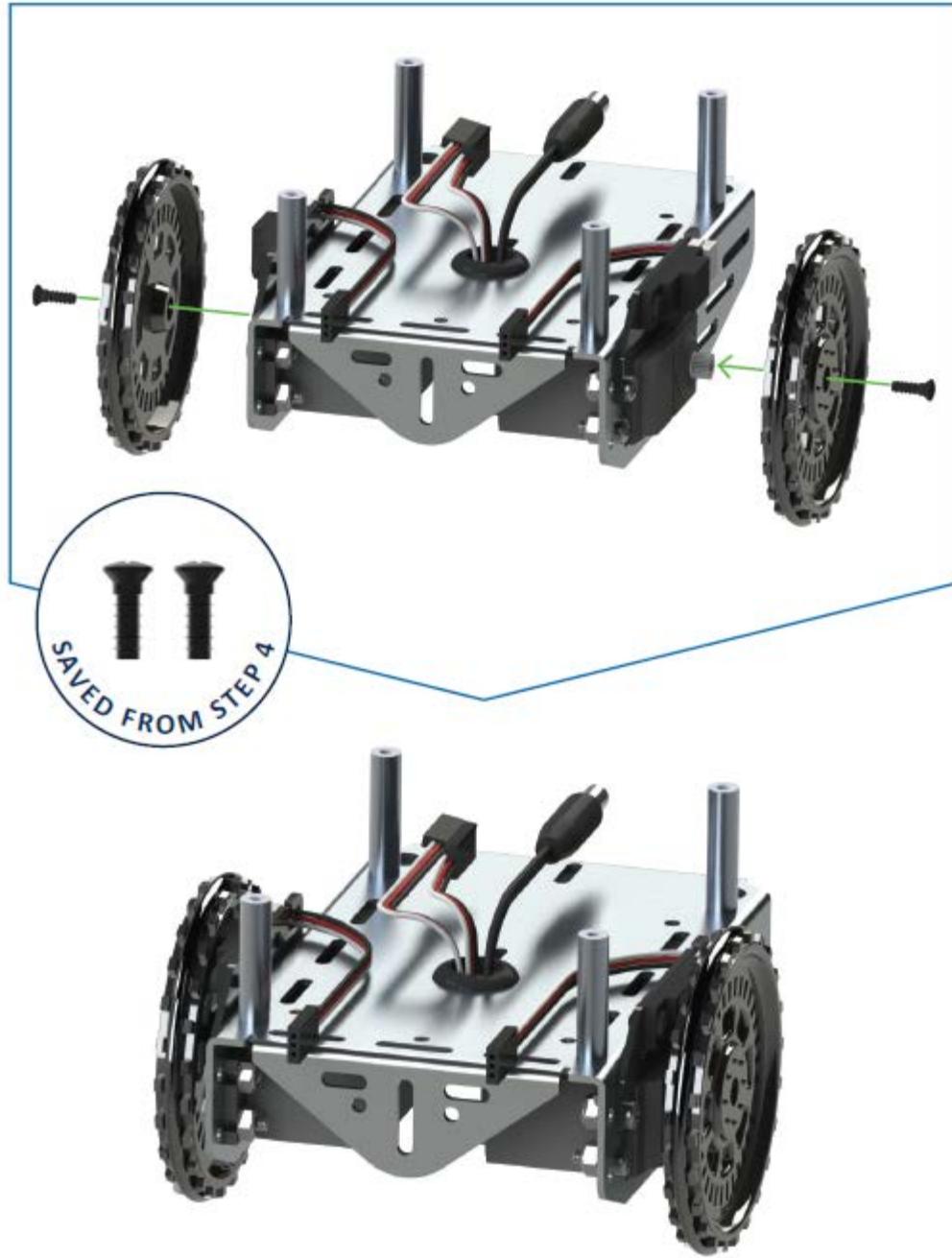
(1) cotter pin 700-00023



Step 11 - Mount the Drive Wheels

STEP 11

Mount drive wheels.



Step 12 - Mount the Activity Board

STEP 12

Mount board.



(2) 7/8" panhead screw
710-00007



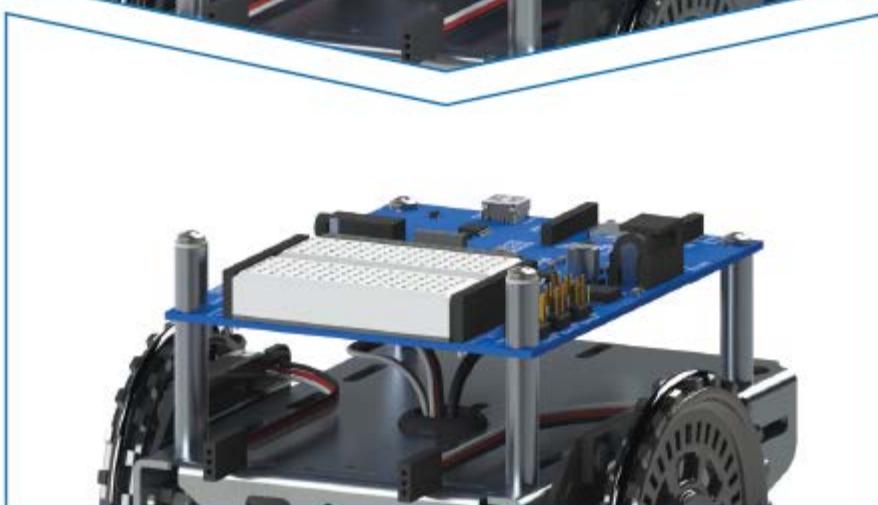
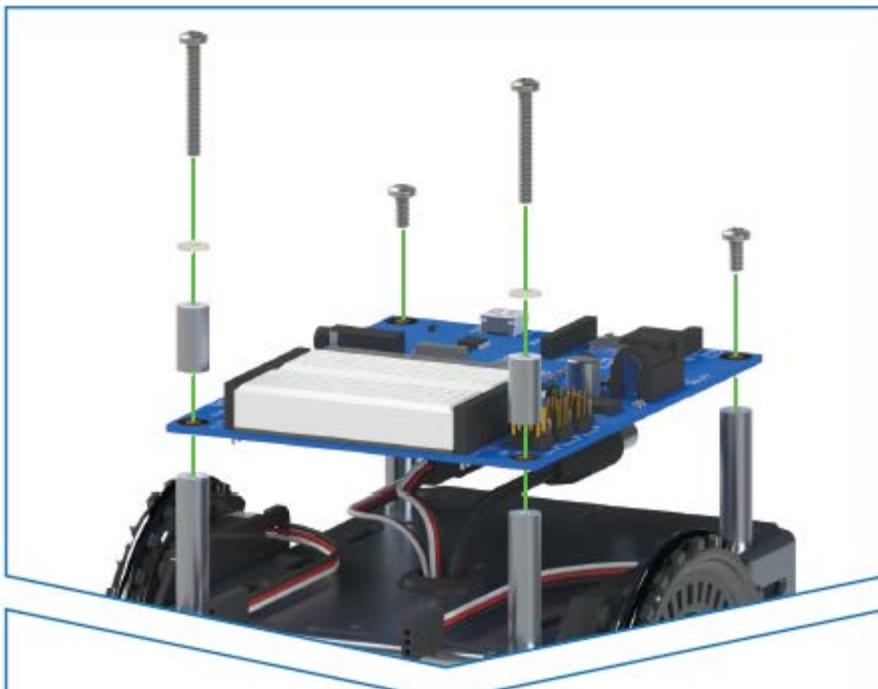
(2) nylon washer 700-00015



(2) 1/4" panhead screw 700-00028



(2) 1/2" round spacer 713-00007



Electrical Connections

Gather the Parts

For this Propeller ActivityBot construction step, you will need:

Assembled robot

(2) 20 k-ohm resistors (red-black-orange)

(5) 1.5 V AA batteries

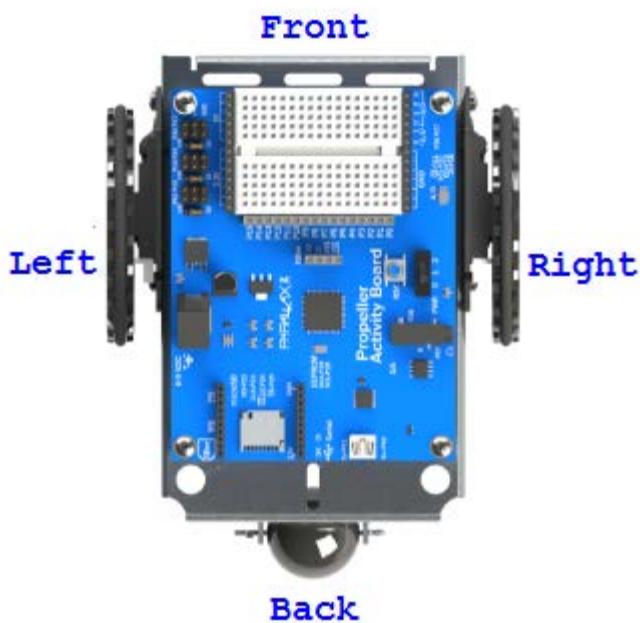
Needle-nose pliers

Masking tape

Pen

Label the Cables

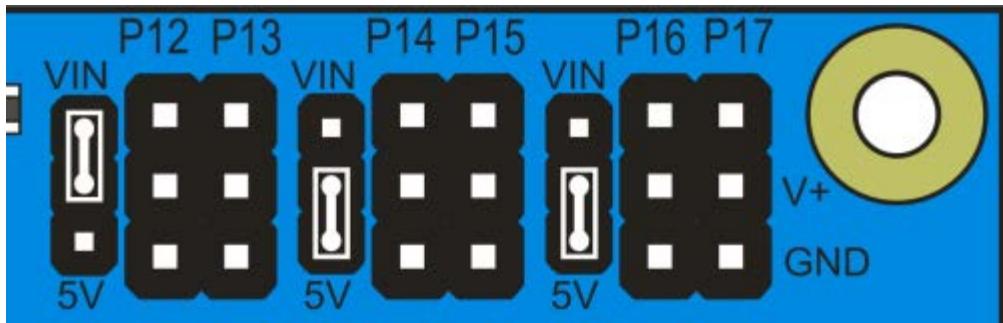
- ✓ Put a masking tape label on the end of each encoder cable and servo cable, near the 3-pin socket.
- ✓ Trace each cable back to its origin to see what it is connected to.
- ✓ Label each cable Right or Left, Encoder or Servo.



Prepare the Servo Ports

Each pair of 3-pin servo ports along the top of the Propeller Activity Board has a jumper on power-select pins to its immediate left.

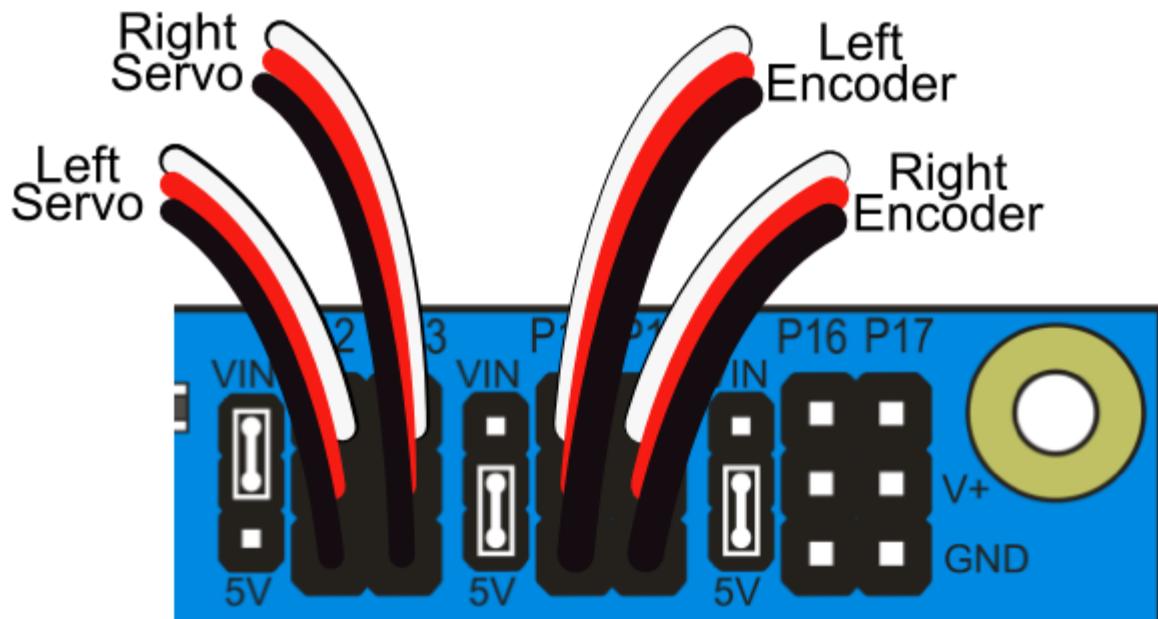
- ✓ **Important!** Make sure there is no USB cable or battery pack cable plugged into the ActivityBoard - moving jumpers with power connected could damage your board.
- ✓ Move the shunt jumpers into the configuration shown below. (If your jumpers don't have a little tab on them, needle-nose pliers will help.)
P12 & P13: VIN
P14 & P15: 5V
P16 & P17, 5V



Plug In the Cables

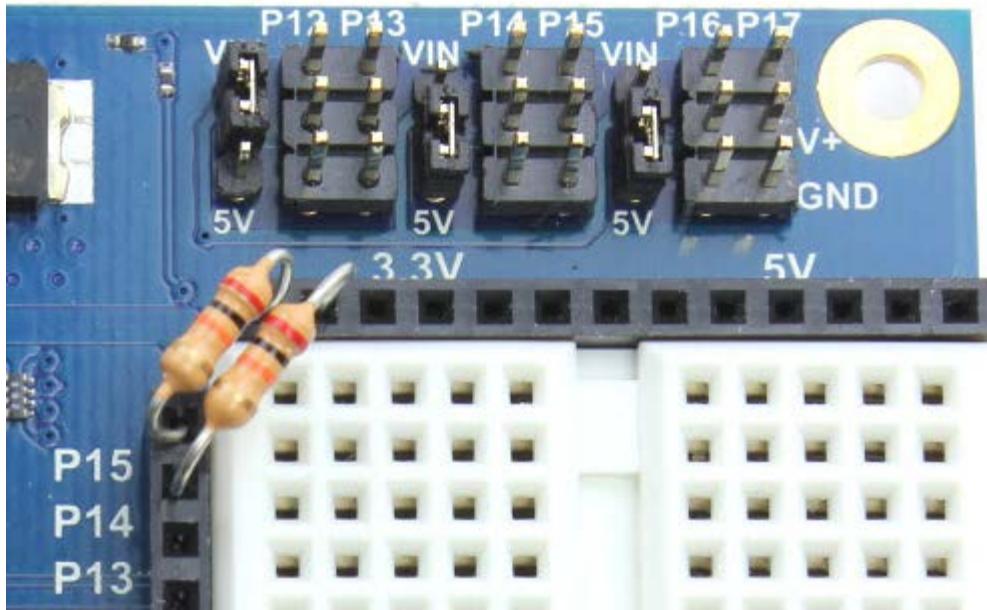
The encoder and servo cables have 3 wires, white-red-black. When you plug these cables into the servo headers, be sure to align white wires with the pin labels along the top of the board, and the black wires along the row labeled GND.

- ✓ Plug the encoder cables and servo cables into the servo ports
P12: Left Servo
P13: Right Servo
P14: Left Encoder
P15: Right Encoder



Add the Resistors

- ✓ Using the 20 k-ohm resistors, connect the P14 and P15 sockets to the left of the white breadboard to the 3.3 V power sockets just above the breadboard.



Double-check your Connections



STOP - CHECK EACH CONNECTION NOW!

The ActivityBot won't work if a connection is wrong. Double-checking now is easier than troubleshooting later.

- ✓ P12 & P13 servo port shunt jumper — Set to VIN.
- ✓ P14 & P15: servo port shunt jumper — Set to 5V.
- ✓ Left servo cable — P12 servo port.
- ✓ Right servo cable — P13 servo port.
- ✓ Left encoder cable — P14 servo port.
- ✓ Right encoder cable — P15 servo port.
- ✓ All servo and encoder cables — Each plug is over all 3 pins of its servo port, with white cable closest to edge of board.
- ✓ Left encoder cable — other end is firmly plugged into encoder sensor.
- ✓ Right encoder cable — other end is firmly plugged into encoder sensor.
- ✓ P14 socket — connected to 3.3V header with 20 k-ohm resistor (red-black-orange).
- ✓ P15 socket — connected to 3.3V header with 20 k-ohm resistor (red-black-orange).
- ✓ Both 20 k-ohm resistors — if you trimmed them, make sure the leads are long enough to sit all the way down in the socket for a good electrical connection.

Insert the Batteries

- ✓ Place five 1.5 V AA batteries into the battery pack.
- ✓ Plug the battery pack's barrel plug into the Propeller Activity Board's barrel jack.



Congratulations, construction is complete!

- ✓ Keep following the links below to set up your software and start programming.

Software and Programming

It's time to program your ActivityBot!

SimpleIDE Software

The software you will use is called SimpleIDE. It is available for Windows and Mac (and even Linux if you

are willing to compile it). It's an open-source C programming environment for the multi-core Propeller microcontroller. It includes special Simple Libraries written just for the Propeller C Tutorials. Sometimes, updates to the libraries and example code are distributed separately as a Learn Folder release. It is important to always use the most recent version of the software and the Learn folder.

- ✓ Go to the [Set Up SimpleIDE tutorial](#) [9] and follow the instructions to get the current USB drivers, software, and Learn folder, then come back here when you are done.

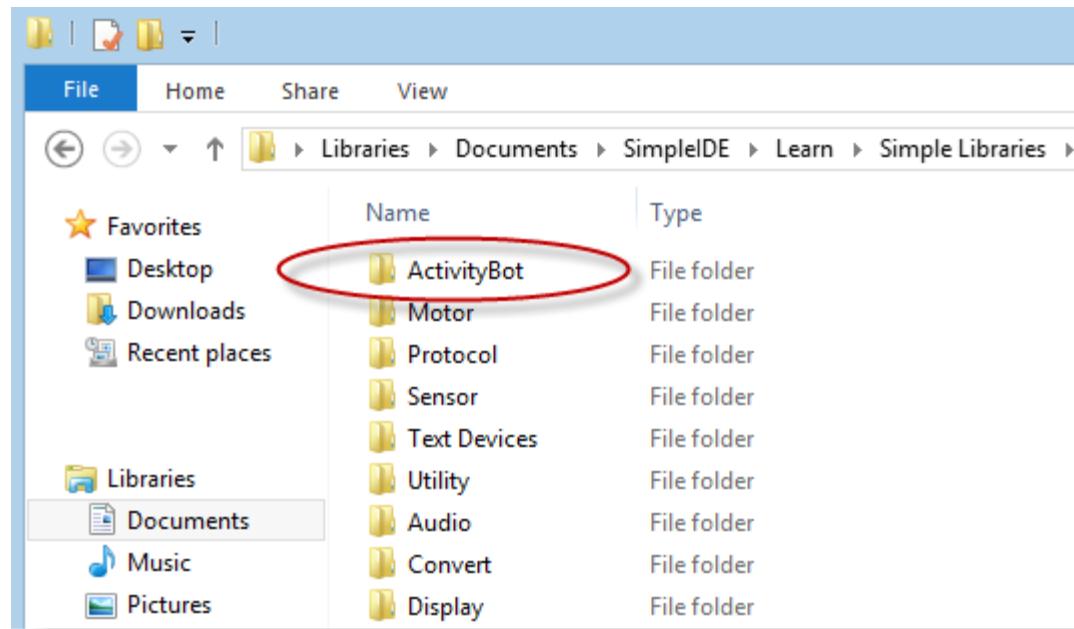


ActivityBot Library

Welcome back!

It's important to always use the latest version of the ActivityBot library, which may be updated frequently as we add features and improvements. The latest release date is shown in the "Stay Current!" block to the right.

- ✓ Completely **remove** any existing versions of the ActivityBot library in your Simple Libraries folder
- ✓ [**Download the latest ActivityBot library \(2013-10-31\)**](#) [10]
- ✓ **Un-zip** the archive.
- ✓ **Copy the folder to ...Documents\SimpleIDE\Learn\Simple Libraries.** (You can leave the date code on the folder if you like, or rename it just ActivityBot as shown below. The important thing is that it's an unzipped folder inside Simple Libraries.)
- ✓ **Important! Close SimpleIDE, and then re-open it.** This allows the software to cache the location of the ActivityBot library, so it will recognize it when it is included in example programs.



You are free to use these libraries for your own amazing inventions! They are distributed under the [MIT License](#) [11].

Programming Practice

Now it's time to try a little programming. This will help you get familiar with the SimpleIDE software and the Propeller C language and Simple Libraries.

- ✓ Go to the [Start Simple](#) [12] tutorial and try the programming examples, then come back here when you are done.

```
Hello Message.c * [X]
9 #include "simpletools.h"
10
11 int main()
12 {
13     pause(1000);
14     print("Hello!!!");
15 }
```

[12]

Welcome Back!

- ✓ Now follow the links below to continue with the ActivityBot tutorials.

Circuit Practice

It's tempting to dive right into robot navigation. We understand! But, as with any vehicle, it's wise to get familiar with the controls before attempting to drive.

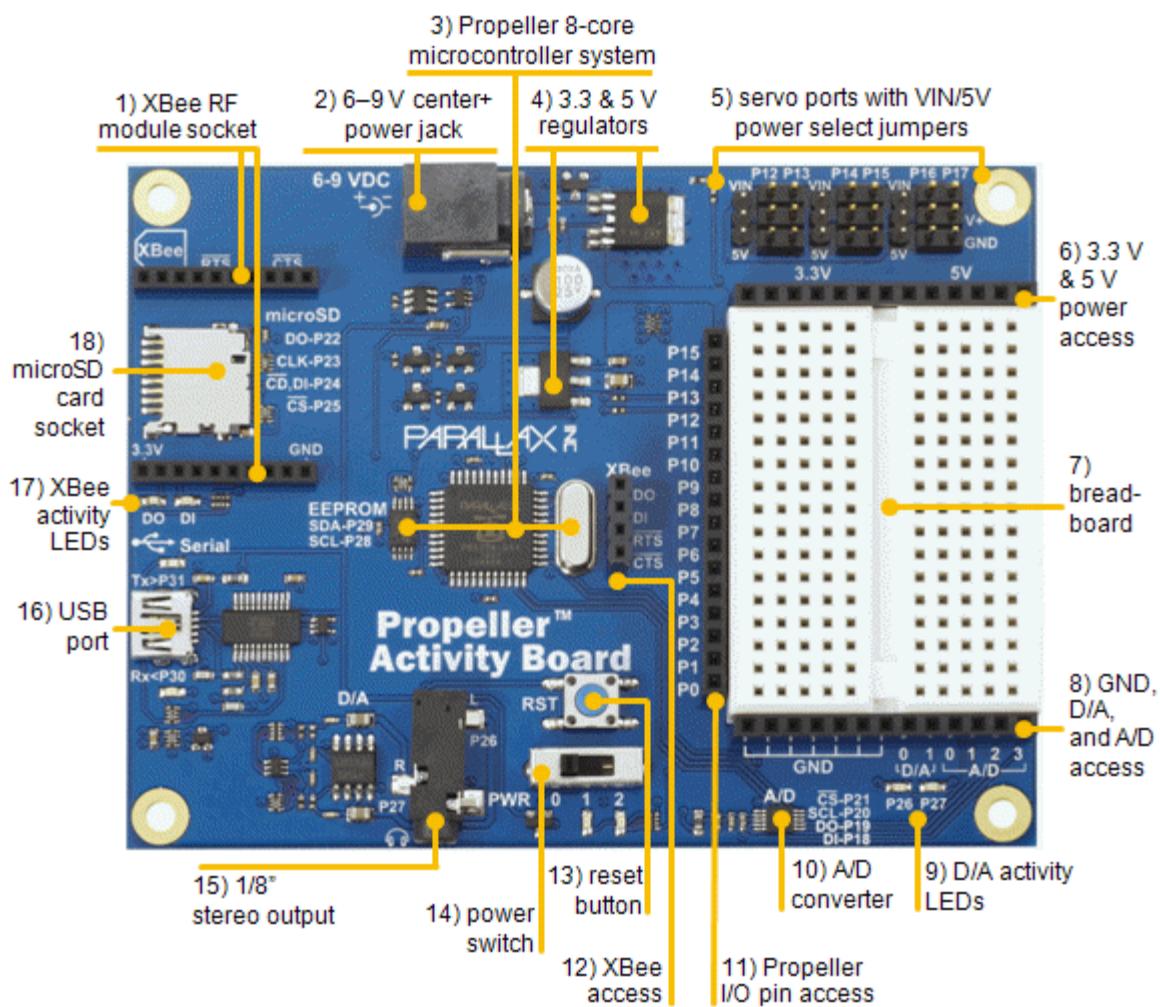
In this section you will:

- Take a look at the Propeller Activity Board's features
- Take a closer look at the power switch and breadboard
- Try blinking the built-in light-emitting diodes
- Add a piezo speaker to your board and make it beep

Know your Board

Let's start with a tour of the Propeller Activity Board. You've already connected cables to the servo ports and have plugged the battery pack into the power jack. You've been programming it a bit already, so you have been using the USB port.

✓ Now, take a moment to look over the diagram below to see the rest of the features.



You can read more about each of the board's features in the Activity Board Documentation available on

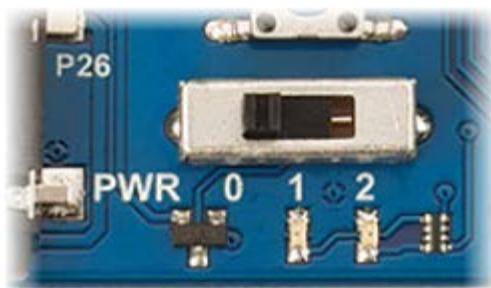
the 32910 product page at Parallax.com.

Next, let's take a closer look at two features in particular: the power switch and the breadboard.

- ✓ Follow the links below to continue with the ActivityBot tutorial.

Powering & Connecting Circuits

The 3-Position Power Switch



The power switch has three positions:

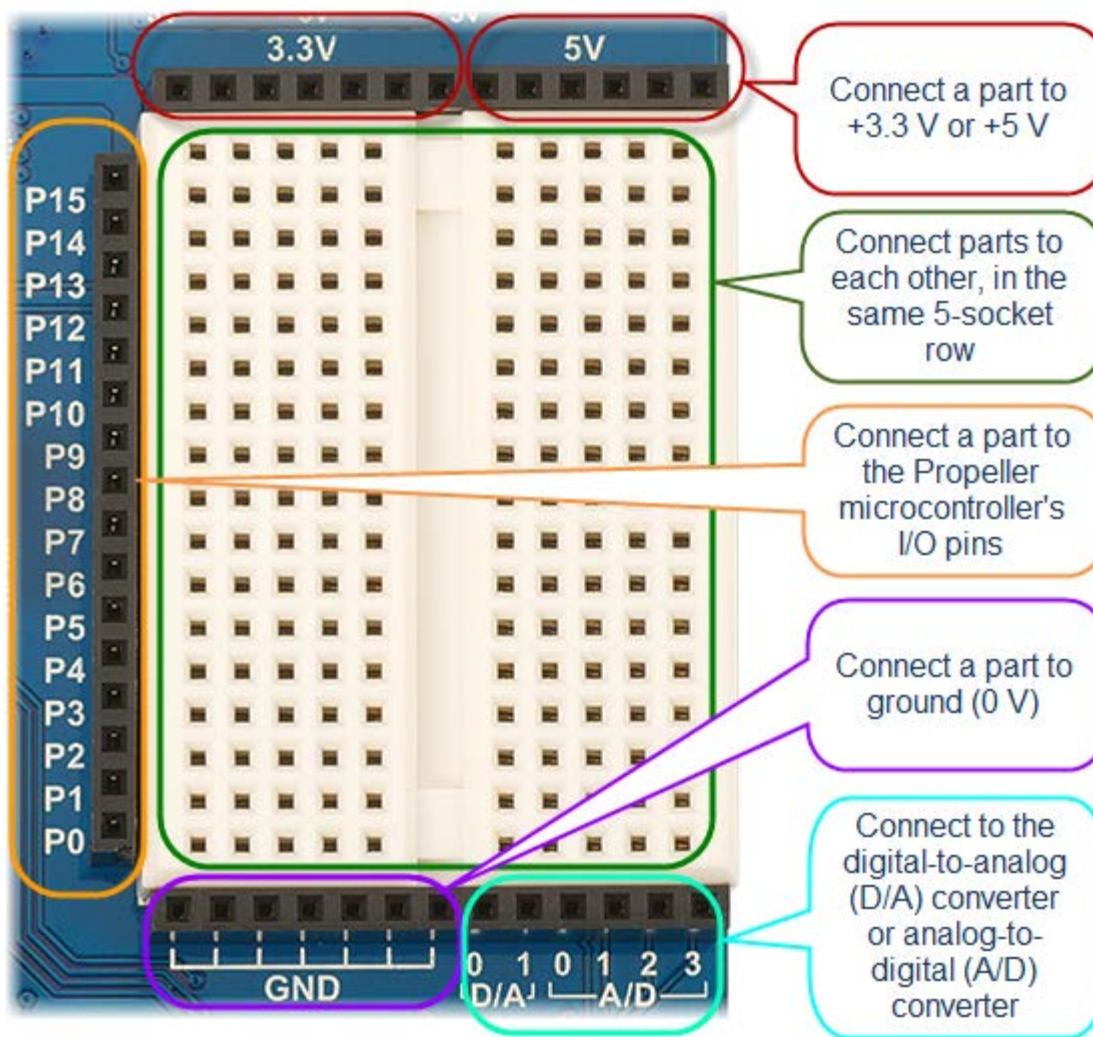
0. **Position 0** turns off power to the circuits on the board. (If you plug your board into a USB port, you might see little lights near the USB connector, since it is receiving power from your computer). Always set the power switch to Position 0 before building or modifying circuits.
1. **Position 1** connects power to *most* of the circuits on the board, including the black sockets along the three edges of the white breadboard. It does NOT connect power to the 3-pin servo ports labeled P12-P17 that are above the white breadboard. Put the switch in Position 1 before loading a navigation program that will make the robot's wheels turn, and then load the program into EEPROM. This will keep the robot from driving right off the table as soon as the program is loaded.
2. **Position 2** powers all the circuits on the board, including the 3-pin servo ports. After loading a navigation program into EEPROM, you can put the robot on the floor, hold the reset button down, and put the switch in position 2. When you let go of the reset button, the program will start running and the robot will drive in the (hopefully) safe place you placed it.

The Breadboard

The breadboard lets you build your own circuits with common electronic components. It's a great way to learn about electricity, and to experiment with making your own inventions. Building experimental circuits to design your own projects is called prototyping, and it is a real-world engineering skill.

The Activity Board's breadboard is surrounded on three sides by black sockets. These make it convenient to connect circuits on the breadboard to power, ground, and the Propeller I/O pins. There are also sockets

to connect to a digital-to-analog converter and an analog-to-digital converter.



- ✓ If you have never built circuits on a breadboard before, go to the [Breadboard Basics page](#) [13], then return here.

Welcome back! Did you watch the video? If you did, you now know why it's called a "breadboard." History can be pretty interesting!

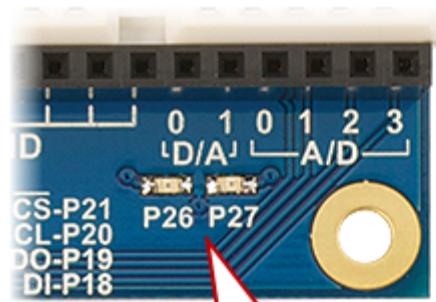
- ✓ Follow the links below to continue with the ActivityBot tutorial.

Blinks

Next, let's test-drive two built-in light circuits on your Activity Board.

Your Activity Board has two built-in lights, near the bottom-right corner of the board. These tiny light-emitting diodes (LEDs) are already electrically connected to I/O pins P26 and P27. These LEDs are helpful when developing applications that use sensors. The idea is to write your program so that if a sensor is activated, an LED lights up to give you, the roboticist, a quick visual cue that the sensor is actually

detecting something.



Built-in light-emitting diodes (LEDs) connect to I/O pins P26 and P27

The Simple Circuits tutorial has an example program for blinking these LEDs.

✓ Go try the [Blink a Light tutorial](#) [14], then come back here when you are done.

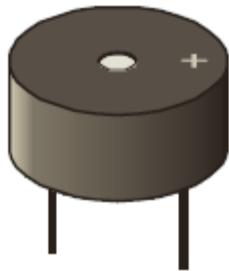
Welcome back!

✓ Follow the links below to continue with the ActivityBot tutorials.

Beeps

Even though the Propeller chip can play WAV files and synthesize speech, sometimes a simple beep is all the noise you need to get the job done.

The first breadboard circuit we'll build for the ActivityBot is a simple piezo speaker.



It is very easy to make this speaker beep with a single line of code:

```
freqout(4, 1000, 3000); // pin, duration, frequency
```

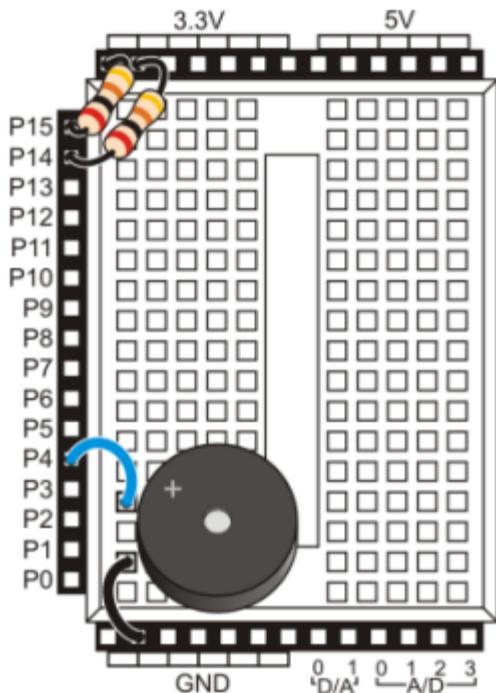
You can optionally add this of code at the beginning of your ActivityBot example programs. Then, If your robot's batteries run low, the Propeller microcontroller will reset and restart the program—the beep will let you know if this has happened. We include it because it can be very helpful when experimenting with

navigation programs, and trying to figure out why your robot is behaving in an unexpected manner.

- ✓ Go try the [Piezo Beep tutorial](#) [15], then return here when you are done.

Welcome back!

- ✓ Now, adjust your piezo speaker circuit to a new position on your ActivityBot's breadboard, as shown below. This will leave room to add a variety of sensor circuits later on.



- ✓ Re-test your piezo speaker circuit using the program from the Piezo Beep tutorial.

Well done! Now you are ready to go on to robot navigation.

- ✓ Follow the links below to continue with the ActivityBot tutorials.

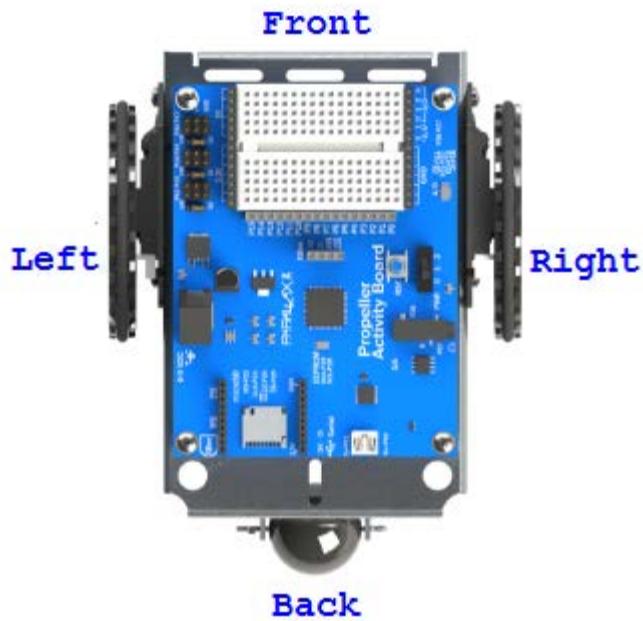
Navigation Basics

This section shows you how to make your ActivityBot move. It covers:

- Calibrating your servo/encoder system
- Testing and tuning the system for optimal performance
- How to drive a specific distance
- How to drive a specific speed.

Get Oriented

From here forward, we will be talking about the robot's right and left sides, and its front and back. These terms are from the perspective of a tiny person sitting on the white breadboard, with a hand on each post and feet dangling past the edge of the board.



Now you are ready to roll!

Get the Example Code

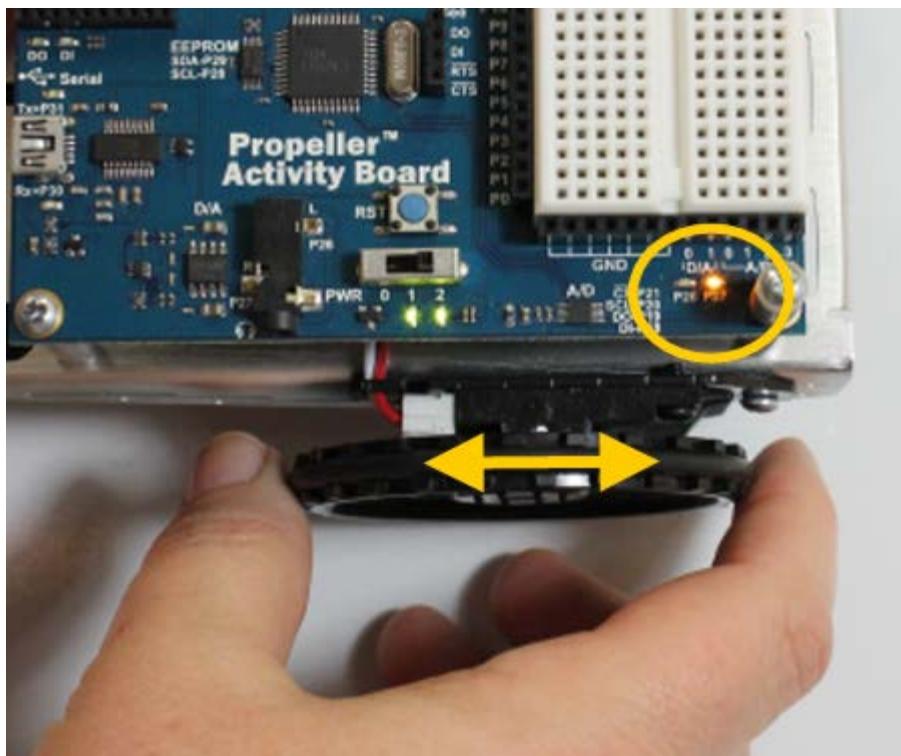
- ✓ Make sure you have the latest [SimpleIDE software, Learn folder, and ActivityBot Library](#) [16].
- ✓ [Download the Navigation Basics Code \(updated 2013-11-18\)](#) [17]
- ✓ Unzip the folder, and copy the contents to Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Follow the links below to continue with the ActivityBot tutorial.

Test the Encoder Connections

This short test program makes sure each encoder sensor can see the spokes in the wheel right next to it. If the encoder is working properly, a built-in LED on the board will go each time it detects a spoke, and go off when the hole between the spokes passes by.

- ✓ Browse to [Documents/SimpleIDE/Learn?Examples/ActivityBot](#)

- ✓ In SimpleIDE, open Test Encoder Connections.side
- ✓ Turn the robot's PWR switch to Position 2
- ✓ In SimpleIDE, click Load EEPROM & Run.
- ✓ With your hand, gently twist the ActivityBot's right wheel. This should make the P27 LED turn on and off as you rotate the wheel.
- ✓ Repeat for the left wheel. Turning the left wheel should make the P26 LED turn on and off.



```
/*
Test Encoder Connections
*/
#include "simpletools.h"

int main()
{
    low(26);
    low(27);
    while(1)
    {
        set_output(26, input(14));
        set_output(27, input(15));
    }
}
```

If It Doesn't Work...

Here are some symptoms and causes:

P26 Light stays off while turning the right wheel.

- The right encoder cable may be plugged into the P14 servo port backwards.
- 20 k resistor (red-black-brown) may not be making contact at either the P14 or 3.3 V socket.

P27 Light stays off while turning the left wheel.

- The left encoder cable may be plugged into the P15 servo port backwards.
- 20 k resistor (red-black-brown) may not be making contact at either the P15 or 3.3 V socket.

P27 light instead of P26 light blinks while wheel turning the right wheel (or vice versa).

- The encoder cables are swapped! Switch the encoder cables plugged into P14 and P15.

P26 or P27 light stays on while turning wheel.

- Resistor connecting P14 or P15 socket to 3.3 V socket is too small. It should be 20 k-ohm (red-black-orange-gold). This resistor came in the bag with the encoder parts, not with the rest of the resistors in the kit.

The encoder's light comes on most of time, but occasionally flickers off.

- For kits made before October 2013, the encoder bracket may be mounted just out of alignment so it sees the rim below the spokes instead of the spaces in between the spokes. If there is a gap between the bottom of the servo case and the lower edge of the chassis mounting hole, this is likely the problem. It is easily corrected by loosening the servo screws slightly, then re-tightening while at the same time pulling the encoder bracket and servo away from each other.

If Both Encoders Work

Congratulations! It is time to calibrate your ActivityBot.

- ✓ Follow the links below to continue with the ActivityBot tutorial.

Calibrate Your ActivityBot

Before running any other example programs, your ActivityBot needs to be calibrated. This is a one-time calibration that the abdrive library needs for measuring and correcting distances and speeds, using information from the ActivityBot encoders.

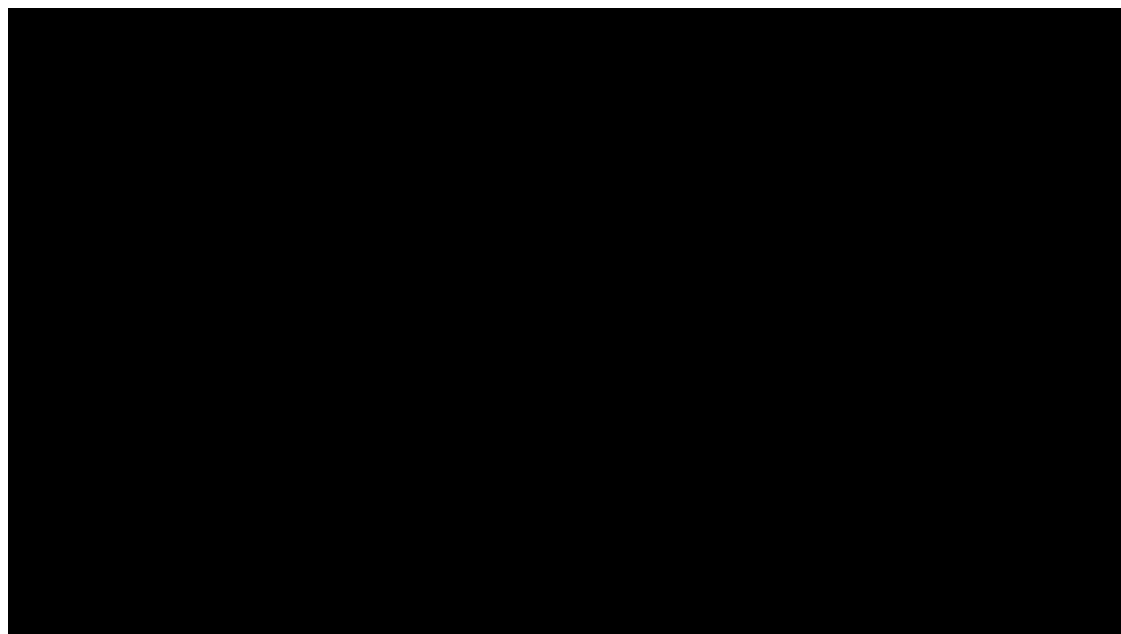
The calibration collects requested speed vs. measured speed data and stores it in a part of the ActivityBoard's EEPROM memory, where it can retain data even after you turn the power off. That way, the calibration data is available every time you turn the robot back on.

When your program asks for a certain wheel speed, the abdrive library will use the calibration data from EEPROM to start driving the motors at speeds close to what your program asks for. That way, abdrive doesn't have to make large corrections, just small ones, which improves overall accuracy.

Circuit

- ✓ If you haven't already, complete the [Electrical Connections](#) [6] first, then return here.

Test Code



The calibration program takes less than 2 minutes to collect all of its data. You will need a smooth and obstacle-free floor area that's roughly one meter square. While the calibration program is running, the ActivityBot will go in circles at various speeds, using only one wheel at a time. It will let you know when its done by turning off its P26 and P27 lights (below the breadboard). The above video shows an ActivityBot correctly performing the calibration.

IMPORTANT! Your ActivityBot Library needs to be installed in the correct location **BEFORE** attempting to run this, or any other, ActivityBot program.

The ActivityBot library folder should be in





...Documents\SimpleIDE\Learn\Simple Libraries and SimpleIDE should be closed and then re-opened before continuing on to allow the software to cache the library location.

Return to [Software and Programming](#) [16] for the library download and additional instructions, if necessary.

- ✓ Set the ActivityBot's power switch to 1
- ✓ Click SimpleIDE's Open Project button
- ✓ Open ActivityBot Calibrate from ...Documents/SimpleIDE/Learn/Examples/ActivityBot Tutorial
- ✓ Click the Load EEPROM & Run button. (**Important:** this program *needs* to be in EEPROM)
- ✓ When the program is finished loading, the P26 and P27 lights will turn on. When they come on, turn off the robot's power (Slide PWR switch to 0)
- ✓ Disconnect the ActivityBot from its programming cable and set it in a 1 meter, obstacle-free, smooth floor area
- ✓ Set the power switch to 2 and move back to give it room to spin in place and slowly roam while it gathers wheel speed data.
- ✓ **Leave it alone until the P26 and P27 lights turn off** (about 2 minutes). After that, calibration is complete and you can turn the power off again.

What if it Didn't Work?

If your robot didn't move when you started the calibration program, or it started going backwards first instead of forwards, or if it started and stopped right away or just twitched, go to the [Troubleshooting](#) [18] page for help.



Re-starting the Calibration Program

If you need to re-start the calibration process, you can push the reset button any time while the P26 and P27 lights are on. Once the calibration process is complete, the program image gets modified so that it cannot run a second time. This keeps it from trying to re-calibrate the next time you turn power on to load a new program. If you want to re-run the calibration, you will need to use SimpleIDE to load ActivityBot Calibrate into EEPROM again.

How it Works

This example program turns on the P26 and P27 LEDs with `high(26)` and `high(27)`. It then calls the abcalibrate library's `cal_activityBot` function, which takes the robot through the sequence of motions you

just observed. While performing the maneuvers, the `cal_activityBot` function builds a list of wheel speeds that correspond to the various drive levels it applies to the servos. It stores those values in EEPROM so that every ActivityBot navigation program can use them to find out what drive level is needed for going a given speed. When the `cal_activityBot` function is done, the program turns off the P26 and P27 lights with `low(26)` and `low(27)`.

```
/*
ActivityBot Calibrate.c
*/
#include "simpletools.h"
#include "abcalibrate.h"

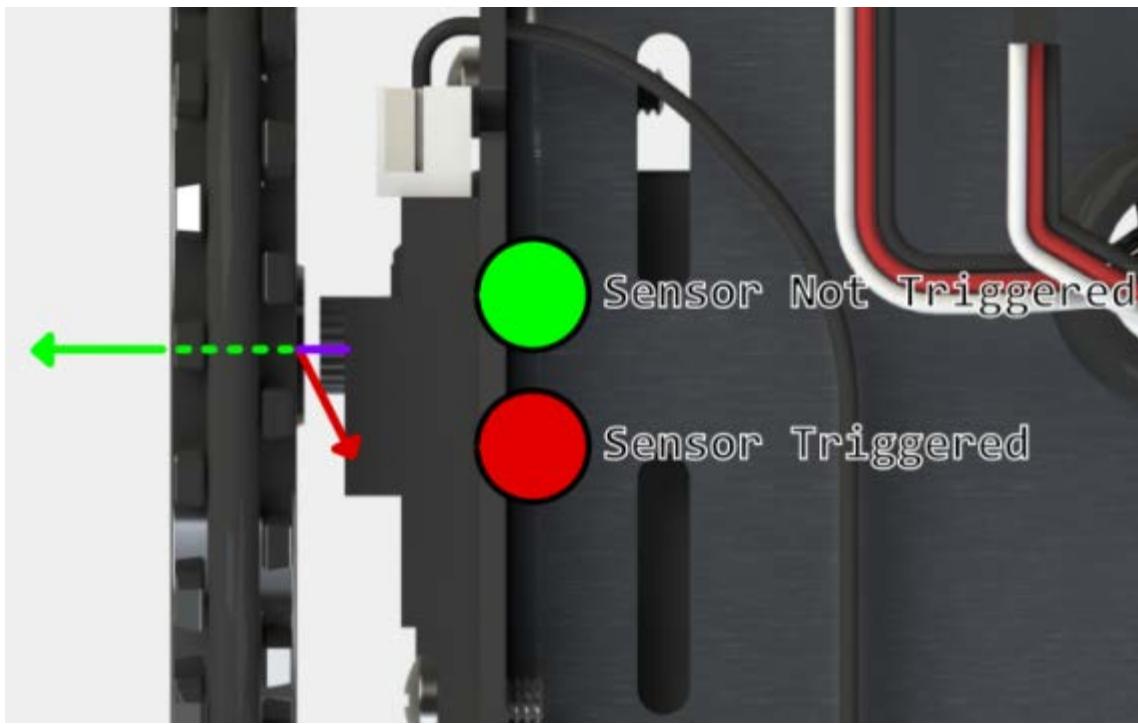
int main()
{
    servo_pins(12, 13);
    encoder_pins(14, 15);

    high(26);
    high(27);
    cal_activityBot();
    low(26);
    low(27);
}
```

Did You Know?

Encoder Ticks - Each ActivityBot encoder shines infrared light at the ring of 32 spokes in the wheel next to it. If the light passes between the spokes, the encoder sends the Propeller a high signal. If it bounces off a spoke and reflects back to the encoder's light sensor, it sends a low signal to the Propeller. Each time the signal changes from high to low, or low to high, the Propeller chip counts it as an *encoder tick*.

Sensing Direction — The Propeller chip knows what direction the servos turn based on the signal it uses to make the servo move. All it needs from the encoder is to know how fast it's turning. It does this by counting encoder ticks over a period of time. The libraries keep track of all this for you, so your programs just need to tell the robot how far or how fast to go.



Interpolation — The `cal_activityBot` function builds a table of data points for motor drive level and the actual speed the wheels turned. Navigation programs will use those data points to figure out how hard to drive a wheel to get a certain speed. If the speed the program asks for is between two data points, the navigation library will figure out the best drive value between two of the known values. For example, if the table has data points for 60 and 80 encoder ticks per second, and your program asks for 70 ticks per second, the navigation library will use a motor drive level that's half way between the 60 and 80 ticks per second levels. This process is called *interpolation* and the data set is called a *linear interpolation table*.

Try This

Want to see the interpolation table? It's the data that shows drive levels and measured wheel speeds in encoder ticks per second.

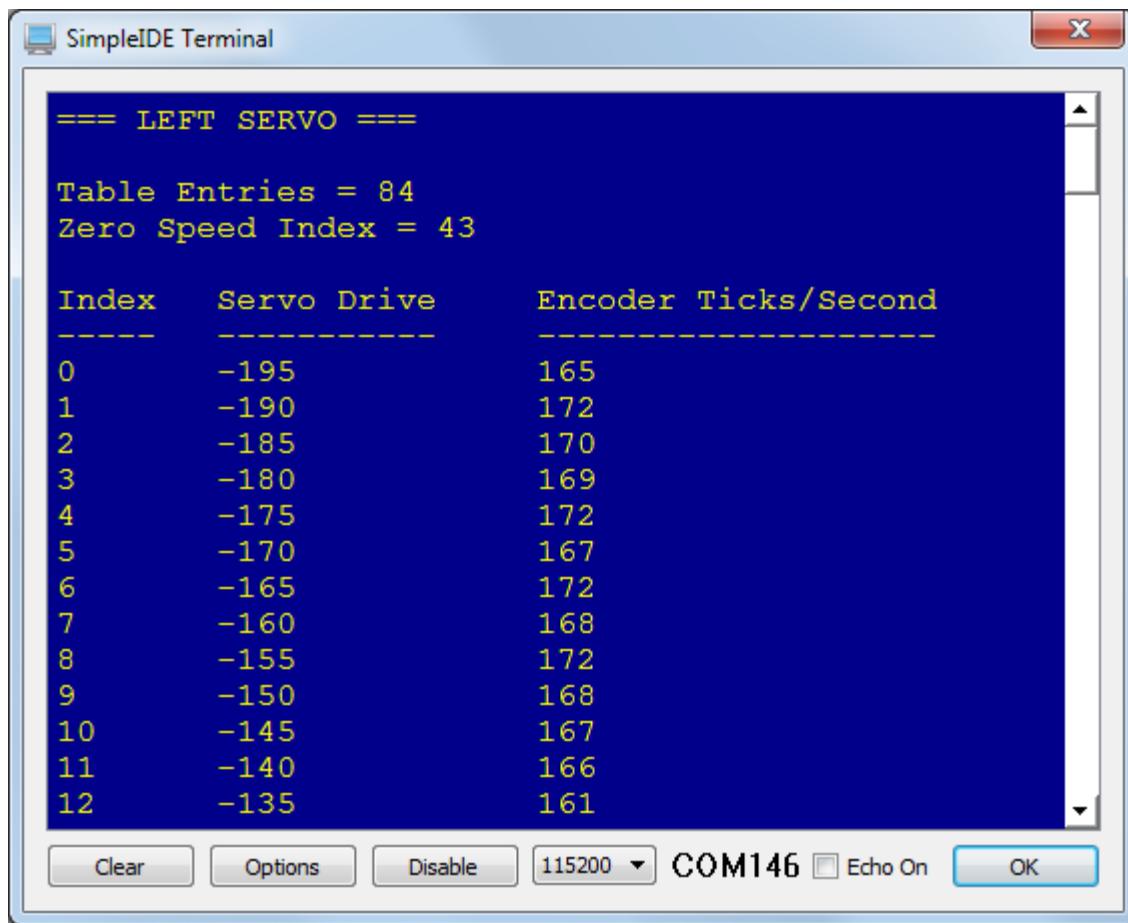
You can use the program below to view it.

- ✓ In SimpleIDE, click New Project.
- ✓ Click Save Project As, use the name ActivityBot Calibration Display, and save it to ...Documents/SimpleIDE/Learn/Examples/ActivityBot Tutorial.
- ✓ Enter the program below.
- ✓ Set the PWR switch to 1.
- ✓ Then, click the Run with Terminal button to see the interpolation table values in the SimpleIDE

terminal.

```
#include "simpletools.h"
#include "abdrive.h"

int main()
{
    drive_displayInterpolation();
}
```



The interpolation table data can be very helpful for [troubleshooting](#) [18] if your robot does not drive as expected after calibration.

Test and Tune Your ActivityBot

Now that your ActivityBot has been calibrated, it is time to run a simple test program to make it drive straight forward. The test code will make both wheels turn at 32 encoder ticks per second for four seconds. This will make the ActivityBot go about 80 cm forward, and both wheels will travel the same

distance at the same speed.

Test Code

- ✓ Click SimpleIDE's Open Project button.
- ✓ Open Test for Trim from ...Documents/SimpleIDE/Learn/Examples/ActivityBot Tutorial.
- ✓ Set the 3-position switch to position-1, then click the Load EEPROM & Run button.
- ✓ After the program is done loading, set the 3-position switch to 0 and disconnect the USB cable.
- ✓ While holding the reset button down, set the 3-position switch to position-2, and set the robot down on a hard, smooth floor.
- ✓ Release the reset button, and monitor the robot's travel.

Your robot should travel forward for about 80 cm in a straight line. You may see slight shifts as the robot adjusts its course based on encoder feedback.

What if it Didn't Work?

If your robot didn't move, or went backwards, or moved in a series of short, jerky arcs instead of heading forward, see the [Troubleshooting](#) [18] page.

What if I want it to go really, really straight?

It is normal for your ActivityBot to be a few centimeters to the right or left of perfectly straight. This can be caused by uneven floor surfaces, or by the sum total of all the slight variances in the robot's mechanical parts. For many robot activities, this does not matter, especially when using sensors to navigate. If you have a robot activity where extra-straight travel is important, you can use the `drive_trimset` function in the optional [Adjusting the Trim](#) [19] activity.

How it Works

Except for `pause`, which is part of simpletools, all the other calls in this program are to functions in the abdrive library.

First, `drive_trimset(0, 0, 0)` clears any trim settings a robot might already have (such as from a previous student in a robotics class).

Next, `drive_speed(32, 32)` sets both wheels to go at a speed of encoder 32 ticks (1/2 a turn) per second. Since it is followed by `pause(8000)`, the servos continue turning at that speed for 8 seconds. Finally, `drive_speed(0, 0)` stops the servos.

```
#include "simpletools.h"
```

```
#include "abdrive.h"
int main()
{
    drive_trimSet(0, 0, 0);
    drive_speed(32, 32);
    pause(8000);
    drive_speed(0, 0);
}
```

Did You Know?

Updates — The ActivityBot updates the motor speeds 50 times per second, based on encoder feedback, to correct any small differences between how far it *should have* turned and how far it *actually* has turned.

Encoder Ticks and Distance — Each encoder tick makes the wheel travel 3.25 mm forward. Remember, an encoder tick is counted when the encoder sensor detects a transition from spoke to hole or hole to spoke. Since there are 32 spokes and 32 holes, there's a total of 64 encoder ticks per wheel turn.

Ramping — Ramping is the term for gradually increasing or decreasing wheel speed so the ActivityBot can ease into and out of maneuvers, instead of starting and stopping abruptly.

Try This

Want to go the same distance, but twice as fast?

- ✓ Click on Save Project As, rename the program Go Faster.
- ✓ Modify the code as shown below.
- ✓ Put the PWR switch to 1, then click Load EEPROM & Run.
- ✓ Set PWR to 0, and disconnect from the USB cable.
- ✓ Now set the robot down again while holding the reset button, set PWR to 2, then let go and watch it drive.

```
#include "simpletools.h"
#include "abdrive.h"

int main()
{
    // drive_trimSet(0, 0, 0);    <-- delete
    drive_speed(64, 64);        // <-- modify
    pause(4000);                // <-- modify
    drive_speed(0, 0);
}
```

Your Turn

The program below lets you see the concept of ramping, both as code and with the robot in action. A `for` loop and a variable named `speed` are used to gradually increase and then decrease the rate at which the wheels turn. The abdrive library also has a built-in functions for ramping, but we'll use that in a later activity.

- ✓ In SimpleIDE, click Project and then New, and give this program a name.
- ✓ Enter the code below, and then load it into EEPROM.
- ✓ Can you see robot's speed gradually increase, then decrease?

```
#include "simpletools.h"
#include "abdrive.h"

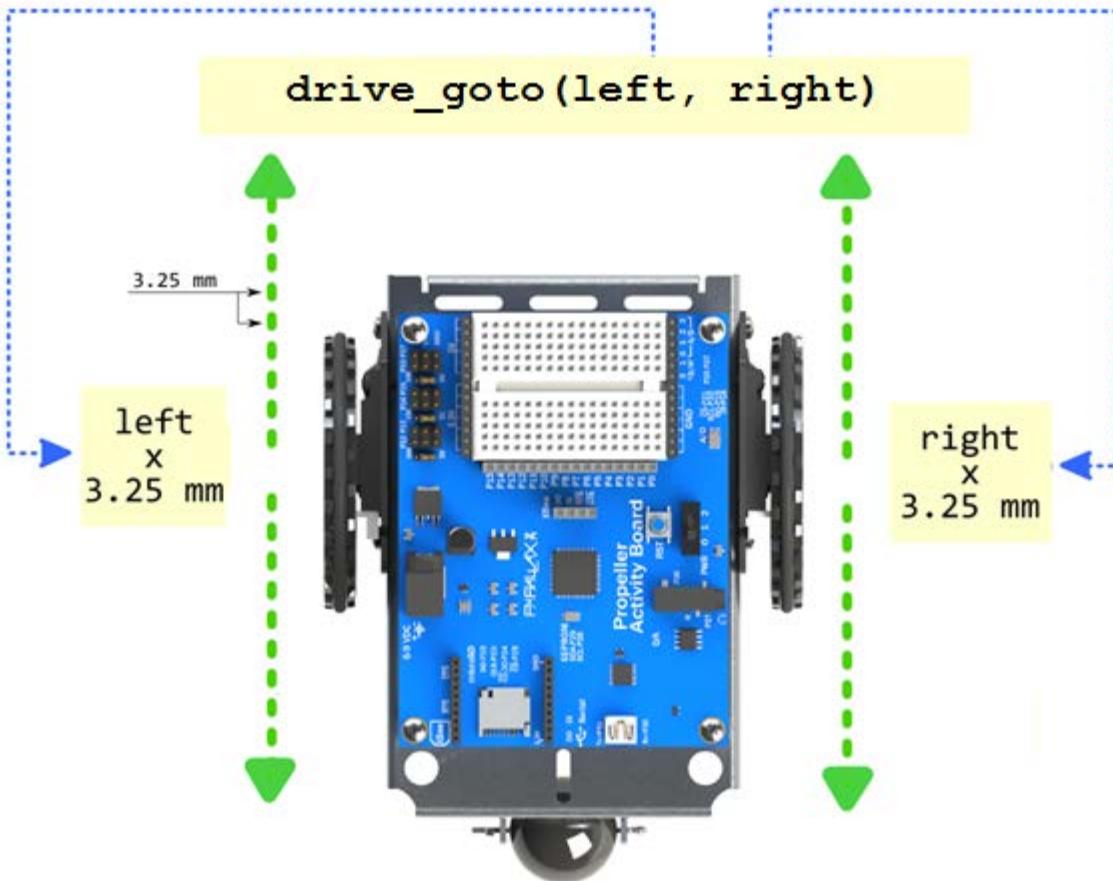
int main()
{
    for(int speed = 0; speed <= 128; speed += 2)
    {
        drive_speed(speed, speed);
        pause(20);
    }

    pause(2000);

    for(int speed = 128; speed >= 0; speed -= 2)
    {
        drive_speed(speed, speed);
        pause(20);
    }
}
```

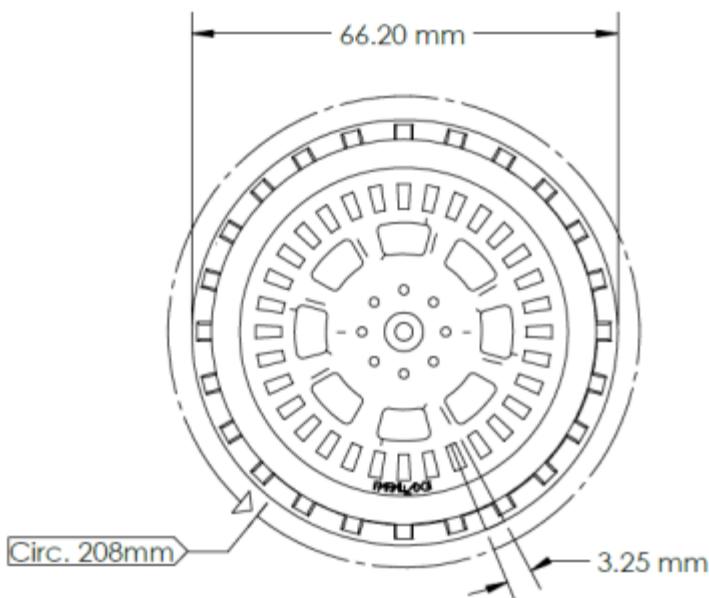
Go Certain Distances

Although it's possible to go a certain distance by choosing a speed and calculating the time, there is an easier way. And, the coding can get complicated if you want to add ramping at the start and stop. Fortunately, the abdrive library has a function that takes care of all that. It's called `drive_goto`, and you can use it to tell the ActivityBot how far each wheel should turn in terms of 3.25 mm increments.



Straight Lines, Turns, and Encoder Ticks

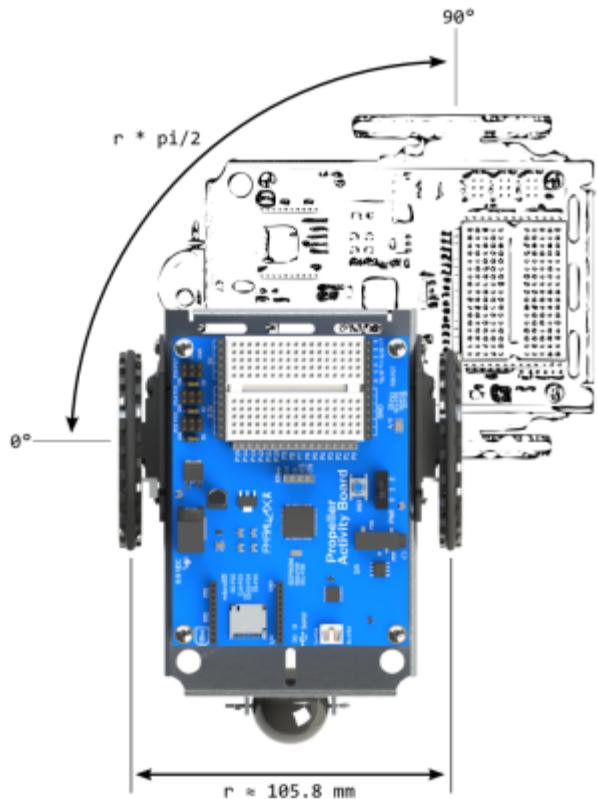
Recall that the term “tick” indicates a transition from either spoke detected to hole detected, or vice-versa. The Propeller ActivityBot wheel has 32 spokes, separated by 32 spaces, for a total of 64 ticks. If the wheel turns $1/64^{\text{th}}$ of a turn, (that’s a tick’s worth), it will travel 3.25 mm.



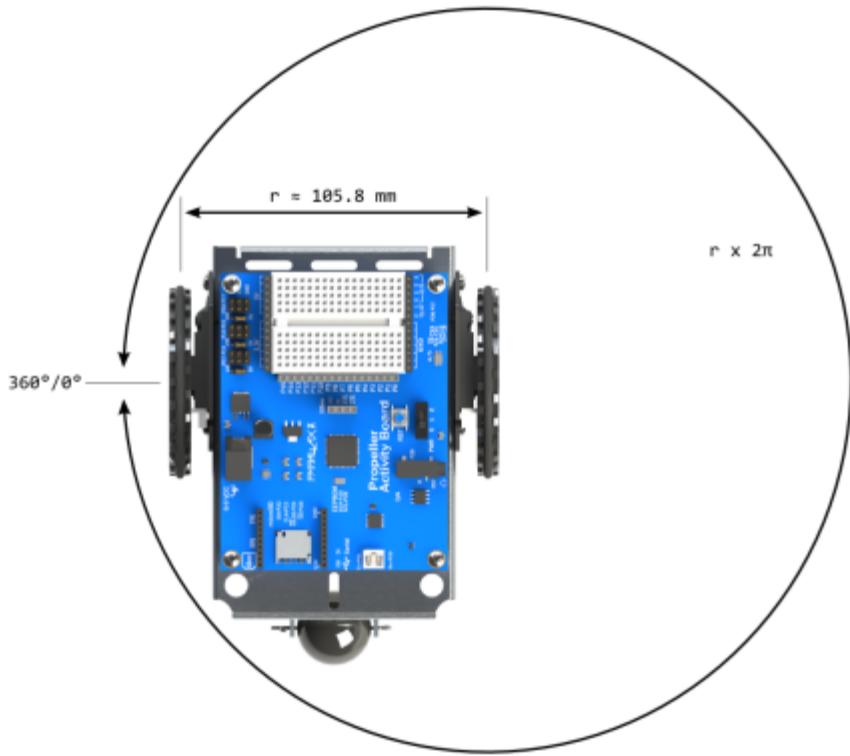
If you know how far you want your ActivityBot to roll, just divide the distance by 3.25 mm (or 0.325 cm or 0.00325 m) to find out how many ticks your program needs to tell the ActivityBot to travel.

$$\text{ticks} = \text{distance mm} \div 3.25 \text{ mm/tick}$$

The ActivityBot's turning radius is typically 105.8 mm.



If you hold the right wheel still and make the left wheel turn, it will have to turn $2 \times \pi \times r$. In this case, the r is the turning radius, so that's $2 \times \pi \times 105.8 \text{ mm} \approx 664.76 \text{ mm}$. If you want the ActivityBot to make a $\frac{1}{4}$ turn, that would be $664.76 \text{ mm} \div 4 \approx 166.2 \text{ mm}$. How many wheel ticks is that? $166.2 \text{ mm} \div 3.25 \text{ mm/tick} \approx 51.14 \text{ ticks} \approx 51 \text{ ticks}$.



TIP: If you make one wheel go forward 26 ticks, and the other go backwards 25 ticks, that's still a total of 51 ticks for $\frac{1}{4}$ of a turn.

Test Code

This test code makes the ActivityBot go forward 256 ticks, and then turn and face 90-degrees to the right.

- ✓ Click SimpleIDE's Open Project button.
- ✓ Open Forward Stop Face Right.side from ...Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Set the PWR switch to 1.
- ✓ Click the Load EEPROM & Run button. Set the PWR to 0 after loading has finished.
- ✓ Disconnect from programming cable, set the ActivityBot on the floor and switch the PWR to position 2.
- ✓ Verify that the ActivityBot goes forward four wheel turns, stops, and faces right.

How it Works

The program makes the ActivityBot go forward by 256 ticks (four wheel turns). Then, `drive_goto(26, -25)` makes the ActivityBot execute a 90-degree right turn.

```
#include "simpletools.h"
#include "abdrive.h"

int main()
{
    drive_goto(256, 256);
    pause(200);
    drive_goto(26, -25);
}
```

Did You Know?

Overshoots — If the `drive_goto` function overshoots, it backs up to the correct number of ticks.

Adjusted for Trim? — If you have `trim` set, the forward distance is for the wheel that goes less far.

Try This

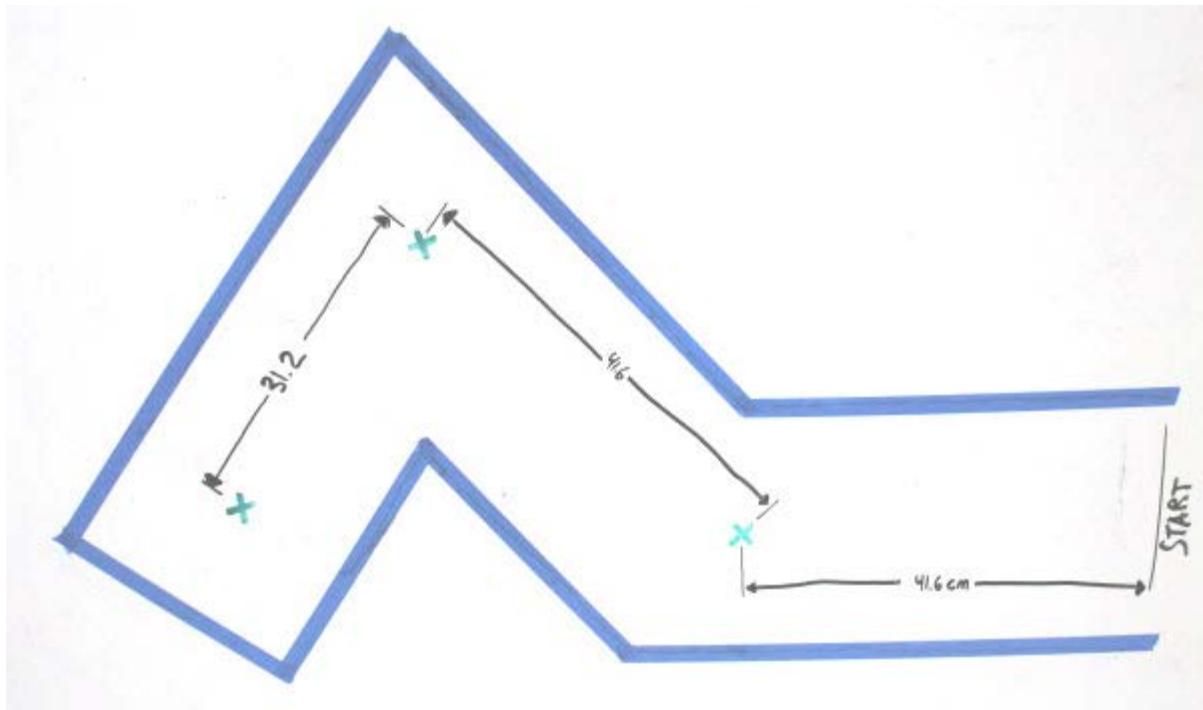
Expand the program to make the ActivityBot retrace its path back to where it started, after going forward and turning right. There are two ways you could accomplish this:

- ✓ Have your ActivityBot turn left by 90 degrees and then reverse back into its starting position.
- ✓ Turn an additional 90 degrees to the right, and drive forward into its original starting position.

Your Turn

Simple waypoint navigation challenges involve programming your robot to drive a pre-determined course. An example course is shown below. To run this course, you would need to program your ActivityBot for the following manuevers (this will take a bit of math):

1. From START, go straight 41.6 cm forward to X.
2. Turn 45 degrees right, and 41.6 cm to the next X.
3. Turn 90-degrees left, and go 31.2 cm to finish X.



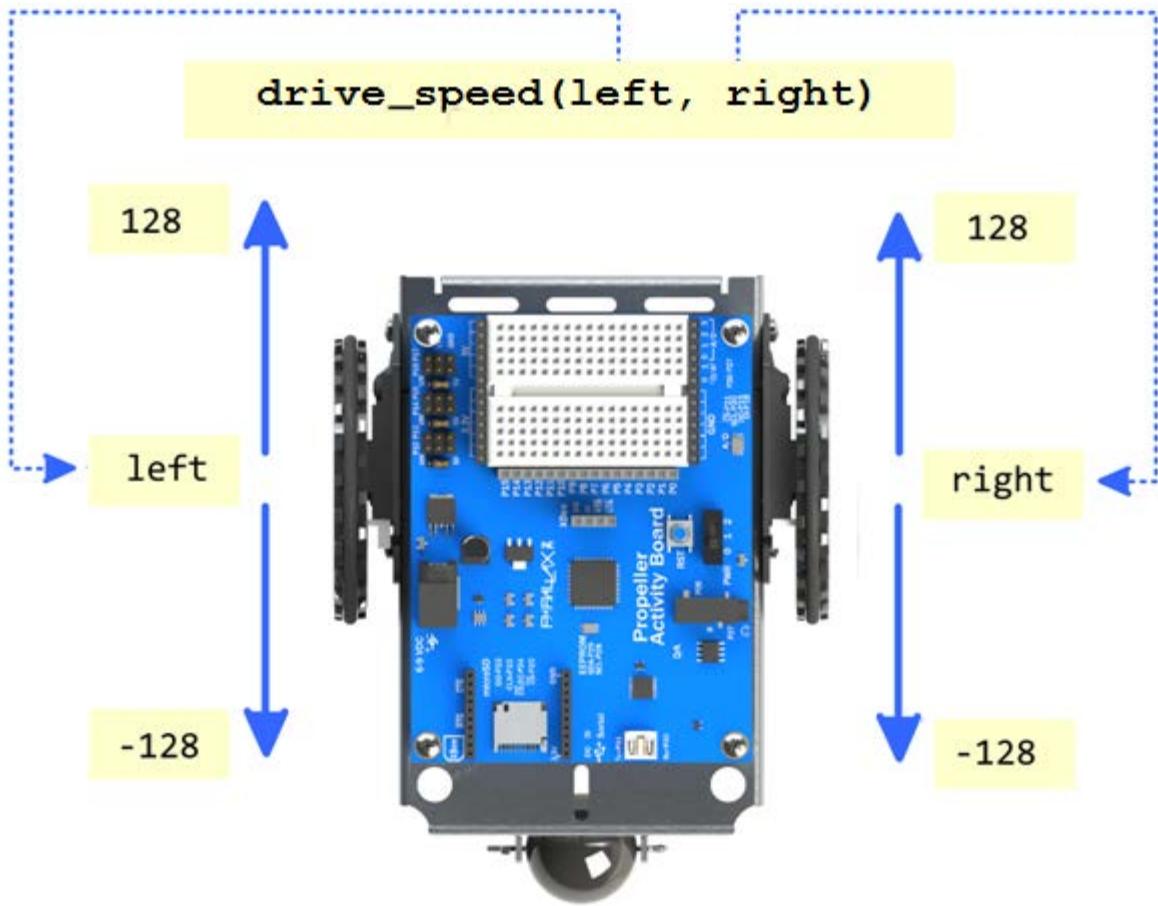
- ✓ Make a waypoint maze — masking tape corridors work well — and write a program to navigate from start to finish.

Set Certain Speeds

Sometimes your ActivityBot might need to go a certain direction until it runs into an obstacle. Setting the robot to go a certain speed, instead of a certain distance, is helpful for that kind of application.

We can do this with the abdrive library's `drive_speed` function. It sets each drive servo to a certain speed, in encoder ticks per second. Since there are 64 ticks in a wheel turn, if you set a wheel speed to 64 ticks per second, that's one full turn per second. Use positive values for forward, and negative values for reverse.

We'll also try a `drive_ramp` function that lets the ActivityBot speed up and slow down gradually. This prevents its wheels from losing traction, and helps keep the robot from tipping forward from abrupt stops.



Speed Control Example

This example sets the robot to go straight ahead at 64 ticks/second until it has gone more than 200 ticks forward. Then it executes a 45-degree right turn with the left wheel turning 32 ticks per second. After that, it continues for another 200+ ticks at 128 ticks per second.

- ✓ Click SimpleIDE's Open Project button.
- ✓ Open Speeds for Navigation.side from ...Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Click the Load EEPROM & Run button.
- ✓ Verify that it goes forward 200 (or maybe a few more) ticks, turns right by about 45-degrees, and goes 200 + ticks again.

How it Works

The program sets out by setting both servos to turn at 64 ticks per second with `drive_speed(64, 64)`. Since 64 ticks per second is one full wheel revolution per second, and since a `pause(2000)` follows it, we know the wheels should both turn at 64 ticks per second for two seconds. In terms of distance, that should

be 64 ticks/second \times 2 seconds = 128 ticks. Keep in mind that this may not be precise because the robot will have some momentum when it gets to `drive_speed(0, 0)`. Also keep in mind that if you do want a more precise distance traveled, just use the `drive_goto` function.

```
#include "simpletools.h"                                // simpletools library
#include "abdrive.h"                                    // abdrive library

int main()
{
    drive_speed(64, 64);                                // Forward 64 tps for 2 s
    pause(2000);
    drive_speed(0, 0);

    drive_speed(26, 0);                                 // Turn 26 tps for 1 s
    pause(1000);
    drive_speed(0, 0);

    drive_speed(128, 128);                             // Forward 128 tps for 1 s
    pause(1000);
    drive_speed(0, 0);
}
```

Next, `drive_speed(26, 0)` makes the left wheel turn at about 26 ticks per second, while the right wheel stays still. Since it's going for 1 second, the left wheel should turn about 26 ticks for a 45-degree turn. `drive_speed(0, 0)` stops it after the 1 second. The last maneuver should go about the same distance as the first maneuver, but it's going twice as fast (128 ticks per second instead of 64) for half the time (1 second instead of 2 seconds).

Did You Know?

Velocity is speed, but it can be either positive (like forward) or negative (like backward). Speed itself is just velocity without the sign. So, if you are going backwards at a speed of 5 km/hr, you could also call that a velocity of -5 km/hr.

Here is an equation you can use to calculate distance (s) given velocity (v) and time (t).

$$s = v \times t$$

Example: A car goes 40 km/hr, how many km does it travel in 0.25 hours? Answer: $s = v \times t = 40 \text{ km/hr} \times 0.25 \text{ hr} = 10 \text{ km}$.

Applied to the ActivityBot, if a wheel goes 64 ticks per second for 2 seconds, that's $v = 64 \text{ ticks per second}$ and $t = 2 \text{ seconds}$. So:

$$s = 64 \text{ ticks/second} \times 2 \text{ seconds} = 128 \text{ ticks.}$$

For a distance in centimeters, remember that a tick is 0.325 cm, so the wheel travels:

$$s = 128 \text{ ticks} \times 0.325 \text{ cm/tick} = 41.6 \text{ cm.}$$

If you divide both sides of $s = v \times t$ by v, you get:

$$t = s \div v$$

...and that's really useful for picking a time to go a certain direction if you know the speed and distance you want the ActivityBot to go.

Example: The ActivityBot needs to go 192 ticks forward at a rate of 64 ticks per second.

- How long should that take? Answer: $t = 192 \text{ ticks} \div 64 \text{ ticks/second} = 3 \text{ seconds}$.
- How long of a `pause` is that? Answer: The `pause` function is in milliseconds (thousandths of a second), so multiply 3 by 1000. Your program would need `pause(3000)`;

Try This

If you noticed some sudden starts and stops in the previous example programs, these can be cushioned with `drive_ramp`. This function gradually speeds up or slows down the wheels.

- ✓ Try replacing `drive_speed` with `drive_ramp`, re-run and observe the difference. Note that it'll go a little further because `drive_ramp` takes some time to reach cruising speed.

```
#include "simpletools.h"
#include "abdrive.h"

int main()
{
    drive_ramp(64, 64);                                // <-- modify
    pause(2000);
    drive_ramp(0, 0);                                  // <-- modify

    drive_ramp(26, 0);                                 // <-- modify
    pause(1000);
    drive_ramp(0, 0);                                  // <-- modify

    drive_ramp(128, 128);                            // <-- modify
    pause(1000);
    drive_ramp(0, 0);                                  // <-- modify
}
```

You can also use `drive_speed` to make the ActivityBot travel in curves.

- ✓ Try setting one wheel to 80 ticks/second and the other one to 60 ticks/second. What happens?

```
#include "simpletools.h"
#include "abdrive.h"

int main()
{
    drive_speed(80, 60);
    pause(2000);
    drive_speed(0, 0);
}
```

Your Turn

If your program ever needs to know how many ticks each wheel has traveled, you can use the `drive_getTicks` function. Here is an example that records the number of ticks traveled before and after the Try This program that curved to the right. Use Run with Terminal, and leave your robot plugged in. (Try setting the robot on a stand or hold it above the desk so that it doesn't go too far while attached to the USB cable.)

Your challenge is to add two more maneuvers and display the distance traveled for each maneuver.

Hint: You can calculate distance for each maneuver by using something like: `distanceLeft = distLeft[1]`

- `distLeft[0];`

```
#include "simpletools.h"
#include "abdrive.h"

int distLeft[4], distRight[4];

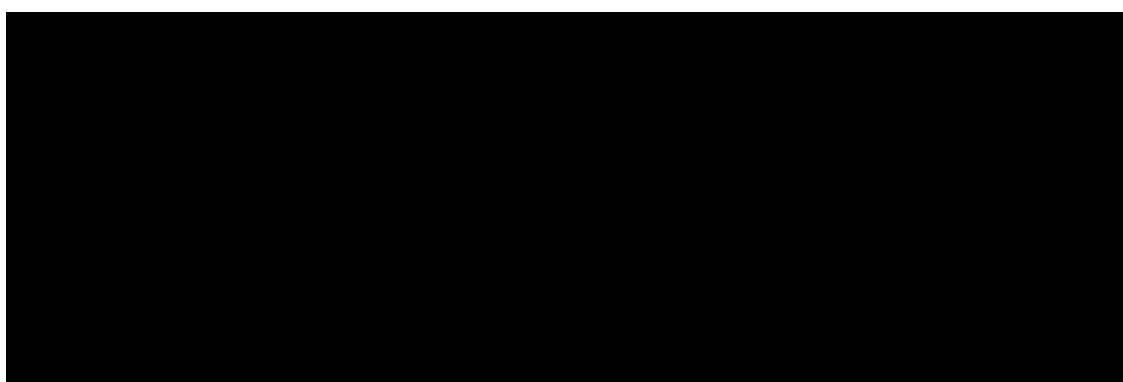
int main()
{
    drive_getTicks(&distLeft[0], &distRight[0]);
    print("distLeft[0] = %d, distRight[0] = %d\n", distLeft[0], distRight[0]);
    drive_speed(80, 60);
    pause(2000);
    drive_speed(0, 0);

    drive_getTicks(&distLeft[1], &distRight[1]);
    print("distLeft[1] = %d, distRight[1] = %d\n", distLeft[1], distRight[1]);
}
```

Navigate by Touch

Detect Obstacles by Bumping into Them

Whisker sensors allow your robot to detect obstacles by bumping into them. When the robot's whisker bumps into an object, it makes an electrical connection between the whisker and a circuit on the breadboard. In this lesson, you will program your robot's Propeller microcontroller to detect those contacts and make the robot avoid obstacles.





Get the Example Code

- ✓ Make sure you have the latest [SimpleIDE software](#), [Learn folder](#), and [ActivityBot Library](#) [16].
- ✓ [**↓ Download the ActivityBot Navigate by Touch code**](#) [20]
- ✓ Unzip the folder, and copy it to Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Follow the links below to continue with the ActivityBot tutorial.

Build the Whiskers

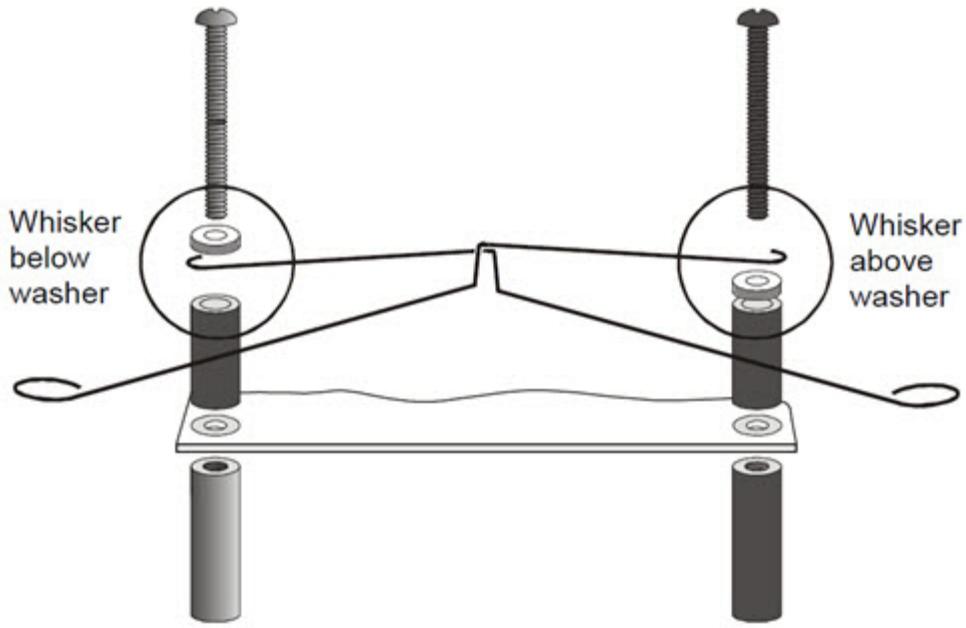
Parts List

- (2) Whisker wires
- (2) 3-pin headers
- (2) 10 k Ω resistors (brown-black-orange)
- (2) 220 Ω resistors (red-red-brown)
- (misc) Jumper wires



Adding the Whiskers

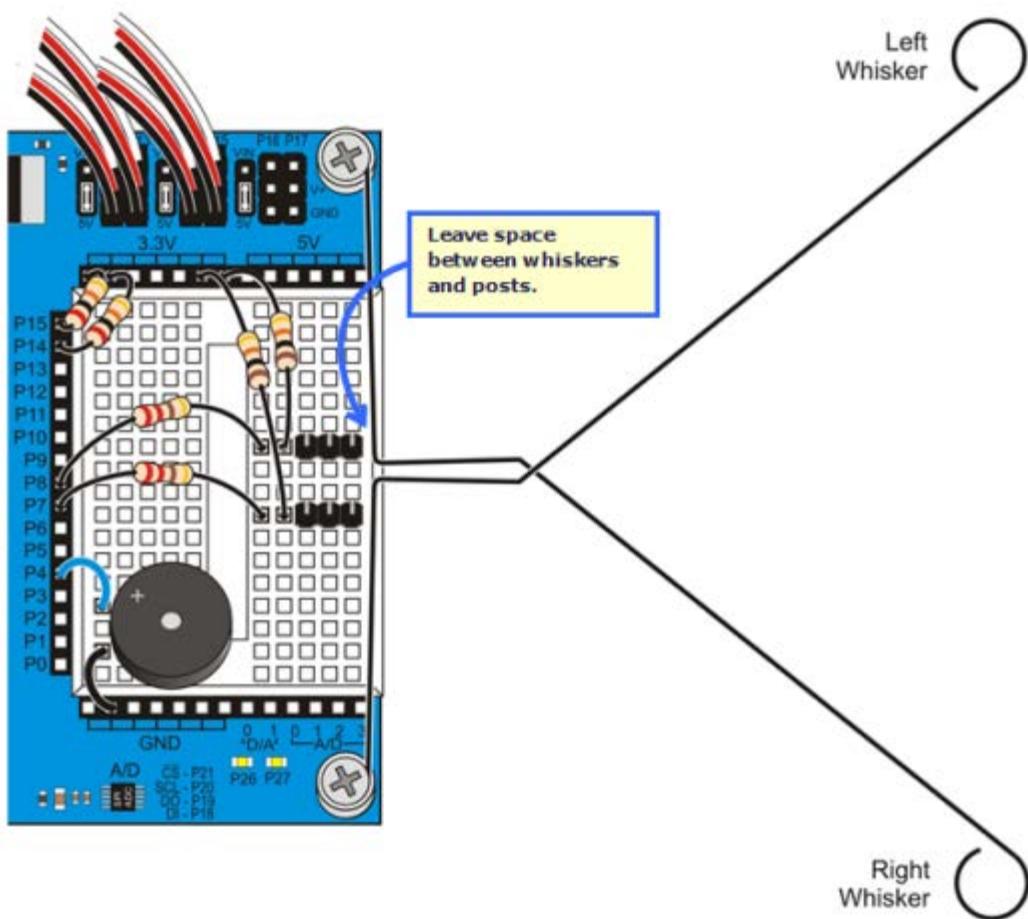
- ✓ Loosen the two screws that attach the front of your board to the chassis standoffs.
- ✓ Hook a whisker wire onto each screw, positioned as shown in the diagram below.
- ✓ Make sure to set one whisker above the nylon washer, and the other below, as shown here.
- ✓ Gently re-tighten the screws, making sure the edge of the wire is parallel to the edge of the breadboard.

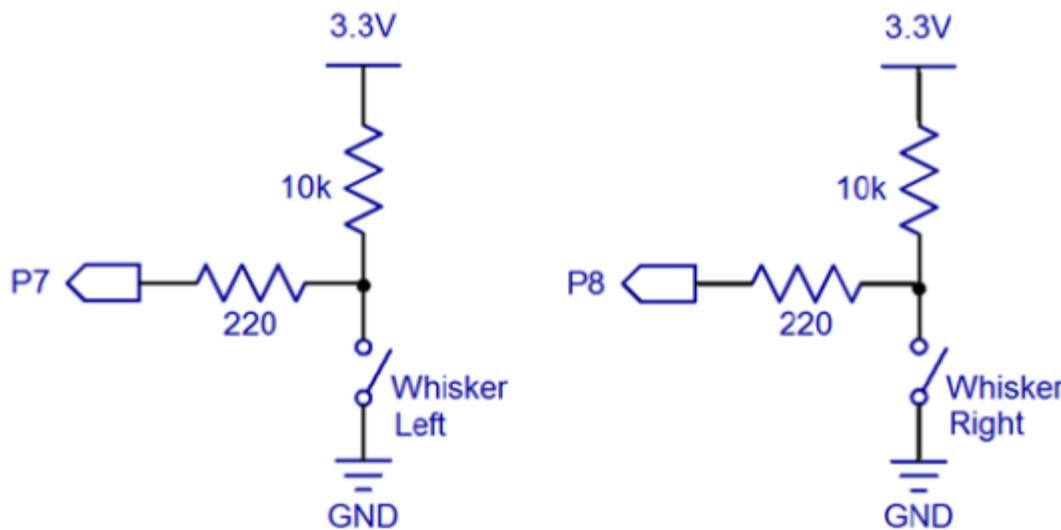


Build the Circuit

The next picture shows whisker wiring. Along with the whisker circuits, it has the same piezospeaker we've been using.

- ✓ Build the circuit as in the picture.
- ✓ Make sure to leave some space between the whiskers and the 3-pin headers.
- ✓ Also make sure that the $10\text{ k}\Omega$ resistors (brown-black-orange) are the ones that connect 3.3 V sockets to the rows with the 3-pin headers.
- ✓ Likewise, the $220\ \Omega$ (red-red-brown) resistors should connect P8 and P7 to the 3-pin header rows.





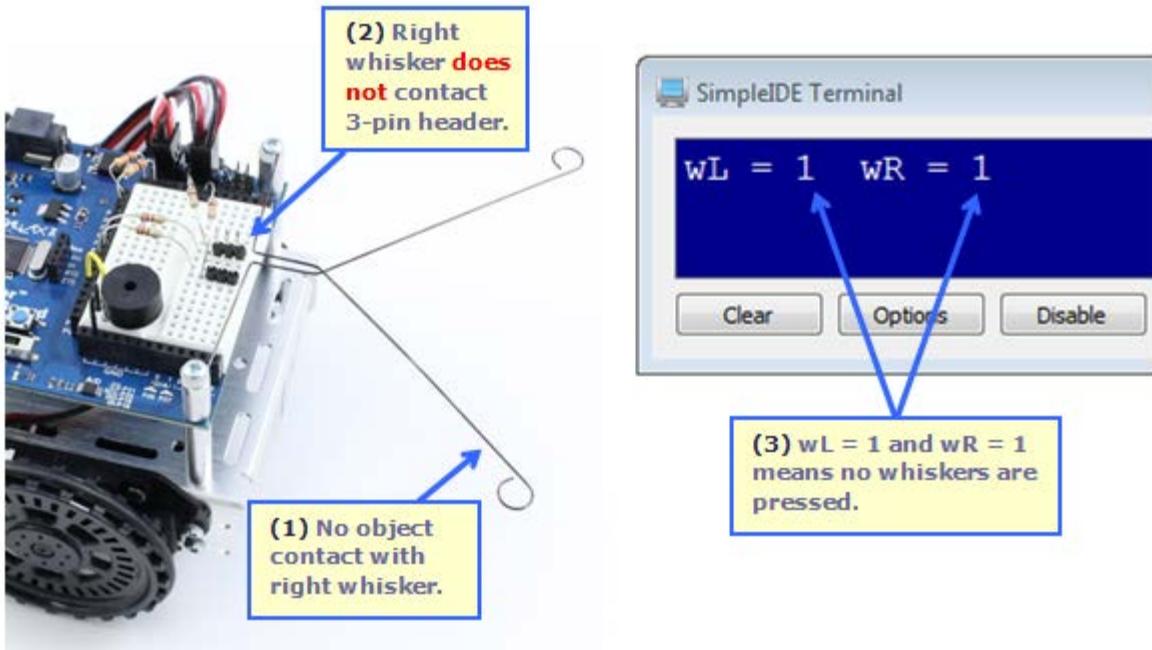
Test the Whiskers

It's important to test the whisker contacts before programming your robot to navigate with them. That way, you can be sure the whiskers are sending the right messages. Otherwise, they might be telling the Propeller that they detect something even though they didn't, or vice-versa.

- ✓ Make sure the robot's battery pack is plugged into the board.
- ✓ Set the PWR switch to 1.
- ✓ Click SimpleIDE's Open Project button.
- ✓ Navigate to SimpleIDE > Learn > Examples > ActivityBot Tutorial.
- ✓ Open Test Whiskers with Terminal.side.
- ✓ Click the Run with Terminal button. Without pressing any whiskers, it should display $wL = 1$ $wR = 1$.

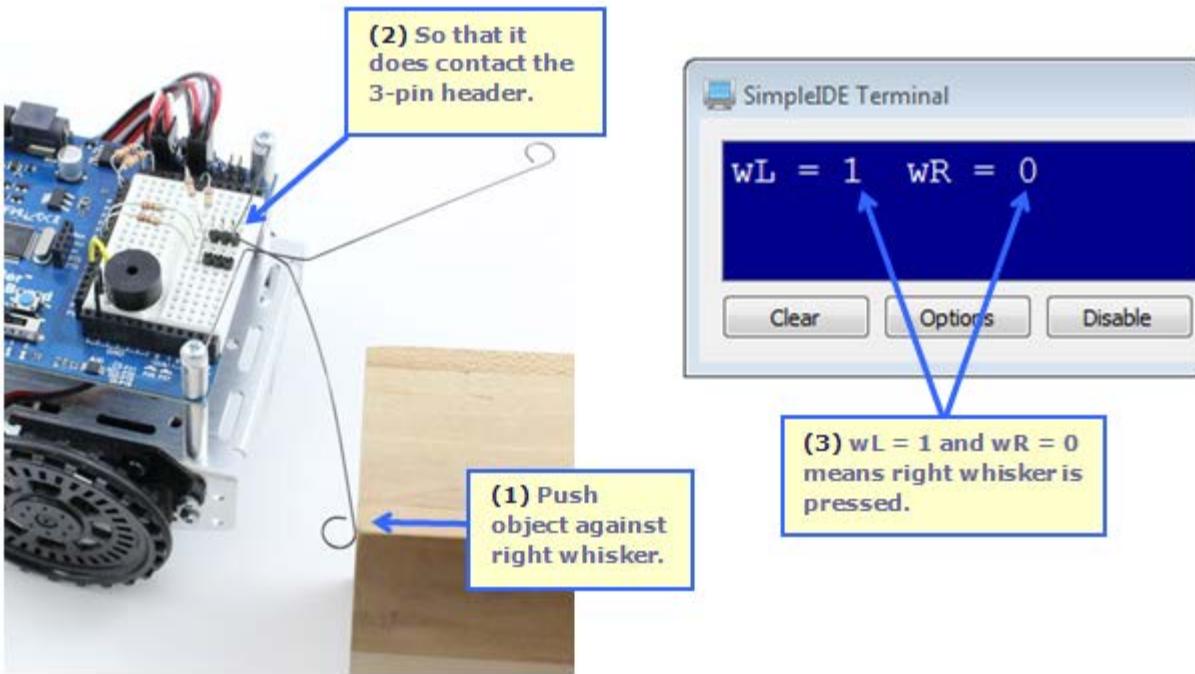
If either of the numbers are zeros, or if either of them flicker between 1 and 0, there's probably a circuit error.

- ✓ If you have a circuit error, go back and double check to make sure all of your connections are correct.



Next, let's check to make sure the Propeller chip detects when the right whisker is pressed or makes contact with an obstacle.

- ✓ Press the right whisker so that it makes contact with the 3-pin header's front post (see 2 in the picture).
- ✓ Verify that the SimpleIDE Terminal displays $wL = 1$ $wR = 0$, which means that the right whisker is pressed.
- ✓ If the SimpleIDE Terminal displays anything other than a steady $wL = 1$ $wR = 0$ while you have the right whisker pressed, find and fix your circuit error.
- ✓ Repeat for the left whisker. You'll want to see $wL = 0$ $wR = 1$ in the Parallax Serial Terminal. Make sure it displays that way. If it doesn't, find and fix any circuit errors.
- ✓ If you press and hold both whiskers against posts, the Parallax Serial Terminal should display $wL = 0$ $wR = 0$. Try that too.



How it Works

The `main` function starts with our now-familiar reset indicator code: `freqout(4, 2000, 3000)` makes the speaker connected to P4 beep for 2 seconds at a frequency of 3 kHz.

After that, the program enters a `while(1)` loop that repeats endlessly. Inside that loop, `int wL = input(7)` copies the value that `input(7)` returns into an `int` variable named `wL` (short for whisker-Left). The `input(7)` call might return a 1, which would indicate that the left whisker is not pressed. Or, it might return a 0, indicating that it is pressed. In either case, that 1 or 0 value gets copied to the `wL` variable. The `int wR = input(8)` statement works roughly the same way, except that it stores the 1 or 0 value for the right whisker circuit, which is connected to P8, into a variable named `wR` (short for whisker-Right).

```
/*
Test Whiskers with Terminal.c
Display whisker states in terminal. 1 = not pressed, 0 = pressed.

*/
#include "simpletools.h"                                // Include simpletools header
int main()                                              // main function
{                                                       // Speaker tone: P4, 2 s, 3 kHz
    freqout(4, 2000, 3000);                           // Endless loop
    while(1)
    {
        int wL = input(7);                            // Left whisker -> wL variable
        int wR = input(8);                            // Right whisker -> wR variable
        print("%c", HOME);                          // Terminal cursor home (top-left)
        print("wL = %d wR = %d", wL, wR);           // Display whisker variables
        pause(50);                                    // Pause 50 ms before repeat
    }
}
```

After storing the whisker states in the `wL` and `wR` variables, `print("%c", HOME)` sends the cursor to the home position. The `%c` format string tells the `print` function to send the value of a char variable. `HOME` is

the number 1, and when `print` sends that value to the SimpleIDE Terminal, the terminal moves its cursor to the top-left “home” position. After that, `print("wL = %d wR = %d", wL, wR)` displays the values of the `wL` and `wR` variables. This time, the `%d` format string made the `print` function display values as decimal characters, in this case, either 1 or 0 for the whisker states.

Did You Know?

The simpletools library has 16 control characters you can use to do things like clear the screen and position the cursor.

Name	Value	Description
HOME	1	Send cursor to the top-left (home) position
CRSRXY	2	Position the cursor. Follow with x and y values. Example places cursor 10 spaces in and 5 lines down: <code>print("%c%c%c ", CRSRXY, 10, 5);</code>
CRSRLF	3	Move cursor one space to the left
CRSRRT	4	Move cursor one space to the right
CRSRUP	5	Move cursor one line up
CRSRDN	6	Move cursor one line down
BEEP	7	Make the host computer beep
BKSP	8	Backspace
TAB	9	Tab
NL	10	Send cursor to next line
CLREOL	11	Clear text to right of cursor
CLRDN	12	Clear text below cursor
CR	13	Carriage return
CRSRX	14	Position cursor x spaces to the right
CRSRY	15	Position cursor y lines down
CLS	16	Clear the display

Try This

Here is a program you can try that positions the cursor at 10 spaces over and 5 lines down before printing the whisker states.

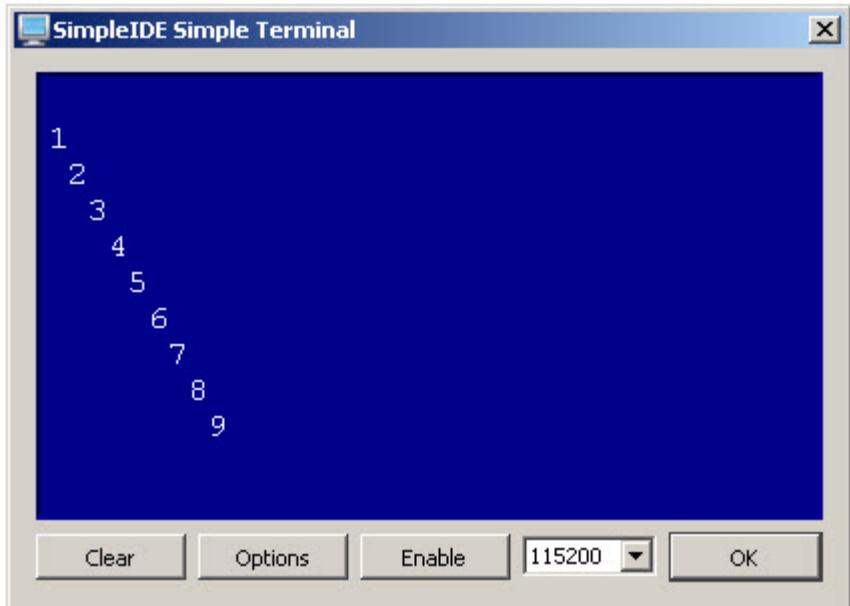
```
#include "simpletools.h"

int main()
{
    freqout(4, 2000, 3000);
    while(1)
    {
        int wL = input(7);
        int wR = input(8);
        print("%c%c%c", CRSRXY, 10, 5);           // <- modify
        print("wL = %d  wR = %d", wL, wR);         // <- add
        print("%c", CLREOL);                      // <- add
        pause(50);
    }
}
```

- ✓ Click the Save As Project button.
- ✓ Name the project Display Whiskers with CRSRXY and save it to the ActivityBot Tutorial folder.
- ✓ Update the `main` function so that it matches the one above.
- ✓ Click the Run with Terminal button and examine the table the SimpleIDE Terminal displays.

Your Turn

- ✓ Write a `for` loop with a `print` call that uses `CRSRXY` to create this output.



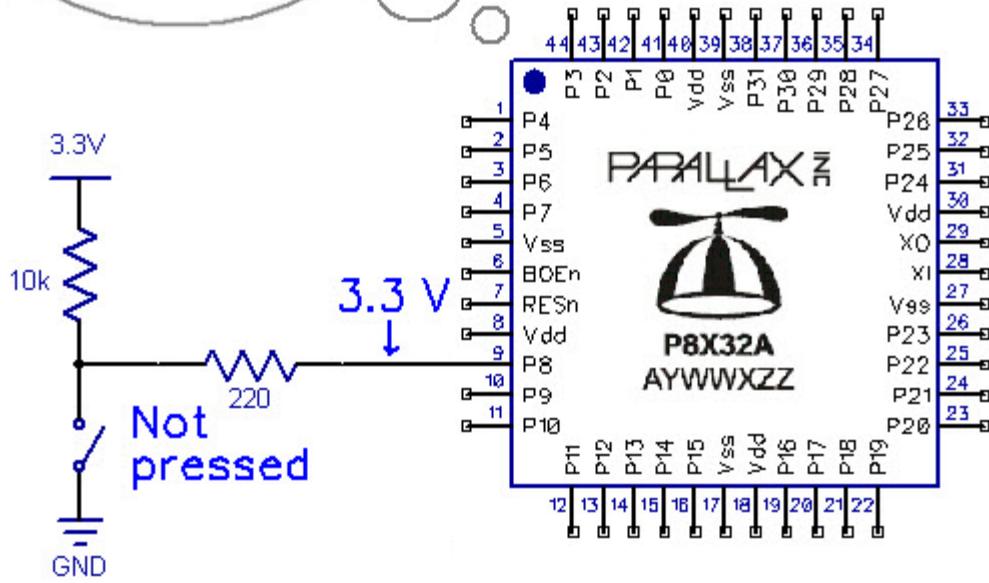
Inside the Whisker Circuit (Optional)

Let's take a look at the right whisker and see what happens when it is not pressed, and then pressed.

When it's not pressed the whisker does not touch the 3-pin header on the breadboard, so there is no electrical contact between the two. Because of that, the $10\text{ k}\Omega$ resistor that's connected to the 3.3 V socket above the breadboard applies 3.3 V to the breadboard row with the 3-pin header post. The $220\ \Omega$ resistor that connects that row to P8 applies the 3.3 V to P8.

The whisker is like a normally open switch. In the schematic, P8 detects 3.3 V through the $220\ \Omega$ and $10\text{ k}\Omega$ resistors. Whenever a circuit applies 3.3 V (or anything over 1.65 V) to P8, `input(8)` returns 1. That's why `wR = input(8)` copied the value of 1 to the `wR` variable (and you saw it displayed in the SimpleIDE Terminal).

`input(8) == 1`



Why 3.3 V at P8? In case you're wondering why the voltage at P8 is the same as the voltage above the 10 k Ω resistor, here's what's going on:

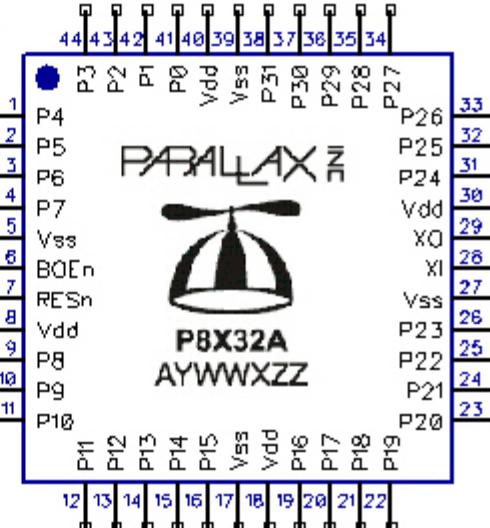
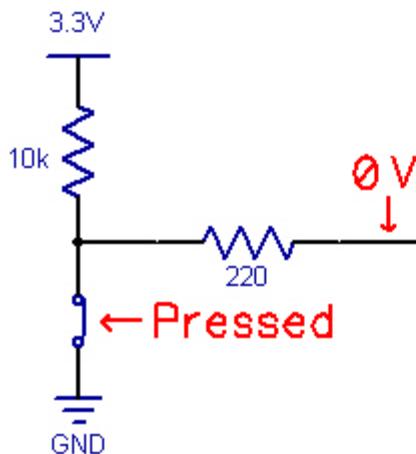
i First of all, the schematic shows an open switch with the "Not pressed" label. That open switch is the whisker, which is not in contact with the 3-pin header, and it keeps ground (GND = 0 V) out of the circuit. So, all we have is 3.3 V on one end of a pair of resistors, and P8 on the other end. Since P8 is set to input, it looks invisible to the circuit. As far as the circuit's concerned, the right side of that 220 Ω resistor might as well be disconnected from everything and sticking up in the air. When only one voltage is applied to a resistor, or even a chain of resistors, that same voltage will appear at the other end as well. So, P8 as an input is invisible to the circuit, but it can detect the voltage the circuit applies to it. In this case, that voltage is 3.3 V, which causes the Propeller to store 1 in its P8 input register, and so `input(8)` returns the value 1.

When the right whisker is pressed because the robot has bumped into an obstacle, then the right whisker makes contact with the front post on the 3-pin header. The whisker is electrically connected to that plated hole on the corner of the board, which eventually makes its way to the battery's negative terminal. That negative terminal is commonly called ground, and has a value of 0 V. Since the whisker touches that 3-pin header post, it connects that row to ground, so P8 sees 0 V through the 220 Ω resistor.

Take a look at this schematic. It shows how the whisker connects that node where the 10 k Ω and 220 Ω resistors meet to ground (0 V). ...and that's what P8 detects. As a result, a call to `input(8)` returns zero. That zero value gets copied to `wR` with `wR = input(8)`, and that's what gets displayed by the Parallax

Terminal Window when the whisker is pressed.

input(8) == 0



Why 0 V at P8? Now that the whisker is pressed, the schematic above shows 0 V applied to P8. That's because pushing the whisker up against the post connects GND (0 V) to the node where the 10 kΩ and 220 Ω resistors meet. Now, instead of 3.3 V, the circuit applies 0 V to P8. Reason being, 0 V is being applied to the left side of the 220 Ω resistor, and the circuit still thinks the right side of that resistor is floating in the air. So, the rule for a high impedance input still applies, and the voltage that's on the right side of the 220 Ω resistor will be the same as the voltage applied to its left side.

Now that there's 3.3 V on one end of the the 10 kΩ resistor and 0 V on the other end, it applies electrical pressure that causes current to flow through that resistor. You can use the Ohm's Law equation of $V = I \times R$ to figure it out. This equation says the voltage (V) difference at two ends of a resistor is equal to the current (I) passing through it multiplied by the resistor's resistance (R). With a little algebra, we have $I = V \div R = 3.3 V \div 10,000 \Omega \approx 0.00033 A$. That's 0.00033 A, or in terms of milliamps, 0.33 mA, a very small amount of current, especially compared about 140 mA, which is what the servos need to make the Propeller ActivityBot move.

The 10 kΩ resistor is called a pull-up resistor. It pulls the voltage at P8 up to 3.3 V when the whisker is not pressed. It's important to have either a pull-up or pull-down resistors in switch circuits. A pull-up or pull-down resistor applies a voltage that's opposite of the voltage the I/O pin will detect when the switch/button contact is made. Without it, the I/O pin will allow nearby electric fields to affect whether it reports 1 or 0 to the input

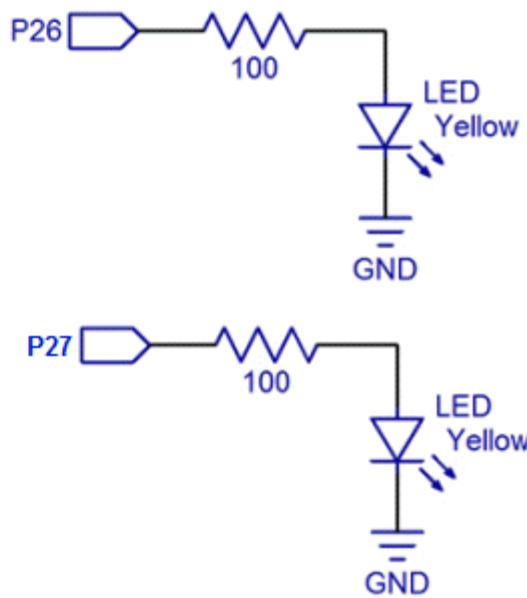
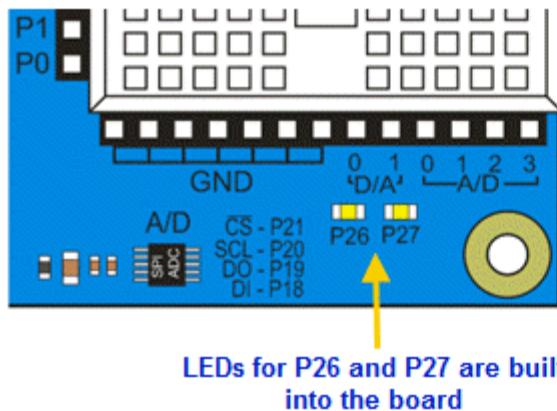


function. For example, without that pull-up/pull-down resistor, simply putting your hand near the 220 Ω resistor might change the value the input function reports.

Add Indicator Lights

As you built the whisker circuits on your board, you connected two Propeller I/O pins to LEDs that are built into the circuit board. That means you can program the Propeller to send high/low signals to these I/O pins, which will turn the connected LEDs on/off.

- ✓ The Propeller Activity Board uses built-in P26 and P27 LEDs for the LED circuits; see [Blink a Light](#) [14] in the Simple Circuits tutorials before moving on.



- ✓ Navigate to SimpleIDE > Learn > Examples > ActivityBot Tutorial.
- ✓ Open Test Whiskers With LEDs.side.
- ✓ Click the Run with Terminal button.
- ✓ Try pressing each whisker. The P26 light should turn on while the left whisker is pressed, and the P27 light should turn on while the right whisker is pressed.

How it Works

Test Whiskers with LEDs.c is just the previous program with two `if...else` statements added. First, `if(wL == 0) high(26); else low(26)` does one of two things. If `wL` stores 0, it means the left whisker is pressed,

so `high(26)` turns on the P26 LED. If `wL` stores 1, it means the whisker is not pressed. In that case, `low(26)` turns the LED off. The second line that was added is `if(wR == 0) high(27); else low(27)`. It does the same job, except that it turns the P27 LED on/off depending on whether `wR` stores a 1 or 0.

```
/*
Test Whiskers with LEDs.c

Display whisker states in terminal. 1 = not pressed, 0 = pressed.

*/
#include "simpletools.h"                                // Include simpletools header
int main()                                              // main function
{
    freqout(4, 2000, 3000);                            // Speaker tone: P4, 2 s, 3 kHz
    while(1)                                            // Endless loop
    {
        int wL = input(7);                             // Left whisker -> wL variable
        int wR = input(8);                             // Right whisker -> wR variable
        if(wL == 0) high(26); else low(26);           // Light for left whisker
        if(wR == 0) high(27); else low(27);           // Light for right whisker
        print("%c", HOME);                           // Terminal cursor home (top-left)
        print("wL = %d  wR = %d", wL, wR);          // Display whisker variables
        pause(50);                                    // Pause 50 ms before repeat
    }
}
```

Try This

You can modify the program to make the lights blink when the whiskers are pressed like this:

```
int main()
{
    freqout(4, 2000, 3000);
    while(1)
    {
        int wL = input(7);
        int wR = input(8);
        if(wL == 0)          // replace if statements...
        {
            high(26);
            pause(50);
            low(26);
            pause(50);
        }
        if(wR == 0)
        {
            high(27);
            pause(50);
            low(27);
            pause(50);
        }                  // ...with all of this
        print("%c", HOME);
        print("wL = %d  wR = %d", wL, wR);
        pause(50);
    }
}
```

- ✓ Click the Save As Project button.
- ✓ Name the project Test Whiskers with Blinking LEDs.
- ✓ Update the `main` function so that it matches the one above.
- ✓ Click the Run with Terminal button.
- ✓ Press and hold each whisker to verify that it makes each light blink.

Your Turn

Your challenge is to modify the Your Turn code to make it blink 10 times each time you press a whisker.
Hint: Review [Counting Loops](#) [21]. You can nest a `for` loop inside an `if` statement.

Whisker-Wheel Response

Now let's try a program that makes the robot back up while you push and hold a whisker up against its breadboard post.

- ✓ Click SimpleIDE's Open Project button.
- ✓ Navigate to SimpleIDE > Learn > Examples > ActivityBot Tutorial
- ✓ Open Whiskers Push Bot.side
- ✓ Set the PWR switch to 1.
- ✓ Click the Load EEPROM & Run button. Wait for it to finish loading and then switch PWR to 0 (off).
- ✓ Disconnect your Bot from the USB cable and place it on the floor (do not let it roam on a desk or other elevated surface).
- ✓ Set the PWR switch to 2.
- ✓ Press and hold one of the whiskers against its contact post. The robot should back up as long as you keep the whisker pressed.

How it Works

Before starting the `while(1)` loop, the program has its usual `freqout` call to make the speaker beep.

Inside the `main` function's `while(1)` loop, the first two lines should look familiar: they test the whisker input states and assign the result to `wL` and `wR`.

Next, we have `if((wL == 0) || (wR == 0))` followed by a code block. It means, “if `wL` stores 0 OR `wR` stores 0, do what is inside the code block.” So, if either variable does store 0, then, `drive_speed(-64, -64)` runs the robot's servos backwards for 20 ms.

If neither whisker variable stores a 0, the program execution skips that `if...` code block and moves on to the `else` code block below. There, it stops both servos with `drive_speed(0, 0)`.

```
/*
Whiskers Push Bot.c
Push the whiskers to make the Propeller ActivityBot back up.
*/
#include "simpletools.h"                                // Include simpletools header
#include "abdrive.h"                                    // Include abdrive header

int main()                                              // main function
{
    freqout(4, 2000, 3000);                            // Speaker tone: 2 s, 3 kHz

    while(1)
    {
        // Check whisker states.
        int wL = input(7);                               // Left whisker -> wL variable
        int wR = input(8);                               // Right whisker -> wR variable

        // If whisker pressed, back up
        if((wL == 0) || (wR == 0))                      // Either whisker detects
        {
            drive_speed(-64, -64);                     // Back up
        }
        else
        {
            drive_speed(0, 0);                          // Neither whisker detects
        }
    }
}
```

Try This

Here is a modified loop for your `main` function. It replaces the `if...else` statements with code that allow you to push one whisker at a time to make it turn away to one side or the other, or both whiskers to make it move straight backwards.

```
int speedLeft, speedRight;

int main()
{
    freqout(4, 2000, 3000);

    while(1)
    {
        // Check whisker states.
        int wL = input(7);
        int wR = input(8);

        // If whisker pressed, back up wheel on that side      // <- modify...
        if(wL == 0) speedLeft = -32; else speedLeft = 0;
        if(wR == 0) speedRight = -32; else speedRight = 0;

        drive_speed(speedLeft, speedRight);                      // <- ...to here.
    }
}
```

- ✓ Click the Save As Project button.
- ✓ Name the project Whiskers Push Bot Improved.
- ✓ Update the `main` function so that it matches the one above.
- ✓ Click the Load EEPROM & Run button.
- ✓ Press and hold each whisker, and verify that you can press and hold individual whiskers to make it turn, or both whiskers to make it back up.

Your Turn

Modify Whiskers Push Bot.side so that it makes the robot back up for one full second each time you press a whisker.

Roaming with Whiskers

Now that you've learned how to make the robot react to its whiskers, let's expand on that and make it roam with whiskers.

To roam, the robot needs to go forward until it bumps into something. When that happens, it needs to back up, turn away from the obstacle, and then go forward again. While the robot is going forward, it should repeatedly check its whiskers with minimal delays between whisker checks. That way, it can know right away when it has encountered an obstacle.

Run the Example Code

- ✓ Navigate to SimpleIDE > Learn > Examples > ActivityBot Tutorial
- ✓ Open Roaming with Whiskers.side.
- ✓ Set the PWR switch to 1.
- ✓ Click the Load EEPROM & Run button, and once it has finished loading, set PWR to 0 (off).
- ✓ Disconnect the robot from the USB cable.
- ✓ Set the robot on the ground in an area with a few obstacles high enough for the whiskers to bump into.
- ✓ Set the power switch to 2.
- ✓ Observe its roaming behavior.

How it Works

Look at the two lines inside the `while(1)` loop. `drive_speed(100, 100)` makes the robot go full speed forward for a 50th of a second.

The next two lines copy the whisker states to the `wL` and `wR` variables.

Next, two `else if` statements check in turn to see if just one whisker or the other is pressed. If just one whisker is pressed, the robot will back up straight for one second, and then turn away from the pressed-whisker side for 0.6 seconds. If both whiskers are pressed, the robot will back up and then decide to turn either left or right.

If no whiskers are pressed, none of the conditions in the entire set of `if...else if...else if` statements will be true, so none of those code blocks will get executed. The result is that the robot just keeps rolling forward and checking its whiskers every 20 ms.

```
/*
Roaming with Whiskers.c
Go forward until object detected by whisker(s). Then, back up, turn
```

```

        and go a new direction.

*/
#include "simpletools.h"
#include "abdrive.h"

int main()
{
    freqout(4, 2000, 3000);

    while(1)
    {
        // Go forward for 1/50th of a second.
        drive_speed(100, 100);

        // Check whisker states.
        int wL = input(7);
        int wR = input(8);

        // If whisker(s) pressed, avoid obstacle.
        if(wR == 0)
        {
            drive_speed(-100, -100);
            pause(500);
            drive_speed(-100, 100);
            pause(220);
        }
        else if(wL == 0)
        {
            drive_speed(-100, -100);
            pause(500);
            drive_speed(100, -100);
            pause(220);
        }
    }
}

// Include simpletools library
// Include abdrive library

// main function
// Speaker tone: 2 s, 3 kHz
// main loop
// Go forward
// Left whisker -> wL variable
// Right whisker -> wR variable
// Just right whisker
// Back up 0.5 seconds
// Turn left 0.22 seconds
// Just left whisker
// Back up 0.5 seconds
// Turn right 0.22 seconds

```

Try This

It's nice to have the lights turn on to indicate which whisker the robot is avoiding. Here is a modified code block for the `if...` statement that makes the LEDs light up for the whiskers that have been pressed.

- ✓ Use Save Project As to give your project a new name.
- ✓ Update the `if...` condition code block as shown, and reload the program.
- ✓ See if you can press both whiskers at exactly the same time to make both of the LEDs light up.

```
// If whisker(s) pressed, avoid obstacle.  
if(wR == 0)  
{  
    high(27); // <-- add  
    drive_speed(-100, -100);  
    pause(500);  
    drive_speed(-100, 100);  
    pause(220);  
    low(27); // <-- add  
}  
else if(wL == 0)  
{  
    high(26); // <-- add  
    drive_speed(-100, -100);  
    pause(500);  
    drive_speed(100, -100);  
    pause(220);  
    low(26); // <-- add  
}  
}  
}
```

Your Turn

- ✓ Modify the program so that it detects if both whiskers are pressed, and turns further. For this you will need to add a 30 ms `pause` between each `while` loop repetition, change the `if` condition to `else if`, and precede it with an `if` condition that check whether both whiskers are pressed.

Navigate by Ultrasound

Testing for Echo

The Ping))) Ultrasonic Distance Sensor lets your ActivityBot detect obstacles and measure the distance to them. Even though the PING))) sensor looks a bit like eyes, it is more like a mouth and an ear.

Much like a bat, the PING))) sensor emits an ultrasonic chirp when triggered, then listens for the chirp's echo and signals its return. The Propeller microcontroller can mark the time between triggering the PING))) sensor and getting a return signal. This echo return time can then be used to calculate the distance to an object.

This ability makes different ActivityBot navigation strategies possible, such as avoiding obstacles while roaming, or maintaining a set distance to an object that is moving.



Get the Example Code

- ✓ Make sure you have the latest [SimpleIDE software, Learn folder, and ActivityBot Library](#) [16].
- ✓ [Download the ActivityBot Navigate by Ultrasound code.](#) [22]
- ✓ Unzip the folder, and copy the contents to Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Follow the links below to continue with the ActivityBot tutorial.

Build and Test the PING))) Sensor Circuit

In this activity, you will build the PING))) sensor circuit and use it to measure distances of a variety of objects to get familiar with what it does and does not detect. The PING))) sensor just needs power, ground, and one signal connection for the Propeller to get distance measurements.

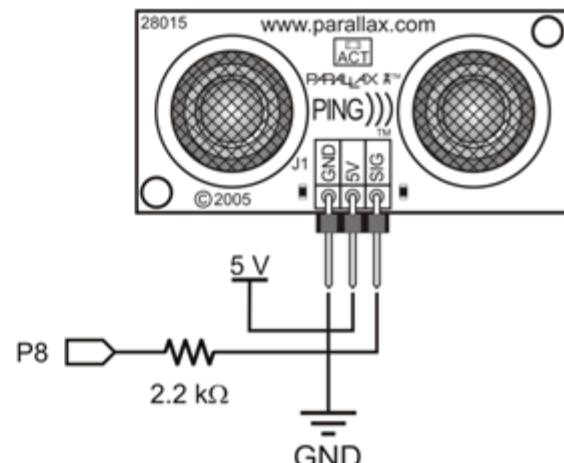
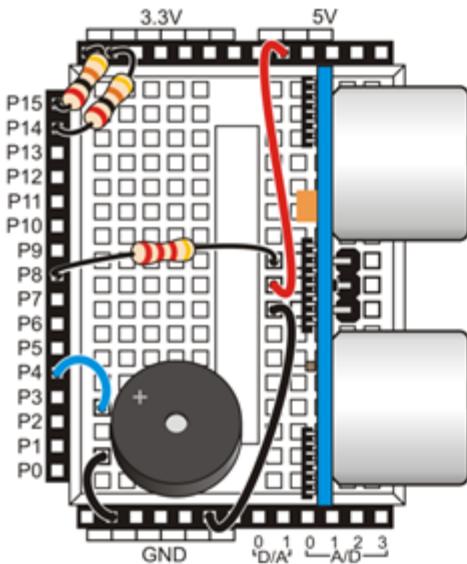
Parts List

- (1) PING))) Sensor (shown below)
- (1) 2.2 k-ohm resistor (red-red-red)
- (1) jumper wire



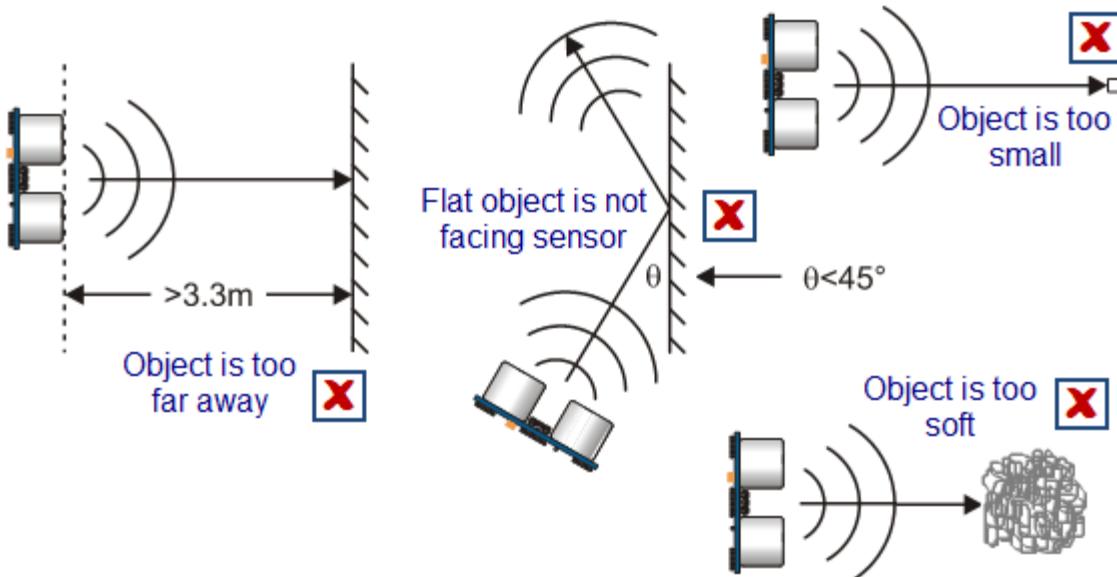
Build the PING))) Sensor Circuit

- ✓ If you haven't done so already, remove the touch-whisker circuits but leave the piezo speaker circuit in place.
- ✓ Build the PING))) sensor circuit on your Propeller Activity Board as shown here.
- ✓ Make sure to use a 2.2 kΩ resistor between the PING))) sensor's SIG pin and the Propeller I/O pin (P8 in this case).



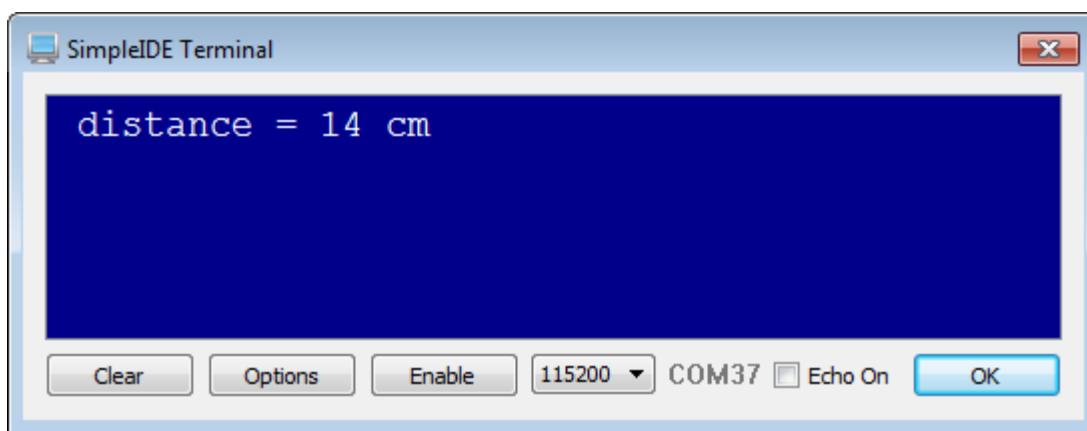
Ping))) Tests

The PING))) sensor's echo-location works on objects that can effectively reflect sound back at the sensor, from up to 3.3 meters away. Small objects don't reflect back enough sound. Hard-surfaced cylinders, ball-shaped obstacles, and walls if faced directly, work well. Walls that are approached at glancing angle will simply reflect the sound further away instead of back at the PING))) sensor. Soft items such as curtains and stuffed animals tend to muffle the sound and not bounce the echo.



For obstacles that do reflect sound back, the PING))) sensor and ping library provide centimeter distance measurements in the 3 cm to 3 m range. Air temperature can affect the accuracy of these distance measurements. [Check out this reference article](#) [23] about the speed of sound vs. air temperature to learn more about why this happens.

The test program will display the PING))) distance measurements in the SimpleIDE terminal. You can use it to test how well the PING))) sensor can detect various objects, and walls at various angles, to get familiar with what your ActivityBot can and cannot do with this sensor.

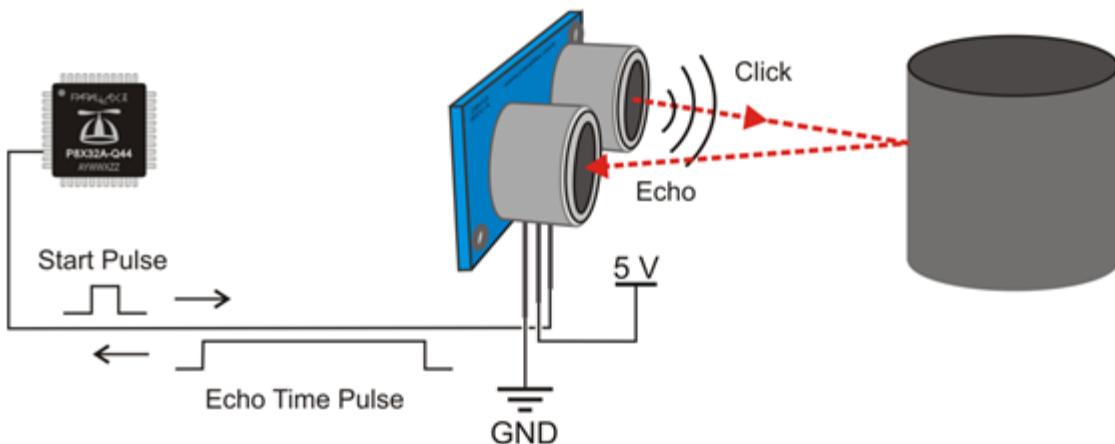


- ✓ Click SimpleIDE's Open Project button.
- ✓ Open Test Ping from ...Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Click the Run with Terminal button.
- ✓ Try measuring the distance to your hand, a cup, and a ball. All should be easy to detect.
- ✓ Try measuring the distance to something soft, like a stuffed animal or a crumpled piece of fabric. Can the PING))) sensor detect an echo from it?
- ✓ Try measuring a wall's distance head-on.

- ✓ Try measuring the distance to a wall, approaching at various angles. How far from 90° can the PING)) sensor still see the object?

How it Works

The PING)) sensor requires a brief high/low signal called a *start pulse* from the Propeller chip to start the measurement. The sensor then makes its ultrasonic click and sets its SIG pin high. When the echo comes back, it sets its SIG pin low. The Propeller chip then can measure the duration of the PING)) sensor SIG pin's high signal, which corresponds to the echo return time.



The ping library takes care of the math for converting the echo return time measurement to distance in centimeters. All your code has to do is request the distance with the `ping_cm` function. In the test code, the function's return value gets stored in a variable named `distance`. Then, it gets displayed in the SimpleIDE terminal with `print("%c distance = %d cm %c", HOME, distance, CLREOL)`. This particular `print` statement puts the cursor in the top-left home position with `HOME`, displays the distance, and then `CLREOL` clears any characters to the end of the line. `CLREOL` is useful if the previous measurement has more digits than the current measurement. (If the program skipped that step, it would display cmm instead of cm after the measurement if it happens to be one less digit than the previous measurement. Try it if you want.)

```
/*
Test Ping.c

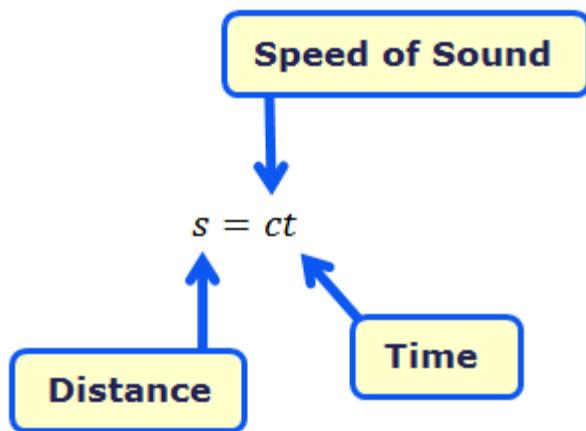
Test the PING))) sensor before using it to navigate with the ActivityBot.

*/
#include "simpletools.h" // Include simpletools header
#include "ping.h" // Include ping header
int distance; // Declare distance variable
int main() // main function
{
    while(1) // Repeat indefinitely
    {
        distance = ping_cm(8); // Get cm distance from Ping)))
        print("%c distance = %d%c cm",
              HOME, distance, CLREOL); // Display distance
    }
}
```

```
    } pause(200); // Wait 1/5 second  
}
```

Did You Know?

The ping library measures the echo pulse in terms of microseconds and then uses the fact that sound travels at 0.03448 cm/ μ s at room temperature. That's 3.448 hundredths of a centimeter per millionth of a second at a temperature of (22.2 °C). Just as a car travels distance = speed x time, so does sound, with an equation of $s = ct$, where s is the distance, c is the speed of sound and t is the time.



To calculate the distance of an echo, you have to remember that the time measurement is for twice the distance (there and back), so the equation would be:

$$2s = ct$$

Divide both sides by 2, and you've got an equation for distance from a time measurement.

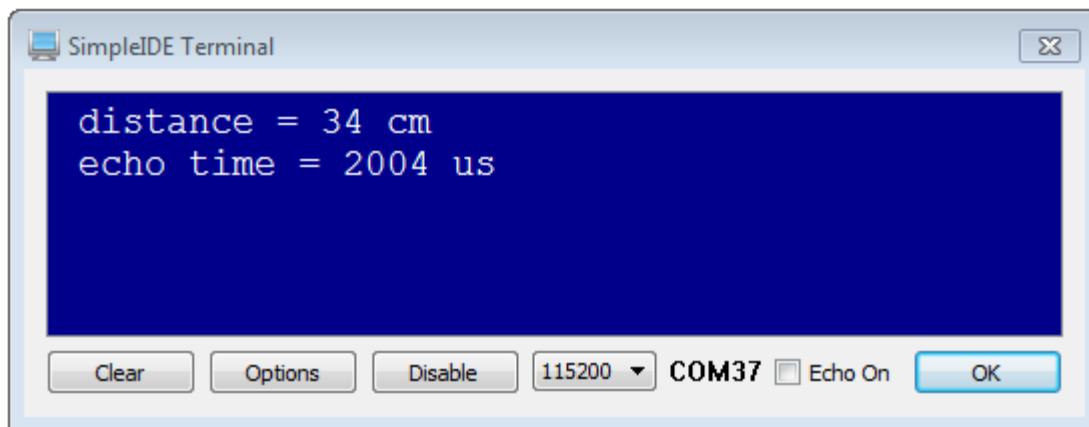
$$s = \frac{ct}{2} = \frac{0.03448 \frac{\text{cm}}{\mu\text{s}} \times t}{2}$$

Since $0.03448/2 \approx 1/58$, the equation for cm distance from microsecond echo time becomes:

$$s \approx \frac{t}{58}$$

Try This

You can display the raw microsecond time measurements with a call to the `ping` function.



- ✓ Use the Save Project As button to save a copy of your project in ...Documents\SimpleIDE\My Projects.
- ✓ Modify the `main` function as shown below.
- ✓ Run the program and verify the output.

```
int main()
{
    while(1)
    {
        distance = ping_cm(8);

        print("%c distance = %d%c cm",
              HOME, distance, CLREOL);

        distance = ping(8); // <- add

        print("\n echo time = %d%c us",
              distance, CLREOL); // <- add

        pause(200);
    }
}
```

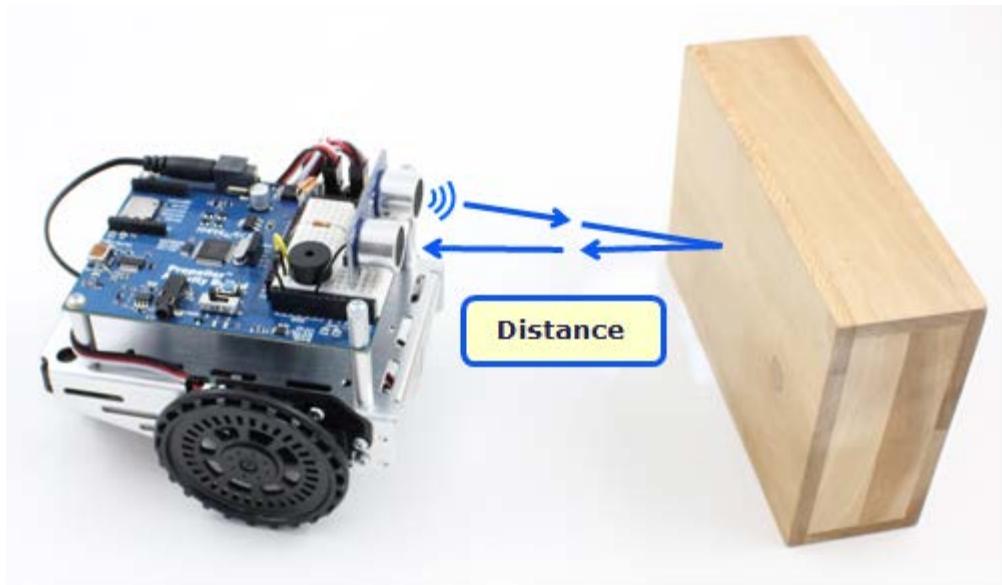
Your Turn

- ✓ Use your calculator to verify that the centimeter distance is the microsecond distance $\div 58$. Keep in mind that C language `int` calculations always round down.

Roaming with Ultrasound

Now that you can measure centimeter distances with the PING))) sensor, and know which kinds of objects it can sense, let's put it to work in navigation.

All your code has to do is make the robot go forward until it receives a distance measurement from the PING))) sensor that's less than some pre-defined value, say 20 cm. Then, slow down to a stop. After that, turn in place until the measured distance is more than 20 cm. At that point, it'll be safe to go forward again. To make the ActivityBot's navigation a little more interesting, your code can also use random numbers to decide which direction to turn.



Test the Sense, Stop, and Turn Maneuver

Before actually roaming, it's important to test a smaller program to make sure your ActivityBot will stop in front of an obstacle, and turn away from it. This example program makes the ActivityBot go forward while the PING))) measured distance is greater than 20 cm. If the distance measures less than or equal to 20 cm, the robot ramps down to a stop. Then, it turns in place until it measures more than 20 cm, which means the robot turned until the object is no longer visible.

- ✓ Click SimpleIDE's Open Project button.
- ✓ Open Detect and Turn from Obstacle from ...Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Click the Load EEPROM & Run button.
- ✓ Disconnect your robot from its programming cable.
- ✓ Set the power switch to 2.
- ✓ Press and release the reset button.
- ✓ Send it toward a wall, block, water bottle, or other obstacle.
- ✓ Verify that it goes full speed until it gets closer than 20 cm, then ramps down and stops, and turns.

- ✓ Try it several times so that you can see that the direction it picks to turn is random.

How it Works

This program doesn't use `distance = ping_cm(8)`. Instead, it just takes the value `ping_cm(8)` returns, and uses it in decisions. So, it doesn't need to declare a `distance` variable; however, it does still need a variable for storing a random number to decide which way to turn. So before the `main` function, there's a declaration for a variable named `turn`.

The program starts with `drive_setRampStep(10)`, which sets the number of ticks per second that the speed is allowed to change for every 50th of a second. Since the function call passes 10, it means the speed can only change by 10 ticks per second, every 50th of a second. Next, `drive_ramp(128, 128)` ramps up to full speed, in steps of 10 ticks per second. With a ramp step of 10, it takes 13 50ths of a second = 0.26 seconds to ramp up to full speed. All the steps are by 10, except for the last one, which is from 120 to 128.

```
/*
Detect and Turn from Obstacle.c

Detect obstacles in the ActivityBot's path, and turn a random direction to avoid them.

*/
#include "simpletools.h"                                // Include simpletools header
#include "abdrive.h"                                    // Include abdrive header
#include "ping.h"                                       // Include ping header

int turn;                                              // Navigation variable

int main()                                             // main function
{
    drive_setRampStep(10);                             // 10 ticks/sec / 20 ms
    drive_ramp(128, 128);                            // Forward 2 RPS

    // While distance greater than or equal
    // to 20 cm, wait 5 ms & recheck.
    while(ping_cm(8) >= 20) pause(5);                // Wait until object in range

    drive_ramp(0, 0);                                 // Then stop

    // Turn in a random direction
    turn = rand() % 2;                               // Random val, odd = 1, even = 0

    if(turn == 1)
        drive_speed(64, -64);                         // If turn is odd
    else
        drive_speed(-64, 64);                         // else (if turn is even)

    // Keep turning while object is in view
    while(ping_cm(8) < 20)                           // Turn till object leaves view
        drive_ramp(0, 0);                            // Stop & let program end
}
```

Next, the program enters a one-line loop: `while(ping_cm() >= 20) pause(5)`. This translates to "check to see if the `ping_cm` function returned a value equal to or greater than 20 (no object is detected within 20 cm), and if this is true, `pause` for 5 ms and then check again." As long as `ping_cm` returns greater than 20, the `while` loop keeps looping, the ActivityBot continues to drive straight forward.

If a `ping_cm` function call returns a value of 20 or less (an object is detected within 20 cm), then the `while` condition is no longer true and the loop stops repeating. This allows the code to move on to `drive_ramp(0, 0)`. This slows the ActivityBot down to a stop, which also takes about 0.26 seconds.

After stopping, the ActivityBot has to decide which direction to turn to avoid the object. This example uses the math library's `rand` function (included by simpletools) to randomly choose between left and right. The `rand` function returns a pseudo random value somewhere in the 0 to 2,147,483,648 range. Each time the code calls `rand`, it returns a new value in its pseudo-random sequence. The statement `turn = rand() % 2` divides 2 into the random number, takes the remainder (which will be a 1 or 0), and copies it to the `turn` variable.

This `turn` variable's random 1 or 0 is used to decide which direction to turn. If `turn` gets a random one, `if(turn == 1) drive_speed(64, -64)` statement makes the ActivityBot rotate right. If instead `turn` gets a random zero, `drive_speed(-64, 64)` makes it rotate left. After that, `while(ping_cm(8) < 20)` keeps repeating while the object is still in view. As soon as the ActivityBot has turned far enough, it stops.

If all that worked, most of the code in the `main` function can be put in a `while(1)` loop to make the ActivityBot go looking for new obstacles.

Did You Know?

Library List — This application uses functions from four libraries, simpletools (`pause`), abdrive (anything starting with `drive`), ping (`ping_cm`), and math (`rand`). The math library is not declared here because simpletools already includes it, but you could just as easily add `#include <math.h>` to the list. It would be enclosed in `<>` symbols instead of quotes by convention since it's part of Propeller GCC and not part of the Propeller C Tutorial's custom libraries (like simpletools, abdrive, and ping).

Try This – Roaming with Ping)))

This program just needs a `while(1)` loop to make it move on in search of the next obstacle and continue to repeat what it just did. The only thing that doesn't need to be part of the `while` loop is the last `drive_ramp(0, 0)`.

- ✓ Use the Save Project As button to save a copy of the example program into My Projects. Use the name Roaming with Ping (Try This).
- ✓ Modify the program to match the example below. You are just adding `while(1)` followed by an opening brace `{`. Just above the last statement you'll be adding a closing brace `}`.
- ✓ For readability, indent all the statements between the braces you just added by two spaces. (You can just shade the block and press the Tab key.)
- ✓ Use the Load EEPROM & Run button to load the program into the ActivityBot. Set PWR to 0 and

disconnect from the programming cable.

- ✓ Set the PWR switch to 2, and hold down the reset button.
- ✓ Set it in an open area with several obstacles, and observe how the ActivityBot goes up to one and then turns away in search for the next obstacle.

```
int turn;

int main()
{
    drive_setRampStep(10);

    while(1)                                //<- add
    {                                         //<- add

        drive_ramp(128, 128);

        // While distance greater than or equal
        // to 20 cm, wait 5 ms & recheck.
        while(ping_cm(8) >= 20) pause(5);

        drive_ramp(0, 0);

        // Turn in a random direction
        turn = rand() % 2;

        if(turn == 1)
            drive_speed(64, -64);
        else
            drive_speed(-64, 64);

        // Keep turning while object is in view
        while(ping_cm(8) < 20);                  //<- add

        drive_ramp(0, 0);
    }
}
```

Your Turn

Here are three challenges for you.

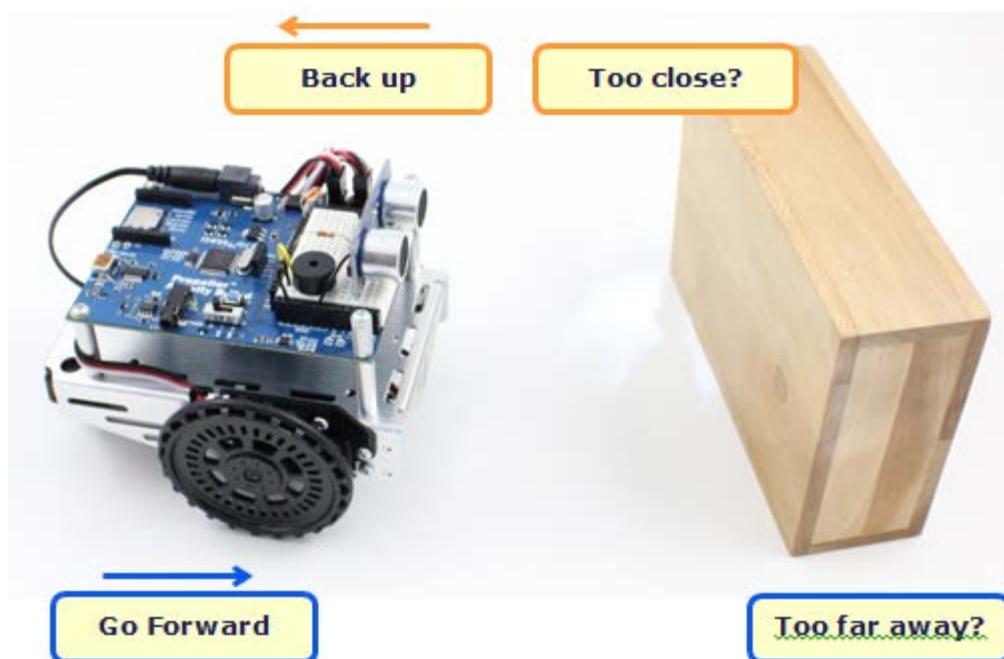
- ✓ Modify the code so that it turns alternate directions with each obstacle.
Hints: Instead of `turn = rand() % 2`, use `turn = (turn + 1) % 2`. This will add 1 to `turn`, then take the remainder of `turn / 2`, which will be 1, then 0, then 1, then 0....
- ✓ Make it so that the ActivityBot stops after it finds four obstacles.
Hints: Declare a variable for counting obstacles with `turn`, and initialize it to 0 before the `while(1)` loop starts. Add 1 to the counting variable each time through the `while(1)` loop. There's more

than one way to limit the repetitions to 4 - can you figure one out?

- ✓ The simplest initial approach to exploring a maze is to take a right turn at each wall. Not necessarily the best approach, but certainly the simplest. Modify the code so that it only turns right when it encounters an obstacle.

Follow Objects with Ultrasound

Instead of finding and avoiding obstacles, how about making the ActivityBot follow a target instead? Same sensor system, different code. All the program has to do is check for a desired distance, and if the object is further away than that, move forward. If the object is closer than the desired distance, move backward. If the object's distance is equal to the desired distance, stay still.



Following Objects

This example makes the ActivityBot try to keep a 32 cm distance between itself and an object in front of it. With no object detected, the robot will go forward until it finds one, then hone in and maintain that 32 cm distance. If you pull the object away from the ActivityBot, it will go forward to catch up. Likewise, if you push the object toward it, it will back up to restore that 32 cm distance.

- ✓ Click SimpleIDE's Open Project button.
- ✓ Open Follow with Ping))) from ...Documents\SimpleIDE\Learn\Examples\ActivitiyBot Tutorial.
- ✓ Click the Load EEPROM & Run button.
- ✓ Send it toward a wall, and verify that it stops and maintains a 32 cm distance.
- ✓ Send it toward an object, and after it locks on, try pushing the object toward and away from the

ActivityBot. It should respond to maintain that 32 cm distance.

How it Works

The program declares five variables: `distance`, `setPoint`, `errorVal`, `kp`, and `speed`.

- `distance` stores the measured distance of an object in front of the ActivityBot.
- `setPoint` stores the distance the ActivityBot should try to maintain between itself and an object it detects in front of it.
- `errorVal` stores the difference between the measured distance and the desired `setPoint` distance.
- `kp` stores a *proportionality constant*; `errorVal` can be multiplied by `kp` for a speed that will help maintain the `setPoint` distance.
- `speed` stores the result of `errorVal * kp`.

```
/*
Follow with Ping.c

Maintain a constant distance between ActivityBot and object.

*/
#include "simpletools.h"                      // Include simpletools header
#include "abdrive.h"                          // Include abdrive header
#include "ping.h"                            // Include ping header

int distance, setPoint, errorVal, kp, speed; // Navigation variables

int main()                                     // main function
{
    setPoint = 32;                             // Desired cm distance
    kp = -10;                                // Proportional control

    drive_setRampStep(6);                     // 7 ticks/sec / 20 ms

    while(1)                                    // main loop
    {
        distance = ping_cm(8);                // Measure distance
        errorVal = setPoint - distance;       // Calculate error
        speed = kp * errorVal;                // Calculate correction speed

        if(speed > 128) speed = 128;          // Limit top speed
        if(speed < -128) speed = -128;

        drive_rampStep(speed, speed);         // Use result for following
    }
}
```

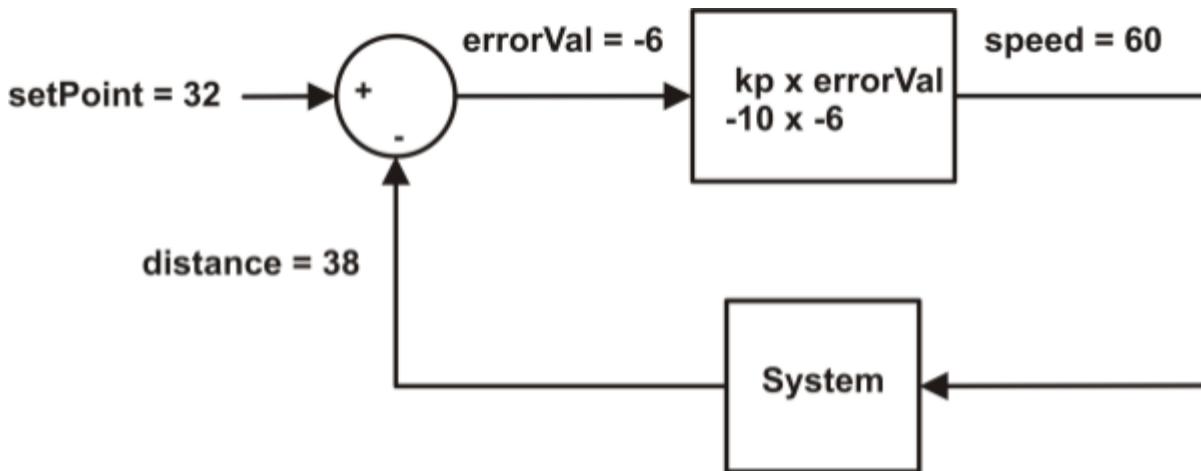
The program starts off by making the `setPoint` 32 cm and the proportionality constant `kp` -10. It also sets the ramp step size to 6, which will cushion sudden changes in speed. Inside the `while(1)` loop the program makes the ActivityBot maintain a distance between itself and the target object by repeating five steps very rapidly:

1. Get the measured distance with `distance = ping_cm(8)`.
2. Calculate the difference between the desired and measured distance with `errorVal = setPoint - distance`.
3. Calculate the speed needed to correct any differences between desired and measured distance with `speed = kp * setPoint`.
4. Limit the output to +/- 128 with two `if` statements.
5. Drive both wheels at the corrected `speed`.

Did You Know?

An automated system that maintains a desired physical output level is called a *control system*. In the case of our ActivityBot, the level it maintains is the distance between itself and the object in front of it. Many control systems use a feedback loop to maintain the level, and our ActivityBot is a prime example of this. The measured distance is fed back into the system and compared against the set point for calculating the next speed, and this happens over and over again in a process called a *control loop*.

Engineers use drawings like this one to think through a control loop's design and also to convey how it works to technicians and other engineers. If the set point (distance the ActivityBot tries to maintain) is 32 and the measured distance is 38, there is an error value. This error value is the set point – measured distance. That's $32 - 38 = -6$. This error is multiplied by a proportionality constant (k_p) for an output that's the speed to correct the error this time through the loop. That's speed = $k_p \times$ error value, or $60 = -10 \times -6$.



The circle is called a *summing junction* (for addition or subtraction), and the rectangular blocks designate operations that can vary. In this example, the top one is a proportional control block that multiplies error by a proportionality constant. The System block represents things we do not necessarily have control over, like if you move the object toward or away from the ActivityBot. Other common control blocks that were not used in this example are Integral, which allows corrections to build up over loop repetitions, and Derivative, which responds to sudden system changes.

Try This

It's really helpful to see how the values respond to different measured distances. To do this, all you have to do is add some `print` values to the screen.

SimpleIDE Terminal

```
distance (38) = ping_cm(8)
errorVal (-6) = setPoint (32) - distance (38)
speed (60) = kp (-10) * errorVal (-6)
After limiting, speed (60)
```

Clear Options Disable 115200 COM37 Echo On OK

- ✓ Use the Save Project As button to save a copy of your project. Name it Display Ping Control System and save it in ...Documents\SimpleIDE\My Projects.
- ✓ Modify the `main` function as shown below.
- ✓ **Set the power switch to 1** (so the ActivityBot does not drive off your desk) and click Run with Terminal.
- ✓ Run the program and monitor the output values as you try placing a target object different distances from the Ping))) sensor. Focus on distances ranging from 20 to 44 cm.



WARNING: This modified program is just for display, not navigation. For navigation, re-open Following with Ping.

```

while(1)
{
    distance = ping_cm(8);
    errorVal = setPoint - distance;
    speed = kp * errorVal;

    print("%c", HOME); // <- Add
    print("distance (%d) = ping_cm(8)%c\n\n", // <- Add
          distance, CLREOL); // <- Add
    print("errorVal (%d) = setPoint (%d) - distance (%d)%c\n\n", // <- Add
          errorVal, setPoint, distance, CLREOL); // <- Add
    print("speed (%d) = kp (%d) * errorVal (%d)%c\n\n", // <- Add
          speed, kp, errorVal, CLREOL); // <- Add

    if(speed > 128) speed = 128;
    if(speed < -128) speed = -128;

    print("After limiting, speed (%d)%c\n\n", // <- Add
          speed, CLREOL); // <- Add
    pause(1000); // <- Add

    drive_rampStep(speed, speed);
}

```

If the measured distance is 40, `errorVal = setPoint - distance` results in `errorVal = 32 - 40 = -8`. Then, `speed = kp * error` is `speed = -10 * -8 = + 80`. So, the ActivityBot will try 80 ticks per second forward. This falls in the $+/- 128$ range, so the ActivityBot goes a little over half speed forward to catch up with the object. Now, try this for a distance of 24. The correct answer is `speed = -80` which will make the ActivityBot back up. If the distance is 28 instead, the ActivityBot will back up less quickly, with a `speed` of -49.

- ✓ Repeat these calculations for distances of 30, 32, 34, and 36 cm.

Your Turn

There are lots of values to experiment with here. For example, if you change the `setPoint` variable from 32 to 20, the ActivityBot will try to maintain a distance of 20 cm between itself and an object in front of it. The value of `kp` can be increased to make it try to go faster to correct with smaller error distances, or slower to correct over larger error distances.

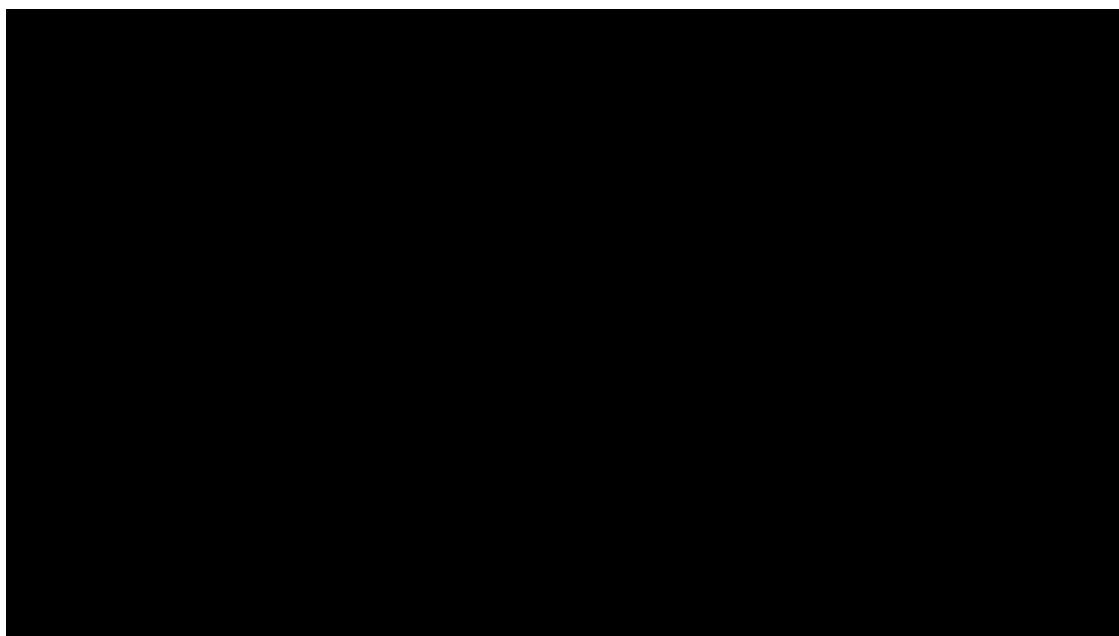
- ✓ Adjust the values of `setPoint` and `kp` to see what happens to the performance.
- ✓ Make notes of what you tried and how the ActivityBot responded. What values of `setPoint` and `kp` seem optimal to you?

If you decrease the `drive_setRampStep` value, it will cushion the changes, but could cause it to respond so slowly that it runs into the object. If you increase it, it will respond more abruptly. Too large, and it could actually change direction so fast that it could even eject the Ping))) sensor from the breadboard!

- ✓ Adjust the `drive_setRampStep` value downward to smooth out motions, or upward to make them more abrupt.
- ✓ Keep experimenting with values as you search for the performance that seems best to you.
- ✓ Again, make notes of what you tried and how the ActivityBot's performance responded

Navigate by Visible Light

Light sensors are used in all kinds of robotics, industrial, and commercial applications. Next, let's teach it to navigate by visible light. By using a pair of small light sensors, called phototransistors, your ActivityBot can measure and compare the light levels on its right and its left. Then, it can turn toward the brighter side as it navigates. This way, the ActivityBot can be programmed to follow a flashlight, or find its way out of a dark room to a bright doorway.



Get Acquainted with the Phototransistor

The Propeller C Simple Circuits tutorial has a page devoted to the same phototransistor that is included in your ActivityBot kit.

- ✓ Go try the [Sense Light tutorial](#) [24], then return here when you are done.

Welcome back!

- ✓ If you happened to remove the piezo speaker, be sure to [put it back where it was](#) [25].

Get the Example Code

- ✓ Make sure you have the latest [SimpleIDE software, Learn folder, and ActivityBot Library](#) [16].
- ✓ [Download the ActivityBot Navigate by Light code.](#) [26]
- ✓ Unzip the folder, and copy its contents to Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Follow the links below to continue with the ActivityBot tutorial.

Build the Light Sensor Circuits

Now, it's time to build and test two phototransistor circuits to give your ActivityBot phototransistor "eyes." Leave the piezo speaker circuit in place.

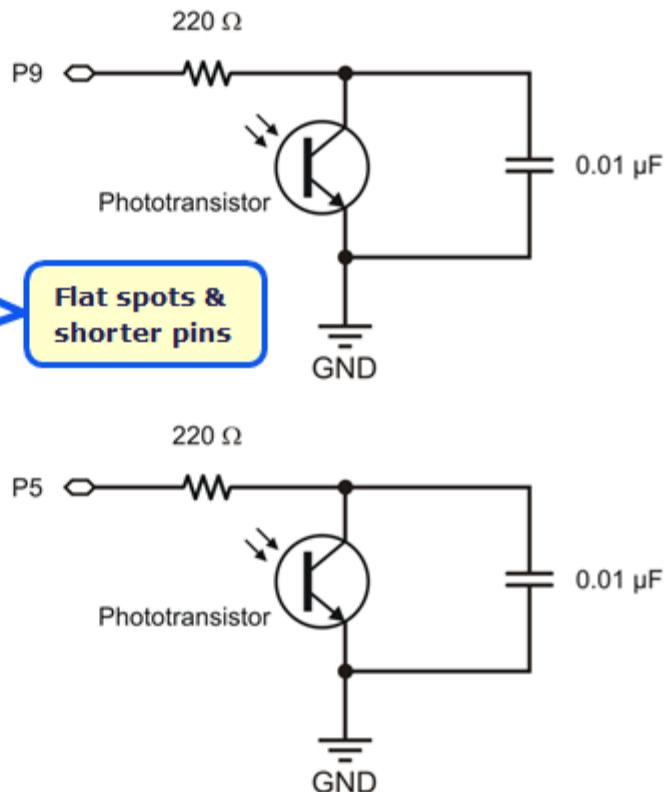
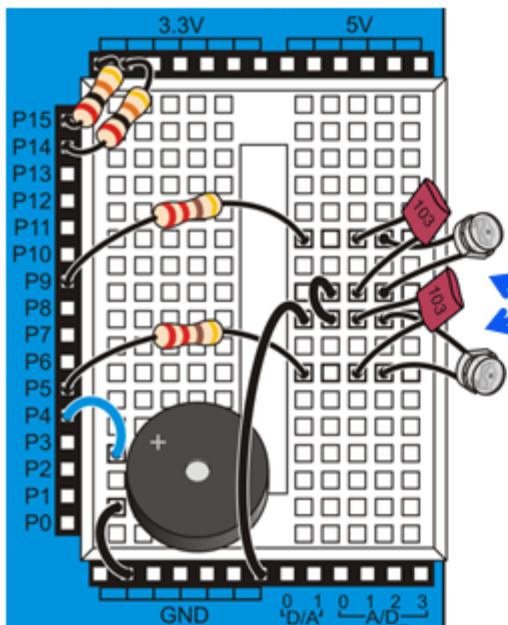
Parts

- (2) Phototransistor (#350-00029)
- (2) 0.01 μ F capacitor (labeled 103)
- (2) 220 ohm resistor (red-red-brown)

Build the Light Sensor Circuits

Build the circuit shown below.

- ✓ Make sure the phototransistor's shorter leads and flat spots are connected to ground, as shown in the wiring diagram below.
- ✓ Use the 0.01 μ F capacitors, labeled "103." The capacitors in your ActivityBot kit are not polar; it does not matter which way you plug them in.
- ✓ Point the phototransistors upwards and outwards, about 90° from each other, and about 45° from vertical.

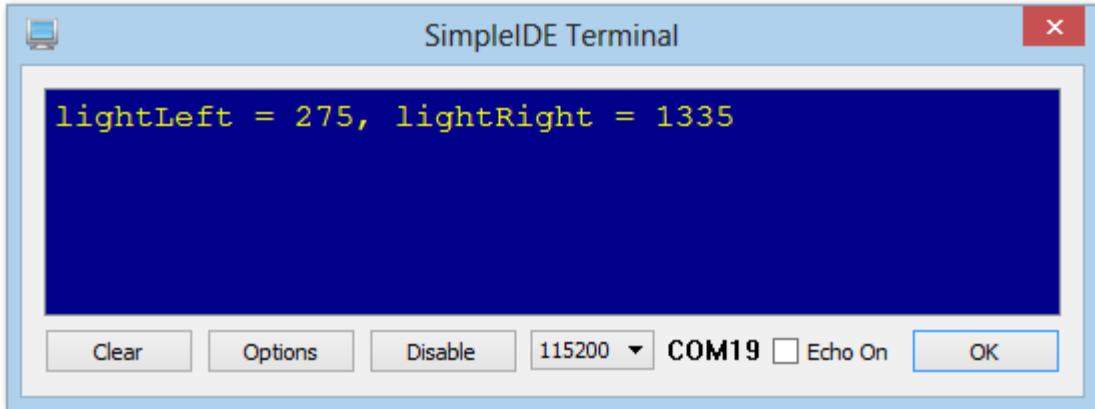


Test the Light Sensor Circuits

This test will display raw light sensor readings in the SimpleIDE Terminal. You will need a flashlight or lamp that is brighter than the overall light level in the room.

The phototransistor circuits are designed to work well indoors, with fluorescent or incandescent lighting. Make sure to avoid direct sunlight and direct halogen lights; they would flood the phototransistors with too much infrared light.

- ✓ In your robotics area, close window blinds to block direct sunlight, and point any halogen lamps upward so that the light is reflected off the ceiling.
- ✓ Click SimpleIDE's Open Project button.
- ✓ Open Test Light Sensors.side from ...Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Put the power switch in Position 1.
- ✓ Click the Run with Terminal button.
- ✓ Hold your hand over one phototransistor, and then the other, while watching the values change.



The values shown above were taken in a room with overhead fluorescent lighting. Notice how the `lightRight` measurement is larger - this was caused by cupping a hand over the right phototransistor.

Mismatched Measurements

It is unlikely that your two phototransistor measurements will match exactly. Each individual sensor will vary slightly as a natural part of the manufacturing process, and the brightness levels are rarely precisely even in any environment. However, if one gives a measurement about 100 times larger than the other when exposed to the same level of light, that phototransistor is probably in backwards. Check your circuit and try again.

How it Works

This simple program starts by declaring two `int` variables, `lightLeft` and `lightRight`, to hold values returned by the left and right phototransistor circuits.

The action happens inside a `while(1)` loop, which takes two light sensor measurements and displays the results over and over again.

Recall from the [Sense Light tutorial](#) [24] that phototransistor measurement is a two-step process. First, `high(9)` connects the phototransistor circuit to 3.3 V, allowing the capacitor in the circuit to charge up like a tiny battery. It only takes a moment to do that; `pause(1)` allows enough time. Immediately after charging the circuit, the next line of code calls the `rc_time` function to take a measurement at I/O pin 9 and store the result in `lightLeft`. The same steps are taken to store a sensor measurement in `lightRight`.

After that, the `print` statement prints the name of each variable followed by its value in decimal. The control character `HOME` returns the cursor to the upper left corner of the SimpleIDE terminal each time, and `CLREOL` clears out any characters left over from the previous measurement.

```
/*
Test Light Sensors.c
*/
#include "simpletools.h"
```

```

int lightLeft, lightRight;

int main()
{
    while(1)
    {
        high(9);
        pause(1);
        lightLeft = rc_time(9, 1);

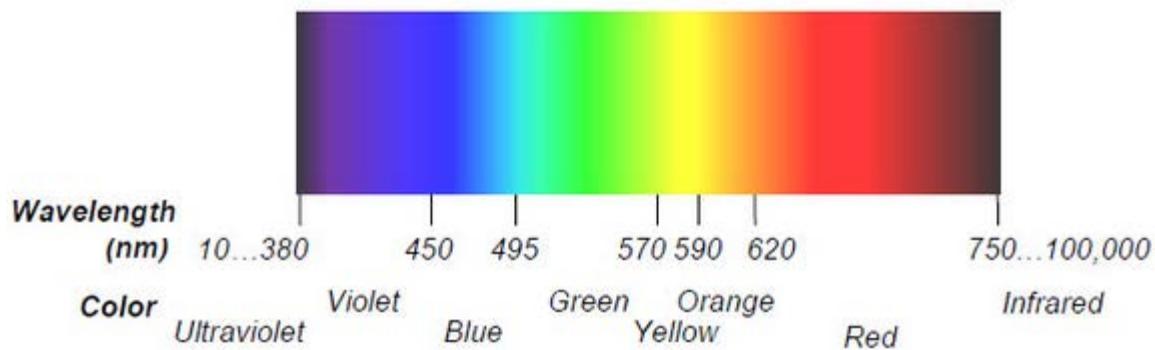
        high(5);
        pause(1);
        lightRight = rc_time(5, 1);

        print("%clightLeft = %d, lightRight = %d%c",
              HOME, lightLeft, lightRight, CLREOL);
        pause(250);
    }
}

```

Did you Know?

Light travels in waves so small that the distance between adjacent peaks is measured in *nanometers* (nm), which are billionths of meters. The figure below shows the wavelengths for colors of light we are familiar with, along with some the human eye cannot detect, such as ultraviolet and infrared. The phototransistor in your ActivityBot kit detects visible light, but is most sensitive to 850 nm wavelengths, which is in the infrared range.



Your Turn

Just for fun, let's see if the phototransistor can detect different colors.

- ✓ Zoom in on the color wavelength graph above.
- ✓ Re-run Test Light Sensors.side.
- ✓ Hold your robot up to the screen, placing one phototransistor very close to the color bar.

- ✓ What color gives the highest reading? What colors give the lowest?

From your observation, do you think a clear phototransistor alone, like this one, could be used as a color sensor? Why or why not?

Using the Measurements

Making the Measurements Useful

For ActivityBot navigation, the raw sensor values don't matter directly. What matters is the *difference* between the light levels detected by each sensor, so the ActivityBot can turn to the side of the one sensing brighter light. It is also nice to have values that will fall into a predictable range that can integrate easily with code to drive the servo motors.

Accomplishing this is surprisingly simple. First, just divide one sensor measurement into the sum of both. The result will always be in the 0 to 1 range. This technique is an example of a *normalized differential* measurement. Here's what it looks like as an equation:

$$\text{normalized differential measurement} = \frac{\text{lightRight}}{\text{lightRight} + \text{lightLeft}}$$

Two more simple operations will make the result easier to use. First, multiplying the result by 200 will make the result a whole number. Second, subtracting 100 from that whole number will always return values from -100 to 100. If the phototransistors sense the same level of light, regardless of how bright or dim, the final whole number will be zero. This is an example of a *zero-justified* normalized differential measurement, let's call it "ndiff" for short. Now our equation looks like this:

$$\text{ndiff} = 200 * \frac{\text{lightRight}}{\text{lightRight} + \text{lightLeft}} - 100$$

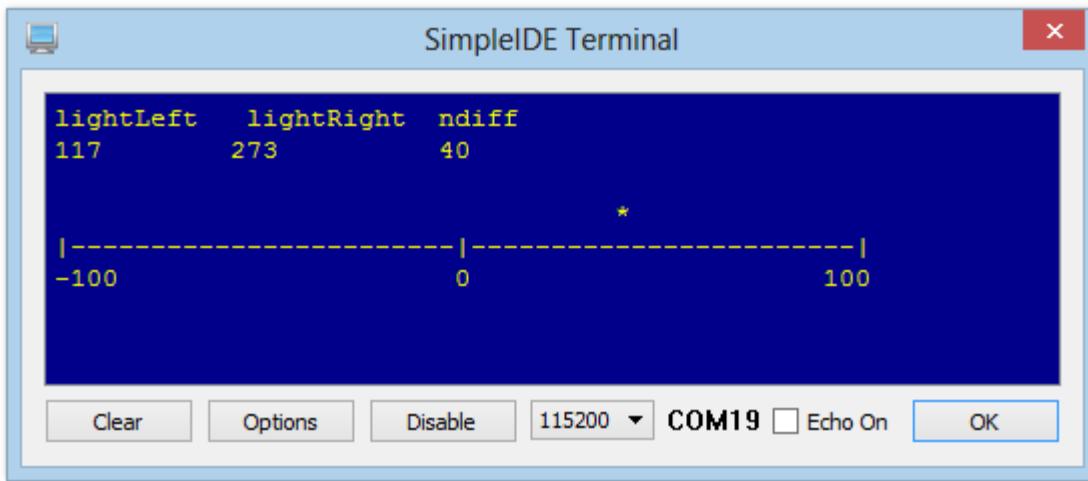
The example code below will display the `lightRight` and `lightLeft` measurements, and also show the value of `ndiff` as an asterisk over a number line. This makes it easier to visualize how the ActivityBot navigation program uses the sensor values to aim for a new trajectory towards the brighter light.

Graphical Test Code

- ✓ Click SimpleIDE's Open Project button.
- ✓ Open Test Light Sensors Graphical.side from

...Documents\SimpleIDE\Learn\Examples\ActivitiyBot Tutorial.

- ✓ Click the Run with Terminal button.
- ✓ Rotate the ActivityBot towards and away from a light source to see the position of the asterisk change.



Display Troubles?

If you have resized your SimpleIDE Terminal window smaller than this example program needs it to be, you will likely get garbled results. Just resize the terminal wider and then click Run with Terminal again.

How it Works

It's a long read, but an interesting one if you want to understand how the graphical display was generated using `print` statements with the `simpletools` library.

This program builds on Test Light Sensors. Two more variables are declared in addition to `lightLeft` and `lightRight`; `ndiff` holds the result of the fancy-named equation, and `position` will be used to place the asterisk in the SimpleIDE terminal.

Inside `main`, a set of three `print` statements define the display layout for the SimpleIDE Terminal.

The first `print` displays the value of the variables listed, and the `\n` (new line) formatters make three blank lines to make room for the asterisk that will display over the number line.

The second `print` displays the number line, followed by a `\n` formatter.

The third `print` displays the labels for the number line, followed by a `\n` formatter.

Then, a `char` array named `s` is declared and initialized with 51 space characters, which will be used to position the asterisk over the number line.

Next, the code enters a `while(1)` loop. The next 8 lines, which obtain the raw phototransistor measurements, should be familiar by now.

The next line beginning with `ndiff` is the same as the `ndiff` equation shown above.

The three `print` statements below the `ndiff` expression (1) position the cursor, (2) display the updated decimal values of `lightRight`, `lightLeft`, and `ndiff`, and clear out any characters left from last time, and (3) move the cursor to the line the asterisk will use.

Next, `position = (ndiff + 100) / 4;` offsets `ndiff` with `+100` to ensure it is a positive number, and then scales it with `/4` to fit the scale of the number line. The result is a number that is useful for positioning an asterisk in the SimpleIDE Terminal, and it is assigned to the `position` variable.

This is how the asterisk gets placed: `s[position] = '*'`; redefines the "position-th" element in the `s` array to be an asterisk instead of a space. Then, the asterisk is displayed in the proper position with `print(s)`; which displays all of the elements in the `s` array - a series of spaces with just one asterisk.

Immediately, `s[position] = ' '`; redefines the "position-th" element to be an empty space again, so the asterisk can be re-positioned on the next run through the loop. Without this line, the display would start accumulating asterisks.

The loop finishes up with `pause(350);` before repeating. If you read this far, thank you for your perseverance!

```
/*
Test Light Sensors Graphical.c
*/
#include "simpletools.h"

int lightLeft, lightRight, ndiff, position;

int main()
{
    print("lightLeft    lightRight    ndiff\n\n\n");
    print("|-----|-----|\n");
    print("-100          0           100\n");
    char s[51] = { " } ;

    while(1)
    {
        high(9);
        pause(1);
        lightLeft = rc_time(9, 1);

        high(5);
        pause(1);
        lightRight = rc_time(5, 1);

        ndiff = 200 * lightRight / (lightRight + lightLeft) - 100;

        print("%c%c", HOME, CRSRDN);
        print("%d      %d", lightLeft, lightRight, ndiff);
        print("%c%c%c", CLREOL, CRSRDN, CR);

        position = (ndiff + 100) / 4;
        s[position] = '*';
        print(s);
        s[position] = ' ';

        pause(350);
    }
}
```

Did You Know?

Parentheses Matter. In C language mathematical expressions, like the one in this example program, multiplication and division operations are performed before addition and subtraction. However, surrounding an operation with parentheses causes it to be evaluated first, working from innermost operation on out in the case of nested parentheses. These rules are part of a larger ruleset called *operator precedence*; each programming language has its own operator precedence ruleset.

Try This

To see what difference a set of parentheses makes, create this small project that uses the same mathematical expression as the program above.

- ✓ Close the Test Light Sensor Graphical project.
- ✓ Click New Project, name it something like Simple Operator Precedence, and save it to SimpleIDE > My Projects.
- ✓ Enter the code shown below, click Run with Terminal, and make note of the answer displayed.

```
/*
  Simple Operator Precedence
*/
#include "simpletools.h"                      // Include simple tools

int answer;

int main()                                     // main function
{
    answer = 200 * 300 / (300 + 187) - 100;   // Evaluate expression
    print("%d", answer);                      // Print the answer
}
```

- ✓ Add parentheses around the numerator, like this: `(200 * 300)`
- ✓ Re-run the program. Did that change the `answer`?
- ✓ Now, try removing the parentheses around `300 + 187`, and re-run the program. What is the `answer` now?

- ✓ Put the parentheses back in to the above statement, and then put a second set around the whole denominator, like this: `((300 + 187) - 100)`
- ✓ Re-run the program a final time. What is the `answer` now?

Are you convinced that parentheses make a difference yet?

Your Turn

- ✓ Look up Operator Precedence in a C reference guide.

Roaming with Light Sensors

Now, let's make the ActivityBot follow light! The following code should allow your robot to follow a flashlight beam, or navigate out of a dark room through a brightly lit doorway.

Navigate by Light

- ✓ Find a flashlight, the brighter the better.
- ✓ Click SimpleIDE's Open Project button.
- ✓ Open Navigate by Light.side from ...Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Click the Load EEPROM & Run button.
- ✓ Disconnect your robot from its programming cable, and set it in an open area.
- ✓ Set the power switch to 2, then press and release the reset button.
- ✓ Shine the flashlight on the floor in front of the ActivityBot, which should turn towards the bright spot. If your flashlight isn't very strong, you may need to point it directly at the sensors.
- ✓ Try shutting off the lights in the room, but opening a door to a brightly lit area outside. The ActivityBot should find its way out of the room.

How it Works

This program uses the simpletools and abdrive libraries. After initializing some variables, the program consists of an infinite loop which receives values from the phototransistors and uses them to adjust the servos' speeds to navigate toward whichever side detects brighter light.

First, the now-familiar `int` variables `lightLeft`, `lightRight`, and `ndiff` are declared for taking and using the sensor measurements. Then, `int` variables `speedLeft` and `speedRight` are declared for use with the `drive_speed` function later on.

The first six lines in the `main` function's `while(1)` loop are straight out the test programs; they take the light sensor measurements and store the values in `lightLeft` and `lightRight`.

The next line is the same `ndiff` equation used in Test Light Sensor Graphical. It divides `lightRight` by `lightRight + lightLeft`. The result is multiplied by 200, and then 100 is subtracted from it. This yields a value in the range of -100 and 100, which is assigned to `ndiff`. Positive `ndiff` values mean the left photoresistor is sensing brighter light, and negative `ndiff` values mean the right photoresistor is sensing brighter light. The farther the value is from zero, the greater the difference between what the two phototransistors are sensing.

The variables `speedLeft` and `speedRight` are then initialized to 100; keep this in mind when looking at the `if` code block that comes next.

The first condition translates to "if `ndiff` is greater than or equal to zero (brighter light on the the left), make `speedLeft` equal to 100 minus `ndiff*4`." The new value for `speedLeft` would be something less than 100, while `speedRight` remains 100, when those variables are used in the `drive_speed` function call right below the `if` block. For example, if `ndiff = 50`, this statement is the same as `speedLeft = 100 - (50*4)`, which equals -100. The result is `drive_speed(-100, 100)` which makes the ActivityBot rotate to the left toward the light.

However, if `ndiff` is NOT positive, the code drops to `else speedRight += (ndiff * 4)`. This translates to "make `speedRight` equal to 100 plus `ndiff*4`." This still yields a number smaller than 100; remember that `ndiff` is negative here. For example, if `ndiff = -25`, this statement is the same as `speedRight = 100 + (-25*4)` so `speedRight` ends up equal to zero, while `speedLeft` is still 100. This giving us `drive_speed(100, 0)`, causing the ActivityBot to pivot right towards the brighter light.

```
/*
 Navigate by Light.c
*/
#include "simpletools.h"
#include "abdrive.h"

int lightLeft, lightRight, ndiff;
int speedLeft, speedRight;

int main()
{
    while(1)
    {
        high(9);
        pause(1);
        lightLeft = rc_time(9, 1);

        high(5);
        pause(1);
        lightRight = rc_time(5, 1);

        ndiff = 200 * lightRight / (lightRight + lightLeft) - 100;

        speedLeft = 100;
        speedRight = 100;
        if(ndiff >= 0) speedLeft -= (ndiff * 4);
        else speedRight += (ndiff * 4);

        drive_speed(speedLeft, speedRight);
    }
}
```

Did You Know?

Phototaxis — This is the term that describes an organism moving its whole body in direct response to light stimulus. Moving towards brighter light is called positive phototaxis, and moving away from brighter light would then be negative phototaxis. While the Navigate by Light program makes the ActivityBot mimic positive phototaxis, in nature, it seems this behavior is usually found in microscopic organisms. Moths are attracted to light, but their varied and circling flight paths around a streetlamp demonstrate that their navigation systems are far more complex.

Try This

Would you like to make your ActivityBot mimic negative phototaxis? It can be done by replacing one single variable in the Navigate by Light code.

- ✓ Click Save Project As, and make a copy of the project with the name Navigate By Dark.
- ✓ In the `ndiff` equation, change the first use of `lightRight` to `lightLeft`, as shown below.

```
//-----change  
ndiff = 200 * lightLeft / (lightRight + lightLeft) - 100;
```

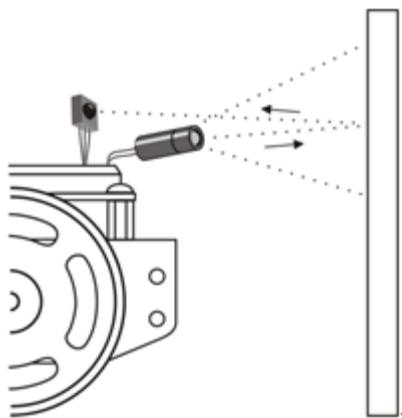
- ✓ With the power switch in position 1, load the code into EEPROM.
- ✓ Put the ActivityBot on the floor, move the switch to position 2, and push the reset button.
- ✓ Shine your flashlight at the ActivityBot. It should turn away from the light and seek shelter under a table or in a dark corner.

Navigate by Infrared Flashlights

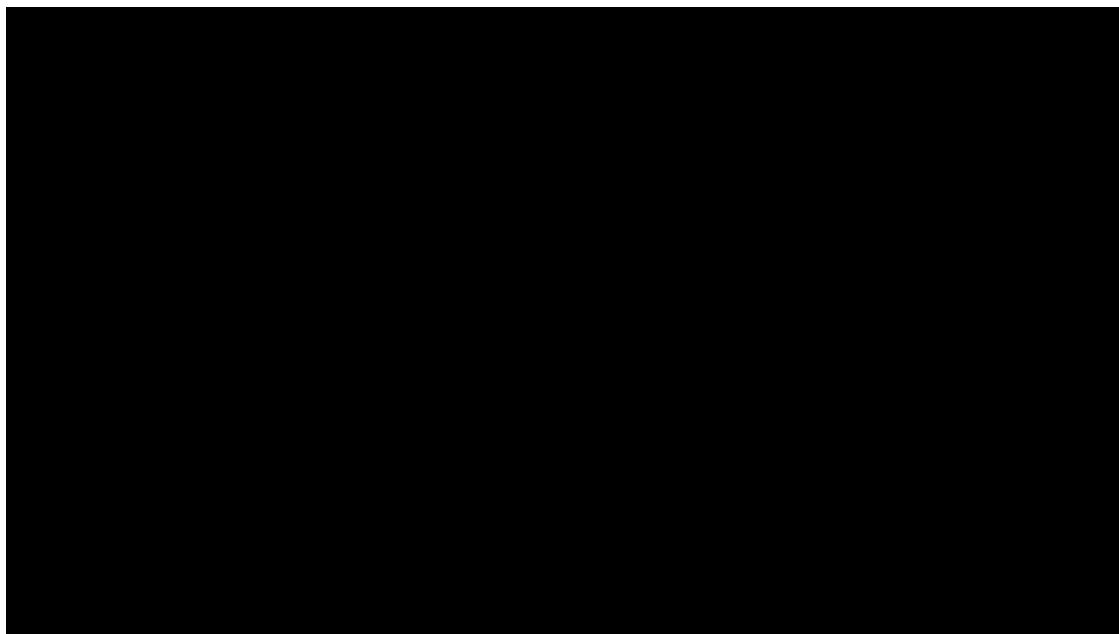
Many TVs and other equipment are controlled by infrared remotes. When you point the remote at the TV and press the channel-up button, it changes the channel for you. Pressing the button makes the remote flash a pattern of infrared light at the TV. Infrared light is not visible to humans, but the TV's infrared

receiver detects the flashing light pattern. The TV's microcontroller decodes this pattern and uses that information to change the channel for you.

The infrared light (IR LED) and receiver work great as "flashlights" and "eyes" for your ActivityBot. The IR LEDs shine forward, and the IR receivers detect reflections off of nearby obstacles.



The Propeller microcontroller receives reflection data from the IR receivers and uses it to make navigation decisions. In this way, your ActivityBot can roam around obstacles without first bumping into them.



Get the Example Code

- ✓ Make sure you have the latest [SimpleIDE software, Learn folder, and ActivityBot Library](#) [16].
- ✓ [Download the ActivityBot Navigate by Infrared Code](#) [27]
- ✓ Unzip the folder, and copy it to Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Follow the links below to continue with the ActivityBot tutorial.

Build the IR Sensor Circuits

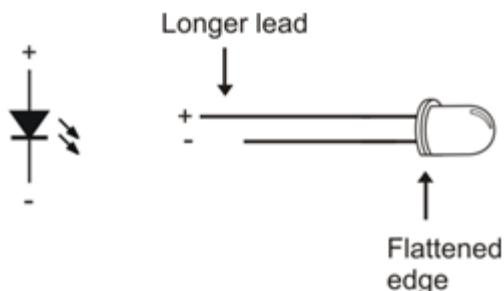
In this activity, you will build and test the infrared object sensors to make sure they detect objects in front of the ActivityBot.

Circuit

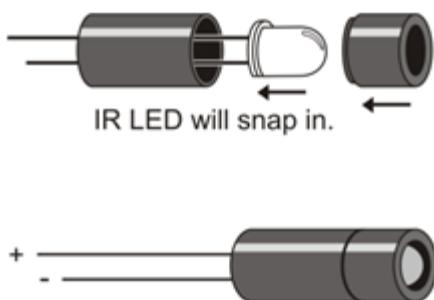
Parts Required

- (2) IR LEDs
- (2) IR receivers
- (2) 1 k-ohm resistors (brown-black-red)
- (2) 220-ohm resistors (red-red-brown)

- ✓ Find the two IR LEDs in your kit — they are the clear ones with dome-shaped (not flat) tops.



To direct the IR LED's light, just like a flashlight beam, we'll use an IR LED standoff (longer tube) and shield (shorter tube).

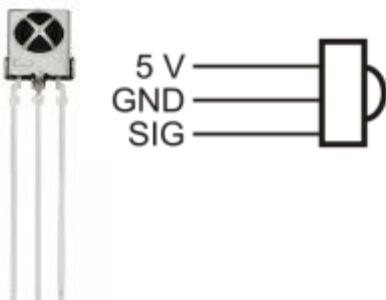


- ✓ Insert the IR LED's leads into the standoff tube, and out through the two holes at the bottom of the standoff.
- ✓ Press firmly and the IR LED should snap into place. If it doesn't, pull it out, give it a half turn, and try again.
- ✓ Fit the LED shield onto the standoff.



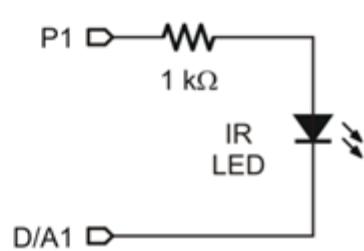
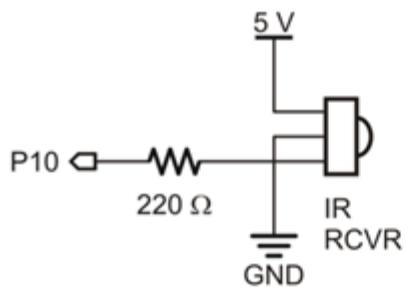
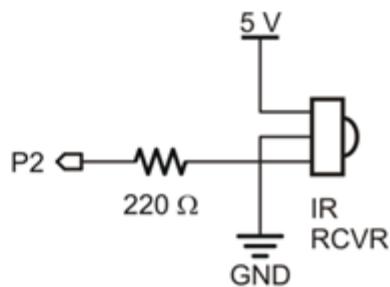
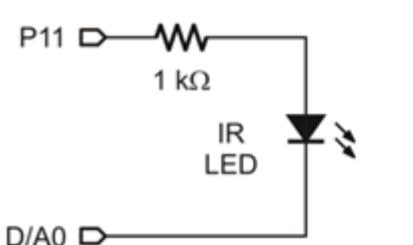
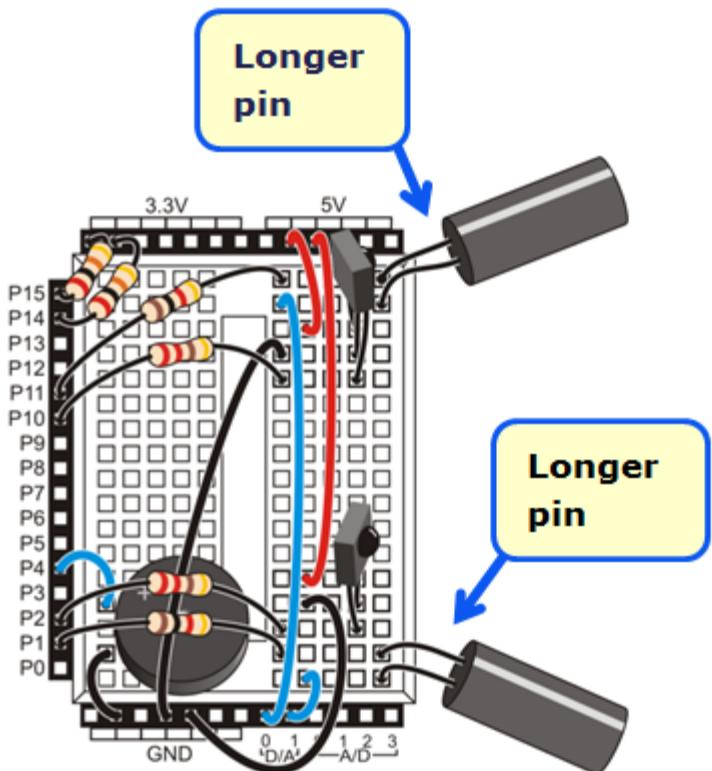
Although the shield friction-fits into the standoff, it can be helpful to place a small piece of clear tape around the tube where they meet to hold them together securely. Make sure the tape does not extend beyond the end of the tube or cover the IR LED.

- ✓ Find the IR receivers in your kit. If you have two different kinds, use the ones with the silver metal cases for this activity.



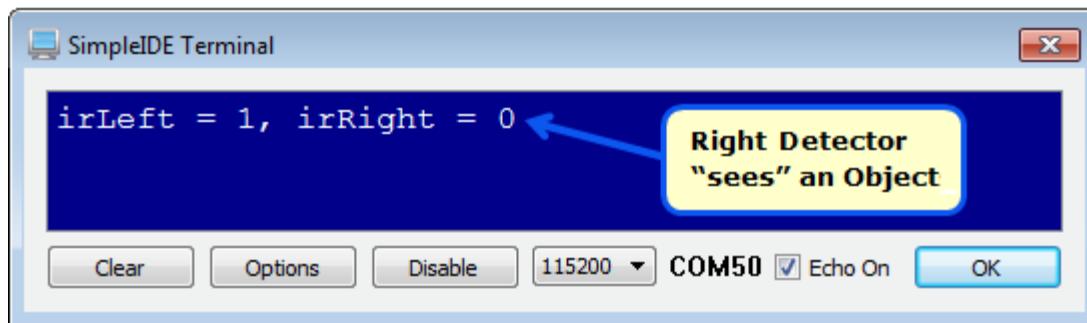
The IR LED's cathodes are connected to D/A0 and D/A1. For object detection, the D/A outputs are set to 0 V = GND. Their voltage can be increased to make the IR LEDs dimmer, for closer range detection. (They can even be tested at different voltages—levels of dimness—to get a rough idea of the object's distance, but that's for another activity...)

- ✓ Use this wiring diagram and schematic to build the infrared detectors.
- ✓ Make sure the longer IR LED anode pins are plugged into the rows shown in the wiring diagram.
- ✓ Make sure your IR receivers are the silver ones with metal cases (not the black plastic ones).



Test the IR Sensor Circuits

The test code displays the state of each IR detector: 1 = no infrared reflection detected, 0 = yes, infrared reflected from an object is detected. Before continuing, make sure that both sides can reliably detect objects.



Test Code

- ✓ Click SimpleIDE's Open Project button, and open Test IR Detectors from ...Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Click the Run with Terminal button.
- ✓ Point your ActivityBot so that both IR detectors are pointing away from any objects, and verify that both detectors report 1.
- ✓ Place your hand in front of (and facing) the right IR detector, about 10 cm (\approx 4 in) away. This should display `irLeft = 1, irRight = 0`.
- ✓ Repeat for the left IR detector and verify that the display updates to `irLeft = 0, irRight = 1`.
- ✓ Repeat for both, and verify that both display 0.
- ✓ If all the tests worked, you're ready to continue to the next section.

Troubleshooting

It's important to make sure both IR detectors work properly before continuing.

- ✓ If your IR output is stuck at 1 or 0, it usually indicates a wiring problem. Go back and check your wiring.
- ✓ If the IR flickers to zero when it shouldn't (because there are no objects in range) try turning off any nearby fluorescent lights, and retest.
- ✓ If the IR only sporadically detects when an object is right in front of it, check the color codes on the

resistors connected to P11 and P1. They should be brown-black-red for 1 kΩ. Also, make sure your ActivityBot is not in direct sunlight.

- ✓ Make sure both IR sensors are working well before continuing to the next activity.

What the IR Sensors Can't See



Remember, the IR sensor system is looking for reflected infrared light. Light-colored objects reflect infrared light well, while dark-colored objects absorb infrared light instead of reflecting it. So, if your IR sensors can't see your black shoes, or a black plastic wastebasket, don't worry, that is normal!

How it Works

First, `int` variables `irLeft` and `irRight` are declared to hold the IR detector output states. Then, inside `main`, the code sets P26 and P27 low to make sure that the D/A terminals are both providing 0 V to the IR LED cathodes.

The IR receivers are designed to only send detect signals if they see infrared light that flashes on/off at 38,000 times per second (38 kHz). So, in the `main` function's `while` loop, `freqout(11, 1, 38000)` sends high/low signals that repeat 38000 times per second to the IR LED's anode. This makes it flash on/off at that rate.

Next, `irLeft = input(10)` saves the IR receiver's output in the `irLeft` variable. If the IR receiver's output is high, meaning no IR reflection detected, `input(10)` returns a 1. If the receiver's output is low, meaning reflected IR was detected, `input(10)` returns a 0. The same process repeats for the right IR detector with `freqout(1, 1, 38000)` and `irRight = input(2)`.

```
/*
Test IR Detectors.c
*/
#include "simpletools.h"                                // Include simpletools
int irLeft, irRight;                                    // Global variables
int main()                                              // main function
{
    low(26);                                            // D/A0 & D/A1 to 0 V
    low(27);

    while(1)                                            // Main loop
    {
        freqout(11, 1, 38000);                          // Left IR LED light
        irLeft = input(10);                             // Check left IR detector
        freqout(1, 1, 38000);                           // Repeat for right detector
        irRight = input(2);

        print("%c irLeft = %d, irRight = %d",
              HOME,           irLeft,          irRight); // Display detector states
        pause(100);                                     // Pause before repeating
    }
}
```

After the two infrared detection results are stored in `irLeft` and `irRight`, `print(%c irLeft = %d, irRight = %d", HOME, irLeft, irRight)` does several things. The `%c` makes it send the `HOME` value (1) to SimpleIDE Terminal. That sends the cursor to the top-left position. After that, `irLeft = %d, irRight = %d` displays `irLeft` = followed by the 1 or 0 that `irLeft` stores, then `irRight` = followed by the 1 or 0 that `irRight` stores. Last, `pause(100)` keeps the rate that the SimpleIDE Terminal is updated down in the 10 times per second neighborhood.

Did You Know?

Infrared — It means “below red”, and it refers to the fact that the light’s frequency is below that of red on the color spectrum.

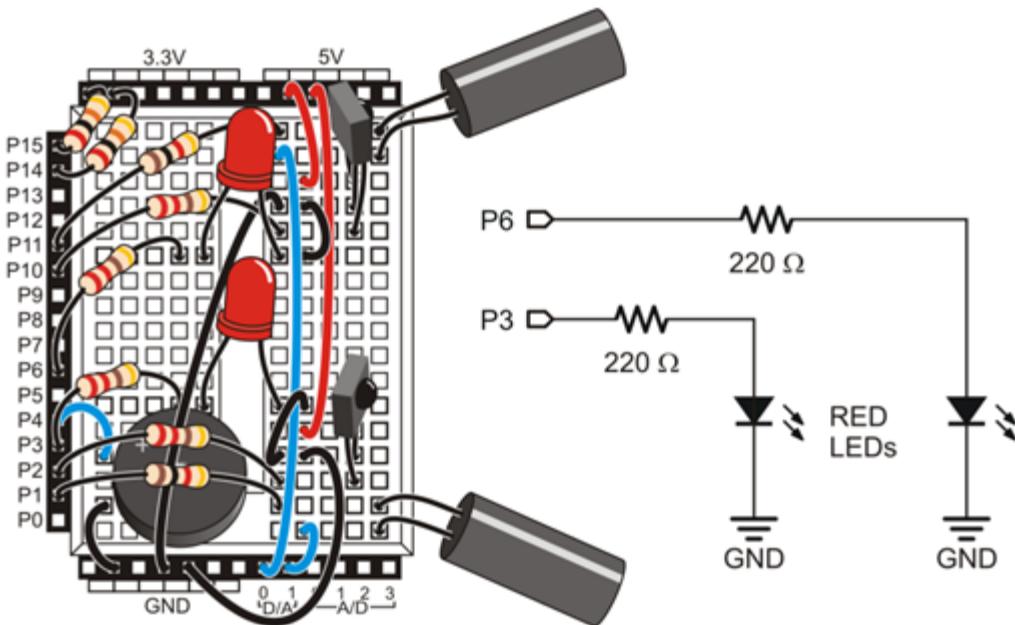
Filtering — The IR receiver has a filter built in that makes it look for infrared that flashes on/off 38000 times per second (38 kHz). This allows it to differentiate between infrared coming from the TV remote and other IR sources such as halogen and incandescent lamps as well as sunlight streaming in through a nearby window.

Sunlight — Even though the IR receiver filters for sunlight, direct sunlight can often swamp the IR LED’s signal. Try to keep it indoors and out of any sunlight streaming in through windows.

Try This

The P26 and P27 lines are currently in use providing 0 V to the IR LED cathodes, so we cannot use their LEDs for indicating the IR detector states like we did with whiskers. So, let’s build some LEDs on the breadboard for indicators.

- ✓ Collect two red LEDs, two $220\ \Omega$ resistors (red-red-brown), and two black jumper wires from your kit.
- ✓ Use those parts to build the two added LED indicator circuits shown here.



Next, modify the code to turn a given light on if its IR receiver returns 0, or off if it returns 1.

- ✓ Use the Save Project As button to make a copy of your project, and name it Test IR Detectors (Try This)
- ✓ Add two `if` statements between the `print` and `pause` function calls, as shown below.
- ✓ Run the program and verify that each light turns on when you put an obstacle in front of the IR object detector that's next to it.

```

print("%c irLeft = %d, irRight = %d",
      HOME,           irLeft,          irRight);

if(irLeft == 0) high(6); else low(6); // <-add
if(irRight == 0) high(3); else low(3); // <-add

pause(100);
}

```

Your Turn

You can further modify your IR detection code from Try This so that it sounds an alarm if it detects IR interference from the environment. This is a useful test to find out if nearby fluorescent lights are sending out signals that are interfering with your detector. The key is to not emit any infrared with the IR LEDs. If the receivers still send a low signal, there must be IR interference coming from another source. If interference is detected, make the piezospeaker get your attention with a series of six 50 ms, 4 kHz chirps separated by six 50 ms pauses. You can use a TV remote to test this.

- ✓ Use the Save Project As button to save a copy of Test IR Detectors (Try This) and rename it Test

IR Detectors (Your Turn).

- ✓ Comment out both `freqout` function calls, along with `print` and `pause`.
- ✓ Point a TV remote at your IR receivers, and press and hold a button. Verify that the indicator lights come on.
- ✓ Add an `if` block that plays the series of six 50 ms, 4 kHz chirps separated by six 50 ms pauses if either `irLeft` or `irRight` hold a 1.
- ✓ Try adding in blinking lights along with the audio alarm.

Roaming with Infrared Flashlights

With your IR object sensors built and tested, you are ready to make your ActivityBot roam and avoid obstacles without bumping into them.

IR Roaming Code

The IR Roaming code example makes the ActivityBot go forward until it detects an obstacle. If it sees an obstacle on the left, it'll turn right. Likewise, if it sees one on the right, it'll turn left, and if it sees obstacles on both left and right, it'll back up.

Not all obstacles are visible.



Many black objects will absorb infrared light instead of reflecting it. If in doubt, use your LED indicator program from the Build and Test the IR Detectors "Try This" section. The LEDs will show you if the object is visible to the IR object detectors.

- ✓ Click SimpleIDE's Open Project button.
- ✓ Open IR Roaming from ...Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorials.
- ✓ Click the Load EEPROM & Run button.
- ✓ Let your robot loose on the floor, or in an obstacle course you have created and see how it does.

How it Works

This example is almost identical to Test IR Detectors.c from Build and Test the IR Sensor Circuits. The code changes are:

- A `drive_setRampStep` call was added to the initialization.
- The `print` statement was replaced by an `if...else if...else if...else` statement.
- The `pause(100)` was removed.

```
/*
IR Roaming.c

Use IR LEDs and IR receivers to detect obstacles while roaming.

*/
#include "simpletools.h"                                // Library includes
#include "abdrive.h"

int irLeft, irRight;                                    // IR variables

int main()                                              // main function
{
    low(26);                                            // D/A0 & D/A1 to 0 V
    low(27);

    drive_setRampStep(12);                             // Max step 12 ticks/s every 20 ms

    while(1)
    {
        freqout(11, 1, 38000);                         // Check left & right objects
        irLeft = input(10);

        freqout(1, 1, 38000);
        irRight = input(2);

        if(irRight == 1 && irLeft == 1)                  // No obstacles?
            drive_rampStep(128, 128);                   // ...full speed ahead
        else if(irLeft == 0 && irRight == 0)           // Left & right obstacles?
            drive_rampStep(-128, -128);                 // ...full speed reverse
        else if(irRight == 0)                           // Just right obstacle?
            drive_rampStep(-128, 128);                  // ...rotate left
        else if(irLeft == 0)                           // Just left obstacle?
            drive_rampStep(128, -128);                  // ...rotate right
    }
}
```

The `drive_rampStep` function is designed to be used in loops. After our call to `drive_setRampStep(12)`, if a loop calls `drive_rampStep(128, 128)` it will increase speed 12 ticks/second at a time toward a final 128 ticks per second every time it's called. If it's calling 50 times per second (20 ms pauses), it will take 11 repeated calls (11/50ths of a second) to get to full speed. After the ActivityBot reaches full speed, repeated calls to `drive_rampStep(128, 128)` don't change the speed any because it's already there. If the loop then starts repeating `drive_rampStep(-128, -128)` calls, it'll take 22/50ths of a second (getting close to half a second) to ramp from full speed forward to full speed reverse.

The `if...else if...else if...else` statement is what uses the `irLeft` and `irRight` detection result values for navigation. For example, `if(irRight == 1 && irLeft == 1)` it means that no objects are detected, so `drive_rampStep(128, 128)` sends the ActivityBot a 12 ticks/second step toward 128 ticks per second if it both wheels aren't already running at that speed. If the ActivityBot sees an obstacle on its right, the `else if(irRight == 0)` condition becomes true, so each time through the loop, it takes 12 ticks/second steps toward rotating left in place every 50th of a second. Rotating in in place is achieved with the left wheel turning at full speed backwards at -128 ticks/second and the right turning at 128 ticks/second forward.

As the ActivityBot rotates left to avoid the right obstacle, the obstacle will disappear from the right detector's view. At that point, the detection states will go back to `irLeft == 1` and `irRight == 1`, which will make the first `if` statement true again, and `drive_rampStep(128, 128)` will start ramping back to full speed forward.

A pause between `while(1)` loop repetitions is not needed because the `drive_rampstep` function delays 1/50th of a second before returning.

Did You Know?

drive_setRampStep — This function sets the maximum change in speed that `drive_rampstep` can cause every 1/50 of a second. The abdrive library defaults to 4, which means that the largest change in speed allowed is 4 ticks/second every 50th of a second. Although that default makes `drive_ramp` and `drive_rampstep` maneuvers nice and smooth, it's not enough change in a small amount of time to detect and avoid obstacles at full speed. So, `drive_setRampStep(12)` triples the speed change allowed every 50th of a second. Not quite as smooth, but responsive enough to avoid the obstacles.

Try This

You can reduce the roaming speeds by changing the 128's in the program to lower values.

- ✓ Use the Save Project As button to save a copy of your project in ...Documents\SimpleIDE\My Projects\ActivityBot Projects.
- ✓ Try changing all the 128 values in your program to 64. Make sure to leave all the negative signs in place.
- ✓ Run the modified program and let the ActivityBot roam at half speed.

Your Turn

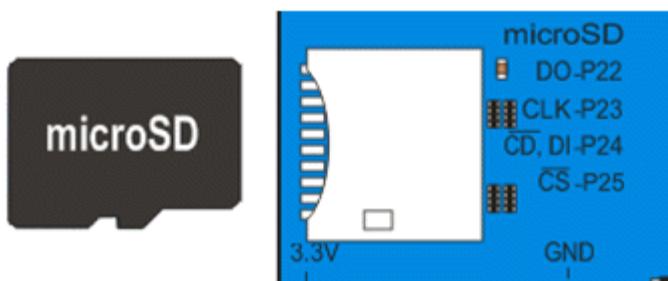
Increasing the value used in the `drive_setRampStep` function call to a higher value will make the robot more responsive, but also more twitchy. Reducing the value will make it smoother, but at some point, it won't respond quickly enough and it will run into obstacles.

- ✓ Re-open the original, unmodified IR Roaming example from ...Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorials.
- ✓ Click the Save Project As button to save another copy and give it a new name.
- ✓ Experiment with increasing and decreasing the `drive_setRampStep` parameter.
- ✓ How low can you go before it starts running into the same obstacles it used to be able to see and avoid.

- ✓ How large can you make it before the ActivityBot's behavior becomes noticeably twitchy?

SD Card Games

Your ActivityBot Kit includes a microSD card, which fits the built-in card reader on the Activity Board. The card reader is connected to the Propeller I/O pins labeled next to it.



There are lots of things you can do with an SD card on the ActivityBot:

- Store a list of maneuvers in a text file for the C program to fetch and perform.
- Log sensor data as the robot drives.
- Store WAV files to play music, speech or sound effects (if you add a speaker!)

The first step is to get familiar with using the microSD card.

- ✓ If you have not used the microSD card with your Activity Board yet, try the [SD Card Data tutorial](#) [28], then return here.

Welcome back!

Get the Example Code



Library Alert! This activity requires abdrive version 0.5.5 or higher, from the ActivityBot Library (2013-10-31). Click the links under "Stay Current!" to update your SimpleIDE and libraries

- ✓ [Download the ActivityBot SD Card Games Code](#) [29]
- ✓ Unzip the folder, and copy its SD Navigation folder to ...
Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial.
- ✓ Follow the links below to continue with the ActivityBot tutorials.

Text File Maneuver List

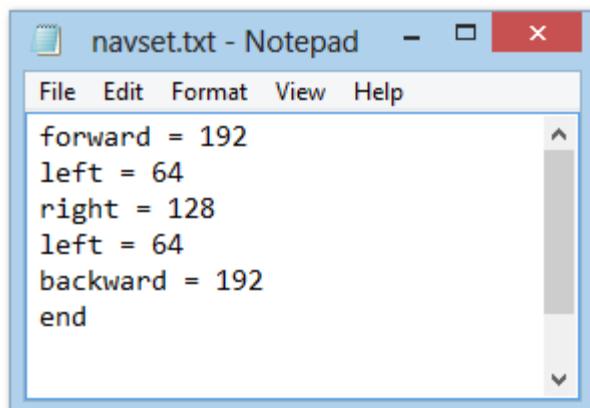
So far in this series, we've been using C language programs to make the ActivityBot go. This time, we will use a C program that opens a text file from the SD card. The text file contains a program in another super-simple language we are inventing right now. Let's call it "AB."

The AB programming language has only five commands:

- forward
- backward
- left
- right
- end

Each command has a single argument: the number of encoder ticks that set the distance the ActivityBot should travel.

The C program will *parse* each line of text, that is, analyze the characters and relate them to predefined actions elsewhere in the C code. In this way, the C code *interprets* the AB commands to make ActivityBot go.



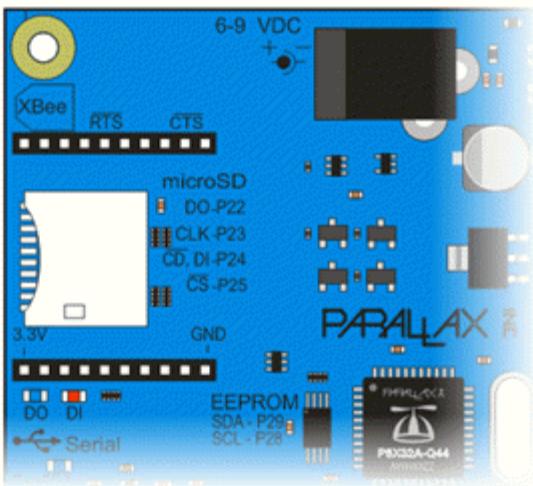
A screenshot of a Windows Notepad window titled "navset.txt - Notepad". The window shows the following text content:

```
forward = 192
left = 64
right = 128
left = 64
backward = 192
end
```

Circuit

We will not be using any sensor circuits here, just the SD card.

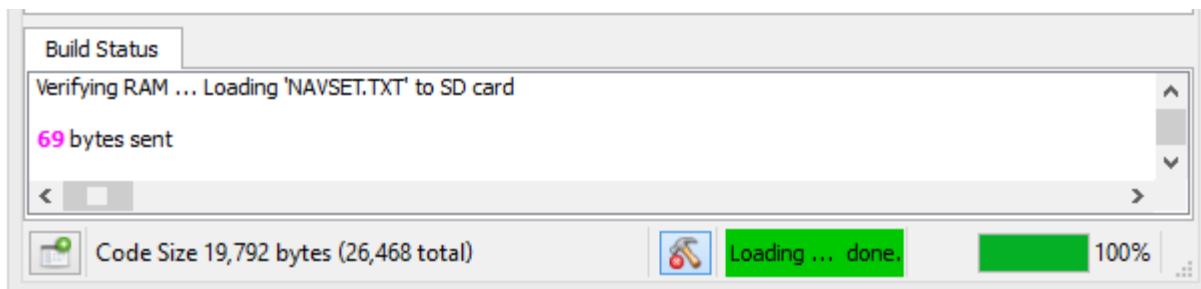
- ✓ Insert your kit's microSD card into your Propeller Activity Board's SD socket.



Test Code

The zip for this application contains a text file that you will copy to your SD card. After that, you will load a C application into the Propeller that will read the file and execute its AB language commands.

- ✓ Click SimpleIDE's Open Project button.
- ✓ Open AB Interpreter.side from ...Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial\SD Navigation.
- ✓ Click Program and select File to SD Card.
- ✓ Browse to ...Documents\SimpleIDE\Learn\Examples\ActivityBot Tutorial\SD Navigation again, select NAVSET.TXT and click Open. This will load the file onto the SD card.
- ✓ Check the build status pane to make sure it loaded — click the little hammer icon in the bottom to see the status.



Now it is time to run the AB Interpreter.side project.

- ✓ Click Load EEPROM and Run.
- ✓ Put your ActivityBot on the floor where it has some room. Switch power to 2, and reset.

- ✓ Verify that your ActivityBot goes forward, then spins in place to the left, then to the right, then to the left again, and backs up to about where it started.

How it Works

The code starts with a file pointer and declarations for character buffers and a variable to use while analyzing the AB command text strings.

```
FILE *fp;                                // File pointer for SD
char str[512];                           // File buffer

char cmdbuf[10];                          // Command buffer
char valbuf[4];                           // Text value buffer
int val;                                  // Value
```

Inside the `main` routine, the first three lines set up the SD card connection. They should be familiar from the SD Card Data tutorial.

```
int main()                                // Main function
{
    int DO = 22, CLK = 23;                 // SD I/O pins
    int DI = 24, CS = 25;
    sd_mount(DO, CLK, DI, CS);           // Mount SD file system
```

The line that follows opens the `navset.txt` file on the SD card for reading. Then, the `fread` function copies up to 512 bytes from `navset.txt` to a 512 byte buffer named `str`. Next, `strlen(str)` figures out how many bytes are actually in the string and copies that number to the `strLength` variable. Before any modifications, there are about 90 characters in the `navset.txt` file.

```
fp = fopen("navset.txt", "r");             // Open navset.txt
fread(str, 1, 512, fp);                  // navset.txt -> str
int strLength = strlen(str);             // Count chars in str
```

There are just two more setup tasks: declaring a variable for indexing, and making sure the robot is not moving to begin with:

```
int i = 0;                                // Declare index variable
drive_speed(0, 0);                         // Speed starts at 0
```

Now the action happens! The program's first task inside the `while(1)` is *parsing* the content of `navset.txt` is to find the first *printable character* — not spaces or line feeds. So, `while(!isalpha(str[i])) i++` repeatedly checks each successive character in the `str` string until it finds an alphanumeric character, which is assumed to be the first character in the next command. Then, `sscanf(&str[i], "%s", cmdbuf)` copies the word — all the characters until it reaches another non-alphanumeric — into a smaller character array named `cmdbuf`. Then, the index of the next character to check in `str` is updated with `i += strlen(cmdbuf)`. Finally, `if(!strcmp(cmdbuf, "end")) break` checks to see if the `end` AB command in the text file has been reached. If it has, `break` exits the `while(1)` loop.

```
while(1)                                    // Loop through commands
```

```

{
    // Parse command
    while(!isalpha(str[i])) i++;           // Find 1st command char
    sscanf(&str[i], "%s", cmdbuf);          // Command -> buffer
    i += strlen(cmdbuf);                  // Idx up by command char count
    if(!strcmp(cmdbuf, "end")) break;      // If command is end, break
}

```

The distance argument for the AB command is parsed using similar techniques. The statement `while(!isdigit(str[i])) i++` checks and rejects any character that's not a digit. After it does find a digit, `sscanf(&str[i], "%s", valbuf)` copies the value to a smaller character array named `valbuf`. Again, the index of for the next character to check in str is updated, this time with `i += strlen(valbuf)`. After figuring out the position of the next character to check in the str array, it's time to convert the character representation of the distance argument to a value the program can use. That's what `val = atoi(valbuf)` does. The term `atoi` is short for ascii to integer.

```

// Parse distance argument
while(!isdigit(str[i])) i++;           // Find 1st digit after command
sscanf(&str[i], "%s", valbuf);          // Value -> buffer
i += strlen(valbuf);                  // Idx up by value char count
val = atoi(valbuf);                  // Convert string to value
}

```

After parsing the command and distance arguments, all that's left is to check what the command is and use the distance arguments accordingly. To figure out what command string was actually stored in `cmdbuf`, we use `strcmp`. This function returns a zero if two strings match, or nonzero if they do not. So, `if(strcmp(cmdbuf, "forward") == 0)` `drive_goto(val, val)` checks to see if the contents of `cmdbuf` match the string "forward". If it does, then it uses the distance argument stored in `val` and `drive_goto` to make the activitybot go the distance. If `cmdbuf` instead stored the string "backward", the first if statement would not be true, but `else if(strcmp(cmdbuf, "left") == 0)`... would be true.

```

// Execute command
if(strcmp(cmdbuf, "forward") == 0)           // If forward
    drive_goto(val, val);                     // ...go forward by val
else if(strcmp(cmdbuf, "backward") == 0)        // If backward
    drive_goto(-val, -val);                   // ... go backward by val
else if(strcmp(cmdbuf, "left") == 0)            // If left
    drive_goto(-val, val);                    // ...go left by val
else if(strcmp(cmdbuf, "right") == 0)           // If right
    drive_goto(val, -val);                   // ... go right by val
}

```

Once the parsing detects the AB end command and thus breaks out of the `while(1)` loop, the only thing left to do is close the `navset.txt` file on the SD card:

```

}
fclose(fp);                                // Close SD file
}

```

Here is the full code listing:

```

/*
AB Interpreter.c

http://learn.parallax.com/activitybot

Fetches text commands from file on SD card, interprets them,
and executes maneuvers. Requires abdrive 0.5.5 from
ActivityBot library 2013-10-31 or later.

*/
#include "simpletools.h"                      // Library includes

```

```

#include "abdrive.h"

FILE *fp;                                // File pointer for SD
char str[512];                            // File buffer

char cmdbuf[10];                          // Command buffer
char valbuf[4];                           // Text value buffer
int val;                                  // Value

int main()                                 // Main function
{
    int DO = 22, CLK = 23;                 // SD I/O pins
    int DI = 24, CS = 25;
    sd_mount(DO, CLK, DI, CS);
    fp = fopen("navset.txt", "r");

    fread(str, 1, 512, fp);               // navset.txt -> str
    int strLength = strlen(str);          // Count chars in str
    int i = 0;                            // Declare index variable

    drive_speed(0, 0);                   // Speed starts at 0

    while(1)                             // Loop through commands
    {
        // Parse command
        while(!isalpha(str[i])) i++;      // Find 1st command char
        sscanf(&str[i], "%s", cmdbuf);   // Command -> buffer
        i += strlen(cmdbuf);             // Idx up by command char count
        if(strcmp(cmdbuf, "end")) break;  // If command is end, break

        // Parse distance argument
        while(!isdigit(str[i])) i++;      // Find 1st digit after command
        sscanf(&str[i], "%s", valbuf);   // Value -> buffer
        i += strlen(valbuf);             // Idx up by value char count
        val = atoi(valbuf);              // Convert string to value

        // Execute command
        if(strcmp(cmdbuf, "forward") == 0) // If forward
            drive_goto(val, val);        // ...go forward by val
        else if(strcmp(cmdbuf, "backward") == 0) // If backward
            drive_goto(-val, -val);       // ... go backward by val
        else if(strcmp(cmdbuf, "left") == 0) // If left
            drive_goto(-val, val);        // ...go left by val
        else if(strcmp(cmdbuf, "right") == 0) // If right
            drive_goto(val, -val);        // ... go right by val
    }

    fclose(fp);                            // Close SD file
}

```

Did You Know?

Many of the functions used for files, strings, and characters come from the Standard C Libraries built into Propeller GCC. The simpletools library includes them for you, so they do not need to be included at the top of AB Interpreter. Here is a list of standard libraries and the functions used:

- **stdio.h:** `fopen`, `fread`
- **string.h:** `strcmp`, `strlen`
- **ctype.h:** `isalpha`, `isdigit`
- **stdlib.h:** `atoi`

My favorite resource for looking up and reading the documentation on these functions is IBM's C library reference: <http://pic.dhe.ibm.com/infocenter/is...sc41560702.htm> [30]

Try This

Let's try expanding our AB language with two new commands: `pivotLeft` and `pivotRight`. These commands will keep one wheel still while turning the other so that the ActivityBot "pivots" on one wheel.

- ✓ Modify NAVSET.TXT to use the new commands as shown below, then re-save it to the SD card.

```
forward = 64
pivotLeft = 53
pivotRight = 106
forward = 64
end
```

- ✓ Update AB Interpreter.side so it knows what to do with the new commands. This means adding two new `else if (strcmp...` conditions and `drive_goto` calls.

```
// Execute command
if(strcmp(cmdbuf, "forward") == 0)           // If forward
    drive_goto(val, val);                      // ...go forward by val
else if(strcmp(cmdbuf, "backward") == 0)        // If backward
    drive_goto(-val, -val);                    // ... go backward by val
else if(strcmp(cmdbuf, "left") == 0)            // If left
    drive_goto(-val, val);                     // ...go left by val
else if(strcmp(cmdbuf, "right") == 0)           // If right
    drive_goto(val, -val);                     // ... go right by val
else if(strcmp(cmdbuf, "pivotLeft") == 0)         // <-- Add
    drive_goto(0, val);                       // <-- Add
else if(strcmp(cmdbuf, "pivotRight") == 0)        // <-- Add
    drive_goto(val, 0);                       // <-- Add
}
```

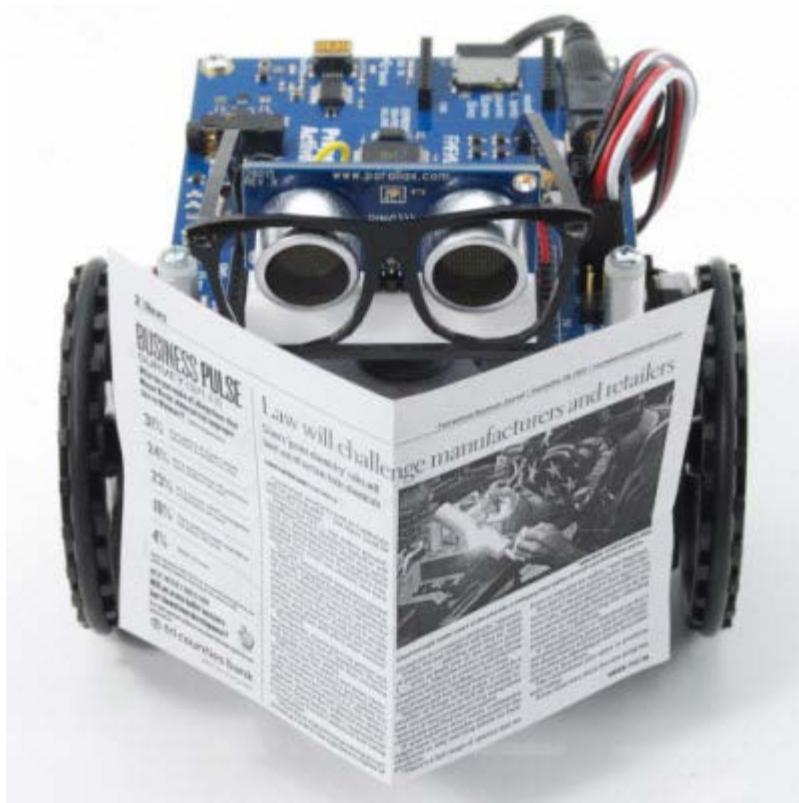
- ✓ Load EEPROM and run to run the modified example.
- ✓ Verify that the ActivityBot goes forward a wheel turn, pivots about a quarter turn to the left, then about a half turn to the right, then goes forward and stops.

Your Turn

This program can also be modified to detect and handle line comments. For example, you could place REM (for remark) at the beginning of the line that contained programmer's notes. You could support this in the program by checking for the string, and then using `while(str[i] != '\n' && str[i] != '\r') i++` to disregard all the characters to the end of the line.

Also consider that this current example limits the program size to 512 characters. Modifying the program to load the next 512 characters could be tricky. How would you approach it?

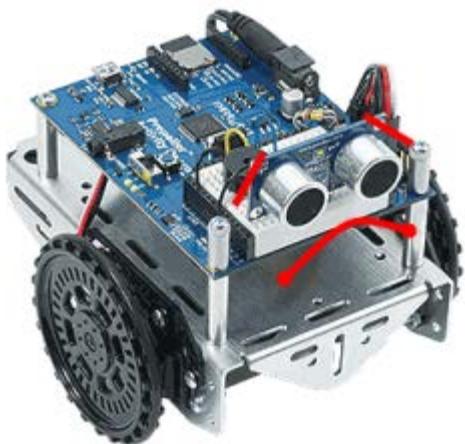
Extras



The "Extras" section includes optional activities, troubleshooting tips, and links to other interesting things to do with your ActivityBot, or the Propeller Activity Board. Sort of like an appendix.

Troubleshooting

Is your ActivityBot having trouble?



This page lists some of the most common problems encountered when calibrating the ActivityBot, or programming it to drive for the first time.

If you don't find your specific problem here, or need more help, contact Parallax Technical Support by email (support@parallax.com [31]), or by phone 916 -624-8333 (toll-free: 888 99-STAMP within USA only). You can also [post a question in the Learn Forum](#) [32] to get help from the community.

Issue: The ActivityBot is on and receiving power, but will not move when programmed, or resets itself while running.

Solution: First, if the servos do not turn at all, make sure the power switch is in Position 2 (which powers the servo headers), and not Position 1. If that doesn't help, check the batteries. Low batteries or batteries that are placed in backward will not provide enough power to the ActivityBot to run all of its components effectively. This can result in slower speeds, resets, or loss of functionality.

Issue: The ActivityBot is on and receiving power (with new batteries), but it will not move at all or moves slowly in a twitchy manner.

Solution: First, if the servos do not turn at all, make sure the power switch is in Position 2 (which powers the servo headers), and not Position 1. If that doesn't help, check the servo port jumper positions. The jumper for P12 and P13 should be set to VIN — if it is set to 5V the servos will not receive enough power. Follow the [Electrical Connections](#) [33] page instructions for moving the jumper, and then run the calibration again.



DO NOT MOVE THE JUMPER WHILE POWER PLUG OR USB CABLE IS CONNECTED TO THE BOARD.

Issue: The ActivityBot's servos and encoders are powered when the 3-position switch is in position 1.

Solution: Potentially, a short-circuit has damaged your ActivityBoard. Position 1 on the 3-position switch should not power the 3-pin headers above the breadboard that the servos and encoders are plugged into. This problem can often be caused if the shunt jumper for P12 & P13 was moved while the Activity Board was receiving power from the USB port or barrel jack. Unfortunately, there is no solution for this problem once it has occurred, please contact technical support (see contact information at the top of this page).

Issue: When the ActivityBot is connected to the computer via USB cable, no COM port registers for it and/or the computer displays an error (no board detected, or board may not be working properly).

Solution: Check the USB connection on the Activity Board. The USB connection port on the ActivityBot is designed to fit tightly to the Mini B connector. Even though it may feel secure, sometimes the cable may not be inserted fully into the port, and this will cause your computer not to recognize the connection or give an error.

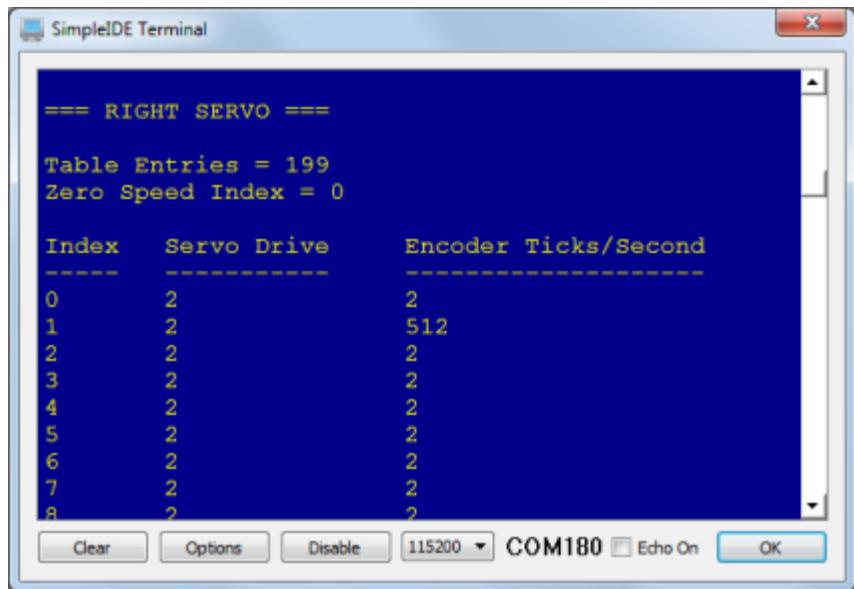
Issue: The ActivityBot moves the in opposite direction from what it was programmed to do.

Solution: Check your servo cables. A common mistake is to accidentally switch the cables for left and right servos, which causes the ActivityBot to go backwards when it should go forward, left when it should go right, etc. Check to make sure the left and right connections match the [Electrical Connections](#) [33] page instructions, and then run the calibration again.

Issue: The ActivityBot moves correctly during calibration, but then doesn't move when trying any other tutorial activity.

Solution: The right and left encoder cables may be swapped. This prevents the ActivityBot from correctly calibrating or using information from the encoders. A way to check if this is the problem is to look at the calibration table or [try the test program here](#) [34]. A successful calibration table shows columns of ascending and descending values, like the one on the [Calibrate Your ActivityBot](#) [35] page. If the encoder cables are swapped, there will be long sequences of identical numbers instead, like in the image below. Reconnect the encoder cables following the [Electrical Connections](#) [33] page instructions, and then run the

calibration again.



SimpleIDE Terminal

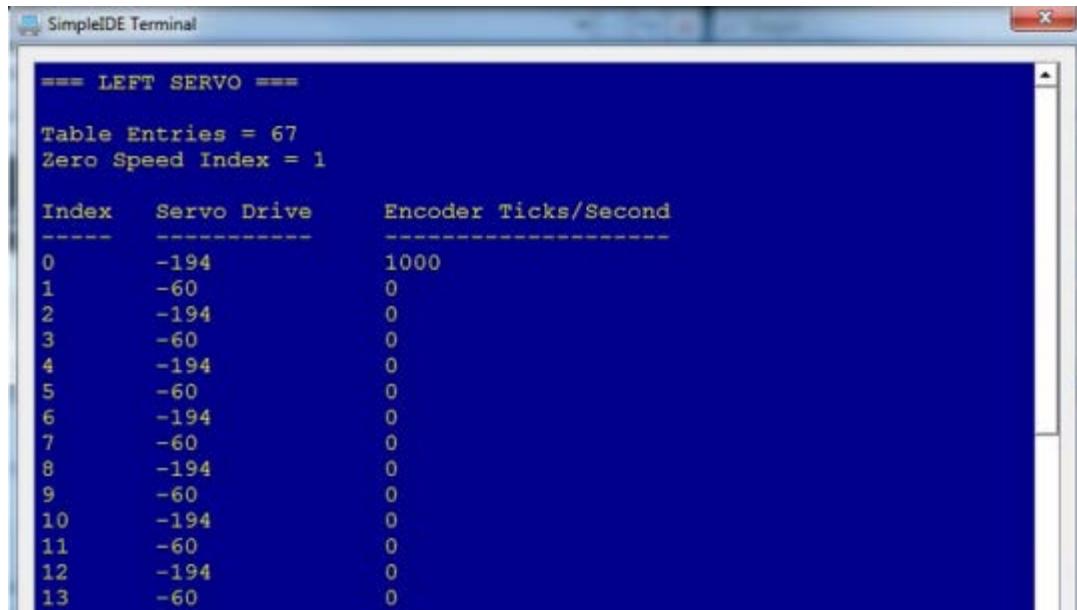
```
==== RIGHT SERVO ====
Table Entries = 199
Zero Speed Index = 0

Index   Servo Drive    Encoder Ticks/Second
-----  -----
0       2               2
1       2               512
2       2               2
3       2               2
4       2               2
5       2               2
6       2               2
7       2               2
8       2               2
```

Clear Options Disable 115200 COM180 Echo On OK

Issue: During calibration, the ActivityBot sits still for an unusually long time, and one or both of its wheels do not start turning.

Solution: Check that you don't have a servo and encoder cable swapped (to quickly test this, [try the test program here](#) [34]). If so, your Interpolation table will look similar to the one below (the image is for a left servo for left encoder switch). If the left wheel doesn't turn, your left servo and encoder cables might be switched. If the right wheel doesn't turn, your right servo and encoder cables might be switched. If no wheel turns, you might have both right and left swapped. Recheck your connections using the [Electrical Connections](#) [33] page and try the calibration again.



SimpleIDE Terminal

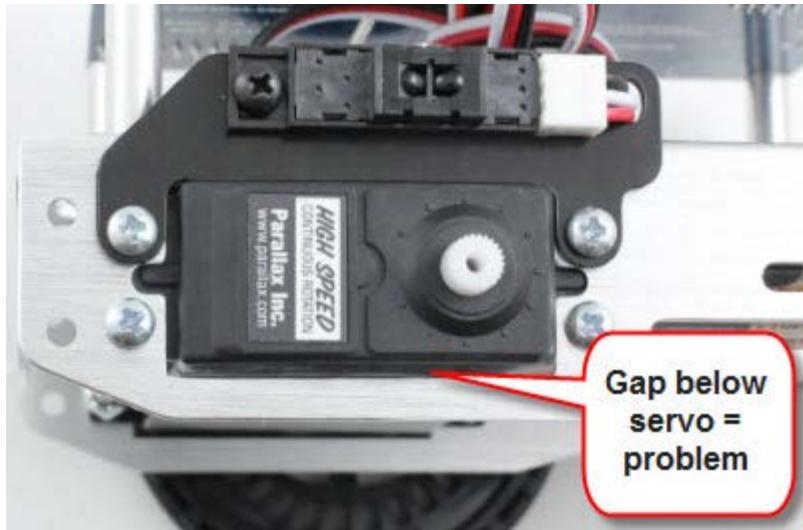
```
==== LEFT SERVO ====
Table Entries = 67
Zero Speed Index = 1

Index   Servo Drive    Encoder Ticks/Second
-----  -----
0       -194            1000
1       -60              0
2       -194            0
3       -60              0
4       -194            0
5       -60              0
6       -194            0
7       -60              0
8       -194            0
9       -60              0
10      -194            0
11      -60              0
12      -194            0
13      -60              0
```

Issue: The ActivityBot calibration routine went fine, but instead of driving straight, it drives in a jerky, wavy line.

Solution: Check the position of the servo inside its hole in the chassis.

Notice that there is a little bit of space around the servo. If the servo is tight against the top edge of the hole close to the encoder, with a gap left below the servo, the beam of infrared light coming from the encoder sensor might be missing the wheel spokes and hitting the solid ring below them instead.



To fix this, loosen the servo's locknuts and then reposition the servo so the gap is between the servo and the encoder. This should ensure that the encoder sees only spokes and holes. Make sure that the servo doesn't shift position when you retighten the locknuts.



(Note: the encoder bracket was redesigned in October 2013 to mitigate this issue.)

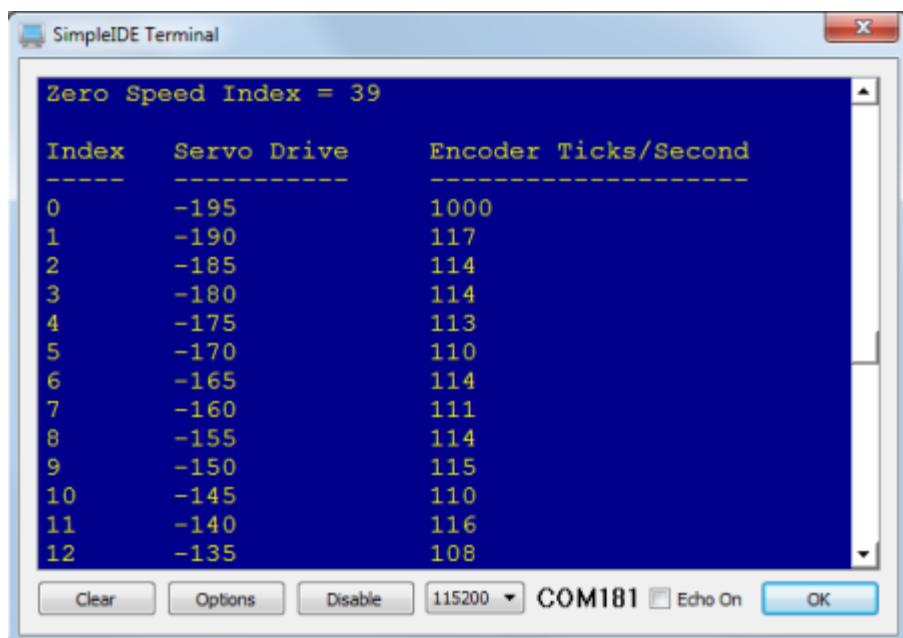
Issue: The calibration table display shows null values, or values that alternate from high numbers to low numbers quickly (repeating 161 - 0, for example).

Solution: The encoder IR sensor may not be functioning correctly. Check the encoder cable plug on both ends to make sure it is seated properly. Re-run the calibration a second time and check the interpolation table again using the Display Calibration program. If there is no change, please get in touch with Technical Support using the contact information above. There is no fix for a malfunctioning encoder; it may need to be replaced.

Issue: The ActivityBot appears to run the calibration routine, but it takes a long time. When I tried to look at the calibration with ActivityBot Calibration Display.side, the SimpleIDE terminal was blank.

Solution: First, check to make sure the resistors connecting P14 and P15 to 3.3 V are marked red-black-orange (20 k-ohm). Next, check to make sure your encoder cables are not plugged into the 3-pin headers upside down. White wire should be near the top edge of the board, black wire should be near the 5V labels.

Issue: My ActivityBot completed the calibration, but seemed to move very slowly. The interpolation table's highest values were more like the table below than the expected output shown in [Calibrate Your ActivityBot](#) [35].



The screenshot shows a Windows-style terminal window titled "SimpleIDE Terminal". The window contains a table of calibration data. At the top, it says "Zero Speed Index = 39". Below that is a table with three columns: "Index", "Servo Drive", and "Encoder Ticks/Second". The data is as follows:

Index	Servo Drive	Encoder Ticks/Second
0	-195	1000
1	-190	117
2	-185	114
3	-180	114
4	-175	113
5	-170	110
6	-165	114
7	-160	111
8	-155	114
9	-150	115
10	-145	110
11	-140	116
12	-135	108

At the bottom of the window, there are several buttons: "Clear", "Options", "Disable", a dropdown menu set to "115200", a checkbox for "Echo On", and an "OK" button.

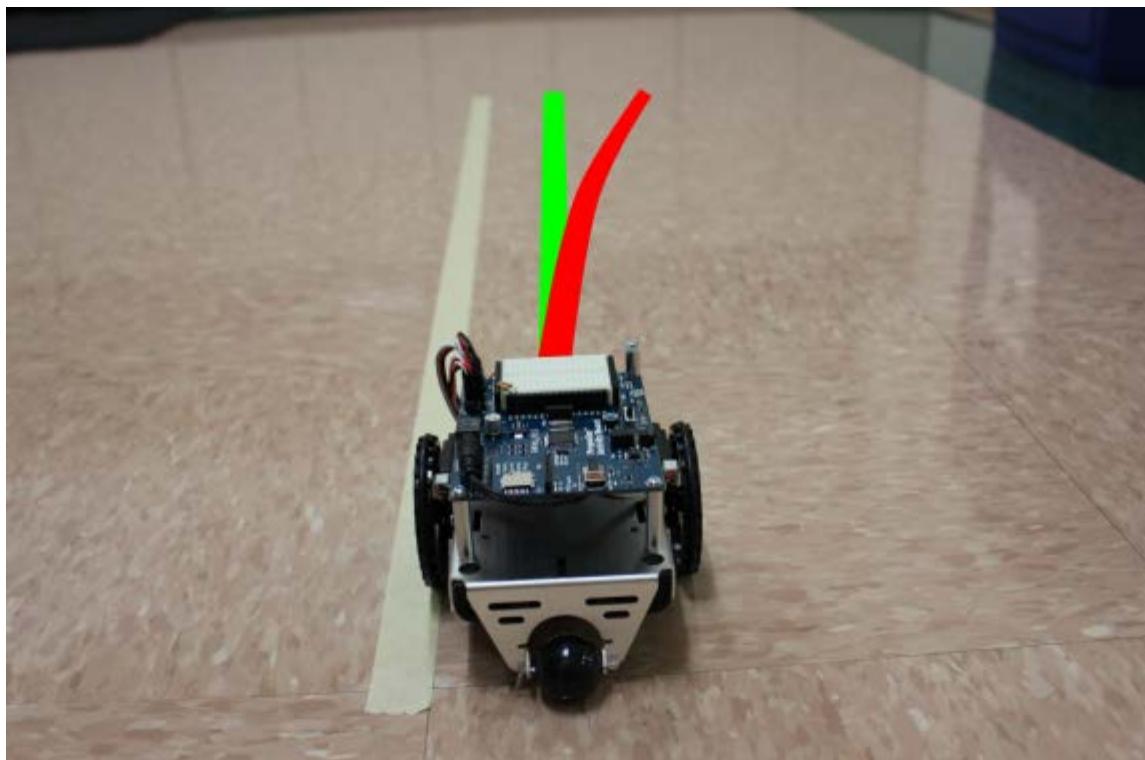
Solution: Check your jumper settings for the servos. They should be set to VIN, not 5V. If they are set to 5V, disconnect all power (barrel jack **AND** USB must be unplugged, power switch set to 0) to the board and move the jumper to VIN as shown on the [Electrical Connections](#) [33] page. Once this is completed, re-run the calibration.

Adjusting the Trim

For some projects, you might want your ActivityBot to go really, *really* straight. Most of the time this is not at all necessary, especially with sensor-based navigation where the robot might change course frequently. So, this activity is completely optional.

It is not uncommon for the ActivityBot to turn slightly even when the encoders go the same distance (red line, below). This can be due to a variety of small mechanical misalignments that add up to a visible curve in travel.

It can be fixed using a technique called *trim*, which will make one wheel turn just a little further than the other to straighten out the robot's trajectory (green line, below). In this optional activity, you will test for straight-ahead travel, and correct with trim settings if needed. You can also test for straight-back travel and use trim settings for going in reverse if you like.

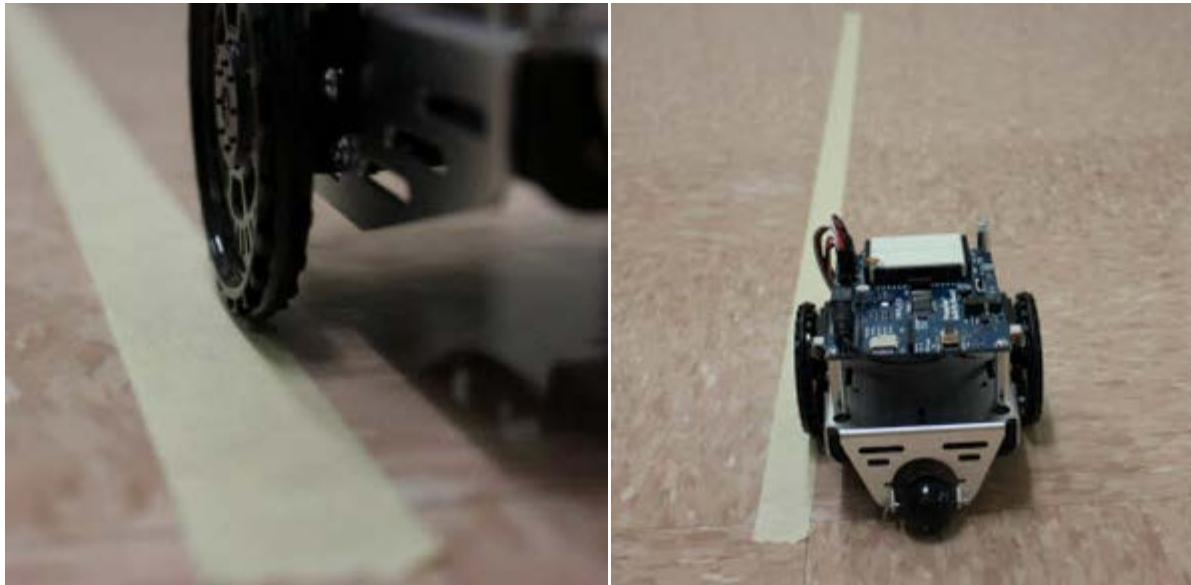


Setup

- Find a hard, smooth floor, preferably with a straight seam (somewhere in the 1 yard to 1 meter

range).

- Lay down about 1 yard or 1 meter of masking tape along the edge of the seam, or use a ruler to get a straight edge if necessary.



Test Code

The test code will make both wheels turn at 32 encoder ticks per second for four seconds. This will make the ActivityBot go forward. If the robot goes straight, then no trim adjustments are needed. If it instead travels in a curve, you can simply set the trim to make one of the wheels turn a little further than the other for any given distance, to correct the path.

- ✓ Click SimpleIDE's Open Project button.
- ✓ Open Test for Trim from ...Documents/SimpleIDE/Learn/Examples/ActivityBot Tutorial.
- ✓ Set the 3-position switch to position-1, then click the Load EEPROM & Run button.
- ✓ After the program is done loading, set the 3-position switch to 0.
- ✓ Take the ActivityBot to the straight line.
- ✓ Set the 3-position switch to position-2.
- ✓ While holding the reset button down, carefully set the robot next to the straight line and **line it up so the robot is parallel to the line.**
- ✓ Release the reset button, and monitor the robot's travel.
- ✓ If the robot goes straight, skip the trim instructions for forward motion. You can still do a backward motion test if you like.
- ✓ If the robot curves more than you want it to, continue with these trim instructions.

Keep in mind that after your trim adjustments are done, you do not need to have `drive_trimset` in your programs. The values will be stored in EEPROM and fetched with the start of each program that uses the abdrive library.

Trim Instructions

To add trim, figure out which wheel needs to go a little further with every turn. For example, if the ActivityBot veers to the right, you can make the right wheel go a little further than the left to straighten it out.

The function call to add trim is:

```
drive_trimSet(int direction, int side, int value)
```

- *int direction* can be `AB_FORWARD` or `AB_BACKWARD`.
- *int side* is the wheel to speed up or slow down: `AB_LEFT` or `AB_RIGHT`.
- *int value* is how often you want to add (or subtract) a tick.

And, `drive_trimSet(0, 0, 0)` clears all trim settings.

If your robot veers to the right when going forward, here is an example that makes the right wheel go an extra encoder tick for every 64:

```
drive_trimSet(AB_FORWARD, AB_RIGHT, 64);
```

If it's curving just slightly to the left, maybe you only need to make the left wheel add a tick for every 200. That would be:

```
drive_trimSet(AB_FORWARD, AB_LEFT, 200);
```

- ✓ Modify the `drive_trimset` function and re-run the Test for Trim.
- ✓ Keep adjusting until you have your robot traveling forward in a line that's straight.

How it Works

Except for `pause`, which is part of simpletools, all the other calls in this program are to functions in the abdrive library. `drive_trimSet(0, 0, 0)` clears all the trim settings so that the distances match. Then, `drive_speed(0, 0)` starts the servos off with zero speed and holds that speed for 0.2 seconds. Next, `drive_speed(32, 32)` sets both wheels to go for encoder 32 ticks (1/2 a turn) per second. Followed by `pause(8000)`, it makes the servos-encoder systems complete 4 wheel turns. Last, `drive_speed(0, 0)`

stops the servos.

```
#include "simpletools.h"
#include "abdrive.h"

int main()
{
    drive_trimSet(0, 0, 0);
    drive_speed(32, 32);
    pause(8000);
    drive_speed(0, 0);
}
```

Did You Know?

Turning Off Trim — The abdrive library has a function named `drive_trim()`, which can be set to 0 (off) or 1 (on). In case you want to suspend trim for a maneuver, you can use `drive_trim(0)` before the maneuver, and `drive_trim(1)` to resume trim after the maneuver.

Try This

For some projects, you might want to set trim for both forward and backward travel. Luckily, the trim function can be called multiple times to handle that situation. After setting your trim for one direction, modify the `drive_trimSet()` call for the other. Trim settings won't necessarily be the same in both directions; test forward and backward motion independently and make adjustments only as needed.

- ✓ Modify your program's first `drive_speed` function call as shown below.

```
int main()
{
    drive_trimSet(0, 0, 0);
    drive_speed(-32, -32);      // <- modify
    pause(8000);
    drive_speed(0, 0);
}
```

- ✓ Set the 3-position switch to position-1, then click the Load EEPROM & Run button.
- ✓ After the program is done loading, set the 3-position switch to 0.
- ✓ Take the ActivityBot to the straight line.
- ✓ Set the 3-position switch to position-2.
- ✓ While holding the reset button down, carefully set the robot next to the straight line again, **this**

time facing backward

- ✓ Release the reset button, and monitor the robot's travel.
- ✓ If the robot curves away from the line, modify the `drive_trimset` parameters to speed up or slow down one wheel.

For example, if your robot veered to the right away from the line, you could make the right wheel go an extra tick per two revolutions (128 ticks) like this:

```
drive_trimSet(AB_BACKWARD, AB_RIGHT, 128);
```

Remember, once you have adjusted trim settings for both forward and backward travel, you do not need to adjust them again. They will be stored in EEPROM until you overwrite them with `drive_trimset(0, 0, 0)`.

[VISIT THE FORUMS](#) ♦

[Terms of Use](#) ♦ [Feedback: learn@parallax.com](#) ♦ Copyright©Parallax Inc. 2013 (unless otherwise noted)

Source URL: <http://learn.parallax.com/activitybot>

Links:

- [1] <http://learn.parallax.com/propeller-activity-board>
- [2] <http://learn.parallax.com/propeller-brains-your-inventions>
- [3] <http://www.parallax.com/propeller/qna/>
- [4] <http://www.parallax.com/downloads/p8x32a-propeller-datasheet>
- [5] <http://learn.parallax.com/activitybot/mechanical-assembly>
- [6] <http://learn.parallax.com/activitybot/electrical-connections>
- [7] <http://www.parallax.com>
- [8] <mailto:sales@parallax.com>
- [9] <http://learn.parallax.com/propeller-c-set-simpleide>
- [10] <http://learn.parallax.com/sites/default/files/content/propeller-c-tutorials/ActivityBot/Software/ActivityBot%202013-10-31.zip>
- [11] <http://learn.parallax.com/reference/mit-license>
- [12] <http://learn.parallax.com/propeller-c-start-simple>
- [13] <http://learn.parallax.com/reference/breadboard-basics>
- [14] <http://learn.parallax.com/propeller-c-simple-circuits/blink-light>
- [15] <http://learn.parallax.com/propeller-c-simple-circuits/piezo-beep>
- [16] <http://learn.parallax.com/activitybot/software-and-programming>
- [17] http://learn.parallax.com/sites/default/files/content/propeller-c-tutorials/ActivityBot/Nav-Basics/ActivityBot_Navigate%282013-11-18%29.zip
- [18] <http://learn.parallax.com/activitybot/troubleshooting>
- [19] <http://learn.parallax.com/activitybot/adjusting-trim>
- [20] http://learn.parallax.com/sites/default/files/content/propeller-c-tutorials/ActivityBot/NavByTouch/ActivityBot_Touch%282013-09-03b%29.zip
- [21] <http://learn.parallax.com/propeller-c-start-simple/counting-loops>
- [22] http://learn.parallax.com/sites/default/files/content/propeller-c-tutorials/ActivityBot/NavbySound/ActivityBot_Ultrasound%282013-09-03b%29.zip
- [23] <http://learn.parallax.com/reference/speed-sound-air-vs-temperature>
- [24] <http://learn.parallax.com/propeller-c-simple-circuits/sense-light>
- [25] <http://learn.parallax.com/activitybot/beeps>
- [26] http://learn.parallax.com/sites/default/files/content/propeller-c-tutorials/ActivityBot/NavByLight/ActivityBot_Light%282016-09-03a%29.zip
- [27] http://learn.parallax.com/sites/default/files/content/propeller-c-tutorials/ActivityBot/NavByIR/ActivityBot_IR%282013-09-13a%29.zip
- [28] <http://learn.parallax.com/propeller-c-simple-devices/sd-card-data>

- [29] <http://learn.parallax.com/sites/default/files/content/propeller-c-tutorials/ActivityBot/SD/SD-CardGames.zip>
- [30] <http://pic.dhe.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Freref%2Fsc41560702.htm>
- [31] mailto:support@parallax.com
- [32] <http://forums.parallax.com/forumdisplay.php/49-Learn>
- [33] <http://learn.parallax.com/activitybot/electrical-connections#AB-Check-Connections>
- [34] <http://learn.parallax.com/activitybot/test-encoder-connections>
- [35] <http://learn.parallax.com/activitybot/calibrate-your-activitybot>