


What's a Multicore Microcontroller?

Student Guide, Part 1

VERSION 1.0

PARALLAX 

Warranty

Parallax warrants its products against defects in materials and workmanship for a period of 90 days from receipt of product. If you discover a defect, Parallax will, at its option, repair or replace the merchandise, or refund the purchase price. Before returning the product to Parallax, call for a Return Merchandise Authorization (RMA) number. Write the RMA number on the outside of the box used to return the merchandise to Parallax. Please enclose the following along with the returned merchandise: your name, telephone number, shipping address, and a description of the problem. Parallax will return your product or its replacement using the same shipping method used to ship the product to Parallax.

14-Day Money Back Guarantee

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a full refund. Parallax will refund the purchase price of the product, excluding shipping/handling costs. This guarantee is void if the product has been altered or damaged. See the Warranty section above for instructions on returning a product to Parallax.

Copyrights and Trademarks

This documentation is Copyright 2015 by Parallax Inc. By downloading or obtaining a printed copy of this documentation or software you agree that it is to be used exclusively with Parallax products. Any other uses are not permitted and may represent a violation of Parallax copyrights, legally punishable according to Federal copyright or intellectual property laws. Any duplication of this documentation for commercial uses is expressly prohibited by Parallax Inc. Duplication for educational use, in whole or in part, is permitted subject to the following conditions: the material is to be used solely in conjunction with Parallax products, and the user may recover from the student only the cost of duplication. Check with Parallax for approval prior to duplicating any of our documentation in part or whole for any other use.

BASIC Stamp, Board of Education, Boe-Bot, Stamps in Class, and SumoBot are registered trademarks of Parallax Inc. HomeWork Board, PING))), Parallax, the Parallax logo, and Spin are trademarks of Parallax Inc. If you decide to use any of these words on your electronic or printed material, you must state that "(trademark) is a (registered) trademark of Parallax Inc." upon the first use of the trademark name. Other brand and product names herein are trademarks or registered trademarks of their respective holders.

Disclaimer of Liability

Parallax Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, or any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax is also not responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your Propeller application, no matter how life-threatening it may be.

Errata

While great effort is made to assure the accuracy of our texts, errors may still exist. Occasionally an errata sheet with a list of known errors and corrections for a given text will be posted on the related product page at www.parallax.com. If you find an error, please send an email to editor@parallax.com.

Table of Contents

Preface	5
Audience.....	5
A Bit About Propeller C and Simple Libraries	5
About the Author.....	7
Contributors	7
Chapter 1 : Getting Started	9
How Many Microcontrollers Did You Use Today?	9
The Propeller Activity Board – Your New Embedded System	9
Amazing Inventions with Microcontrollers.....	10
Activity #1 : What’s a “Multicore” Microcontroller?	12
Activity #2 : Set up Software and Hardware	17
Activity #3 : Learn Just a Little Programming	18
Activity #4 : Variables and Math	23
Activity #5 : When You are Done For Now	26
Summary	26
Chapter 2 : Lights On – Lights Off	30
Indicator Lights	30
Making a Light-Emitting Diode (LED) Emit Light	30
Activity #1 : Building and Testing the LED Circuit.....	31
Activity #2 : On/Off Control with the Microcontroller	40
Activity #3 : Counting and Repeating.....	47
Activity #4 : Building and Testing a Second LED Circuit	51
Activity #5 : Control a Bicolor LED with Current Direction	55
Summary	62
Chapter 3 : Digital Input – Pushbuttons	67
Found on Calculators, Handheld Games, and Appliances	67
Receiving vs. Sending High and Low Signals	67
Activity #1 : Testing a Pushbutton with an LED Circuit.....	67
Activity #2 : Reading a Pushbutton with the Propeller	71
Activity #3 : Pushbutton Control of an LED Circuit	76
Activity #4 : Two Pushbuttons Controlling Two LED Circuits.....	80
Activity #5 : Reaction Timer Test.....	88
Summary	96
Chapter 4 : Control Position and Motion	103
Microcontrolled Motion.....	103
Introducing the Servo.....	103
Activity #1 : Safely Connecting the Servo.....	106

Activity #2 : Test and Adjust Range of Motion	110
Activity #3 : Program to Hold Positions.....	116
Activity #4 : Controlling Position with your Computer	120
Activity #5 : Converting Position to Motion	125
Activity #6 : Pushbutton-Controlled Servo	127
Summary	132
Chapter 5 : Write Multicore Code	136
Introducing the Function	137
Activity #1 : Test the Multi-HelloFunction	138
Activity #2 : Parameters and Return Values.....	142
Activity #3 : Variable Scope.....	146
Activity #4 : Run Functions in Other Processors (cogs)	153
Activity #5 Sharing Global Variables Between Cogs	160
Activity #6 : Self-terminating Cogs.....	164
Activity #7 : Printing and Terminating from a launched Cog.....	168
Summary	173
Chapter 6 : Measure Voltage and Position	177
The Variable Resistor – a Potentiometer.....	177
Activity #1 : Set Voltages with Two Resistors.....	178
Activity #2 : Read the Position with the Propeller	184
Activity #3 : Calibrate D/A Outputs	189
Activity #4 : Potentiometer Controlled LED	197
Activity #5 : Measure Input, Scale Value, Set Output.....	198
Activity #6 : Potentiometer Controlled Servo	202
Activity #7 : Potentiometer Controlling Other Cog	209
Summary	214

Preface

This tutorial answers the question “What’s a multicore microcontroller?” and shows students how to use one to design their own “smart” invention. It features Parallax Inc.’s Propeller microcontroller, which is built into the Propeller Activity Board. The tutorial activities are designed to appeal to a student’s imagination by using motion, light, sound, and tactile feedback to explore new concepts. Along the way, students encounter basic principles in the fields of computer programming, electricity and electronics, mathematics, and physics. Many activities give hands-on experience with design practices and common electronic components used by engineers and technicians in the creation of modern machines and appliances. At the end of this course students will understand the capabilities of microcontrollers, design their own projects, and build them. In short, they will be able to use multicore microcontrollers as another tool to equip their genius.

AUDIENCE

This tutorial is designed to be an entry point to technology literacy, and an easy learning curve for embedded programming and device design. The text is organized so that it can be used by the widest possible variety of students as well as by independent learners. Middle-school students can try the examples in this text in a guided tour fashion by simply following the check-marked instructions with instructor supervision. At the other end of the spectrum, engineering students’ comprehension and problem-solving skills can be tested with the questions, exercises, and projects (with solutions) in each chapter summary. The independent learner can work at his or her own pace, and obtain assistance through the Learn forum cited below.

A BIT ABOUT PROPELLER C AND SIMPLE LIBRARIES

Propeller C is introduced here: <http://learn.parallax.com/propellerc>. This program combines the most popular elements of the Stamps in Class program for the BASIC Stamp with the multicore Propeller microcontroller and the C programming language.

Those of you who got started with Stamps in Class tutorials will probably recognize many of the PBASIC features that made getting started with microcontrollers and electronics so fun with the BASIC Stamp. For example, blinking a light in PBASIC uses commands named **high**, **low** and **pause**. The Propeller C tutorials use a library called **simpletools**, which has equivalent functions named: **high**, **low** and **pause**. A few more

examples from simpletools you might recognize: `pulse_in`, `pulse_out`, `shift_in`, `shift_out`, `i2c_in`, `i2c_out`, `freq_out`, `count`. You can go here for the complete list, and click the simpletools.h link:

<https://prosideworkspace.googlecode.com/hg/Learn/Simple%20Libraries%20Index.html>

The portions of PBASIC that were incorporated into the simpletools library were the P (for Parallax) part of PBASIC, not the BASIC part. In other words, the simpletools library made functions out of the commands that can be applied in any language to simplify the the I/O control and timing commonly used to make microcontrollers interact with the circuits inside products, robots, and inventions. The rest is C language, built by the fully C compliant Propeller GCC compiler. So, instead of putting `high`, `low`, and `pause` commands in a PBASIC `do...while` loop for the BASIC Stamp, to blink that light with a BASIC Stamp, we have `high`, `low`, and `pause` function calls inside a C language `while(1){...}` loop for the Propeller.

Launching into the larger world of C language and multicore microcontroller applications, Propeller C uses both custom and standard libraries. Some custom libraries support popular devices. So instead of needing to write a function to check a PING))) Ultrasonic Distance Sensor and convert the echo time measurement to centimeters, you can just include the `ping.h` header, and call its `ping_cm` function. Some libraries even provide simple function calls that launch and manage multicore processes like wav files, VGA display, and/or controlling many servos with a single core.

Standard libraries are used whenever possible. Trig functions, random numbers, and string comparisons are all examples that can be found in the tutorials. The original intent was to also use the stdio library's `printf` and `scanf` functions. Another original intent that guided the design of the Propeller Activity Board was to have all Propeller C examples executed in CMM (compact memory) mode so that programs could be run by just the Propeller chip on most existing boards without requiring extended memory hardware add-ons.

The first stdio obstacle was encountered when floating point was added to a simple program that wrote to/from SD. Even in CMM mode, the program overflowed the Propeller chip's 32 KB memory. Although a library called libtiny reduced the size of some programs with `printf` and `scanf`, it didn't support floating point or file I/O. In contrast, the simpletext library's `print` and `scan` functions supported a more well-rounded subset of `printf` / `scanf` features, including floating point. In addition, `print`

supports a binary formatter that is not available to `printf`, yet is extremely useful for microcontroller applications and was expected by BASIC Stamp users. The `simpletext` library also has other functions for communicating with peripheral devices, including simple and full duplex serial, and VGA. This, combined with the fact that it supported floating point and allowed judicious use of `fread` and `fwrite` for SD I/O and still left space for application code, made it the best choice at the time.

The `simpletext` library has been a great tool for terminal examples, as an addition to libraries for serial peripheral devices, and for applications with multiple peripherals that handle text input and/or display. Examples on learn.parallax.com include many terminal examples, serial LCD, XBee, RFID, VGA, and more. The list will grow as as more libraries are submitted.

The Propeller C Simple Libraries are part of an open source project, and we highly encourage submissions of new libraries, especially for supporting devices. Simple Libraries posted to obex.parallax.com will be evaluated and considered for inclusion in future revisions. Libraries that are compatible with existing simple libraries and follow the existing Simple Library API pattern and Doxygen comments will be more readily incorporated. For examples, look for similar devices in the Library Index, Learn Tutorials, and in the Simple Libraries folder that SimpleIDE places in `...Documents/SimpleIDE/Learn/`. If another subset version of `stdio` is submitted, it will also be carefully considered. It would need to be able support all the existing tutorials and device drivers with an equivalent level of simplicity to what's already there and either use equal or less code space.

ABOUT THE AUTHOR

Andy Lindsay joined Parallax Inc. in 1999, and has since authored numerous books, articles, product documents, and web tutorials for the company. Andy also travels the nation and abroad teaching Parallax Educator Courses and events, and gathers feedback from educators' observations to improve the material. Andy studied Electrical and Electronic Engineering at California State University, Sacramento. When he's not writing educational material, Andy does product and application engineering for Parallax.

CONTRIBUTORS

This Parallax-authored tutorial includes application engineering, activity design, technical writing, photographs, and C code by Andy Lindsay; technical illustration by Andy

Lindsay and Courtney Jacobs; editing and layout by Courtney Jacobs, and technical nitpicking/general prodding by Stephanie Lindsay.

A very special thank you goes to educator and customer John Kauffman for extensive edits and suggestions made to the *What's a Multicore Microcontroller?* draft, for test-driving the activities in the classroom, and for creating and sharing the Educators Guide and assessment material.

Chapter 1: Getting Started

HOW MANY MICROCONTROLLERS DID YOU USE TODAY?

A microcontroller is a kind of miniature computer brain that you can find in all kinds of devices. Some common, every-day products that have microcontrollers inside are shown in Figure 1-1. If it has buttons and a digital display, chances are it also has a programmable microcontroller brain.



Figure 1-1
Many devices
contain
microcontrollers

Try counting how many devices with microcontrollers you use in a day. If you hit your alarm clock's snooze button a few times in the morning, the first thing you did is interact with a microcontroller. Heating up some food in the microwave oven and making a call on a mobile phone also involve interacting with microcontrollers. Each of those microcontrollers is doing several jobs at once; the radio calculates the time, displays the numbers, tunes to the right station and reacts when you hit the snooze button. All of those devices have microcontrollers inside them that interact with you.

THE PROPELLER ACTIVITY BOARD – YOUR NEW EMBEDDED SYSTEM

Parallax Inc.'s Propeller Activity Board shown in Figure 1-2 has a multicore microcontroller built onto it; it is the largest black chip just above the Propeller Activity Board label. That chip has eight cores inside that perform the actual computing functions. The rest of the parts on the board support the microcontroller by providing power, a USB connection to your computer, extra memory, and sockets for connecting other devices. When all of these parts work together they are called an *embedded computer system*. This name is almost always shortened to just “embedded system.”



Explore the Board. Learn more about each part by reading the Propeller Activity Board's product documentation, a free download from www.parallax.com/product/32910.

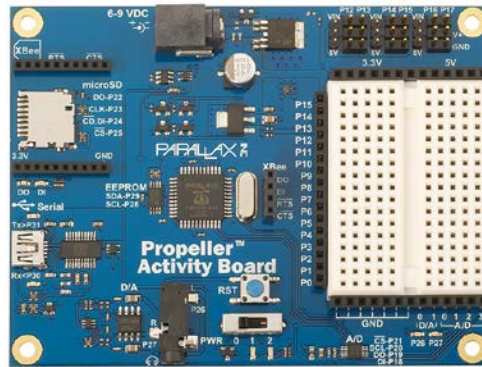


Figure 1-2
Propeller Activity Board with Built-in Propeller Microcontroller

The activities in this tutorial will guide you through building circuits with electronic parts similar to the ones found in consumer appliances and high-tech gadgets. You will also write computer programs that the Propeller chip will run. These programs will make the Propeller Activity Board monitor and control these circuits so that they perform useful functions.

AMAZING INVENTIONS WITH MICROCONTROLLERS

Consumer appliances aren't the only things that contain microcontrollers. Robots, machinery, aerospace designs, and other high-tech devices are also built with microcontrollers. Let's take a look at two in Figure 1-3 shows two robotic examples. On each of these robots, students use the Propeller microcontroller to read sensors, control motors, and communicate with other computers.

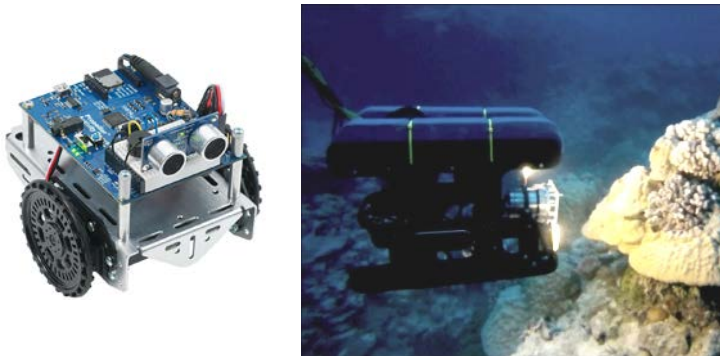


Figure 1-3
Educational Robots

*ActivityBot robot
(left)
CSU Monterey Bay
Ulithi ROV Project
(right)*

The robot on the left is Parallax Inc.'s [ActivityBot](#) robot. It uses the Propeller Activity Board mounted on a small chassis with servo motors, wheels, and sensors to navigate by touch, visible light, infrared light, or ultrasound. The robot on the right is called an underwater ROV (remotely operated vehicle) and it was constructed at California State University of Monterey to study coral reefs in the Ulithi Atoll. Operators see what the ROV sees through a video camera feed, and control the ROV with a combination of hand controls and a laptop. Its Propeller microcontroller monitors sensors on the ROV and reports that information to the operator. At the same time, it also processes signals received from the operator's hand controls and relays them to the ROV's onboard motor controllers.

The flying quadcopter in the left of Figure 1-4 is called the ELEV-8. It was developed by Parallax, and its Propeller microcontroller manages the four motor-driven flight propellers so the aircraft remains stable and responsive to the operator's joystick controls. The millipede-like robot on the right of Figure 1-4 was developed by a professor at Nanyang Technical University, Singapore. It has more than 50 simple microcontrollers on board, and they all communicate with each other in an elaborate network that helps control and orchestrate the motion of each set of legs. In both of these vehicles, microcontrollers solve complex mechanical control problems. Robots not only help us better understand designs in nature, but they are being used to explore remote locations, disaster sites, and even other planets.



Figure 1-4
Research Robots

*Parallax's ELEV-8
quadcopter (left) and
Millipede Project at
Nanyang University
(right)*

Microcontrollers are used in environmental applications, both unique and common. The weather station shown on the left of Figure 1-5 is part of a coral reef decay study. The microcontroller inside it gathers weather data from a variety of sensors and stores it for later retrieval by scientists. Even common traffic lights use sophisticated embedded systems to sense the presence of vehicles, coordinate with other lights to keep traffic moving smoothly, and detect preemption signals sent by emergency vehicle operators.

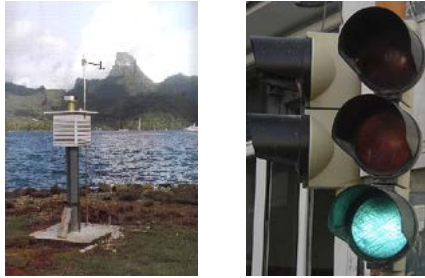


Figure 1-5
Environmental Devices

*Ecological data collection by
EME Systems (left); traffic light
in Greece (right)*

From your first project all the way through scientific applications, the microcontroller basics needed to get started on projects like these are introduced in this book. By working through the activities, you will get to experiment with a variety of building blocks like the ones found in all these inventions. You will build circuits for displays, sensors, and motion controllers. You will learn how to connect these circuits to the Propeller Activity Board's microcontroller, and then write programs that make it collect data from sensors, make decisions, and control lights or motion. Along the way, you will learn many important electronic and computer programming concepts and techniques. By the time you're done, you might find yourself well on the way to inventing a device of your own design.

ACTIVITY #1: WHAT'S A "MULTICORE" MICROCONTROLLER?

Okay, so now that we have seen where microcontrollers are used, what is a *multicore* microcontroller, and why would you want to use one? In each of the examples above, the microcontroller is doing more than one task at the same time. While a few tasks can be juggled at once by a single-core microcontroller, the more processes running at once, the more complicated it gets, especially with time-sensitive tasks like playing WAV files and controlling motors. Computers now come with dual, quad, and even eight cores to handle multiple tasks with speed and precision. Since a microcontroller is like a miniaturized computer that's designed to be the brains of products and inventions, it was inevitable that microcontrollers would also be designed with more cores.

The web article [Propeller Brains for Your Inventions](#) included below, breaks down and illustrates these concepts.

Propeller Brains for Your Inventions

The Propeller microcontroller on the Activity Board can be the brains for your own inventions, such as a robot. It is the brains of the ActivityBot robot, for example.

So, what is a microcontroller? It is an integrated circuit (computer chip) that includes a tiny processor to do the “thinking” and some memory so it can keep track of what it is doing. Microcontrollers also have input/output pins, **I/O pins** for short, which can exchange electrical signals with other devices such as lights, switches, beepers, motors, and sensors.

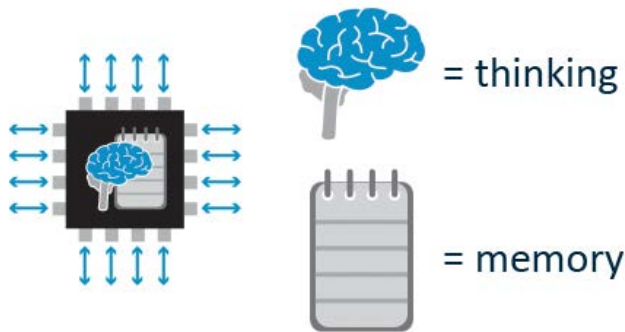


Figure 1-6
So What Is a
Microcontroller?

A **single-core microcontroller** has just one processor inside. A **multicore microcontroller** has two or more processors, also called cores, inside one chip.

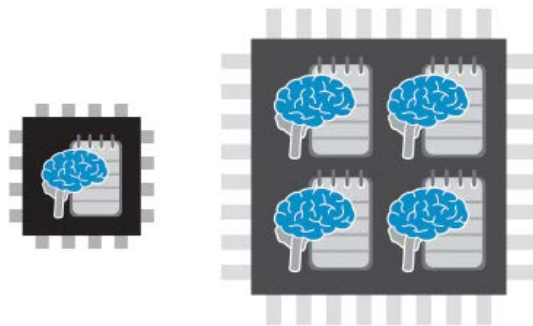


Figure 1-7
Single Core vs.
Multicore

A single-core microcontroller is **multitasking** when it executes several tasks that must share its single processor. The processor must interrupt each task to switch briefly to another, to keep all of the processes going.

Imagine a chef in a kitchen alone, making bread, roast beef, and sauce. The chef must knead the bread dough for 15 minutes, interrupt that task every minute to stir the sauce, and remove the roast from the oven as soon as a thermometer reaches 120 °F. At any moment, the chef (processor) is executing only one task, while keeping all three processes (kneading, stirring, roasting) going at once.

Now imagine being that chef. The more tasks you must do at once, the more difficult it gets to keep track of them all, and keeping the timing right becomes more of a challenge.



Figure 1-8
Multitasking

A multi-core microcontroller is **multiprocessing** when it executes several tasks at once, with each task using its own processor. This is also referred to as **true multitasking**.

Now, imagine a chef in a kitchen with three assistants, making bread, roast beef, and sauce. The chef puts one assistant at the stove to stir the sauce every minute. Another assistant is sent to keep watch on the thermometer, and remove the roast when it reaches 120 °F. Now the chef is free to knead bread dough for 15 minutes. The three cooks (processors) are keeping all three processes (kneading, stirring, roasting) executing at the same time, without any task-switching interruptions, and without missing the moment when the thermometer reaches 120 °F. There is even an extra assistant ready to help if something more is needed.

Having multiple cores makes it easier to do many tasks at once, especially if precise timing is needed.



Figure 1-9
Multiprocessing

The Propeller microcontroller has 8 cores, and can therefore do multiprocessing, also called true multitasking. The cores are all the same. It has 32 I/O pins, which are also all the same. Each core can work with every I/O pin. This means that all of the Propeller cores and I/O pins are equally good at any tasks they must perform. Each core has a bit of its own memory. Each core also takes turn accessing a larger Main Memory, where they can share information.

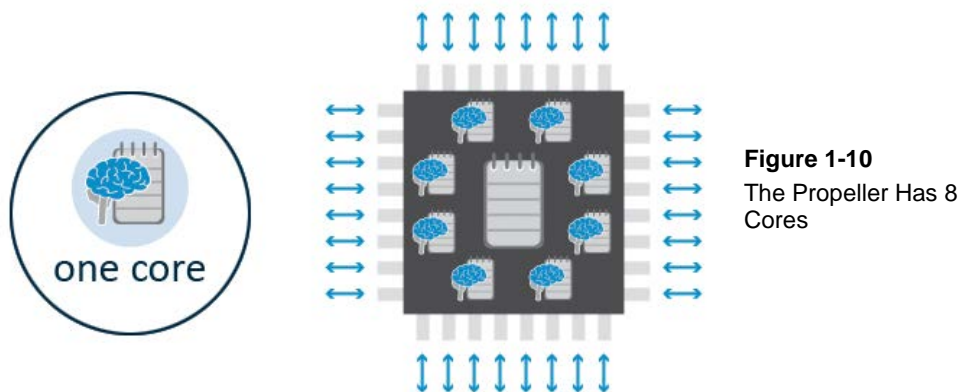


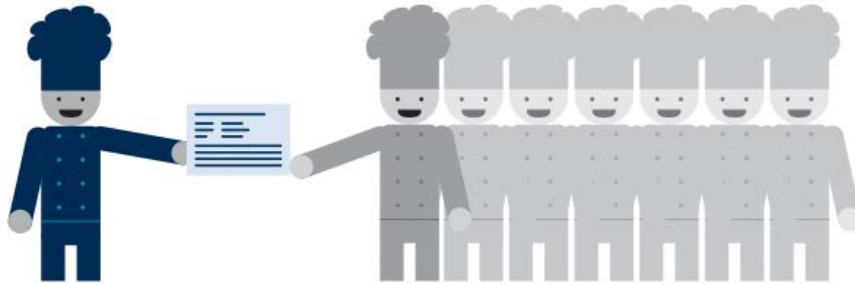
Figure 1-10
The Propeller Has 8
Cores

Eight cores in one microcontroller might sound intimidating. It might seem complicated to have to write programs for all of them. But the Propeller C language has pre-written code tasks, called **functions**, which make it easy.

Just think of functions as recipes the head chef can hand over to assistants, instead of having to explain to each one how to cook. One assistant can even ask another assistant for help, without bothering the head chef. Just as a team of 8 chefs can efficiently

manage great meals, the Propeller with its eight cores can efficiently manage great inventions. Now that's teamwork!

Figure 1-11 Multicore Is Easy with C Functions



A **C library** is a collection of functions, sort of the way a cookbook is a collection of recipes.

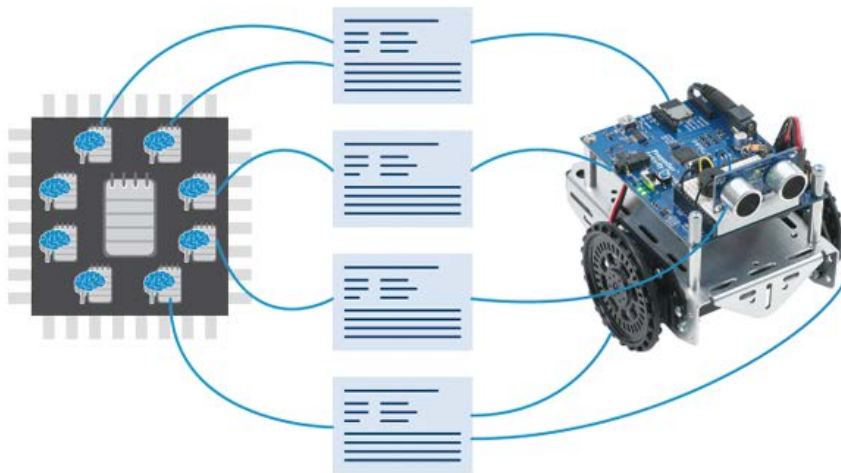
Figure 1-12 A C Library is a Collection of Functions



So what does multicore processing look like for an invention like the ActivityBot?

Your C program might start a motor function, which makes another core manage the motors to make the robot move. Then, it might call a sensor function so the robot can “see” if there is an obstacle in its path. If an object is detected, your program then might call a music function, which will task other cores with the jobs of fetching songs from an SD card and playing music on an audio jack.

The ActivityBot is just one example of a microcontroller invention. You can use the Activity Board to build other projects or create your own inventions.

Figure 1-13 From Cooking to Robotics

In this book, you will learn to use the Propeller microcontroller with electronic components you can think of as ingredients for your own inventions. So let's get started!

ACTIVITY #2: SET UP SOFTWARE AND HARDWARE

Getting started with the Propeller Activity Board is similar to getting started with a brand-new PC or laptop: take it out of the box, power it up, download and test some software. If this is your first time using the Propeller Activity Board, you will be doing all these same activities plus (most importantly) learning to write software of your own in a programming language for the Propeller called “C”.

If you are in a class, your hardware may already be all set up for you and your teacher may have other instructions. If not, it is time to go to online resources for downloading and installing the software, connecting the hardware to your computer, and testing to make sure your computer can load programs into your board.

- ❑ Using a web browser, go to the web tutorial [Propeller C – Set Up SimpleIDE](http://learn.parallax.com/propeller-c-set-simpleide). (<http://learn.parallax.com/propeller-c-set-simpleide>)
- ❑ Click the link for your operating system (Windows, Mac, or Linux).

- Follow the instructions to:
 - Download and install the USB driver.
 - Download and install the SimpleIDE software.
 - Connect your Activity Board to the computer.
 - Run a test program that displays a “Hello” message.
- Once you are sure your board and software are working, [update your learn folder](#) to make sure you have the most current example programs and libraries.
(<http://learn.parallax.com/propeller-c-set-simpleide/update-your-learn-folder>)



What do I do if I get stuck? If you run into problems, you have many options to obtain free Technical Support:

- **Forums:** sign up and post a message in our free, moderated Learn forum at <http://forums.parallax.com>.
- **Email:** send an email to support@parallax.com.
- **Telephone:** In the Continental United States, call toll-free 888-997-8267. All others call (916) 624-8333.

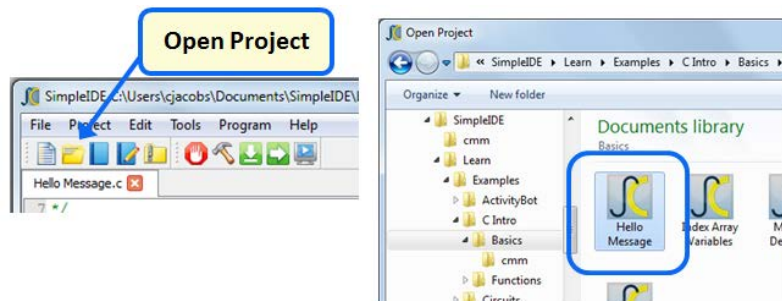
ACTIVITY #3: LEARN JUST A LITTLE PROGRAMMING

In this book, you will build lots of useful circuits and write programs to monitor and control them. Most of the programming and circuit-building will be learn-as-you-go, and just a little at a time. But before moving on to that, let's try two of the tutorials from the [Propeller C – Start Simple web tutorial series](#) that have been included below; one in this activity and one in the next. They will help you to you get familiar with the SimpleIDE programming software.

Simple Hello Message Tutorial

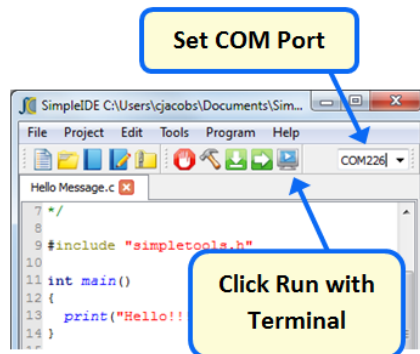
This C program will make the Propeller microcontroller send a "Hello!!!" message to the SimpleIDE Terminal on your computer.

- Click the Open Project button.
- Navigate to My DocumentsSimpleIDE\Learn\Examples\C Intro\Basics.
- Select Hello Message.side, and click Open.

Figure 1-14 Opening a Project in SimpleIDE

When SimpleIDE opens the project, it will open Hello Message.c into its text editor pane.

- ❑ Click the COM Port dropdown on the right and select the com port your board is connected to. If in doubt, disconnect/reconnect the board and click it again to see which one disappeared/reappeared.
- ❑ Click the Run with Terminal button.

**Figure 1-15**
Setting the COM Port and Running the Terminal

A single "Hello!!!" message should appear in the Simple IDE Terminal.

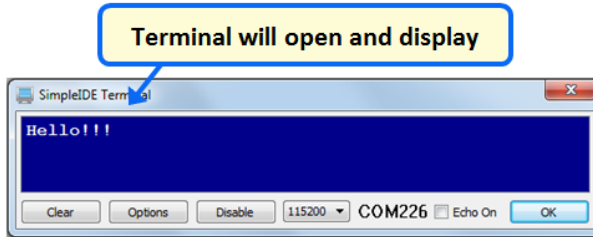


Figure 1-16
SimpleIDE Terminal

How Hello Message.c Works

The `print("Hello!!!")` makes the Propeller chip send its message to your computer through its programming port. The SimpleIDE terminal displays the message on your computer screen.

The `print("Hello!!!")` is followed by a semicolon (`;`). The semicolon is what tells the PropGCC compiler that it has reached the end of an instruction statement.

The `print` statement is inside curly braces `{ }` below `main()`, and so we call it part of the `main` function's *code block*. **A C program always starts with the first statement in the main function.**

The `print` command is also a function, but it is stored in other files called *library files*. Later on, you'll get to search for libraries that contain useful functions to add to your own projects. For now, just keep in mind that your program needs `#include "simpletools.h"` because it has information about `print`, and many other functions.

Try This – Print Another Message

The program has one statement: `print("Hello!!!");`. Let's save this project under a new name, and add a second `print` statement. We can then say that the program is *calling* the `print` function twice.

- Click the Save Project As button.

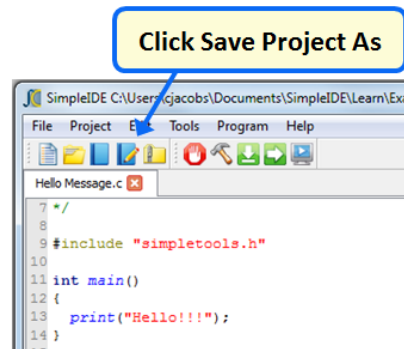


Figure 1-17
Save Project As

- Browse to My Documents\SimpleIDE\My Projects.
- Type Hello Again into the File name field.
- Click the Save button.

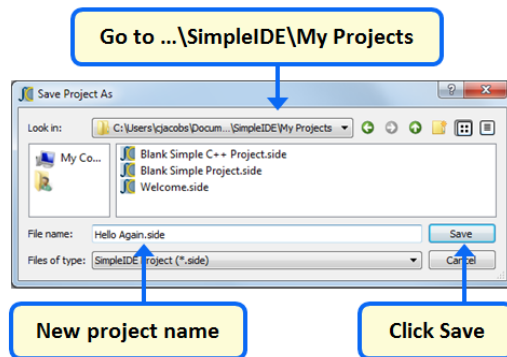


Figure 1-18
Saving a Copy to the
My Projects Folder

- Modify the `main` function to add a second `print` function call, like this:

Figure 1-19 Modified Main Function

```

int main()
{
    print("Hello!!!\n");
    print("Hello again!!!");
}
// <- modify this line
// <- add this line

```

- Click the Run with Terminal button, and observe the output.
- What effect does the `\n` have? Delete `\n`, then run the program a third time.



Saving Programs

SimpleIDE saves your program each time you run or compile it. As you progress through these tutorials you will notice that we ask you to save a new copy of any program you'll be modifying to prevent you from overwriting the original project with one you have changed.

Did You Know?

C is case-sensitive. You have to use the correct capitalization when programming in C. If you make an error, such as typing `Print`, for example, SimpleIDE will let you know:

Figure 1-20 Build Failure Message

Done. Build Failed!

Check source for bad function call or global variable name `'_Print'`

- **newline** — `\n` is called the newline character, and it is an example of a control character used for positioning a cursor in a serial terminal.
- **int (main)** — the `int` in `int main()` is part of the C compiler's programming convention. It is used no matter what you include inside the `main` function's code block. You will learn more about how `int` is used in other ways as you go through the tutorials.

Your Turn – Using Comments

Comments are notes about your code that help explain it to other people that have to work with it. Also, it is good to leave comments as notes to yourself about what you were doing in case you need a reminder days (or months or years) later.

If you want to comment all or part of a single line, use two forward slashes `//`. Everything to the right of `//` will be ignored by the C compiler. Block comments can span multiple lines. They start with `/*` and end with `*/`, and everything in between will be ignored by the C compiler.

- Click the Save As Project button again and save the project as Hello Again Commented.
- Add the comments shown below.
- Run it again to verify that the comments do not have any actual effect on the way your program runs. (If your comment prevents the program from running, you may have a typing error!)

Figure 1-21 Hello Again Commented.c in SimpleIDE

```

/*
  Hello Again Commented.c

  Display a hello message in the serial terminal.
*/

#include "simpletools.h"      // Include simpletools header

int main()                  // main function
{
  print("Hello!!!\n");      // Display "Hello" Message
  print("Hello again!!!");  // 2nd "Hello" message on new line
}

```

ACTIVITY #4: VARIABLES AND MATH

A *variable* is a name you give to a section of microcontroller memory so your program “remembers” values and works with them. In this activity, the Propeller microcontroller will do some simple math problems, using variables to store the values and the answers.

- Click SimpleIDE’s Open Project button.
- If you’re not already there, navigate to ...\\SimpleIDE\\Learn\\Examples\\C Intro\\Basics.
- Open Variables and Calculations.side.
- Examine Variables and Calculations.c, and try to predict what SimpleIDE Terminal will display.
- Click the Run with Terminal button to run the program, and compare the actual output to your predicted output.

```

/* Variables and Calculations.c */

#include "simpletools.h"           // Include simpletools

int main()                       // main function
{
    int a = 25;                  // Initialize a variable to 25
    int b = 17;                  // Initialize b variable to 17
    int c = a + b;               // Initialize c variable to a + b
    print("c = %d ", c);        // Display decimal value of c
}

```

How Variables and Calculations.c Works

Variables and Calculations.c declares an integer variable named **a** and assigns it the value 25 with `int a = 25`. Then, it declares a second variable named **b** and initializes it to 17 with `int b = 17`. The last integer variable it declares is named **c**, and stores the result of $a + b$ in it.

Finally, it displays the value of **c** with `print("c = %d", c)`. This variation on `print` displays a sequence of characters called a *string*, followed by a variable. The `%d` is called a *format placeholder*, and it tells `print` how to display the value stored in that variable as a decimal number, 42 in this case.



The add operator (+) is a binary operator, meaning that it needs two inputs to perform an operation. Here are some common binary operators:

+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus (remainder of a division calculation)

Try This – Test Binary Operators

Here is a modified version of the `main` routine that displays "a = , b = " with their values, and then "a + b = " and the value of **c** on a new line. Then, it repeats for $a - b$.

Notice that the second time it calculates the value of **c**, we don't need to declare it with `int`. It's just `c = a - b`. Notice also that `print` allows you to display more than one

numeric value within your string. All it takes is two format placeholders in the string and two values, separated by commas, after the string.

- Click Save As Project button, and name it Test Binary Operators.
- Modify the `main` function as shown below.
- Run the program and verify the output.

Figure 1-22 Modified Main Function

```
int main()
{
    int a = 25;
    int b = 17;
    int c = a + b;
    print("a = %d, b = %d\n", a, b);           // <- modify
    print("a + b = %d\n", c);                 // <- add
    c = a - b;                                // <- add
    print("a = %d, b = %d\n", a, b);         // <- add
    print("a - b = %d\n", c);                 // <- add
}
```

Your Turn – More Binary Operators

- Expand Test Binary Operators.c so that it goes through testing all five binary operators in the information box, above.
- Try changing `a` to 17 and `b` to 25, then re-run.
- Try declaring `int` variables of `y`, `m`, and `b`, and then use them to calculate and display `y = m * x + b`.



PRO TIP: Displaying % with print

To display the output of the Modulus operator, use ("`a mod b = ...`") or ("`a %% b = ...`") in the `print` function. Since `%` has another purpose in `print` strings, just saying ("`a % b = ...`") will give unexpected results.

There are many additional pages in the Propeller C – Start Simple tutorial series that introduce more C programming. Those topics will be introduced as you go through the upcoming activities as needed for a certain circuit or project. But if you want to go

online to try more programming language web tutorials now, have fun. Just make sure to pick back up here when you're done. (<http://learn.parallax.com/propeller-c-start-simple>)

ACTIVITY #5: WHEN YOU ARE DONE FOR NOW

Ready to take a break? Whenever you leave your Activity Board unattended, it's best to set its PWR switch to 0 (off) and disconnect its USB cable (and/or batteries). Also, it is wise to always wash your hands after working with electronics.

- Set the Activity Board's PWR switch to 0.
- Disconnect the USB cable.
- If you happen to have batteries connected to the Activity Board's 6-9 VDC jack, unplug them now and store them where they cannot touch other components.

SUMMARY

This chapter guided you through the following:

- An introduction to some devices that contain microcontrollers.
- An introduction to the Activity Board and its Propeller multicore microcontroller.
- A tour of some interesting inventions made with the Propeller microcontroller and other embedded systems.
- How to install the USB drivers for loading programs into the Propeller microcontroller.
- How to download and install the SimpleIDE for writing programs for the Propeller microcontroller.
- How to make the Propeller send messages to your computer, and how to make your computer display them in the SimpleIDE Terminal.
- Using the `print` function in a program to make the Propeller send messages for your computer to display.
- Using the `\n` (newline) character to move the SimpleIDE Terminal's cursor to the next line.
- How to use variables to store values.
- How to use operators to perform simple math operations.
- What to do when you are finished working with your Propeller Activity Board.

Questions

1. What is a microcontroller?
2. Is the Propeller Activity Board a microcontroller, or does it contain one?
3. What clues would you look for to figure out whether or not an appliance like a clock radio or a cell phone contains a microcontroller?
4. What effect does the double-slash `//` have on code to the right of it?
5. What character tells the C compiler it has reached the end of a statement?
6. Let's say you want to take a break from your Propeller microcontroller project to go get a snack, or maybe you want to take a longer break and return to the project in a few days. What should you always do before you take your break?

Exercises

1. Explain what the `\n` does in this function call:

```
print("Hi \n there!");
```

2. What would the SimpleIDE Terminal display in response to this statement:

```
print("Line1\nLine2\nLine3");
```

3. This statement was written so that it would display `a = 1, b = 2`, but it instead displayed `a = 1, b = 1960`. (The value of `b` may be different for you, but it is not guaranteed to be 2.) What's the problem, and how would you correct it?

```
int a = 1, b = 2;
print("a = %d, b = %d\n", a);
```

Projects

1. Use `print` to display the solution to the math problem: $1 + 2 + 3 + 4$.
2. Make a program that computes $z = c \times (a + b)$. Test with `a = 1, b = 2, c = 3`. The result should be 9.

Solutions

- Q1. A microcontroller is a kind of miniature computer found in electronic products.
- Q2. The Propeller Activity Board contains a Propeller microcontroller chip. The rest of the board provides support for the Propeller and sockets for connecting other devices.

- Q3. If the appliance has buttons and a digital display, these are good clues that it has a microcontroller inside.
- Q4. Whatever notes you put to the right of `//` will be ignored by the C compiler. It's great for inserting notes about the code into the program.
- Q5. The semicolon (`;`)
- Q6. Set the PWR switch to 0. Disconnect the USB cable. Disconnect power cable if connected. Wash your hands!

- E1. It causes “ there!” to appear on the line below “Hi” in the SimpleIDE Terminal.
- E2. The SimpleIDE Terminal would display each item on its own line, like this:
Line1
Line2
Line3
- E3. The `print` statement's text had two `%d` formatters, but it was missing `b` in the variable list following the text. Here is the code with the corrected `print` statement.

```
int a = 1, b = 2;  
print("a = %d, b = %d\n", a, b);
```

- P1. Here are two examples of programs that display a solution to the math problem: $1+2+3+4$. There will be lots of possible solutions, so if it displayed the correct answer in the SimpleIDE terminal, you got it right.

```
/* Intro-P1-Solution1.c */  
  
#include "simpletools.h"  
  
int main()  
{  
    int sum = 1 + 2 + 3 + 4;  
    print("1 + 2 + 3 + 4 = %d\n", sum);  
}
```

```
/* Intro-P1-Solution2.c */  
  
#include "simpletools.h"  
  
int main()  
{  
    print("1 + 2 + 3 + 4 = %d\n", 1 + 2 + 3 + 4);  
}
```

P2. Here is an example of a program that makes the calculation correctly. Again, keep in mind that this is just one of many possible correct solutions.

```
/* Intro-P2-Solution.c */  
  
#include "simpletools.h"  
  
int main()  
{  
    int a = 1, b = 2, c = 3;  
    int z = c * (a + b); // add b and c together before multiplying by c  
  
    print("z = %d \n", z);  
}
```

Chapter 2: Lights On – Lights Off

INDICATOR LIGHTS

Indicator lights are so common that most people tend not to give them much thought. Figure 2-1 shows three indicator lights on a laser printer. Depending on which light is on, the person using the printer knows if it is running properly or needs attention. Car stereos, televisions, DVD players, disk drives, printers, and alarm system control panels all use indicator lights. Look around — can you see any from where you are sitting?



Figure 2-1
Indicator Lights on a Printer

Indicator lights are common on many everyday devices.

Turning an indicator light on and off is a simple matter of connecting and disconnecting it from a power source. In some cases, the indicator light is connected directly to the battery or power supply, like the power indicator lights on the Propeller Activity Board. Other indicator lights are switched on and off by a microcontroller inside the device. These are usually status indicator lights that tell you what the device is up to.

MAKING A LIGHT-EMITTING DIODE (LED) EMIT LIGHT

Most of the indicator lights you see on devices are called *light-emitting diodes*. It is abbreviated LED, and pronounced as three letters: “L-E-D.” If you build an LED circuit and connect power to it, the LED emits light. If you disconnect the power from an LED circuit, the LED stops emitting light.

When an LED circuit is connected to the Activity Board, its Propeller microcontroller can be programmed to connect and disconnect the LED circuit's power. This is much easier than manually changing the circuit's wiring or connecting and disconnecting the battery, but we will try both ways. Here are a few more things we will do in this chapter:

- Turn an LED circuit on and off at different rates
- Turn an LED circuit on and off a certain number of times
- Control more than one LED circuit
- Control the color of a bicolor (two-color) LED circuit

ACTIVITY #1: BUILDING AND TESTING THE LED CIRCUIT

It's important to test components individually before building them into a larger system. This activity focuses on building and testing two different LED circuits. The first circuit makes the LED emit light. The second circuit makes it *not* emit light. You will be connecting the LED to battery power, but not to the microcontroller. (In the activity following this one, you connect the LED the Propeller microcontroller and write programs to turn it on and off.)

Introducing the Resistor

A *resistor* is a component that “resists” the flow of electricity. This flow of electricity is called *current*. Each resistor has a value that tells how strongly it resists current flow. This resistance value is called the *ohm*, and the sign for the ohm is the Greek letter omega: Ω . Later in this tutorial you will see the symbol $k\Omega$, meaning kilo-ohm, or one thousand ohms.

Let's look at an example: the $470\ \Omega$ resistor shown in Figure 2-2. This resistor has two wires — called *leads* and pronounced “leeds” — one coming out of each end. There is a ceramic case between the two leads, and it contains the part that resists current flow. Many circuit diagrams use the jagged-line symbol shown on the left. This tells the circuit-builder to use a resistor, and the number indicates the required resistance value in ohms. This is numbered jagged line is an example of a *schematic symbol*. The drawing on the right is a part drawing used in this entry-level tutorial to help you identify the resistor needed, and where to place it when you build a circuit.

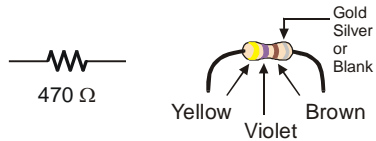


Figure 2-2
470 Ω Resistor Schematic Symbol (left) and Part Drawing (right)

Resistors like the ones in this activity have color-coded stripes to indicate their resistance values. There is a different color combination for each resistance value. For example, the color code for the 470 Ω resistor is yellow-violet-brown.

There may be a fourth stripe that indicates the resistor's *tolerance*. Tolerance is measured in percent, and it tells how far off the part's true resistance might be from the labeled resistance. The fourth stripe could be gold (5%), silver (10%) or no stripe (20%). For the activities in this book, a resistor's tolerance does not matter, but its value does.

Each color bar corresponds to a digit, and these colors/digits are listed in Table 2-1. Figure 2-3 shows how to use each color bar with the table to determine the value of a resistor. Always hold a resistor with the silver or gold band on the right side when reading its value.

Table 2-1 Resistor Color Code Values	
Digit	Color
0	Black
1	Brown
2	Red
3	Orange
4	Yellow
5	Green
6	Blue
7	Violet
8	Gray
9	White

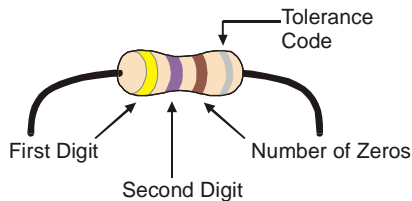


Figure 2-3
Resistor Color Codes

Here is an example that shows how Table 2-1 and Figure 2-3 can be used to figure out a resistor value by proving that yellow-violet-brown is really 470 Ω :

- The first stripe is yellow, which means the leftmost digit is a 4.
- The second stripe is violet, which means the next digit is a 7.
- The third stripe is brown. Since brown is 1, it means add one zero to the right of the first two digits.

$$\text{Yellow-Violet-Brown} = 4\text{-}7\text{-}0 = 470 \Omega.$$

You will be using a different resistor in this activity, with a value of 220 Ω .

- Use the table to figure out the color code for a 220 Ω resistor.

What did you come up with? The answer is red-red-brown. If you came up with a different answer, try again to make sure you've got the steps right.

Introducing the LED

A *diode* is a one-way current valve, and a light-emitting diode (LED) emits light when current passes through it. Unlike the color codes on a resistor, the color of the LED usually just tells you what color it will glow when current passes through it. The important markings on an LED are contained in its shape. Since it is a one-way current valve it is important to connect it the right way in your circuit or it won't work as intended.

Figure 2-4 shows an LED's schematic symbol and part drawing. An LED has two leads. One connects to the LED's *anode*, and the other connects to its *cathode*. In this activity, you will build the LED into a circuit, paying attention to make sure the anode and cathode leads are connected to the circuit properly: anode to power, cathode to ground.

On the part drawing, the anode lead is longer and is labeled with the plus-sign (+). On the schematic symbol, the anode is the wide part of the triangle.

In the part drawing, the cathode lead is the shorter, unlabeled pin, and on the schematic symbol, the cathode is the line across the point of the triangle.

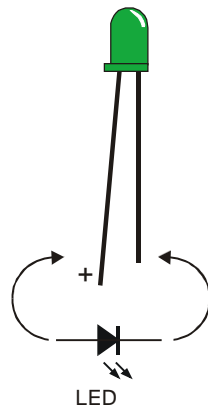


Figure 2-4
LED Part Drawing and Schematic
Symbol

*Part Drawing (above) and schematic
symbol (below).*

*The LED's part drawings in later
pictures will have a + next to the
anode leg.*

When building your circuit, check it against the schematic symbol and part drawing. Note that the LED's leads are different lengths. The longer lead is connected to the LED's anode; connect this lead to power. The shorter lead is connected to its cathode; connect this lead to ground.

Also, if you look closely at the LED's plastic case, it's mostly round, but there is a small flat spot right near the shorter lead that tells you it's the cathode. This is useful if an LED's leads have been cut to equal lengths.

LED Test Circuit Parts

- (1) LED – Green
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Jumper Wire (black)



Identifying the parts: In addition to the part drawings in Figure 2-2 and Figure 2-4, you can use the photo on the last page of the book to help identify the parts in the kit needed for this and all other activities.

Building the LED Test Circuit

You will build a circuit by plugging the LED and resistor leads into small holes called *sockets* on the prototyping area, shown in Figure 2-5. This prototyping area has black sockets along the top, left and bottom. The black sockets along the top have labels above them: 3.3 V and 5 V, and there are some sockets along the bottom labeled GND. These

are called the *power terminals*, and they will be used to supply your circuits with electricity.

The black sockets on the left have labels P0, P1, up through P15. Use these sockets to connect your circuit to the Propeller microcontroller’s input/output pins (called I/O pins).

The sockets labeled D/A and A/D are the *analog terminals*, which we will use later in the tutorial.

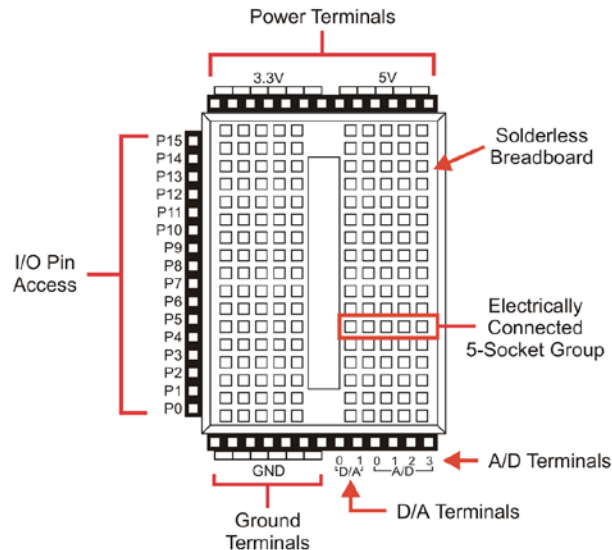


Figure 2-5
Prototyping
Area



Input/output pins are usually called I/O pins. After connecting your circuit to one or more of these I/O pins, you can program your Propeller microcontroller to monitor the circuit (input) or send “on” or “off” signals to the circuit (output). You will try this in the next activity.

The white board with lots of holes in it is called a *solderless breadboard*. You will use this breadboard to connect components to each other and build circuits. This breadboard has 17 rows of sockets. In each row, there are two five-socket groups separated by a trench in the middle. All the sockets in a 5-socket group are electrically connected together by a metal clip under the breadboard. So, if you plug two wires into the same 5-socket group, they will make electrical contact.

Two wires in the same row but on *opposite* sides of the center trench will *not* be connected. Many devices are designed to be plugged in *over* this trench, such as the pushbutton we will use in Chapter 3. There is no connection between the black sockets and the white breadboard. You will make connections to the I/O pins with short wires or component leads when you build circuits on the breadboard.



More about breadboards and connecting circuits: To learn about the history of breadboards, how modern breadboards are constructed, and how to use them, watch the video on our [Breadboard Basics](#) page.

(<http://learn.parallax.com/reference/breadboard-basics>)

The left side of Figure 2-6 shows a *circuit schematic*, a drawing that uses symbols and lines to show how electrical components need to be connected together. On the right is a *wiring diagram*, which is a drawing of how that circuit might look when it is built on the prototyping area.

For this circuit, the resistor and the LED's anode are connected because each one has a lead plugged into the same 5-socket group. The resistor's other lead is plugged into 3.3V so the circuit can draw power. The LED's cathode lead is plugged into a different 5-socket row, along with a wire whose other end is connected to GND (0 V, ground) completing the circuit. (Note that in this case, the circuit is not connected to an I/O pin. We will get to that in the next activity, we promise!)

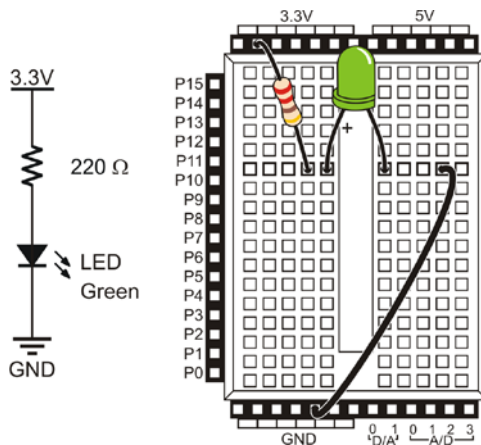


Figure 2-6
LED On, Wired Directly to Power
Schematic (left) and Wiring Diagram (right).

Follow the checklist below to build the circuit shown in Figure 2-6:

- Set the Activity Board's PWR switch to 0.
- Plug one end of the 220 Ω resistor into one of the sockets labeled 3.3 V. It doesn't matter which end.
- Plug the resistor's other end into the white breadboard.
- Use Figure 2-4 to decide which LED lead is the anode and which is the cathode.
- Plug the LED's longer anode lead into the same 5-socket row as the resistor. This connects those two leads together.
- Plug the LED's shorter cathode lead into a different 5-socket row. (Remember, 5-socket rows on opposite sides of the trench are not connected to each other.)
- Plug one end of a wire into the same 5-socket row with the LED's cathode.
- Plug the other end of the wire into one of the sockets labeled GND.



Direction does matter for the LED, but not for the resistor or the wire. If you plug the LED in backward, the LED will not emit light when you connect power. The resistor just resists the flow of current. There is no backwards or forwards for a resistor. Likewise, a wire conducts current either way, so it doesn't have a backwards or forwards either.

- Power your Activity Board by plugging it into your computer's USB port.
- Double-check your circuit connections, and then set the PWR switch to 1.
- Is your LED emitting light? It should glow green.

If the green LED does not emit light when you connect power to the board:

- Try looking straight down onto the dome part of the LED's plastic case from above. Some LEDs are brightest when viewed from above.
- If the room is bright, try turning off some of the lights, or use your hands to cast a shadow on the LED.

If you still do not see any glow, try these steps:

- Double-check that the LED's cathode and anode are connected properly. If not, turn off power, then simply remove the LED, give it a half-turn, and plug it back in. Then turn the power back on. (It will not hurt the LED if you plugged it in backwards, it just doesn't emit light.)
- Check that you are using the correct resistor, marked red-red-brown, with the gold band on the right. A high-value resistor will make the light dimmer.

- ❑ Double-check that two leads that need to be connected together are actually in the same 5-socket row, as shown in Figure 2-6.
- ❑ If you are using a component kit that somebody used before you, the LED may be damaged, so try a different one.
- ❑ If you are in a lab class, check with your instructor.

How the LED Test Circuit Works

The 3.3V socket is like a battery's positive terminal. The GND socket is like a battery's negative terminal. Figure 2-7 shows how connecting a to a battery's terminals causes electrons to flow. This flow of electrons is called *electric current*, which is what causes the diode to emit light. The current is limited by the resistor, so it does not apply more “pressure” than the LED can tolerate.

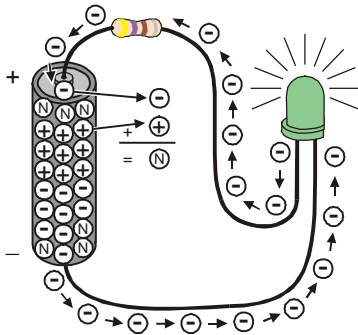


Figure 2-7
LED On, Circuit Electron Flow

The minus signs with the circles around them show electrons flowing from the battery's negative terminal to its positive terminal.



Chemical reactions inside the battery supply the circuit with current. The battery's negative terminal contains a compound that has molecules with extra electrons (shown in Figure 2-7 by minus-signs). The battery's positive terminal has a chemical compound with molecules that are missing electrons (shown by plus-signs). When an electron leaves a molecule in the negative terminal and travels through the wire, it is called a *free electron* (also shown by minus-signs). The extra electrons at the negative end of battery create a force, or electrical pressure, to go through your circuit and get to the molecules that need electrons at the positive end of the battery.

Figure 2-8 shows how the flow of electricity through the LED circuit is described using *schematic notation*. The electrical pressure across the circuit is called *voltage*. The + and – signs show the voltage applied to a circuit. The arrow shows the current flowing through the circuit.

This arrow is almost always shown pointing the opposite direction of the actual flow of electrons. Benjamin Franklin is credited with not having been aware of electrons when he decided to represent current flow as charge passing from the positive to negative terminal of a circuit. By the time physicists discovered the true direction of electric current, the convention was already well established.

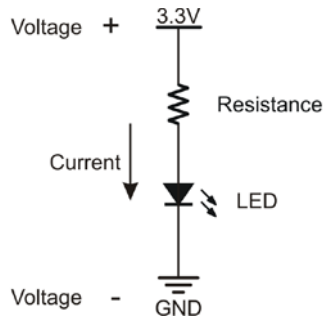


Figure 2-8

LED On, Circuit Schematic Showing Conventional Voltage and Current Flow

The + and – signs show voltage applied to the circuit, and the arrow shows current flow through the circuit.

Your Turn – Modifying the LED Test Circuit

Now you will modify the circuit by connecting the resistor to GND instead of 3.3V, and verify that the LED will then *not* emit light.

- Set your board's PWR switch to 0.
- Unplug the resistor lead from the 3.3V socket, and plug it into a socket labeled GND as shown in Figure 2-9.
- Set the PWR switch back to 1.
- Check to make sure your LED is not emitting light. It should not glow anymore.

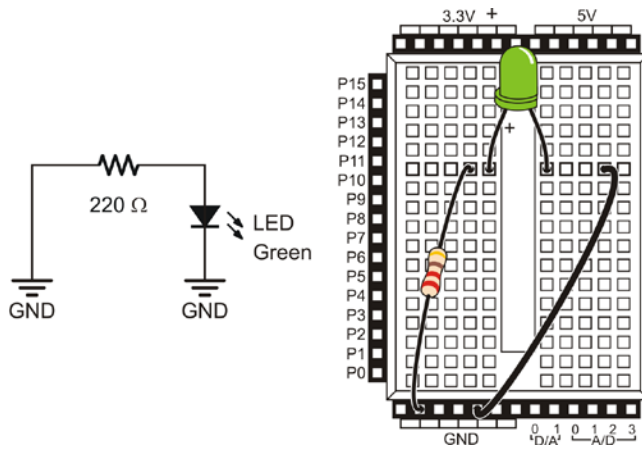


Figure 2-9

LED Off Circuit

Schematic (left) and wiring diagram (right).

Why does the LED not glow? Since both ends of the circuit are connected to the same voltage (GND), there isn't any electrical pressure across the circuit. So, no current flows through the circuit, and the LED stays off.

Now you have experienced turning the LED on and off by moving the resistor lead from 3.3V to GND by hand. It is effective, but not at all convenient. Imagine if you needed the LED to blink very quickly, over and over again! This is a perfect job for a microcontroller. In the next activity you will connect the resistor's lead to a Propeller I/O pin. Then, you will write a program that tells the Propeller to internally connect that resistor to 3.3V or GND to turn the LED on or off.

ACTIVITY #2: ON/OFF CONTROL WITH THE MICROCONTROLLER

In Activity #1, two different circuits were built and tested. One circuit made the LED emit light while the other did not. Figure 2-10 shows how the Propeller microcontroller can do the same thing if you connect an LED circuit to one of its I/O pins. In this activity, you will connect the LED circuit to the Activity Board and program its Propeller Microcontroller to turn the LED on and off. You will also experiment with programs that make the Propeller do this at different rates.

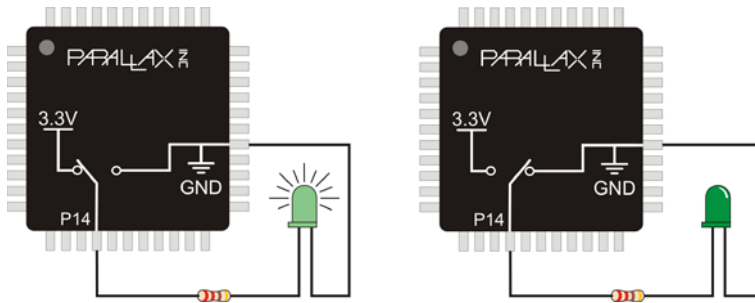


Figure 2-10
Switching Inside
the Propeller

A Propeller can be programmed to internally connect the LED circuit's input to 3.3V or GND.

There are two big differences between changing the connection manually and having the Propeller microcontroller do it. First, the Propeller doesn't have to cut the power to the development board when it changes the LED circuit's supply from 3.3V to GND. Second, while a human can make that change several times a minute, the Propeller can do it thousands or even millions of times per second!

LED Test Circuit Parts

Same as Activity #1.

Connecting the LED Circuit to the Propeller Microcontroller

The LED circuit shown in Figure 2-11 is wired almost the same as the circuit in the previous exercise. The difference is that the resistor's lead that was manually switched between 3.3V and GND is now connected to a Propeller I/O pin.

- Set your board's PWR switch to 0.
- Modify your circuit from Activity #1 so that it matches Figure 2-11.

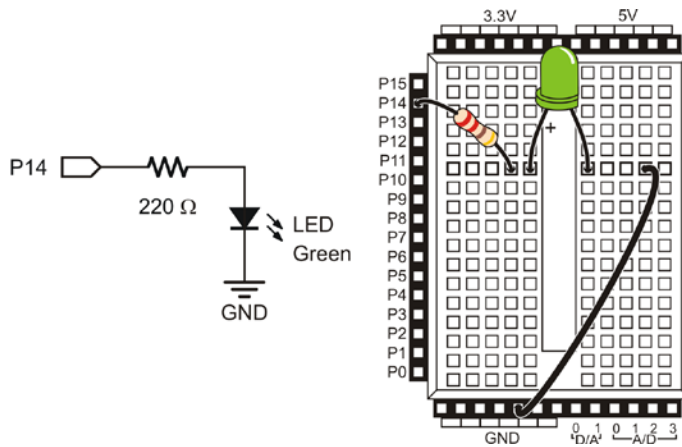


Figure 2-11
Propeller-controlled
LED Circuit

*The LED circuit's
input is now
connected to a
Propeller I/O pin
instead of 3.3V or
GND.*





Resistors are essential. Always remember to use a resistor. Without it, too much current will flow through the circuit, and it could damage any number of parts in your circuit, Propeller, or Activity Board.

Turning the LED On/Off with a Program

The example program makes the LED blink on and off one time per second. It introduces several new programming techniques at once. After running it, you will experiment with different parts of the program to better understand how it works.

Example Program: LED-OnOff

- Click SimpleIDE's New Project button .
- Name the new project LED-OnOff, and click Save.
- Enter the LED-OnOff.c code into SimpleIDE.
- Make sure your Activity Board is connected to your computer, and set the PWR switch back to 1.
- Click SimpleIDE's Run with Terminal button .
- Verify that the LED blinks on and off once per second.
- Disconnect power when you are done with the program.

```

/* LED-OnOff.c */

#include "simpletools.h"

int main()
{
    print("The LED connected to P14 is blinking!\n");

    while(1)
    {
        high(14);
        pause(500);
        low(14);
        pause(500);
    }
}

```



Build your programming muscles – type it in!

Experience shows that students learn better by typing in the example programs themselves, to save in the SimpleIDE > My Projects folder. However, if you just can't get an activity to work, you can find the example programs in the Learn > Examples > WAMM folder. This can be helpful if you need to find out if the problem is in your code, or your circuit. Be careful not to save changes to the examples for the Try This or Your Turn activities. Use Save Project As, rename them, and save them in your My Projects folder.

How LED-OnOff Works

Lines of code or comments between `/*` and `*/` are ignored by the C compiler. This lets you add the program name and instructions on how to use it. Just make sure your comments are between `/*` and `*/`.

The line `#include "simpletools.h"` is a directive that makes SimpleIDE add a library named `simpletools` to your project. Remember from the Propeller Brains for Your Inventions article that libraries are collections of functions — useful pre-written code that takes care of basic tasks — a big time saver. This library has code that makes the `high`, `low`, `pause`, and many other functions do their jobs. Simpletools also includes other libraries, such as `simpletext` that has code to make `print` do its job. You will see `simpletools.h` included in every program in this tutorial.

Remember that every program starts by executing the the first line of code inside of the `int main()` curly braces: `{ }`. Here, that is `print("The LED connected to P15 is blinking!")`. It's a lot like `print("Hello!")` but with different text between the quotes.

Next comes `while(1)`. The `while` command repeats the code inside of its own curly braces. Its syntax is:

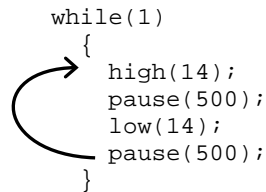
`while(condition){one or more statements to repeat}`

As long as the condition is not zero, the `while` loop will keep repeating. Since the `while` loop's condition is set at 1, the four statements between its curly braces keep repeating in order, as shown in Figure 2-12.

```

while(1)
{
  high(14);
  pause(500);
  low(14);
  pause(500);
}

```


 A diagram showing a code block for a while loop. The code is:


```

while(1)
{
  high(14);
  pause(500);
  low(14);
  pause(500);
}

```

 A curved arrow starts from the closing curly brace of the loop and points back to the opening curly brace, indicating that the code inside the braces repeats.

Figure 2-12

An infinite `while` Loop

The statements between the `while` loop's opening { and closing } braces get executed over and over endlessly.

The first statement in the `while` loop is `high(14)`. This makes the Propeller's P14 I/O pin internally connect to 3.3V, like the left side of Figure 2-10. This makes the LED light up. You will see this called “sending a high signal” throughout this tutorial.

The second statement is `pause(500)`, which makes the Propeller processor executing this program do nothing for 500 milliseconds. A *millisecond* is one thousandth of a second, so this `pause(500)` lasts one half of a second, keeping the LED lit.

The third statement is `low(14)`. This makes the I/O pin internally connect to GND, like the right side of Figure 2-10. This turns off the LED. We'll be calling that “sending a low signal.”

The fourth statement keeps LED off for a half second; it is another `pause(500)`. That's the last statement inside the curly braces, so the program execution returns to the first statement inside the braces.

Since there is nothing in the code that can cause the `while` condition to change, these four statements get executed over and over again until the power turns off or runs out! This is an example of what's called an *endless loop* or *infinite loop*.




More simpletools syntax

If you want to see the syntax for the `high`, `low`, and `pause` functions, click the SimpleIDE Help menu and select Simple Library Reference. When you get there, click the `simpletools.h` link under the Utility header. On the Simpletools Library page, scroll down to see all of the functions available, including `high`, `low`, and `pause`. We will dig deeper into function syntax later in this tutorial.

Your Turn – Timing and Repetitions

A *parameter* is a bit of information that a function needs to do its job. Your code needs to provide a value that parameter, also known as an *argument*, each time the function is called. For example, in `pause(500)`, the value 500 is provided for (often phrased as “passed to”) the `pause` function’s *time* parameter. By changing the value passed to the *time* parameter, you can change how long the LED stays on or off. For example, if you changed both instances of `pause(500)` to `pause(250)`, what do you think will happen?

- Use SimpleIDE’s Save Project As button  to save a copy of LED-OnOff.
- Name it LED-OnOff-YourTurn1, and save it to My Projects.
- Update the project name at the top of the code.
- Change both instances of `pause(500)` to `pause(250)` and re-run the program using SimpleIDE’s Run with Terminal button.

Did you correctly anticipate what would happen? The LED should now blink twice as fast as it did before, completing two on/off cycles in about 1 second.

The “on” time and “off” time do not have to be the same. For example, let’s make the LED blink on and off once every three seconds, with the low time twice as long as the high time. To do this, use `pause(1000)` after `high(14)` so that the LED stays on for one second. Then, use `pause(2000)` after `low(14)` to keep the LED off for 2 seconds:

```
while(1)
{
  high(14);
  pause(1000);
  low(14);
  pause(2000);
}
```

- Save another copy as LED-OnOff-YourTurn2, and then update the arguments in the `pause` function calls as shown above.

A fun experiment is to see how short you can make the pauses and still see that the LED is flashing. When the LED is flashing very fast, it looks like it's just staying on, a phenomenon called *persistence of vision*.

To test your own persistence of vision threshold:

- Set your `pause` calls' *time* values to 100.
- Save a copy as LED-OnOff-YourTurn3 and re-run your program and check for flicker.
- Reduce the value for both *time* values by 5 and try again.
- Keep reducing *time* values until the LED appears to be on all the time with no flicker. When you cross from flicker to solid you have reached the minimum time your eye can detect. After persistence of vision kicks in the LED will be dimmer than normal, but it should not appear to flicker.

One last thing to try is to just flash the LED once. This is a way to look at the functionality of the `while` loop. You can de-activate a line of code by placing two forward slashes `//` at the beginning of the line. This is also called “commenting out” the line, and it is a very useful tool to test the effect of changes to your code while you are developing programs. (Two forward slashes can also be placed at the end of the line of code, followed by notes about what that line does. These comments will be ignored by the code compiler, but are very useful to other humans who might want to read and understand your code.)

To de-activate the `while` loop, you will need to comment out three lines of code that make up its syntax: the `while(1)` itself, and the lines with its opening and closing curly braces `{` and `}`. If you have been playing the persistence of vision game, you will also need to make your `pause` times longer again so you don't miss the flash.

```
// while(1)
// {
    high(14);
    pause(1000);
    low(14);
    pause(2000);
// }
```

- Modify, save a copy as LED-OnOff-YourTurn4, and re-run the program using the code snippet above.
- Explain what happened. Why did the LED only flash once?

ACTIVITY #3: COUNTING AND REPEATING

In the previous activity, the LED circuit either flashed on and off endlessly, or it flashed once and then stopped. What if you want the LED to flash on and off exactly ten times? Computers (including the Propeller) are great at keeping running totals of how many times something happens. Computers can also be programmed to make decisions based on a variety of conditions. In this activity, you will program the Propeller to stop flashing the LED on and off after ten repetitions.


Counting Parts and Test Circuit

Continue using the example circuit shown in Figure 2-11 on page 42.

Counting with a While Loop

LED-OnOffTenTimes shows how just a few updates to the previous activity's LED-OnOff program can use counting and comparing in the `while` loop's condition to limit the light to ten blinks. Counting takes two steps. First, declare a variable to count the number of times the LED blinks. This example uses `int x = 1` to declare a variable named `x` and set it to 1. Then, at the very end of the `while` loop, insert the line `x = x + 1`. Now, every time through the `while` loop, the value of `x` will increase by 1. Inside the `while` loop's condition, change `1` to `x <= 10`. That means the `while` loop will only keep repeating while `x` is less than or equal to 10. To help us see what the loop is doing, a `print` statement to display the value of `x` was added at the start of the `while` loop.

Example Program: LED-OnOffTenTimes

- Use SimpleIDE's Open Project button to open LED-OnOff .
- Use SimpleIDE's Save Project As button and name a copy LED-OnOffTenTimes. Don't forget to update the project name at the top of the code!
- Delete the line `print("The LED connected to P14 is blinking!\n")`.
- Make the additions and changes shown below.

```

/* LED-OnOffTenTimes.c */

#include "simpletools.h"

int main()
{
    int x = 1;                                // <- Add

    while(x <= 10)                            // <- Change
    {
        print("x = %d\n", x);                 // <- Add
        high(14);
        pause(500);
        low(14);
        pause(500);
        x = x + 1;                             // <- Add
    }
}

```

- Use the Run with Terminal button, and verify that the LED stops blinking after ten reps. The blinks start quickly so begin watching right after you click on Run with Terminal.
- Use Run with Terminal a second time and watch the SimpleIDE Terminal to verify that the value of **x** counts from 1 to 10.

Easier Counting with a For Loop

There's a special kind of loop called a *for loop* that makes this job easier. Instead of using three lines (to declare a variable, use **while** with a condition, and add to variable in the loop) you can do it all with just one line. Try it!

Example Program: LED-OnOffTenAgain

- Use SimpleIDE's Save Project As button to save LED-OnOffTenTimes as LED-OnOffTenAgain.
- Delete these two lines: **int x = 1; x = x + 1;**
- Change **while(x <= 10)** to **for(int x = 1; x <= 10; x++)**
- Use the Run with Terminal button, and verify that the value of **x** counts from 1 to 10 as the LED blinks 10 times.


```

/* LED-OnOffTenAgain.c */

#include "simpletools.h"

int main()
{
  for(int x = 1; x <= 10; x++)           // <- Change
  {
    print("x = %d\n", x);
    high(14);
    pause(500);
    low(14);
    pause(500);
  }
}

```

How LED-OnOffTenAgain Works


This `for` loop has three parameters:

1. A variable declaration and starting value. Here, the argument is `int x = 1`,
2. A condition. Here, the argument is `x <= 10`;
3. A variable operation. Here, the argument is `x++`

The third argument is something you might not have seen yet, `x++`. The operator `++` means “add one to the variable” and its position after the `x` means “do the addition after it gets used in this instruction.” It’s called the *post-increment* operator, and it is a shortcut to writing `x = x + 1`.

- Try replacing `x++` with `x = x + 1`.
- Re-run the program and verify that it still works the same.


The `for` loop depends on a variable to track how many times the LED has blinked on and off. Recall from the [Variables and Math lesson](#) that a variable is a name you give to a section of microcontroller memory so your program “remembers” values and works with them. We used the name `x`, but you could also have picked something more self-explanatory, like `blinkReps`.



Variable Name Rules:

1. The name cannot be a word that is already used by the C language, like `for`, `while`, and `main`. These words are called *reserved words*.
2. The name cannot contain a space.
3. Even though the name can contain letters, numbers, or underscores, it must begin with either a letter or an underscore.
4. Give each variable a unique name. (A good practice, but not technically necessary in certain circumstances — we will explore variable scope in Chapter 5.)

The `int` in `int x = 1;` tells the C compiler that the `for` loop will use the letter `x` as a variable that can store an `int` variable's worth of information.



What's an int? An `int` is enough memory to store a number in the approximately negative 2 billion to positive 2 billion range. Here are some examples of C language variable types supported by the Propeller.

Table 2-2: Variable Types	
Variable type	Range of Values
char	-128 to 127
unsigned char	0 to 255
short	-32768 to 32767
unsigned short	0 to 65535
int	-2,147,483,648 to 2,147,483,647
unsigned int	0 to 4,294,967,296
float	For numbers with decimal point that can range from very large to very small, with six digits of precision
double	Like float, but with ten digits of precision

Your Turn – Other Ways to Count

In the `LED-OnOffTenAgain`, replace the statement:

```
for(int x = 1; x <= 10; x++)
```

...with this:

```
for(int x = 1; x <= 20; x++)
```

- Save a copy of the program and name it LED-OnOffTenAgain-YourTurn1. Re-run the program. What was different, and was it expected?
- Save another copy as LED-OnOffTenAgain-YourTurn2. Try a second modification to the `for` statement. This time, change it to:

```
for(int x = 1; x <= 20; x = x + 4)
```

How many times did the LED flash? What values displayed in the Debug Terminal?

- Save a third copy as LED-OnOffTenAgain-YourTurn3. This time, replace `x = x + 4` with `x += 4`.

Did you get the results you expected?

ACTIVITY #4: BUILDING AND TESTING A SECOND LED CIRCUIT

Indicator LEDs often tell a machine's user many things. Many devices need two, three, or more LEDs to tell the user if the machine is ready or not, if there is a malfunction, if it's done with a task, and so on.

In this activity, you will repeat the LED circuit test in Activity #1 for a second LED circuit. Then, you will adjust the example program from Activity #2 to make sure the LED circuit is properly connected to the Propeller. After that, you will modify the example program from Activity #2 to make the LEDs operate in tandem.

Extra Parts Required

In addition to the parts you used in Activities 1 and 2, you will need these parts:

- (1) LED – yellow
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Jumper wire (black)

Building and Testing the Second LED Circuit

In Activity #1, you manually tested the first LED circuit to make sure it worked before connecting it to the Propeller. Before connecting the second LED circuit to the Propeller, it's important to test it too.

- Set your board's PWR switch to 0.
- Construct the second circuit as shown in Figure 2-13.
- Make sure the leads of the different resistors are not touching each other.
- Set your board's PWR switch to 1.

Did the LED circuit you just added turn on? If yes, then continue. If no, Activity #1 has some troubleshooting suggestions that you can repeat for this circuit.

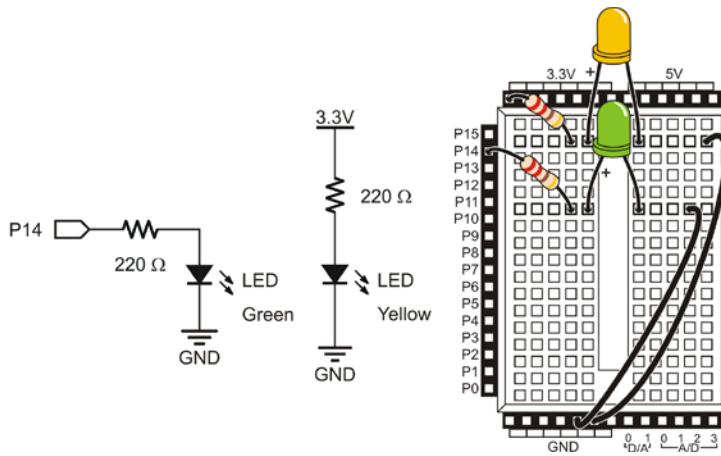


Figure 2-13
Manually Test the
Second LED

- Turn off power, and then connect the second LED circuit's resistor lead to P15 as shown in Figure 2-14.

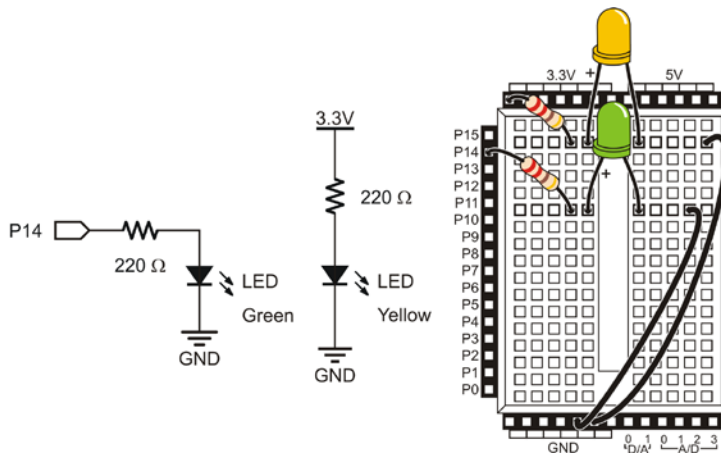


Figure 2-14
Connect the Second
LED to I/O Pin 15

Using a Program to Test the Second LED Circuit

In Activity #2, you used an example program with **high**, **low**, and **pause** statements to control the P14 LED circuit. These statements can be modified to control the P15 LED circuit. It is simply a matter of passing 15 instead of 14 to the **high** and **low** functions' pin parameters.

Example Program: LED-TestSecond

- Use SimpleIDE's Open Project button to open LED-OnOff.
- Use SimpleIDE's Save Project As button to save it as LED-TestSecond.
- Change the values passed to the **high** and **low** pin parameters from 14 to 15.
- Click SimpleIDE's Run with Terminal button and verify that the LED in the circuit connected to P15 blinks.

```

/* LED-TestSecond.c */
#include "simpletools.h"

int main()
{
    print("The LED connected to P15 is blinking!\n");

    while(1)
    {
        high(15);                // <- Change
        pause(500);
    }
}

```

```

low(15); // <- Change
pause(500);
}
}

```

Controlling Both LEDs

Yes, you can flash both LEDs at once! One way you can do this is to use two **high** statements before the first **pause** statement, one for P14 and one for P15. You will also need two **low** statements to turn both LEDs off. It's true that both LEDs will not turn on and off at exactly the same moment because one is turned on or off after the other, but the difference will be infinitesimal compared to how long it would take for the human eye to actually detect it.

Example Program: LED-FlashBoth

- If it's not already open, open LED-TestSecond, and then use Save Project As to save it as LED-FlashBoth.
- Add the **high(14)** and **low(14)** lines as shown below.
- Click the Run with Terminal button.
- Verify that both LEDs appear to flash on and off at the same time.

```

/* LED-FlashBoth.c */

#include "simpletools.h"

int main()
{
    print("The LEDs connected to P14 and P15 are blinking!\n");

    while(1)
    {
        high(15);
        high(14); // <- add
        pause(500);
        low(15);
        low(14); // <- add
        pause(500);
    }
}

```

Your Turn – Alternate LEDs

What if you want one LED to be on while the other LED is off, and vice versa? Think about how you would need to change your code. If you thought of swapping the `high` and `low` statements that control one of the I/O pins, you thought right.

- Modify LED-FlashBoth so that the statements in the `while` loop look like this:

```

high(15);
low(14);           // <- change
pause(500);
low(15);
high(14);         // <- change
pause(500);

```

- Save a copy and name it LED-FlashBoth-YourTurn. Run the modified code and verify that the lights are on alternately.

ACTIVITY #5: CONTROL A BICOLOR LED WITH CURRENT DIRECTION

The device shown in Figure 2-15 is a security system’s card reader for electronic key cards. When key card with the right code is held near the device, the LED changes color from red to green, and a door unlocks. This kind of LED is called a *bicolor* LED. This activity answers two questions:

1. How does the LED change color?
2. How can you control a bicolor LED with the Propeller microcontroller?



Figure 2-15
Bicolor LED in a Security Device

When the door is locked, this bicolor LED glows red. When the door is unlocked by an electronic key with the right code, the LED turns green.

Introducing the Bicolor LED

The bicolor LED's schematic symbol and part drawing are shown in Figure 2-16. This LED has a rounded dome and a cloudy white color. (You may also have a part with a clear case and flat top – that's a phototransistor, and we will use it later.)

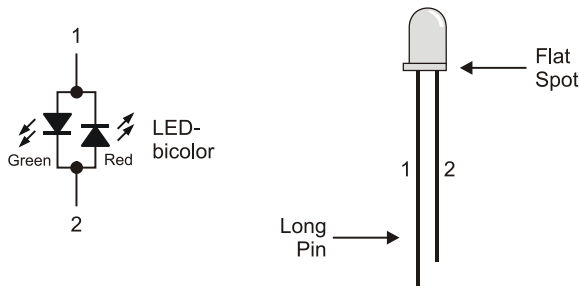
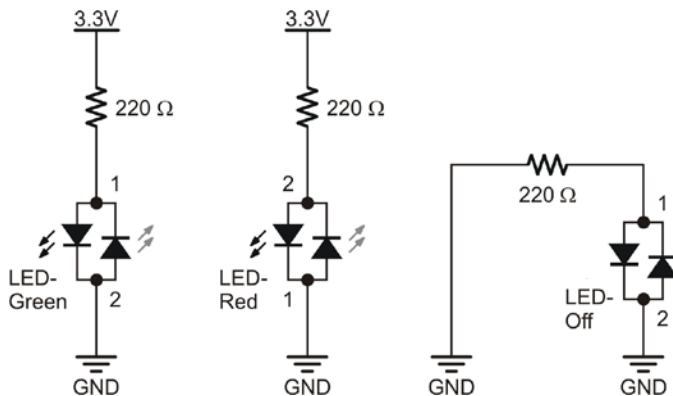


Figure 2-16
Bicolor LED
Schematic symbol (left) and part drawing (right).

The bicolor LED is really just two LEDs in one package. Figure 2-17 shows how you can apply voltage in one direction and the LED will glow green. By disconnecting the LED and plugging it back in reversed, the LED will then glow red. As with the other LEDs, if you connect both terminals of the circuit to GND, the LED will not emit light. Since current could be going either direction in your project, the leads are named 1 and 2 rather than + and -.

**Figure 2-17**

Bicolor LED and
Applied Voltage

*Green (left), red
(center) and no
light (right)*

Bicolor LED Circuit Parts

- (1) LED – bicolor (plastic case is a cloudy white color).
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Jumper wire (black)

Building and Testing the Bicolor LED Circuit

Figure 2-18 shows the manual test for the bicolor LED.

- Set your board's PWR switch to 0, and remove any parts from the last activity.
- Build the circuit shown on the left side of Figure 2-18.
- Set the PWR switch to 1, and verify that the bicolor LED is emitting green light.
- Set PWR to 0 again.
- Modify your circuit by turning the LED around so that its leads swap position, as shown in the right side of Figure 2-18.
- Set PWR to 1, verify that the bicolor LED is now emitting red light.
- Set PWR to 0 again.

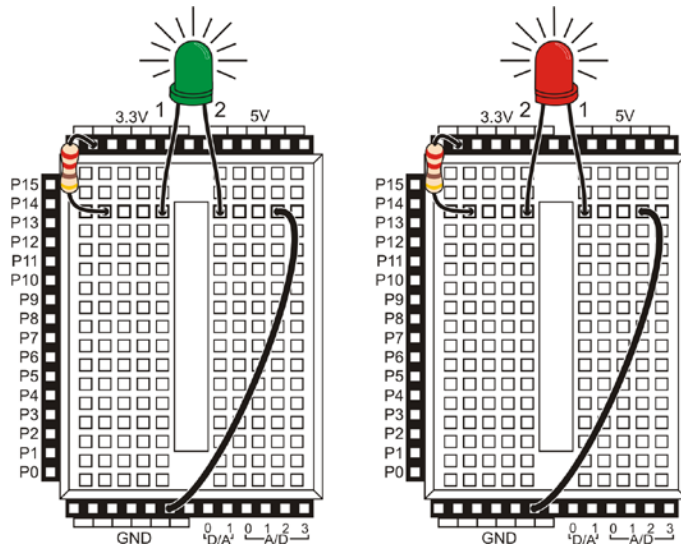


Figure 2-18
Manual Bicolor LED Test

Bicolor LED green (left) and red (right).

Your LED might be manufactured with the colors reversed.



What if my bicolor LED's colors are reversed? If your bicolor LED glows red when it's connected in the circuit that should make it glow green and vice-versa, your LED's colors are reversed inside the case. Just plug pin 1 in where the diagrams show pin 2, and pin 2 where the diagrams show pin 1.

Controlling a bicolor LED with a microcontroller requires two I/O pins. After you have manually verified that the bicolor LED works using the manual test, you can connect the circuit to the Propeller as shown in Figure 2-19.

- Connect the bicolor LED circuit to the Propeller microcontroller as shown in Figure 2-19.
- Double-check that the bare metal leads of the components are not accidentally touching above the breadboard.

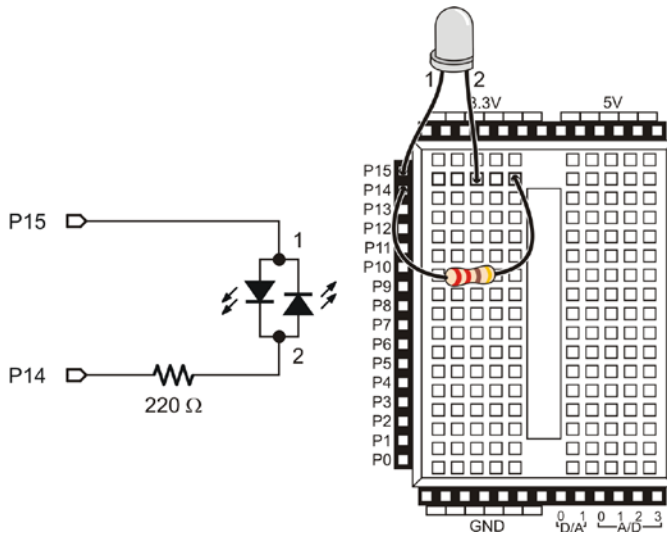


Figure 2-19
Bicolor LED Connected
to Propeller

*Schematic (left) and
wiring diagram (right).*

Propeller Bicolor LED Control

Figure 2-20 shows how you can use P15 and P14 to control the current flow in the bicolor LED circuit. The upper schematic shows how current flows through the green LED when P15 is internally connected to 3.3V and P14 is internally connected to GND. The green LED will let current flow through it when electrical pressure is applied as shown, but the red LED acts like a closed valve and does not let current through it. Therefore, only the bicolor LED glows green.

The lower schematic shows what happens when P15 is instead set to GND and P14 is set to 3.3V. The electrical pressure is now reversed. The green LED shuts off and does not allow current through. Meanwhile, the red LED turns on as current passes through the circuit in the opposite direction.

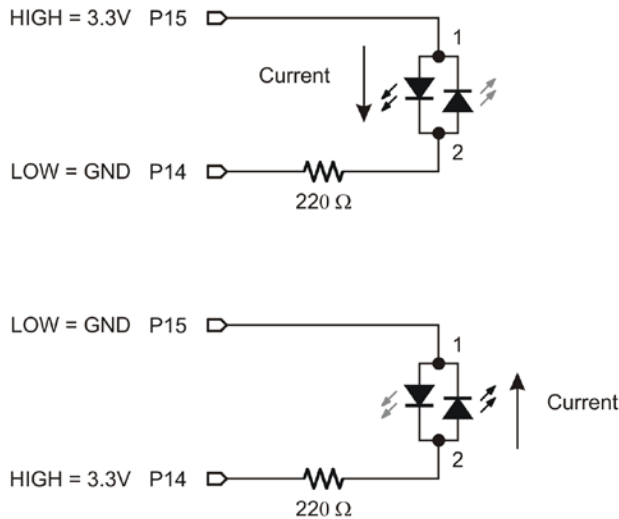


Figure 2-20
Propeller Bicolor LED
Test

*Current through green
LED (above) and red
LED (below).*

Figure 2-20 also shows the key to programming the Propeller to make the bicolor LED glow two different colors. The upper schematic shows how to make the bicolor LED green using `high(15)` and `low(14)`. The lower schematic shows how to make the bicolor LED glow red by using `low(15)` and `high(14)`. To turn the LED off, connect both leads to GND using `low(15)` and `low(14)`.



The bicolor LED will also turn off if you send high signals to both P14 and P15. Why? Because the electrical pressure (voltage) is the same at P14 and P15 if you set both I/O pins high (3.3 V). So, the effect is the same as setting both P14 and P15 low (0 V = GND).

Example Program: LED-TestBicolor

- Set the PWR switch to 1.
- Use SimpleIDE's New Project button to create a project named LED-TestBicolor.
- Enter the code below and test with the Run with Terminal button.
- Verify that the LED cycles through the red, green, and off states.

```
/* LED-TestBicolor.c */
#include "simpletools.h"
```

```

int main()
{
    print("Program running! \n");

    while(1)
    {
        print("Green \n");
        high(15);
        low(14);
        pause(1500);

        print("Red \n");
        low(15);
        high(14);
        pause(1500);

        print("Off \n");
        low(15);
        low(14);
        pause(1500);
    }
}

```

Your Turn – Lights Display

In Activity #3, a variable named **x** was used to control how many times an LED blinked. What happens if you use the value **x** to control the **pause** function's **time** parameter while repeatedly changing the color of the bicolor LED?

- Use the Save as Project button to create a copy of your code named LED-TestBicolor-YourTurn.
- Replace the code inside the **while(1)** loop with this:

```

for(int x = 1; x <= 50; x++)
{
    high(15);
    low(14);
    pause(x);

    low(15);
    high(14);
    pause(x);
}

```

When you are done, your code should look like this:

```
#include "simpletools.h"

int main()
{
    print("Program running! \n");

    while(1)
    {
        for(int x = 1; x <= 50; x++)
        {
            high(15);
            low(14);
            pause(x);

            low(15);
            high(14);
            pause(x);
        }
    }
}
```

At the beginning of each pass through the `for` loop, the `pause` value (*time* parameter) is only one millisecond. Each time through the `for` loop, the `pause` gets longer by one millisecond at a time until it gets to 50 milliseconds. The `while(1)` causes the `for` loop to go through this process over and over again.

Run the modified program and observe the effect. Did it do what you expected?

SUMMARY

This chapter introduced lots of new concepts, electronic components, and programming techniques:

- How common devices use indicator lights.
- What a resistor is, what its schematic symbol looks like, and how to decode its colored markings to determine its resistance value.
- What an LED (light-emitting diode) is, and what its schematic symbol looks like.
- What a bicolor LED is, and what its schematic symbol looks like.

- What a solderless breadboard is, and how to use it to make electrical connections between components for building a circuit.
- How electrons move through a circuit to turn on an LED.
- How to build an LED circuit and turn it on and off by manually connecting it to power.
- How current direction/voltage polarity determine which color a bicolor LED will glow.
- How to build an LED circuit connected to a microcontroller.
- How to write programs to control regular and bidirectional LED circuits.
- How to use the `high`, `low`, and `pause` functions from the `simpletools` library.
- What a parameter is, and how to pass values to a function's parameter.
- How to use a `while` instruction to make a block of code repeat endlessly (an infinite loop), or only while an expression evaluates as true using a variable (a conditional loop).
- How to use a `for` instruction to make a block of code repeat a certain number of times (a counted loop) by using a variable.
- How to use the post-increment operator, as in `x++`.
- How to use `//` at the front of a line of code to deactivate it, or at the end of a line of code to add human-readable comments.
- Several types of C variable names and value ranges.
- Basic rules for naming variables.

Questions

1. What is the name of this Greek letter: Ω , and what measurement does Ω refer to?
2. Which resistor would allow more current through the circuit, a 470 Ω resistor or a 1000 Ω resistor?
3. How do you connect two wires using a breadboard? Can you use a breadboard to connect four wires together?
4. What do you always have to do before modifying a circuit that you built on a breadboard?
5. How long would `pause(10000)` last?
6. How would you cause a Propeller's processor to do nothing for an entire minute?
7. What are three different C variable types?
8. Can a `char` variable hold the value 500?

9. What will the command `high(7)` do?

Exercises

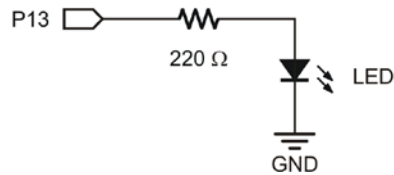
1. Draw the schematic of an LED circuit like the one you worked with in Activity #2, but connect the circuit to P13 instead of P14. Explain how you would modify LED-OnOff on page 42 so that it will make your LED circuit flash on and off four times per second.
2. Explain how to modify LED-OnOffTenTimes so that it makes the LED circuit flash on and off 5000 times before it stops.

Project

1. Make a 10-second countdown using one yellow LED and one bicolor LED. Make the bicolor LED start out red for 3 seconds. After 3 seconds, change the bicolor LED to green. When the bicolor LED changes to green, flash the yellow LED on and off once every second for ten seconds. When the yellow LED is done flashing, the bicolor LED should switch back to red and stay that way for three seconds before turning off.

Solutions

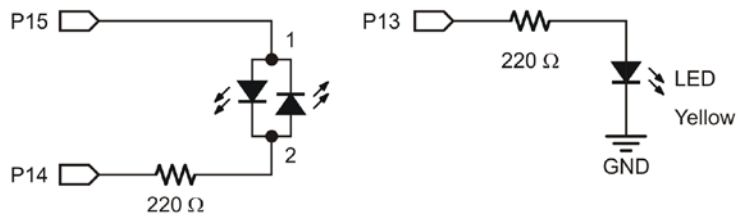
- Q1. Its name is omega, and it refers to the ohm, which measures how strongly something resists current flow.
 - Q2. A 470 Ω resistor: higher values resist more strongly than lower values, therefore lower values allow more current to flow.
 - Q3. To connect 2 wires, plug the 2 wires into the same 5-socket group. You can connect 4 wires by plugging all 4 wires into the same 5-socket group.
 - Q4. Disconnect the power by setting your board's PWR switch to 0.
 - Q5. 10 seconds.
 - Q6. `pause(60000)`
 - Q7. Any three of these would be correct: `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `float`, `unsigned float`.
 - Q8. No. A `char` variable only holds from -128 to 127, so the value 500 is too large. A `short` or `int` would work.
 - Q9. `high(7)` will cause the Propeller to internally connect I/O pin P7 to 3.3V.
-
- E1. The `pause(time)` must be reduced to $500 \text{ ms} / 4 = 125 \text{ ms}$, so `pause(125)`. Propeller C will also accept `pause(500/4)`. To use I/O pin P13, `high(14)` and `low(14)` have been replaced with `high(13)` and `low(13)`.



```
while(1)
{
  high(13);
  pause(125);
  low(13);
  pause(125);
}
```

E2. Change `for(int x = 1; x <= 10; x++)` to `for(int x = 1; x <= 5000; x++)`.

P1. The bicolor LED schematic, on the left, is unchanged from Figure 2-19 on page 59. The yellow LED schematic is based on Figure 2-11 on page 42. For this project P14 was changed to P13, and a yellow LED was used instead of green.



```
/* LED-P1-Solution.c
 10 Second Countdown with Red, Yellow, Green LED
 Red/Green: Bicolor LED on P15, P14. Yellow: P13 */

#include "simpletools.h"

int main()
{
  print("Program Running!");

  low(15); // Bicolor LED Red
  high(14);
  pause(3000); // ...for three seconds

  high(15); // Bicolor LED Green
  low(14);

  for(int x = 1; x <= 10; x++) // ...while yellow flashes
  {
    high(13); // Yellow LED on
    pause(500);
  }
}
```

```
    low(13);                // Yellow LED off
    pause(500);
}

low(15);                  // Bi Color LED Red
high(14);
pause(3000);              // Bi Color LED Red
low(14);                  // Bi Color LED off
}
```

Chapter 3: Digital Input – Pushbuttons

FOUND ON CALCULATORS, HANDHELD GAMES, AND APPLICANCES

How many devices with pushbuttons do you use on a daily basis? Think about a computer, mouse, calculator, microwave oven, TV remote, handheld game, and cell phone. In each device, there is a microcontroller scanning the pushbuttons and waiting for a circuit to change when you push a button. When the microcontroller detects a change, it carries out whatever action you expect the pushbutton to trigger. By the end of this chapter, you will have experience with designing pushbutton circuits and programming the Propeller to monitor them and take action when changes occur.

RECEIVING VS. SENDING HIGH AND LOW SIGNALS

In Chapter 2, you programmed the Propeller make its I/O pins send, or *output*, high and low signals. The LEDs turned on and off (or changed color) to display the state of these signals. In this chapter, you will use a Propeller I/O pin as an input. As an *input*, an I/O pin “listens” for high/low signals instead of sending them. You will send these signals to the Propeller with a pushbutton circuit, and you will program the Propeller to recognize whether the pushbutton is pressed or not pressed.



Other terms that mean send, high/low, and receive: Sending high/low signals is described in different ways. You might see sending referred to as *transmitting*, *controlling*, or *switching*. Instead of high/low, you might see it referred to as *binary(1/0)*, *TTL*, *CMOS*, or *Boolean* signals. Another term for receiving is *sensing*.

ACTIVITY #1: TESTING A PUSHBUTTON WITH AN LED CIRCUIT

If you can use a pushbutton to send a high or low signal to the Propeller, can you also control an LED with a pushbutton? The answer is yes, and you will use it to test a pushbutton in this activity.

Introducing the Pushbutton

Figure 3-1 shows the schematic symbol and the part drawing of the pushbutton included in your kit. It has four pins, connected in two pairs: 1,4 and 2,3.

This means that connecting a wire to pin 1 of the pushbutton is the same as connecting it to pin 4. The same rule applies with pins 2 and 3. The reason the pushbutton doesn't just

have two pins is because it needs stability and strength so it doesn't bend or break when you push on it.

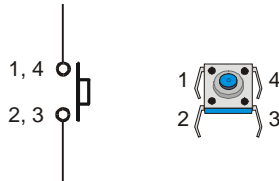


Figure 3-1

Normally Open Pushbutton

Schematic symbol (left) and part drawing (right)

This is a *normally open* pushbutton, and it looks like the left side of Figure 3-2 when it's **not** pressed. Notice there is a gap between the 1,4 and 2,3 terminals. This gap makes it so that the 1,4 terminal cannot conduct current to the 2,3 terminal. This is called an *open circuit*. So, this pushbutton's normal not-pressed state forms an open circuit, which gives us the name "normally open." When the button is pressed, the gap between the 1,4 and 2,3 terminals is bridged by conductive metal, as shown in Figure 3-3. This forms a *closed circuit*, and current can then flow through the pushbutton.

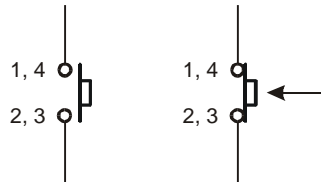


Figure 3-2

Normally-Open Pushbutton

Not pressed (left) and pressed (right)

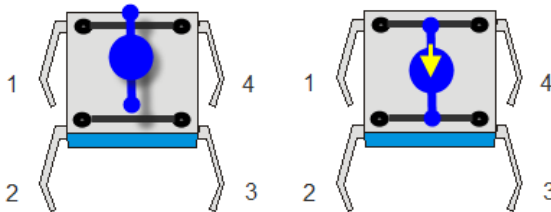


Figure 3-3


Pressing the button bridges the 1,4 and 2,3 terminals with conductive metal

Test Parts for the Pushbutton

- (1) LED – red, yellow, or green, your choice!
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Pushbutton – normally open
- (2) Jumper wires

Building the Pushbutton Test Circuit

Figure 3-4 shows a circuit you can build to manually test the pushbutton.



Always disconnect power from your board by setting the PWR switch to 0 before making any changes to your test circuit. Unplugging the USB cable (which can supply power) and any external power supply will also disconnect power.

Always reconnect power to your board before downloading a program to the Propeller, by setting the PWR switch to 1. Of course, you must have your USB programming cable connected as well, and you should also reconnect any external supply you may have unplugged.

IMPORTANT: From here onward, the instructions will no longer say “Disconnect power...” and “Reconnect power” between each circuit modification. **It is up to you to remember!**

WARNING SIGNS: When doing any circuit activities, if the LEDs below the PWR switch on the Activity Board flicker, go dim, or go out completely when you reconnect power, **DISCONNECT POWER IMMEDIATELY** and re-check your circuit. It may mean that there is a short circuit from 3.3 V or 5 V to GND. Fix any circuit errors before proceeding.

Build the circuit shown in Figure 3-4.

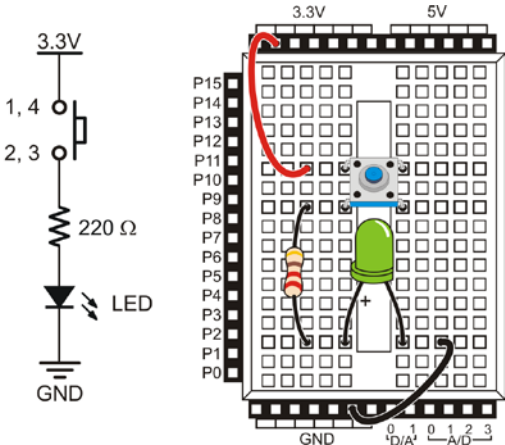


Figure 3-4
Circuit for pushbutton to turn on an LED

Testing the Pushbutton

When the pushbutton is not pressed, the LED will be off. If the wiring is correct, when the pushbutton is pressed, the LED should be on (emitting light).

- Press the pushbutton down — you should hear or feel a little click.
- Verify that when you hold down the pushbutton, the LED emits light, and when you let go of the pushbutton, the LED turns off.

How the Pushbutton Circuit Works

The left side of Figure 3-5 shows what happens when the pushbutton is not pressed. The LED circuit is not connected to 3.3 V. It is an open circuit that cannot conduct current. By pressing the pushbutton, as shown on the right side of the figure, you close the connection between the terminals with conductive metal. This makes a pathway for electrons to flow through the circuit, and so the LED emits light as a result.

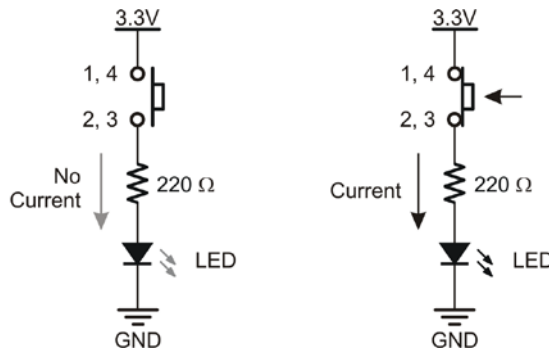


Figure 3-5

Pushbutton Not Pressed, and Pressed

Pushbutton not pressed: circuit open and light off (left)

Pushbutton pressed: circuit closed and light on (right)

Your Turn – Turn the LED off with a Pushbutton

Figure 3-6 shows a different pushbutton and LED circuit. In this case, when the pushbutton is not pressed, the LED stays on; when the button is pressed, the LED turns off. When the pushbutton is not pressed, current flows through the LED and it emits light. But when the pushbutton is pressed, conductive metal connects terminals 1,4 and 2,3, and electricity will take the *path of least resistance* through the pushbutton instead of through the LED.

- Build the circuit shown in Figure 3-6.

- ❑ Make sure the LED's longer anode lead is in the same row with the wire coming from 3.3V, as marked with a (+) sign in the wiring diagram. The LED's shorter cathode lead (by the flat spot on the case) is in the same row with the resistor going to GND.
- ❑ Now, turn on the PWR switch. The LED should turn on.
- ❑ Press and hold down the pushbutton. This should now make the LED turn off.

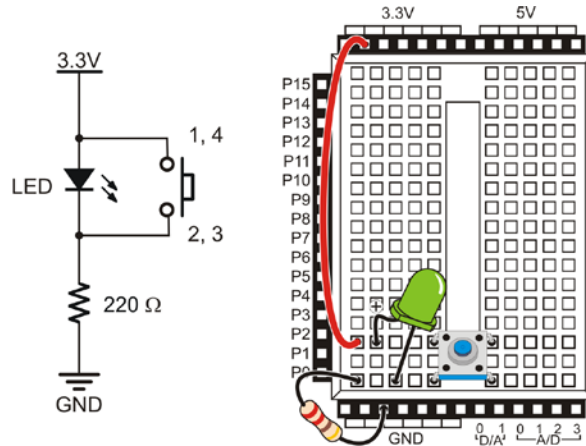


Figure 3-6
Circuit for Pushbutton to Turn Off an LED



Can you really do that with the LED? Up until now, a resistor has always connected the LED's anode to either 3.3V or an I/O pin. But now, the anode is connected directly to 3.3V, and the resistor is connecting the LED's cathode to GND. People often ask if this breaks any circuit rules.

The answer is no! The electrical pressure supplied by 3.3V and GND is 3.3 volts. You might see the term *voltage drop* describing how much voltage to expect across each component's leads. The green LED will always have a voltage drop in the 2.1 V range, and the resistor will use the remaining 1.2 V, regardless of their order.

ACTIVITY #2: READING A PUSHBUTTON WITH THE PROPELLER

In this activity, you will connect a pushbutton circuit to a Propeller microcontroller I/O pin, and display whether or not the pushbutton is pressed. You will do this by writing a C program that checks the state of the pushbutton and prints it in the SimpleIDE Terminal.

Parts for a Pushbutton Circuit

- (1) Pushbutton – normally open
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Resistor – 10 k Ω (brown-black-orange)
- (1) Jumper wire (red)

Building a Pushbutton Circuit for the Propeller Microcontroller

Figure 3-7 shows a pushbutton circuit that is connected to Propeller I/O pin P3.

- Build the circuit shown in Figure 3-7.

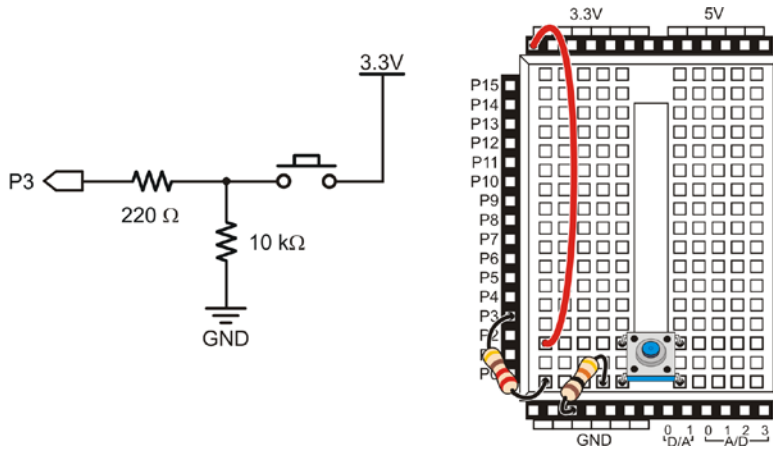
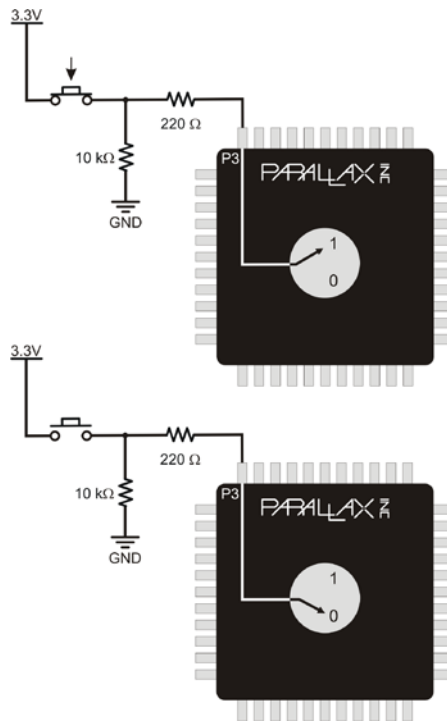


Figure 3-7
Pushbutton
Circuit
Connected to I/O
Pin P3

*On the wiring
diagram, the
220 Ω resistor
(red-red-brown)
is connecting the
pushbutton's
lower terminal to
P3.*

The upper half of Figure 3-8 shows how the Propeller responds when the pushbutton is pressed. The Propeller senses that 3.3 V is connected to P3, and responds by placing the number 1 in a part of its memory that stores information about its I/O pins.

When the pushbutton is not pressed, the lower half of Figure 3-8 shows that the Propeller cannot sense 3.3 V, but it can sense GND through the 10 k Ω and 220 Ω resistors. This causes it to store the number 0 in that same memory location. We can use the simpletools library's `input` function to check that memory location, and take a different action depending on if it's storing a 0 or a 1.

**Figure 3-8**

Propeller Reading a Pushbutton

When the pushbutton is pressed, the Propeller reads a 1 (above). When the pushbutton is not pressed, the Propeller reads a 0 (below).



Binary and Circuits: The base-2 number system uses only the digits 1 and 0 to make numbers, and these binary values can be transmitted from one device to another. The Propeller interprets 3.3 V as binary 1 and GND (0 V) as binary 0. Likewise, when the Propeller sets an I/O pin to 3.3 V using `high`, it sends a binary 1. When it sets an I/O pin to GND using `low`, it sends a binary 0. This is a very common way of communicating binary numbers that is used by many computer chips and other devices.

Example Program: Button-ReadState

This next program uses the `input` function to check the pushbutton every $\frac{1}{4}$ second, and then display the function's return value, which will be a 0 or 1. Figure 3-9 shows the SimpleIDE Terminal while the program is running. When the pushbutton is pressed, SimpleIDE Terminal displays the number 1, and when the pushbutton is not pressed, it displays the number 0.

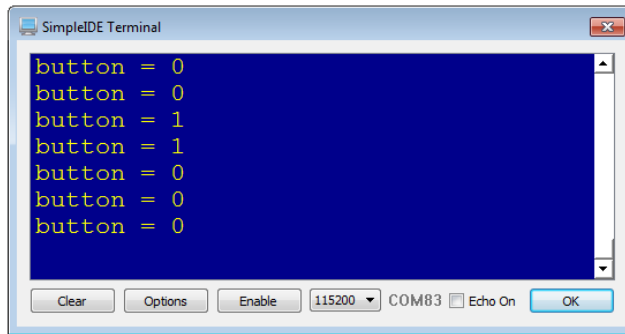


Figure 3-9
SimpleIDE Terminal
Displaying
Pushbutton States

The SimpleIDE Terminal displays 1 when the pushbutton is pressed and 0 when it is not pressed.

- Click the New Project button and name the project Button-ReadState.
- Enter the Button-ReadState.c code into SimpleIDE.
- Click the Run with Terminal button.
- Verify that the SimpleIDE Terminal displays the value 0 when the pushbutton is not pressed, and the value 1 when pressed.

```

/* Button-ReadState.c */

#include "simpletools.h"

int main()
{
    int button;

    while(1)
    {
        button = input(3);
        print("button = %d\n", button);
        pause(250);
    }
}

```

How Button-ReadState Works

The program contains an infinite `while(1)` loop. Just above the loop, `int button;` declares a variable named `button`. The first statement in the loop, `button = input(3)`, translates to “check I/O pin P3, and assign its input state to the variable named `button`.” The `input(3)` function call checks the state of P3, and will return a value of 1 if a button is pressed, or 0 if it is not pressed. So, a 0 or a 1 are the two possible values that could be assigned to `button`.

Next, `print("button = %d\n", button)` displays the string between the quotation marks in the SimpleIDE terminal. The text “button = ” is displayed just as appears in the string. The `%d` flag means “right here, display the value that follows this string as a decimal integer value.” In this case, the `button` variable comes after the string, so its value, which will either be 0 or 1, is printed next. Finally, the `\n` formatter tells the `print` statement to put the cursor at the beginning of the next line.

The last statement is `pause(250)`, which makes the program wait for $\frac{1}{4}$ of a second before allowing the `while(1)` loop to repeat. This makes the terminal display more readable. Without it, the values would just race by.

Your Turn – A Pushbutton with a Pull-up Resistor

The circuit you just finished working with has a resistor connected to GND, called a *pull-down* resistor because it pulls the voltage at P3 down to GND (0 volts) when the button is not pressed. Figure 3-10 shows a pushbutton circuit that uses a *pull-up* resistor.

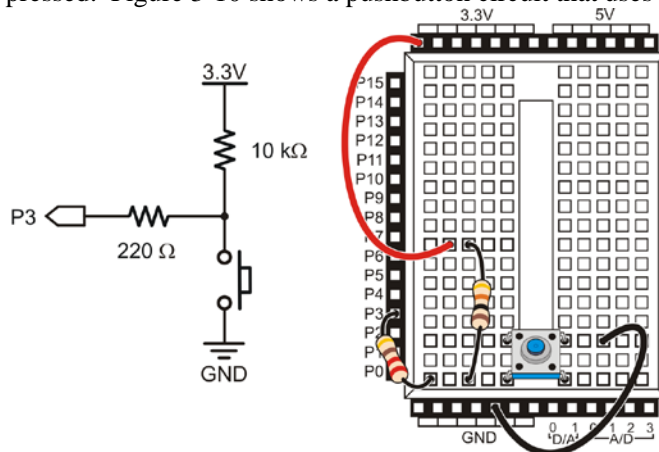


Figure 3-10
Modified Pushbutton
Circuit

This resistor pulls the voltage up to 3.3V (3.3 volts) when the button is not pressed. When the button is pressed, P3 detects GND. So, the rules are now reversed. When the button is *not* pressed, `input(3)` returns the number 1, and when the button *is* pressed, `input(3)` returns the number 0.



The 220 Ω resistor is used in the pushbutton example circuits to protect the Propeller I/O pin. Although it's a good practice for prototyping, in many products this resistor is replaced with a wire (since wires cost less than resistors).

- ❑ Modify your circuit as shown in Figure 3-10.
- ❑ Re-run Button-ReadState with the Run with Terminal button.
- ❑ Use the SimpleIDE Terminal to verify that `input(3)` returns 1 when the button is not pressed and 0 when the button is pressed.



Active-low vs. Active-high: The pushbutton circuit in Figure 3-10 is called *active-low* because it sends the Propeller a low signal (GND) when the button is active (pressed). The pushbutton circuit back in Figure 3-7 is called *active-high* because it sends a high signal (3.3 V) when the button is active (pressed). The “active” direction will always be opposite the “pull” direction.

ACTIVITY #3: PUSHBUTTON CONTROL OF AN LED CIRCUIT

Many devices have pushbuttons to press for changing settings on a device, and LEDs to show you the status of the settings. The example Figure 3-11 shows a zoomed-in view of a pushbutton and LED used to adjust the settings on a computer monitor.



Figure 3-11
Button and LED on a
Computer Monitor

The Propeller microcontroller can be programmed to make decisions based on what it senses. Like Activity #1, this next activity will use a pushbutton to control an LED. However, instead of wiring the pushbutton to directly change the flow of current to the LED, we will connect both the LED and the pushbutton to Propeller I/O pins. Then, your C program can use the input state of the pushbutton's I/O pin in a decision to set the output state of the LED's I/O pin. Using a microcontroller this way opens up many more options than the simple button/LED interaction you've seen so far. The next example program will rapidly flash the LED while the pushbutton is held down — something the Activity #1 circuits could not do.

Pushbutton and LED Circuit Parts

- (1) Pushbutton – normally open
- (1) Resistor – 10 k Ω (brown-black-orange)
- (1) LED – red, yellow, or green, your choice!

- (2) Resistor – 220 Ω (red-red-brown)
- (2) Jumper wires

Building the Pushbutton and LED Circuits

Figure 3-12 shows the active-high pushbutton circuit with pull-down resistor used in the beginning of the last activity, along with the LED circuit from Chapter 2, Activity #2.

- Build the circuits shown in Figure 3-12.

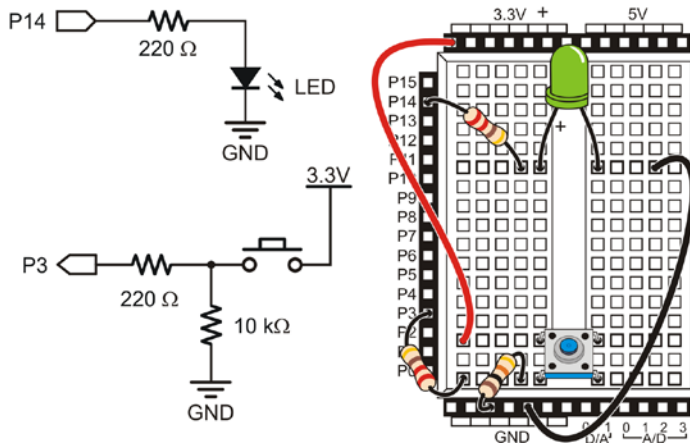


Figure 3-12
Pushbutton and LED
Circuit

Programming Pushbutton Control

The Propeller microcontroller can be programmed to make decisions using an `if...else...` statement. Its syntax, paraphrased, is:

If (condition is true) {execute this code block} else {execute this code block instead}



A **code block** is a group of commands contained by opening and closing braces { }. A code block can be all on one line, or take up multiple lines.

Example Program: Button-ControlOneLED

- Use SimpleIDE's New Project Button to create a new project and name it Button-ControlOneLED.
- Enter the Button-ControlOneLED.c code into SimpleIDE.

- Click the Run with Terminal button.
- Verify that the LED does not flash when you are not pressing the pushbutton.
- Verify that the LED flashes on and off when you hold down the pushbutton.

```

/* Button-ControlOneLED.c */

#include "simpletools.h"

int main()
{
    int button;

    while(1)
    {
        button = input(3);
        print("button = %d\n", button);

        if(button == 1)
        {
            high(14);
            pause(50);
            low(14);
            pause(50);
        }
        else
        {
            pause(100);
        }
    }
}

```

How Button-ControlOneLED Works

This program is a modified version of `Button-ReadState` from the previous activity. Everything stays the same up through `button...` and `print` inside the `while(1)` loop. But then, the `pause(250)` was deleted, and an `if...else...` statement was put in its place. When code execution reaches this spot, it checks to see if `(button == 1)` is actually true. If yes, the statements within its code block braces `{ }` get executed: `high(14); pause(50); low(14);` and another `pause(50);`. As a result, the light blinks on and off very quickly, and then the code execution goes back to the beginning of the `while` loop. So, as long as you are holding the button down, the LED will flash rapidly. If you are not pressing the button, the `if (button == 1)` condition evaluates as false, and the code execution skips down to the `else` block where only `{ pause(100); }` gets executed. There is no code for blinking the LED in the `else` block, so it stays off.



Assign-equals (=) vs. Compare-equals (==)

In the instruction `button = input(3)`, the `=` operator assigns the value on the right to the variable on the left. Another way to say it would be that the `=` operator sets the symbol on the left equal to the value on the right. It is also called the *assignment* operator.

In `if(button == 1)`, the `==` operator performs a comparison to see if the variable on the left is equal to the value on the right. It does not change the value of `button`. It is simply asking a true-or-false question, and will return 1 if it is true, and 0 if it is false. It is also called the *equality* operator.

You can make a detailed list of what a program should do, to either help you plan the program or to describe what it does. This kind of list is called *pseudo code*, and the example below uses pseudo code to describe how Button-ControlOneLED works.

- *Do these commands over and over again*
 - *Copy the 1/0 result of P3 input to a variable named button*
 - *Display the value of button in SimpleIDE Terminal*
 - *If the value of button is 1, Then*
 - *Turn the LED on*
 - *Wait for 50 ms*
 - *Turn the LED off*
 - *Wait for 50 ms*
 - *Else, (if the value of IN3 is not 1)*
 - *Do nothing, but wait for the same amount of time it would have taken to briefly flash the LED (1/10 of a second).*

Your Turn – Alternate Coding Approach

Button-ControlOneLED-YourTurn shows another way to write code that does the same job. Instead of `if...else...`, it just uses an `if...` statement. Also, instead of copying the `input(3)` function's return value to a variable, it uses `input(3)` and the compare-equals operator directly to decide whether or not to turn on the light. If the button is pressed, `if(input(3) == 1 high(14)` turns the light on. If the button is not pressed, it just leaves the light off.

- Use SimpleIDE to create a project named Button-ControlOneLED-YourTurn and try this code. It should do the same job.

```

/* Button-ControlOneLED-YourTurn.c */

#include "simpletools.h"

int main()
{
  while(1)
  {
    print("button = %d\n", input(3));

    if(input(3) == 1)
      high(14);

    pause(50);
    low(14);
    pause(50);
  }
}

```

An **if... statement** is just an **if...else...** statement without the **else**.

If you have just one statement to conditionally execute, you don't need braces { }.

Conditional statements execute the next thing that follows them. That could be a code block contained in braces { }, or it could be a single statement ending with a semicolon ; .

These two pieces of code do the same job: set P14 high if the button is pressed.

SAME

<pre> if(input(3) == 1) high(14); </pre>	<pre> if(input(3) == 1) { high(14); } </pre>
--	--



These next two pieces of code do different jobs. On the left, P14 is set high if the button is pressed. Then, it waits for 50 ms no matter what. The one on the right will do both, but only if the button is pressed, since the **high** and **pause** statements are both inside the braces.

NOT SAME

<pre> if(input(3) == 1) high(14); pause(50) </pre>	<pre> if(input(3) == 1) { high(14); pause(50); } </pre>
---	---

ACTIVITY #4: TWO PUSHBUTTONS CONTROLLING TWO LED CIRCUITS

Now that you know how to use a microcontroller to monitor a pushbutton and control an LED based on the pushbutton's state, let's make things more interesting. In this activity, you will add a second pushbutton and a second LED to the circuits on your breadboard.

Pushbutton and LED Circuit Parts

- (2) Pushbuttons – normally open
- (2) Resistors – 10 k Ω (brown-black-orange)
- (4) Resistors – 220 Ω (red-red-brown)
- (2) LEDs – any color
- (4) Jumper wires (2 red, 2 black)

Adding a Pushbutton and LED Circuit

- Build the circuits shown in Figure 3-13 and Figure 3-14. If you need help building the circuit shown in the schematic, use the wiring diagram in Figure 3-14 as a guide.

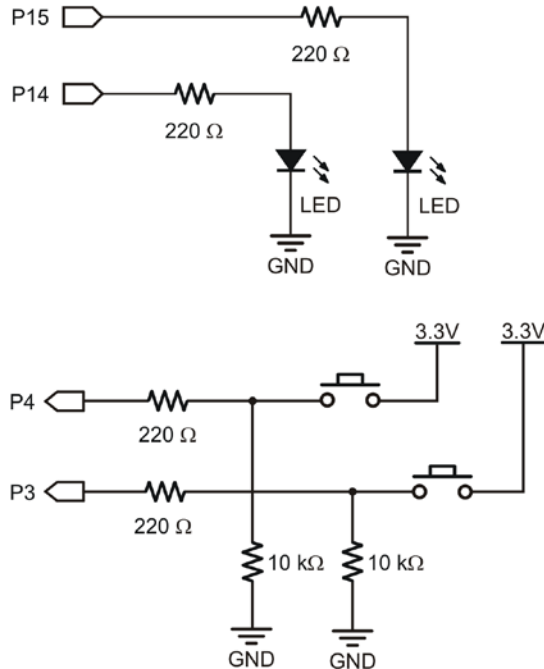


Figure 3-13

Schematic for Two Pushbuttons and LEDs

Dots Indicate Connections

There are three places where lines intersect in Figure 3-13, but only two of those intersections have dots. When two lines intersect with a dot, it means they are electrically connected. When building a circuit on the breadboard, leads connected by a dot are usually in the same 5-socket row.



For example, the 10 k Ω resistor on the lower-right side of Figure 3-14 has one of its terminals connected to one of the P3 circuit's pushbutton terminals and to one of its 220 Ω resistor terminals. When one line crosses another, but there is no dot, it means the two wires DO NOT electrically connect. The line that connects the P4 pushbutton to the 10 k Ω resistor does not connect to the P3 pushbutton circuit because there is no dot at that intersection.

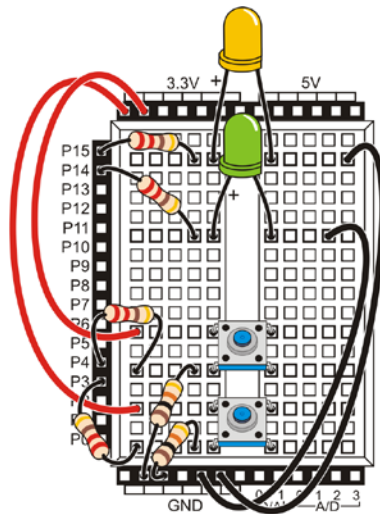


Figure 3-14

Wiring Diagram for Two Pushbuttons and LEDs

- Reopen the Button-ReadState project, and modify it so that it reads `input(4)` instead of `input(3)`, and use it to test your second pushbutton circuit.

Programming Pushbutton Control

In the previous activity, you experimented with making decisions using `if...else...` and `if...` statements. There is also such a thing as an `if...else if...else...` statement. The `else if` part allows you to add additional conditions to test for, if the first `if` condition evaluates to false. It works great for deciding which LED to flash on and off. The next example program shows how it works.

Example Program: Button-ControlTwoLEDs

- Make a new project and name it Button-ControlTwoLEDs.
- Enter Button-ControlTwoLEDs into SimpleIDE.
- Click the Run with Terminal button.
- Verify that the P14 LED flashes while the P3 pushbutton is held down.
- Also verify that the P15 LED flashes while the P4 pushbutton is held down.

```
/* Button-ControlTwoLEDs.c */  
  
#include "simpletools.h"  
  
int main()  
{  
    int button, otherButton;  
  
    while(1)  
    {  
        button = input(3);  
        otherButton = input(4);  
  
        print("%c button = %d, otherButton = %d \n", HOME, button, otherButton);  
  
        if(button == 1)  
        {  
            high(14);  
            pause(50);  
            low(14);  
            pause(50);  
        }  
        else if(otherButton == 1)  
        {  
            high(15);  
            pause(50);  
            low(15);  
            pause(50);  
        }  
        else  
        {  
            pause(100);  
        }  
    }  
}
```

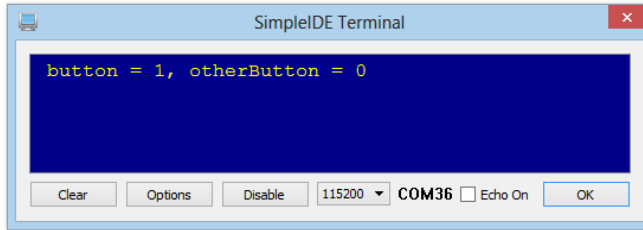


Figure 3-15
SimpleIDE Terminal output of
Button-ControlTwoLEDs

How Button-ControlTwoLEDs Works

The `main` function begins with declaring two `int` variables: `button` and `otherbutton`, just above a `while(1)` loop. All the other instructions in the program are inside this infinite loop.

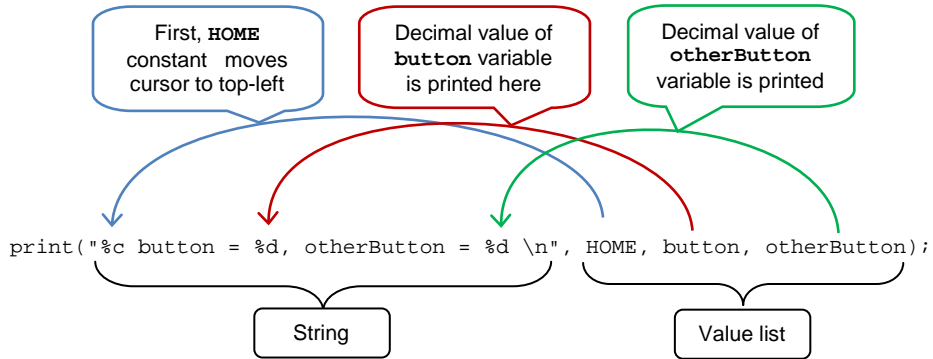
```
int button, otherButton;

while(1)
```

Each time through the `while` loop, the two variables are assigned values from `input` function calls that check P3 and P4.

```
button = input(3);
otherButton = input(4);
```

Next comes an expanded `print` statement, and it's probably the trickiest line to understand. Note that it has three formatting flags in it, denoted by the `%` sign. Each time the `print` statement sees a `%` flag, it displays the next item in the value list that comes after the string between the quotation marks. The first formatting flag is `%c`, which means “display the character without any changes.” The first item in the value list is `HOME`. The `HOME` constant from the `simpletools` library is a value that sends SimpleIDE Terminal's cursor to the top-left home position. This allows this one `print` statement in the `while(1)` loop to keep reprinting in the same place each time through the loop, replacing what was there before with the most up-to date values.



After that, the `print` statement displays the text “button =”. Then, it sees another formatting flag, `%d`, for displaying a decimal integer value. This is the second flag, so the next thing `print` will display is the decimal integer form of the value stored in `button`. Since we know `button` either stores 1 or 0, it’ll display either ‘1’ or ‘0’. Then, the `print` statement displays, “otherButton =”. Finally, the third formatting flag is another `%d`, which prints the value of the `otherButton` variable in decimal integer form too.

After checking the copying the button states to variables and displaying them, the program uses the current values of `button` and `otherButton` to decide what to do. The first part is the same as the previous example program; it blinks the P14 light if the P3 pushbutton is pressed. But, if it’s not pressed, and if the P4 pushbutton is pressed, then the `otherButton == 1` condition will be true, and the P15 light blinks. If neither condition is true, then the `else` condition just pauses for 100 ms (1/10 second) before allowing the `while` loop to repeat.

```
if(button == 1)
{
    high(14);
    pause(50);
    low(14);
    pause(50);
}
else if(otherButton == 1)
{
    high(15);
    pause(50);
    low(15);
}
```

```

    pause(50);
}
else
{
    pause(100);
}

```

Your Turn – What about Pressing Both Pushbuttons?

The example program has a flaw. Try pressing both pushbuttons at once, and you'll see the flaw. You would expect both LEDs to flash on and off, but they don't because only the first code block with a "true" condition in an `if...else if...else...` statement gets executed before the code leaves the decision making process behind and skips to whatever code follows.

What the program needs is an additional condition to test if both buttons are pressed at the same time. Fortunately, it is fair game to have more than one `else if` condition at a time. Let's modify the current program so it has four conditions: `if...else if ... else if ...else`.

- Use the Save Project As button, and rename the project Button-ControlTwoLEDs-YourTurn1.
- Replace this `if` statement and code block:

```

if(button == 1)
{
    high(14);
    pause(50);
    low(14);
    pause(50);
}

```

...with this `if...else if...` statement:

```

if(button == 1 && otherButton == 1)
{
    high(14);
    high(15);
    pause(50);
    low(14);
    low(15);
    pause(50);
}

```

```

}
else if(button == 1)
{
  high(14);
  pause(50);
  low(14);
  pause(50);
}

```

- Run your modified program and see if it handles both pushbutton and LED circuits as you would expect.

Logical AND && and Logical OR ||



The `&&` operator can be used in conditional statements like `if`, `else if`, and `while` to check if more than one condition is true. All conditions with `&&` have to be true for the conditional statement to be true. It is called the *logical-AND* operator.

The `||` operator can also be used in conditional statements, and at least one of the conditions must be true for the conditional statement to evaluate as true. It is called the *logical-OR* operator.

You can also modify the program so that the flashing LED stays on for different amounts of time. For example, you can reduce the value passed to both `pause` function calls' `time` parameters to 25, increase the `pause` for the P14 LED to 100, and increase the `pause` for the P15 LED to 200.

- Use the Save Project As button, and rename the project Button-ControlTwoLEDs-YourTurn2.
- Modify the `pause` commands in the `if` and the two `else if` statements as discussed.
- Run the modified program.
- Observe the difference in the behavior of each light.

A Simplified Approach

In case you're wondering if the Your Turn approach from the previous activity will work here, yes, it will. The code is nice and compact too.

- Use the Save Project As button and rename another copy of the project Button-ControlTwoLEDs-YourTurn3, then try this in place of the `if...else if...else...statement` and code blocks:

```
if(input(3) == 1)
    high(14);

if(input(4) == 1)
    high(15);

pause(50);
low(14);
low(15);
pause(50);
```

- Run the modified code and verify that it works the same.

Why not just use this code and forget about the example in the main activity? Mainly because you'll encounter both when looking at published code solutions for various projects you might work on. So, knowing the rules of how to work with and without braces and using just `if...` or `if...else if...else...` will come in handy.

ACTIVITY #5: REACTION TIMER TEST

Imagine you're an embedded systems engineer at a video game company. The marketing department recommends that the next hand-held game controller should have a circuit and firmware code to test the player's reaction time. Your next task is to develop a proof of concept for the reaction timer test.

The solution you will build and test in this activity is an example of how to solve this problem, but it's definitely not the only solution. Before continuing, take a moment to think about how you would design this reaction timer.

The approach we are taking here is to turn on a bicolor LED, and time how long it takes for the player to release a pushbutton in response to seeing the LED change color.

Reaction Timer Game Parts

- (1) LED – bicolor
- (2) Resistor – 220 Ω (red-red-brown)
- (1) Pushbutton – normally open
- (1) Resistor – 10 k Ω (brown-black-orange)
- (2) Jumper wires

Building the Reaction Timer Circuit

Figure 3-16 shows a schematic and wiring diagram for a circuit that can be used with the Propeller to make a reaction timer game.

- Build the circuit shown in Figure 3-16 on page 89.
- Run LED-TestBicolor from Chapter 2, Activity #5 to test the bicolor LED circuit and make sure your wiring is correct.
- If you just re-built the pushbutton circuit for this activity, run Button-ReadState from Activity #2 in this chapter to make sure your pushbutton is working properly.
- Fix any circuit-building errors your tests uncover before continuing.

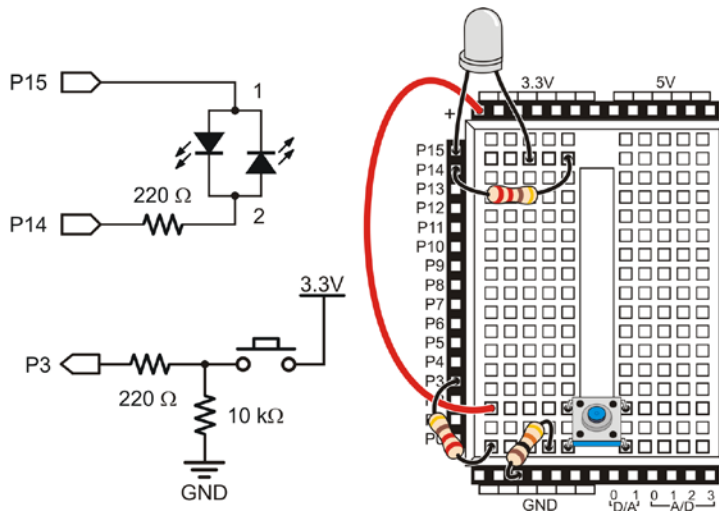


Figure 3-16
Reaction Timer Game Circuit

Programming the Reaction Timer

This next example program will leave the bicolor LED off until the game player presses and holds the pushbutton to start the game. When the pushbutton is held down, the LED will turn red for a short period of time. Then the LED will switch to green, and the player has to let go of the pushbutton as fast as he or she can. The program then measures time it takes the player to release the pushbutton in reaction to the light turning green.

The example program also demonstrates how polling and counting work. *Polling* is the process of checking something over and over again very quickly to see if it has changed. *Counting* is the process of adding a number to a variable each time something does (or does not) happen.

In this program, polling is used twice. Initially, the Propeller polls the pushbutton to see if it has been pressed yet, which starts the game. Then, from the time the bicolor LED turns green the Propeller will start polling the pushbutton again every millisecond (1/1000 of a second) to see if it has been released. Each time it polls and the pushbutton is not yet released, it will add 1 to a counting variable named `timeCounter`. When it senses that the pushbutton is released, the program stops polling and sends a message to the SimpleIDE Terminal that displays the value of the `timeCounter` variable.

Example Program: Button-ReactionTimer

- Enter Button-ReactionTimer into SimpleIDE.
- Click the Run with Terminal button.
- Follow the prompts on the Debug Terminal (see Figure 3-17).

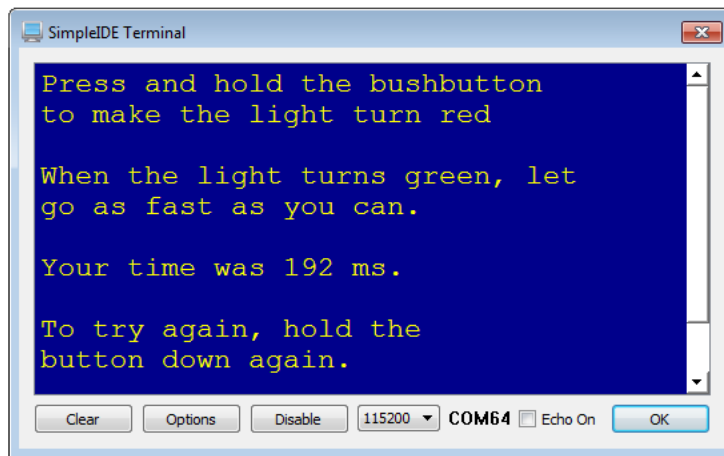


Figure 3-17
Reaction
Timer Game
Instructions in
the Debug
Terminal

```
/* Button-ReactionTimer.c */
#include "simpletools.h"           // Include library
int main()                       // Main function
{
```

```

int timeCounter;

print("Press and hold the pushbutton\n"); // Display instructions
print("to make the light turn red \n\n");
print("When the light turns green, let\n");
print("go as fast as you can.\n\n");

while(1) // Main loop
{
    while(input(3) == 0); // Wait for press

    high(14); // Light red
    low(15);

    pause(1000); // Wait 1 second

    low(14); // Light green
    high(15);

    timeCounter = 0; // timeCounter var -> 0

    do // do
    {
        pause(1); // pause 1 second
        timeCounter++; // Add 1 to timeCounter
    }
    while(input(3) == 1); // ...while pressed

    low(15); // Turn light off

    print("Your time was %d ms. \n\n", // Display timecounter
          timeCounter);
    print("To try again, hold the\n"); // Repeat instructions
    print("button down again.\n\n");
}
}

```

How Button-ReactionTimer Works

Inside `main`, the program declares the `int timeCounter` variable, then makes `print` function calls that display instructions for the player.

```

int timeCounter;

print("Press and hold the pushbutton\n");
print("to make the light turn red \n\n");
print("When the light turns green, let\n");
print("go as fast as you can.\n\n");

```

Next comes nested `while` loops, one `while` loop is inside of the other. The outer `while(1)` is followed immediately with `while(input(3) == 0)`. Notice that this inner, conditional loop actually has *no* code block below it. This translates to “while the P3 pushbutton is not pressed, do nothing.” This coding technique of an empty, conditional `while` loop is very handy when you want program execution to simply pause for an unknown period of time, polling until some external input occurs.

```
while(1)
{
    while(input(3) == 0);
```

When the player presses and holds down the pushbutton to start the game, `while(input(3) == 0)` finally evaluates to false, and code execution resumes to the lines below.

Now that the player has pressed the button, the bicolor LED turns red with the now-familiar `high(14); low(15)` function calls. A `pause(1000)` keeps the LED red for 1 second, and then `low(14); high(15)` reverses the current to turn the LED green.

```
high(14);                // Light red
low(15);

pause(1000);             // Wait 1 second

low(14);                // Light green
high(15);
```

As soon as the bicolor LED turns green, it's time to start polling the pushbutton again, and counting the milliseconds until the player releases it. The next line sets `timeCounter` equal to 0.

```
timeCounter = 0;        // timeCounter var -> 0
```

Immediately after that, a `do...while(condition)` loop starts repeating itself. A `do...while(condition)` loop is like a `while` loop, but the condition test comes after the code block instead of before it to make sure the code inside the block gets executed at least once. As long as `while(input(3) == 1)` evaluates as true, meaning the player is still holding down the pushbutton, the loop will repeat and add 1 to the `timeCounter`

variable with the `++` operator. The `pause(1)` makes the loop repeat about 1000 times per second until the `do...while` loop evaluates as false when the pushbutton is released.

```
do
{
    pause(1);
    timeCounter++;
}
while(input(3) == 1);
```

When the player releases the pushbutton, the code exits the `do...while` loop and the next line of code turns the bicolor LED off.

```
low(15)
```

The game is now over, but the program is not. The next three lines are `print` statements to display in the SimpleIDE Terminal. The first one displays the value of `timeCounter`, so the player can see his or her reaction time in milliseconds. The next two `print` statements invite the player to try again, and give instructions to press the button.

```
print("Your time was %d ms. \n\n",
      timeCounter);
print("To try again, hold the\n");
print("button down again.\n\n");
```

After this, the code reaches the outer `while(1)` loop's closing brace `}`. So, code execution returns the first instruction inside the `while(1)` loop's opening brace `{`. That puts us back at `while(input(3) == 1)`, polling the pushbutton to see if anyone has pressed it to start a new game.

Your Turn

Imagine now that the marketing department gave your prototype to some game testers. When the game testers were done, the marketing department came back to you with details about two problems that have to be fixed before your prototype can be built into the game controller. One can be fixed with code you already know. The other is trickier, and we've introduced some new concepts to solve it.

Problem 1: A player that lets go of the button before the light turns green gets an unreasonably good score (1 ms, since the code in the `do...while` loop gets executed at least once). Your microcontroller needs to figure out if a player is cheating.

This can be fixed with code that checks the value of `timeCounter` to decide whether to display the result or tell the user to try again and wait for the light to change before letting go of the button. Here's an example of some pseudo-code that describes it.

- *If the value of `timeCounter` is greater than 1 (`timeCounter > 1`)*
 - *Display the value of `timeCounter` in ms (just like in `Button-ReactionTimer`)*
- *Else, (if the value of `timeCounter` is 1 or less)*
 - *Display a message telling the player he or she has to wait until after the light turns green to let go of the button.*
- *Display a "To play again..." message. (Unchanged from what's in `Button-ReactionTimer`)*

- Before continuing, stop and consider how you would write that code.
- Save a copy as `Button-ReactionTimer-YourTurn1`. Then try modifying the code, following the pseudo-code, to fix Problem 1.

Here is an `if...else...` solution to the pseudo-code:

```

if(timeCounter > 1)           // <- add
{                               // <- add
    print("Your time was %d ms. \n\n",
          timeCounter);
}                               // <- add
else                           // <- add
{                               // <- add
    print("Wait for the light to change \n"); // <- add
    print("before letting go of the button. \n"); // <- add
    print("Try again. \n\n"); // <- add
}                               // <- add
print("To try again, hold the\n");
print("button down again.\n\n");

```

- Try it!

Optional Tricky Topic – Pseudo-random Number, Scale, and Offset

Problem 2: Players soon figure out that the delay from red to green is 1 second. After playing it several times, they get better at predicting when to let go, and their score no longer reflects their true reaction time.

One solution would be to make the LED stay red for a random number of milliseconds between 500 and 1500 ms, instead of always pausing for 1 second. But how? Fortunately, there is a function named `rand()` in the Standard C Library (called `stdlib`) which is already included by the `simpletools` library.

The `rand` function returns a different pseudo-random number each time you call it. You can use a couple of math tricks to scale down and offset the value returned by `rand` into the desired range. Then, you can assign this scaled & offset value to a variable, let's call it `randomVal`, and then use `pause(randomVal)` in place of `pause(1000)` in the program. This will cause the LED to stay red for a different length of time each game, within the time range you desire. Let's try it first, and then look closer at how it works:

- Save a new copy of `Button-ReactionTimer` as `Button-ReactionTimer-YourTurn2`.
- Add a second `int` variable named `randomVal`.
- Then, replace `pause(1000)` with this:

```
randomVal = 500 + rand() % 1001;
print("randomVal = %d \n", randomVal);
pause(randomVal);
```

- Try the button game a few times, and notice that `randomVal`, indicating the red LED time, changes for each game.

How it Works

The line `randomVal = 500 + rand() % 1001` takes care of the hard part, which is generating a value in the 500 to 1500 range using a trick with modulus operator `%` and the addition operator `+`. There's a lot going on in that one line of code, so let's dissect it.

- (1) First, `rand()` gets a pseudo-random value, which will be in the range of 0 to 2,147,483,647. (We can't use this value range with `pause` as-is; it could make the LED red for more than 24 days!!)
- (2) The operation `% 1001` gives you the remainder of `rand() ÷ 1001`. This will always be a number in the 0 to 1000 range, effectively scaling down the value returned by `rand`.
- (3) `500 +` offsets the scaled value range from 0-1000 to 500-1000.
- (4) `=` copies the scaled and offset result to the `randomVal` variable.

After that, `print("randomVal = %d \n", randomVal)` displays the value, and then `pause(randomVal)` pauses for that random amount of time that's somewhere in the 500 to 1500 ms range. It will be different each time.



What's an algorithm? An *algorithm* is a sequence of mathematical operations.

What's pseudo-random? *Pseudo-random* means that it seems random, but it isn't really. Each time you start the program over again, you will get the same sequence of values. An algorithm is used to create the sequence.

What's a seed? A *seed* is a value that is used to start the pseudo-random sequence. You always get the same sequence of random numbers when using the same seed value. The default seed value is 1. To change the seed value, to 23 for example, you would add `srand(23)` before the `while(1)` loop starts.

SUMMARY

This chapter introduced a new electrical component and many new circuit-building activities, programming concepts, and C language elements:

- Introduced the normally open pushbutton and its schematic symbol.
- What dots at line intersections mean in a circuit schematic.
- How to build and test active-high and active-low pushbutton circuits, using pull-down and pull-up resistors.
- How to control an LED directly with a pushbutton.
- How to use a microcontroller I/O pin as an input to monitor the state of a pushbutton circuit.
- How to use the `input()` function to monitor the state of an I/O pin.
- C operators And `&&`, OR `||`, Modulus `%`, Assign-equals `=`, Compare-equals `==`.
- Decision-making with `if` and `if...else if...else` conditional statements.

- Polling with conditional `while` and `do...while` statements.
- What the `%c` and `%d` formatting flags do in a `print` statement.
- What the `simpletools` library's `HOME` constant does in a `print` statement.
- Generating pseudo-random numbers with the `rand()` function, and scaling and offsetting the return value to the desired range.
- Building a reaction-timer game.

Questions

1. What is the difference between sending and receiving `high` and `low` signals with the Propeller?
2. What does “normally open” mean in regards to a pushbutton?
3. What happens between the terminals of a normally open pushbutton when you press it?
4. What value does `input(3)` return when a pushbutton connects it to 3.3 V? What value does `input(3)` return when a pushbutton connects it to GND?
5. What does `int button = input(3); print("button = %d", button); do?`
6. What kind of statements will conditionally execute blocks of code based on conditions?
7. What does the `HOME` control character do in the statement `print("%c button = %d", HOME, button)?`

Exercises

1. Explain how to modify `Button-ReadState` on page 74 so that it reads the pushbutton every second instead of every $\frac{1}{4}$ second.
2. Explain how to modify `Button-ReadState` so that it reads a normally open pushbutton circuit with a pull-up resistor connected to I/O pin P6.

Project

1. Modify `Button-ReactionTimer` so that it is a two-player game. Add a second button wired to P4 for the second player.

Solutions

- Q1. Sending uses the Propeller I/O pin as an output, whereas receiving uses the I/O pin as an input.

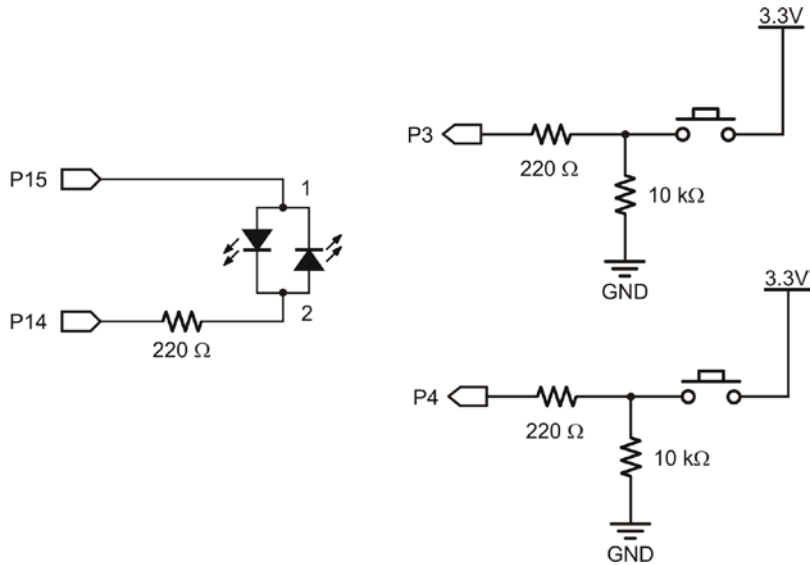
- Q2. Normally open means the pushbutton's normal state (not pressed) forms an open circuit.
- Q3. When pressed, the gap between the terminals is bridged by a conductive metal. Current can then flow through the pushbutton.
- Q4. `input(3) == 1` when pushbutton connects it to 3.3 V; `input(3) == 0` when pushbutton connects it to GND.
- Q5. `int button = input(3)` gets declares an `int` variable named `button`, and then copies the I/O value `input(3)` returns to it. `print("button = %d", button);` displays “button = ” followed by the characters that describe the decimal integer value of `button`. If the button is pressed, it will display “button = 1”. If the button is not pressed, it will display “button = 0”.
- Q6. Conditional statements like `if....`, `if...else....`, `if...else if...else....`, `while`, and `do...while`.
- Q7. The **HOME** control character sends the cursor to the top left position in the SimpleIDE Terminal.
- E1. The `while(1)` loop in the program repeats every $\frac{1}{4}$ second because of the `pause(250)` call. To repeat every second, change the `pause(250)` (250 ms = 0.25 s = $\frac{1}{4}$ s), to `pause(1000)` (1000 ms = 1 s).

```
while(1)
{
    button = input(3);
    print("button = %d\n", button);
    pause(1000); // <- Change this
}
```

- E2. Replace `input(3)` with `input(6)`, to read I/O pin P6. The program only displays the pushbutton state, and does not use the value to make decisions; it does not matter whether the resistor is a pull-up or a pull-down. The `print` call will display the button state either way.

```
while(1)
{
    button = input(6); // <- Change this
    print("button = %d\n", button);
    pause(1000);
}
```

P1. First, add a button for the second player, wired to Propeller I/O pin P4. The schematic is based on Figure 3-16 on page 89.



Snippets from the solution program are included below, but keep in mind that solutions may be coded a variety of different ways. However, most solutions will include the following modifications:

Use two variables to keep track of two players' times:

```
int timeCounterA, timeCounterB;
```

Change instructions to reflect two pushbuttons:

```
print("Press and hold the pushbuttons\n");
print("to make the light turn red \n\n");
```

Wait for both buttons to be pressed before turning LED red, by using the **OR** operator:

```
while(input(3) == 0 || input(4) == 0);
```

Make sure both players' time counters are set to zero:

```
timeCounterA = 0;
timeCounterB = 0;
```

Add logic to decide which player's time is incremented:

```
pause(1);

if(input(3) == 1)
{
    timeCounterA++;
}
if(input(4) == 1)
{
    timeCounterB++;
}
```

Wait for both buttons to be released to end timing, again using the **OR** operator:

```
while(input(3) == 1 || input(4) == 1);    // ...while pressed
```

Change time display to show times of both players:

```
print("Player A Time: %d \n",
      timeCounterA);
print("Player B Time: %d \n",
      timeCounterB);
```

Add logic to show which player had the faster reaction time:

```
if(timeCounterA < timeCounterB)
{
    print("Player A is the winner!\n");
}
else if(timeCounterA > timeCounterB)
{
    print("Player B is the winner!\n");
}
else
{
    print("It's a tie!\n");
}
```

The complete solution is shown below.

```

/* Button-P1-Solution.c */
#include "simpletools.h"           // Include library
int main()                       // Main function
{
    int timeCounterA, timeCounterB; // counter for each player

    print("Press and hold the pushbuttons\n"); // Display instructions
    print("to make the light turn red \n\n");
    print("When the light turns green, let\n");
    print("go as fast as you can.\n\n");

    while(1)                      // Main loop
    {
        while(input(3) == 0 || input(4) == 0); // Wait for both to press

        high(14);                 // Light red
        low(15);

        int delay = 500;
        delay += rand()%1500;
        pause(delay);             // Wait 1 second
        // print("delay = %d\n", delay);

        low(14);                 // Light green
        high(15);

        timeCounterA = 0;        // timeCounterA var -> 0
        timeCounterB = 0;        // timeCounterB var -> 0

        do                       // do
        {
            pause(1);            // pause 1 second

            if(input(3) == 1)
            {
                timeCounterA++;   // Add 1 to timeCounter
            }
            if(input(4) == 1)
            {
                timeCounterB++;   // Add 1 to timeCounter
            }
        }
        while(input(3) == 1 || input(4) == 1); // ...while either pressed

        low(15);                 // Turn light off
    }
}

```

```
print("Player A Time: %d \n",           // Display timecounterA
      timeCounterA);
print("Player B Time: %d \n",         // Display timecounterB
      timeCounterB);

if(timeCounterA < timeCounterB)      // Decide who wins
{                                     // ...and display
    print("Player A is the winner!\n");
}
else if(timeCounterA > timeCounterB)
{
    print("Player B is the winner!\n");
}
else
{
    print("It's a tie!\n");
}

print("To try again, hold the\n");    // Repeat instructions
print("button down again.\n\n");
}
```

Chapter 4: Control Position and Motion

MICROCONTROLLED MOTION

Microcontrollers move mechanical devices in objects you see every day. If you have an inkjet printer, the print head is swept across the page by stepper motor controlled by a microcontroller. The automatic doors at a store are controlled by microcontrollers, and the auto-eject feature in your DVD player is also controlled by a microcontroller. Look around you now — can you spot more examples?

Just about all microcontrolled motors receive sequences of high and low signals that resemble the ones you've been sending to LEDs. The microcontroller has to send these signals much faster, sometimes so fast that the human eye cannot detect the switching.

Some motors require lots of external circuitry in addition to a microcontroller. Others require extra mechanical parts to fit into machinery. The hobby servo that you will experiment with in this chapter is simplest, as it requires neither.

INTRODUCING THE SERVO

A standard hobby servo is a device that controls position. You can find them in just about any radio controlled (RC) car, boat or plane. In RC cars, the servo holds the steering to control turn radius. In an RC boat, it holds the rudder in position for turning in the water. RC planes may have several servos to position the different wing and tail flaps that control the plane's trajectory. In RC vehicles with gas powered engines, a servo moves the engine's throttle lever to control how fast the engine runs.

An example of an RC airplane and its radio controller are shown in Figure 4-1. The hobbyist “flies” the airplane by manipulating thumb joysticks on the radio controller, which causes the servos on the plane to control the positions of the RC plane's elevator flaps and rudder.



Figure 4-1
Model Airplane and
Radio Controller

So, how does this work? The radio controller converts the position of the joysticks into pulses of radio activity. The time each pulse lasts indicates the position of one of the joysticks. On the RC plane, a radio receiver converts these radio activity pulses to digital pulses (high/low signals) and sends them to the plane's servos. Each servo has circuitry inside it that converts these digital pulses to a position that the servo maintains. The amount of time each pulse lasts is what tells the servo what position to maintain. These control pulses only last a few thousandths of a second, and repeat around 40 to 50 times per second to make the servo maintain the position it holds. Between pulses, the servo will hold its position against any outside force that tries to turn it.

Figure 4-2 shows a drawing of a Parallax Standard Servo. The plug (1) is used to connect the servo to a power source (5 V and GND) and a signal source (a Propeller I/O pin). The cable (2) has three wires, and it conducts 5 V, GND and the signal line (white) from the plug into the servo. The horn (3) is the part of the servo that looks like a four-pointed star. When the servo is running, the output shaft turns and holds the horn in different positions. The Phillips screw (4) attaches the horn to the servo's output shaft. The case (5) contains the servo's position-sensing and control circuits, a DC motor, and gears. These parts work together to take high/low signals from the Propeller and translate them into positions held by the servo horn.

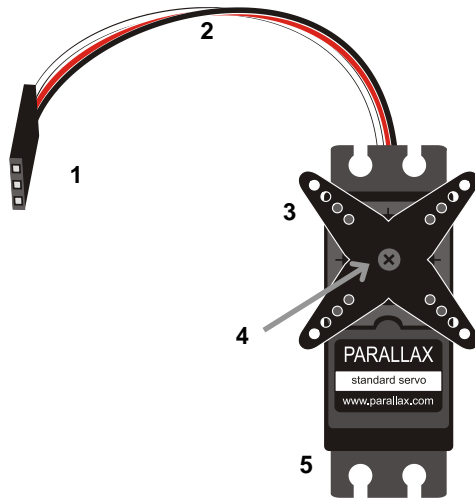
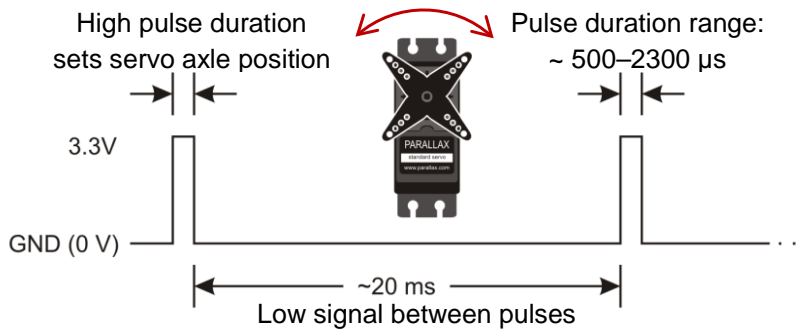


Figure 4-2
The Parallax Standard Servo

- (1) Plug
- (2) Cable
- (3) Screw attaching horn to output shaft
- (4) Screw that attaches the horn to the servo's output shaft
- (5) Case with mounting holes

To turn to and hold a specific position, the servo needs to receive a short high-signal pulse. The duration of the high pulse, between 500–2300 μs determines which position the output shaft will turn to. The high pulse must be repeated every 20 ms for the servo to maintain the position against outside forces.

Figure 4-3: Parallax Standard Servo Control Signal



What's a μs ? This is the standard abbreviation for a *microsecond*, which is a millionth of a second. μ is the Greek symbol Mu (typically pronounced mew).

ACTIVITY #1: SAFELY CONNECTING THE SERVO

Up to now, our Activity Board and breadboard circuits got all the power they needed from the computer's USB port. However, a servo may draw more power than a USB port can provide, and the Activity Board will shut itself down to prevent that from happening.

In this activity, you will connect your Activity Board to an appropriate external power supply, and then connect the servo to the board. You will still keep the USB connection for programming and communication, though we won't be writing programs to control until the next activity.



STOP: Before starting these activities, get an approved power supply with a 2.1 mm, center positive plug. It should either be a 4 or 5 AA cell supply or 6-9 regulated VDC, 800 mA (min) wall mount.

CAUTION! If you get a wall mount supply (other than Parallax part #750-00009 shown below), make sure to test the output with a voltmeter to verify that its actual output matches its rated range. If its actual voltage is outside the 6-9 V range, don't use it.

- Obtain one of the external power supply options shown below.
- Don't plug it in yet!

Figure 4-4: Appropriate Power Supply Options

Use one of the power supplies shown below available from www.parallax.com, or an equivalent.



4-cell pack (#700-00038)
plus four 1.5 V AA
batteries



5-cell pack (#753-00007) plus five 1.2
V rechargeable or 1.5 V AA batteries



7.5 V regulated, 1 A
wall-mount supply
with 2.1 mm center-
positive plug
(#750-00009)

Servo and LED Circuit Parts

- (1) Parallax Standard Servo
- (1) Resistor – 220 Ω (red-red-brown)
- (1) LED – any color
- (1) 2.1 mm, center positive plug supply option from Figure 4-4.
- (1) Jumper wire (black)
- #1 Phillips-tip screwdriver

The LED circuit is not required to help the servo operate. It is just there to help you “see” the control signals.



Use only a Parallax Standard Servo for the activities in this text! Other servos may be designed to different specifications that might not be compatible with these activities.

Building the Servo and LED Circuits

These instructions are for all Propeller Activity Board Revisions.

- Turn off the power as shown in Figure 4-5.

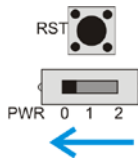


Figure 4-5

Disconnect Power

Set 3-position switch to 0

Figure 4-6 shows the servo header on the Propeller Activity Board. Each larger set of 3 pins is a servo port, with a Propeller I/O pin connection on top, a V+ connection in the middle, and a GND connection at the bottom.

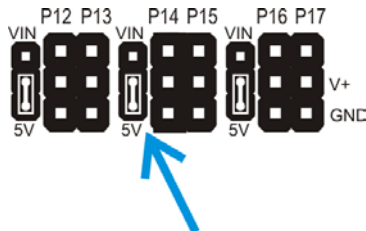


Figure 4-6

Servo Header Jumpers Set to 5V

Each pair of servo ports has a smaller 3-pin power header to its left, along with a removable plastic jumper. You can set the jumper to connect the middle pin to the VIN pin on top, or the 5V pin on the bottom. Whichever one the jumper connects the middle pin to, that will be the power supplied to the V+ pins for the two servo ports to its right.



Setting the jumper to 5V protects your Parallax Standard Servo. The Parallax Standard Servo accepts 4–6 VDC. By setting the power supply to 5V, your servo is protected from excess voltage, no matter which of the three power supply options you chose above.

- ❑ Verify that the jumper is set to 5V as shown in Figure 4-6. If it is instead set to VIN, lift the rectangular jumper up off of the pins it is currently on, and then press it on the two pins closest to the 5V label.

Figure 4-7 shows the schematic of the circuit you will build on your Activity Board.

- ❑ Build the circuit shown in Figure 4-7 and Figure 4-8.
- ❑ Make sure you did not plug the servo in upside-down. The white, red and black wires should line up as shown in Figure 4-8, with the white wire near the top edge of the board.

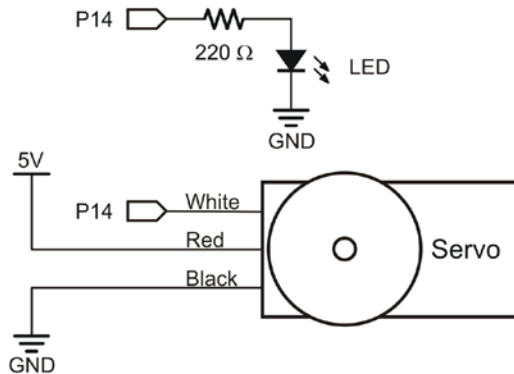


Figure 4-7
Servo and LED Indicator
Schematic for Propeller
Activity Board

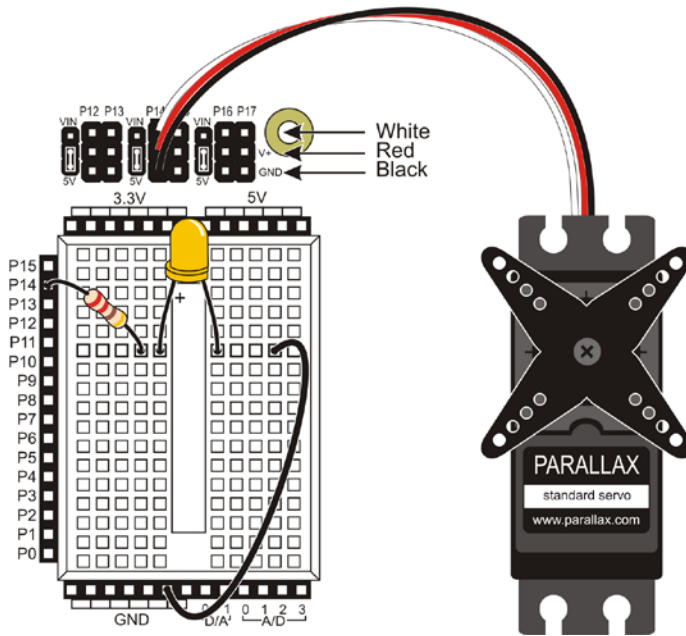


Figure 4-8
Servo and LED Indicator
on Activity Board

You are connecting two devices to one Propeller I/O Pin!

Notice that there are sockets labeled P12–P15 along the left edge of the breadboard, and there are also servo ports labeled P12–P15. A socket and port pin with the same number are connected to the same Propeller I/O pin.

Here, we are taking advantage of that to “see” the high and low signals being sent to the servo. But be careful! In the future, make sure that you connect only one device to an I/O pin at a time unless you are absolutely sure they are fully compatible in your application circuit and program.

- ❑ Connect your external supply to the Activity Board as shown in Figure 4-9.



Figure 4-9
Connect External Supply

When you set the PWR switch to 1 it powers the entire board except for the servo ports. Setting it to 2 also supplies power to the servo headers.

- ❑ Supply power to the board and servo header by setting the PWR switch to 2 as shown in Figure 4-10. Your servo may move a bit when you connect the power. If the board had a P14 blinking light program running from an earlier activity, the servo may also twitch when the light turns on/off. **If the servo "chatters", immediately turn off the power switch.**

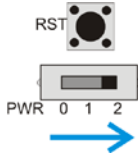


Figure 4-10

Power Turned on to Activity Board and Servo Header

Throughout the rest of the tutorial, if you see instructions that read “Connect power to your board” move the PWR switch to position-2 if you are using the servo. Likewise, if you see instructions in this chapter that read “Disconnect power from your board” move the 3-position switch to position-0.

- ❑ Disconnect power from your board.

ACTIVITY #2: TEST AND ADJUST RANGE OF MOTION

A *degree* is an angle measurement denoted by the $^{\circ}$ symbol. Examples are shown in Figure 4-11 for 30° , 45° , 90° , 135° , and 180° . Each degree of angle measurement represents $1/360^{\text{th}}$ of a circle, so the 90° measurement is $1/4$ of a circle since $90 \div 360 = 1/4$. Likewise, 180° is $1/2$ of a circle since $180 \div 360 = 1/2$, and you can calculate similar fractions for the other degree measurements in the figure.

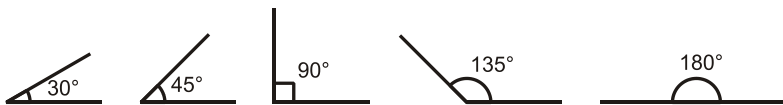
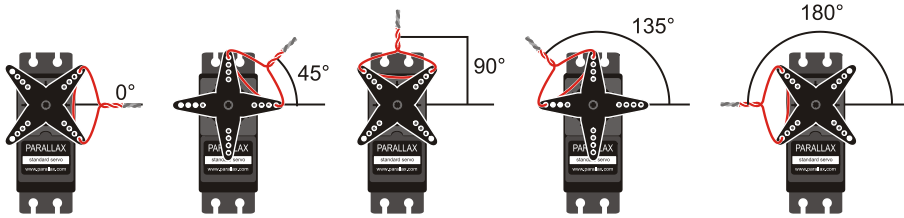


Figure 4-11

Examples of Degree Angle Measurements

The Parallax Standard Servo can make its horn hold positions anywhere within a 180° range. Figure 4-12 shows examples of a servo with a loop of wire that has been threaded through two of the holes in its horn and then twist-tied. The direction the twist tie points indicates the angle of the servo's horn, and the figure shows examples of 0°, 45°, 90°, 135°, and 180°.

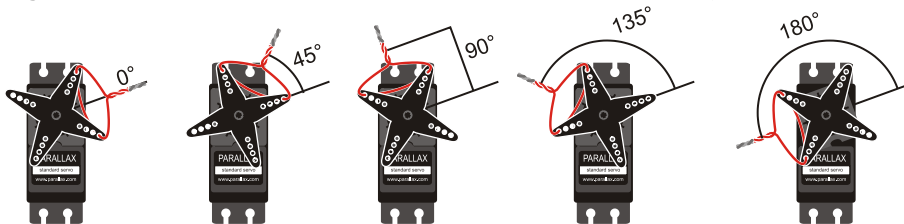
Figure 4-12: Servo Horn Position Examples



Your servo horn's range of motion and mechanical limits will probably be different from what's shown here. Instructions on how to adjust it to match this figure come after the first example program.

In the factory, servo horn mounting can be somewhat random, so your servo horn positions will probably be different from the ones in Figure 4-12. In fact, compared to Figure 4-12, your servo's horn could be mounted anywhere in a +/- 45° range. The servo in Figure 4-13 shows an example of a servo whose horn was mounted 20° clockwise from the one in Figure 4-12. After you find the center of the servo horn's range of motion, you can either use it as a 90° reference or mechanically adjust the servo's horn so that it matches Figure 4-12 by following instructions later in this activity.

Figure 4-13: Servo Horn Position Examples before Mechanical Adjustment



This is an example of a horn that's mounted on the servo's output shaft about 20° counterclockwise of how it was set in Figure 4-12.



In these next steps, twist the servo horn slowly and do not force it! The servo has built-in mechanical limits to prevent the horn from rotating outside its 180° range of motion. Twist the horn gently, and you'll be able to feel when it reaches one of its mechanical limits. Don't try to force it beyond those limits because it could strip the gears inside the servo.

You can find the center of the servo's range of motion by gently rotating the horn to find its clockwise and counterclockwise mechanical limits. The half way position between these two limits is the center or 90° position. The servo's center position could fall anywhere in the region shown in Figure 4-14.

The center of your servo horn's range of motion should fall somewhere in this region

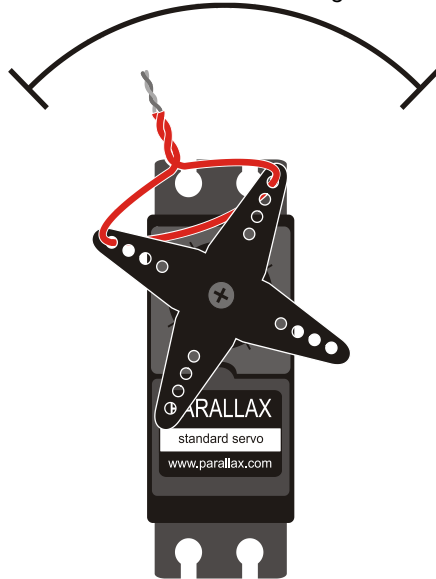


Figure 4-14
Range of Possible
Center Positions

- Verify that the power to your board is still disconnected.
- Gently rotate the servo horn to find the servo's clockwise and counterclockwise mechanical limits. The servo's horn will turn with very little twisting force until you reach these limits. **DO NOT TRY TO TWIST THE HORN PAST THESE LIMITS**; only twist it far enough to find them.
- Rotate the servo's horn so that it is half way between the two limits. This is approximately the servo's "center" position.

- ❑ With the servo horn in its center position, thread a jumper wire through the horn and twist tie it so that it points upward into the region shown in Figure 4-14.

Keep in mind the direction the twist tie is pointing in the figure is just an example; yours might point anywhere in the region. Wherever it points when it's in the center of its range of motion should be pretty close to the servo's 90° position. Again, this position can vary from one servo to the next.

Test and Adjust the Servo's 90° "Center" Position

The servo's 90° position is called its *center position* because the 90° point is in the "center" of the servo's 180° range of motion. You can use a screwdriver to remove and reposition the horn so that 90° makes the jumper wire twist tie point straight up — Instructions for this are coming up in the Your Turn section. But first, let's find what your servo's actual center position is:

- ❑ Gently turn the servo to its clockwise limit.
- ❑ Click SimpleIDE's New Project button and name your project Servo-Center.
- ❑ Enter Servo-Center.c into SimpleIDE.
- ❑ Connect power (plug in external power and set PWR switch to 2).
- ❑ Click SimpleIDE's Load EEPROM button.
- ❑ The P14 LED should glow dimly indicating that the Propeller is sending servo control signals over its P14 I/O pin.
- ❑ The servo should automatically turn to its center position, and hold there indefinitely. We can use this position to fine-tune the servo horn.

Example Program: Servo-Center

```

/* Servo-Center.c */

#include "simpletools.h"           // Include simpletools header
#include "servo.h"               // Include servo header

int main()                       // main function
{
    servo_angle(14, 900);        // P14 servo to 90 degrees
}

```

As soon as the program loads, the P14 LED should glow dimly, indicating that the Propeller is transmitting the servo signal to P14. The servo's horn should rotate to its center position and stay there. The servo "holds" this position, because standard servos

are designed to resist external forces that push against it. That's how the servo holds the RC car steering, boat rudder, or airplane control flap in place.

- Make a note of your servo's center position.
- Apply **very gentle** twisting pressure to the horn like you did while rotating the servo to find its mechanical limits, just enough to feel its resistance to the force you are applying.

If you disconnect power, you can rotate the servo away from its center position. When you reconnect power, the program will restart, and servo will immediately move the horn back to its center position and hold it there.

- Try it disconnected!

How it Works – Servo-Center.c

There's a Simple Library named `servo` with functions for setting a servo's position. For access to its functions, just add `#include "servo.h"` to your program. It's best to keep all the `#include` statements together, so we added it right below `#include "simpletools.h"`.

```
#include "servo.h"
```



For a full list of servo functions: Check the Documentation `servo Library.html` page. It's in `...Documents\SimpleIDE\Learn\Simple Libraries\Motor\Servo\`. You can also access it from SimpleIDE by clicking Help and selecting Simple Library Reference. Then, find and click the `servo.h` link.

Since the program has `#include "servo.h"`, it has access to all the servo library's functions. This program uses a function named `servo_angle` to set the servo to its center 90-degree position. This function has two parameters, `pin` and `degreeTenths`. The statement `servo_angle(14, 900)` makes the Propeller send signals for a servo connected to P14 to hold the 90-degree position.

```
int main()
{
    servo_angle(14, 900);
}
```



The servo library automatically launches its servo control code into another Propeller cog (processor) as soon as your code makes its first `servo_angle` call. The code running in the other cog sends a constant stream of signals to the servo that make it hold its position. That's why the servo continues to hold its position after the `main` program ends.

Your Turn – Adjust Servo Horn to 90° Center

Next, adjust your servo's horn so that it makes the jumper wire twist-tie point straight up when `Servo-Center.c` is running, like it does in the right side of Figure 4-15. This mechanical adjustment will simplify tracking the servo's angles because each angle will resemble the ones in Figure 4-12 on page 111.



You will need a #1 Phillips screwdriver for this adjustment.

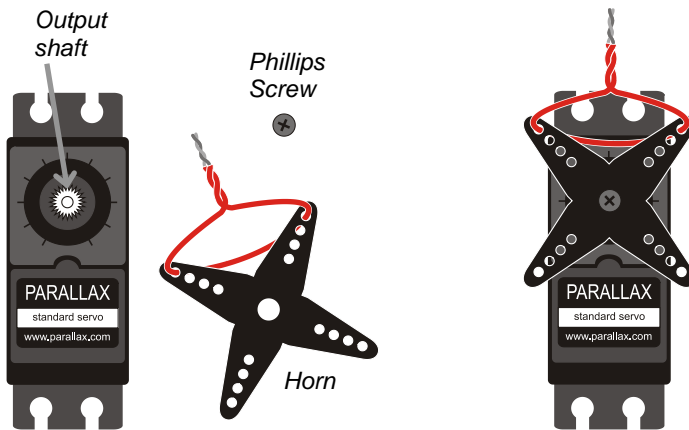


Figure 4-15
Mechanical Servo Centering

You can remove and reposition the servo horn on the output shaft with a small screwdriver.

- Disconnect power from your board.
- Remove the screw that attaches the servo's horn to its output shaft, and then gently pull the horn away from the case to free it. Your parts should resemble the left side of Figure 4-15.
- Reconnect power to your board to run the `Servo-Center.c` program. The program should make the servo hold its output shaft in the center position.

- Slip the horn back onto the servo's output shaft so that it makes the twist-tied wire point straight up like it does on the right side of Figure 4-15.



Alignment Offset: It might not be possible to get it to line up perfectly because of the way the horn fits onto the output shaft, but it should be close. You can then adjust the wire loop to compensate for this small offset and make the twist tie point straight up.

- Disconnect power from your board.
- Gently re-tighten the Phillips screw. It only has to be firm, not tight. The ridges of the servo shaft lock the rotation force. The screw only holds the horn down onto those ridges. Over-tightening will strip the threads and leave the servo useless.
- Reconnect power so that the program makes the servo hold its center position again. The twist tie should now point straight up (or almost straight up) indicating the 90° position.

ACTIVITY #3: PROGRAM TO HOLD POSITIONS

Animatronics uses electronics to animate props and special effects, and servos are a common tool in this field. Figure 4-16 shows an example of a robotic hand animatronics project, with servos controlling each finger. The program that controls the hand gestures has to make the servos hold positions for certain amounts of time for each gesture. In the previous activity, our programs made the servo hold certain positions indefinitely. This activity introduces how to write code that makes the servo hold certain positions for certain amounts of time.



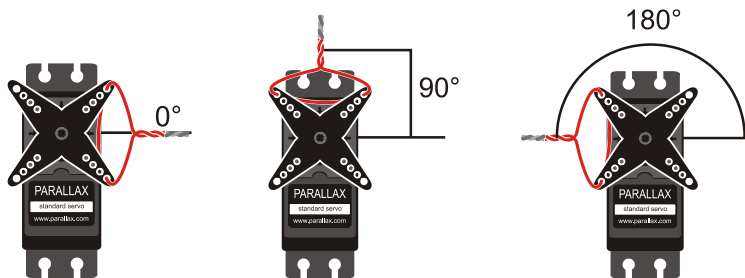
Figure 4-16
Animatronic Hand

Five servos pull bicycle brake cables threaded through the fingers and thumb to make them flex.

This gives the microcontrollers control over each finger.

Let's try a program that sets the servos to the three different positions shown in Figure 4-17. The program will hold each position for 2.5 seconds. It may be difficult to see, but make sure to check the P14 signal LED's brightness with each position. It should be just a little dimmer at 0 degrees, and slightly brighter the further counterclockwise it turns.

Figure 4-17: Servo Test Positions



- Click SimpleIDE's New Project button and name your project Servo-Positions.
- Enter Servo-Positions.c into SimpleIDE.
- Connect power (plug in external power and set PWR switch to 2).
- Click SimpleIDE's Load EEPROM & Run button.
- Verify that the servo holds three positions that are about 90-degrees apart.

- ❑ Verify that the P14 servo signal indicator LED is slightly brighter in the 180 degree position and slightly dimmer in the 0 degree position.

Example Program: Servo-Positions.c

```

/* Servo-Positions.c */

#include "simpletools.h"           // Include simpletools header
#include "servo.h"               // Include servo header

int main()                       // main function
{
    servo_angle(14, 0);          // P14 servo to 0 degrees
    pause(2500);                // ...for 2.5 seconds
    servo_angle(14, 900);        // P14 servo to 90 degrees
    pause(2500);                // ...for 2.5 seconds
    servo_angle(14, 1800);       // P14 servo to 180 degrees
    pause(2500);                // ...for 2.5 seconds
    servo_stop();                // Stop servo process
}

```

How it Works – Servo-Positions.c

As mentioned earlier, the program uses `#include "servo.h"` to access to all the servo library's functions. This program uses a function named `servo_angle` to set the various servo positions. This function has two parameters: `pin` and `degreeTenths`.

The statement `servo_angle(14, 0)` sends signals that make the servo connected to P14 hold the 0-degree position. The `pause(2500)` statement makes the servo hold that position for 2.5 seconds. Next, `servo_angle(14, 900)` makes the servo hold a new position, 90-degrees. Another `pause(2500)` allows it to hold that position for another 2.5 seconds before `servo_angle(14, 1800)` makes the servo hold the 180-degree position.

```

servo_angle(14, 0);
pause(2500);
servo_angle(14, 900);
pause(2500);
servo_angle(14, 1800);
pause(2500);

```

Last, but not least, the servo library's `servo_stop()` function tells the servo to stop holding any position by shutting down the processor that was running the code that sends control signals to the servo.

```
servo_stop();
```



The servo library can control up to 14 servos. If your project needs more than 14, you can include the `servoAux` library, which can control an additional 14 servos.

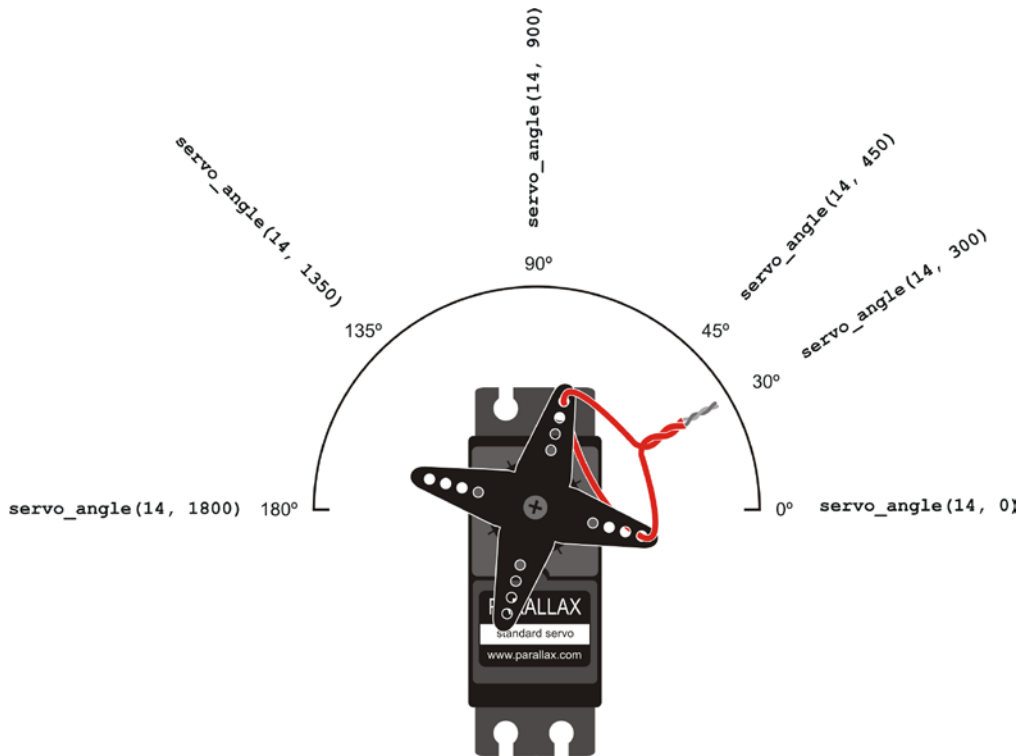
Your Turn – Programs to Point the Servo in Different Directions

Figure 4-18 shows a few `servo_angle` statements that tell the servo to hold certain major positions: 0°, 30, 45°, 90°, 135°, and 180°.

- Use SimpleIDE's Save Project As button, and re-name the project Servo-Positions-YourTurn.
- Modify the program so that it visits each of the positions shown in Figure 4-18. Pick a different amount of time from 1 second to 3 seconds for each hold time.

Figure 4-18: Servo Horn Positions

Note that each one uses a different argument for the `servo_angle` function's `degreeTenths` parameter



ACTIVITY #4: CONTROLLING POSITION WITH YOUR COMPUTER

Factory automation often involves microcontrollers communicating with larger computers. The microcontrollers read sensors and transmit that data to the main computer. The main computer interprets and analyzes the sensor data, and then sends position information back to the microcontroller. The microcontroller might then update a conveyer belt's speed, or a sorter's position, or some other mechanical, motor-controlled task. The control system pattern is *input, process, output*.

In this activity, you will program the Propeller to interact through the SimpleIDE terminal. The program will make the Propeller send messages to the SimpleIDE terminal with instructions to enter servo positions and hold times. The Propeller program will also read the values you type and control the servo accordingly.

Parts and Circuit

Same as Activity #2

Programming the Propeller to Receive Messages from SimpleIDE Terminal

You can use the SimpleIDE Terminal to send messages from your computer to the Propeller as shown in Figure 4-19. The Propeller has to be programmed to listen for the messages you send using the SimpleIDE Terminal, and it also has to store the data you send in one or more variables.

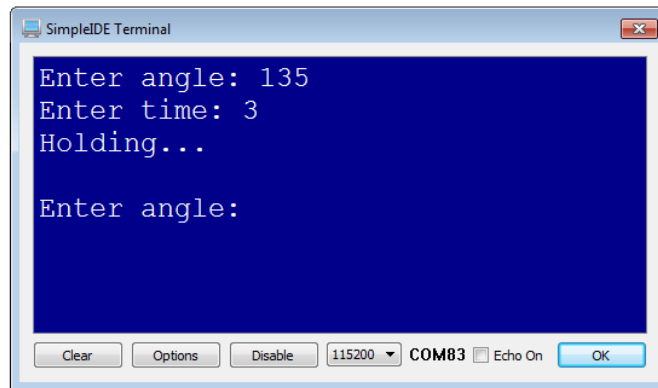


Figure 4-19
Sending Messages
to the Propeller

*Click to the right of
the prompt and type
your answer.*

- Click SimpleIDE’s New Project button, and save as Servo-TerminalControl.
- Click Run with Terminal.
- Click to the right of the “Enter angle:” prompt and type a value in the 0–180 range, then press Enter. (Don’t enter a value larger than 180 here! We’ll get to that later.)
- Repeat for hold time in seconds — use a small number because it’s the number of seconds you’ll have to wait — then press Enter again.
- Verify that the servo holds the specified position for the amount of time before prompting you for a new position and time.

Example Program: Servo-TerminalControl

```

/* Servo-TerminalControl.c */

#include "simpletools.h"           // Include simpletools
#include "servo.h"

int main()                       // Main function
{
    int angle, time;             // Variables

    while(1)                     // Main loop
    {
        print("Enter angle: ");  // Prompt for angle
        scan("%d", &angle);     // Get angle

        print("Enter time: ");  // Prompt for time
        scan("%d", &time);     // Get time

        angle *= 10;            // Degrees -> degree tenths
        time *= 1000;           // Seconds -> milliseconds

        print("Holding...\n\n"); // Indicate holding

        servo_angle(14, angle); // Set servo to position
        pause(time);            // For specified time
    }
}

```

How it Works - Servo-TerminalControl

Inside the `main` function, the first line declares a couple of `int` variables, `angle` and `time`.

```
int angle, time;
```

Inside the `while(1)` loop a `print` statement displays the message "Enter angle: ".

```
print("Enter angle: ");
```

Next we have a new function, `scan`. It's like `print`, but instead of sending info from Propeller to terminal, it makes the Propeller receive information from the Terminal. Like `print`, the `%d` formatting flag specifies a decimal integer value, but this time, it's going to get stored in the `angle` variable.

```
scan("%d", &angle);
```

One key difference between `print` and `scan` is that you have to put the `&` sign in front of the variable that gets the value with `scan`. When we printed a value with `print("%d", angle)`, we used the `%` symbol. That's because the `print` function is designed to send the value of the `angle` variable to SimpleIDE Terminal. In contrast, the `scan` function needs to know the variable's address in RAM, so `scan("%d", &angle)` passes the address of the `angle` variable with the `&` operator. After it converts the characters you type into a decimal integer value, it writes that value to the memory set aside for the `angle` variable in the Propeller chip's RAM.



What's the difference between a variable's value and its address?

Each variable stores a value at a certain location in the Propeller microcontroller's RAM (random access memory), which has 32,768 bytes. There's the 0th byte, the 1st byte, 2nd byte, and so on, up to the 32,767th byte. Each of these bytes can contain a value in the 0 to 255 range (unsigned char) or -128 to 127 (char). Four bytes together form an `int` variable, which can contain a value in the -2,147,483,648 to 2,147,483,647 range.

Let's say that the user types 135, causing `scan` to enter 135 in the `angle` variable and that variable occupies the 32,752th through 32,755th bytes in Propeller RAM. Its address would be 32,752. The variable's *value* would be 135, but its *address* would be 32,752.

The program repeats the same two calls to prompt for a time, and then gets the `time` value from SimpleIDE Terminal.

```
print("Enter time: ");
scan("%d", &time);
```

The `servo_angle` call needs a value that describes the angle in terms of tenths of a degree. So `angle *= 10`, which is the C language shorthand version of `angle = angle * 10`, changes the value `angle` stores from degrees to tenths of a degree. So, if you typed in 135, the result would be 1350. The `time *= 1000` statement performs a similar operation on the `time` variable, converting the seconds value that was entered into thousandths of a second since it's going to be used in the `pause` function. If you entered 3 for time, time it'll be converted to 3000.

```
angle *= 10;
time *= 1000;
```

The program is just about ready to put the servo in a new position for a new amount of time. Just before that, this `print` statement displays the "Holding...\n\n" message.

```
print("Holding...\n\n");
```

Now that we have an angle in tenths of a degree and a time in milliseconds, we can use those variables in `servo_angle(14, angle)` and `pause(time)`. The values stored by each variable get passed to the functions and they use those values to do their jobs, making the servo connected to P14 hold the 1350 tenth of a degree position for 3000 milliseconds, for example.

```
servo_angle(14, angle);
pause(time);
```

Your Turn – Setting Limits in Software

Let's imagine that this computer servo control system is one that has been developed for remote-control. Perhaps a security guard will use this to open a shipping door monitored from a control room. Maybe a college student will use it to control doors in a maze that mice navigate in search of food. Maybe a researcher will use it to aim their high-powered telescope at a certain constellation. If you are designing the product for somebody else to use, the last thing you want is to give the user (security guard, college student, researcher) the ability to enter the wrong number that could damage the equipment or give unexpected—and possibly disastrous—results.

While running Servo-TerminalControl, it is possible to make a mistake while typing the `angle` value into the SimpleIDE Terminal. Let's say the user accidentally typed 220 instead of 20 for the angle, and pressed Enter. The value 220 would cause the servo to try to turn to a 220-degree position, beyond its mechanical limits. Although it won't instantly break the servo, it's certainly not good for the servo or its useful lifespan.

A couple of `if...` statements just before the `servo_angle` call can prevent this problem.

- Click SimpleIDE's Save Project As button, and save as Servo-TerminalControl-YourTurn.
- Add the lines with the `// <- add` comments to your code between the `print("Holding...\n\n")` and `servo_angle(14, angle)` statements.
- Try entering a value that's out of range (like 200 or -10), and verify that it makes the correction before positioning the servo.

```
print("Holding...\n\n");

if(angle > 1800) angle = 1800;           // <- add
if(angle < 0) angle = 0;                 // <- add
```

```

print("angle = %d degree tenths\n",    // <- add
      angle);                          // <- add

servo_angle(14, angle);

```

ACTIVITY #5: CONVERTING POSITION TO MOTION

In this activity, you will program the servo to change position at different rates. By changing position by a few degrees at a time instead of all at once, you can make it seem that the servo horn is rotating more slowly. It's actually advancing positions incrementally, but the motor's response takes the jitter out of those increment changes so that the horn "turns" instead of taking tiny steps.

Programming a Rate of Change for Position

The servo library has to send the servo a signal that repeats itself 50 times per second to make it hold any given position. That's why the library uses another cog so that the exact timing of the signals does not interrupt any other task the Propeller is performing. Use your calculator to divide 50 into 1, and you'll see that $1/50^{\text{th}}$ of a second is 0.02 seconds. That's 20 thousandths of a second, or 20 ms, which is the manufacturer's requirement for the Parallax Servo. So this next example program makes the servo turn from 0 to 180 in steps of 6 every 20 ms. Then, it returns twice as fast, in steps of 12 every 20 ms.

Example Program: Servo-Velocities.c

- Click SimpleIDE's New Project button, and save as Servo-Velocities.
- Enter Servo-Velocities.c into SimpleIDE.
- Click the Load EEPROM and Run button. Don't forget to set your power switch to 2.
- Monitor the servo as it turns gradually from 0 to 180, then twice as quickly back to zero.

```

/* Servo-Velocities.c */

#include "simpletools.h"          // Header includes
#include "servo.h"

int main()                      // main function
{
    int angle;                  // Declare angle variable

    while(1)                    // Main loop

```

```

{
  for(angle = 0; angle <= 1800; angle += 6)      // 0 to 180, steps of 6
  {
    servo_angle(14, angle);
    pause(20);
  }

  for(angle = 180; angle > 0; angle -= 12)      // 180 to 0, steps of 3
  {
    servo_angle(14, angle);
    pause(20);
  }
}
}

```

How Servo-Velocities.c Works

We only need one variable for this operation, an `int` variable named `angle`.

```
int angle;
```

A `for... loop` starts with `angle = 0`. Each time through the loop, `angle` advances by 6. The `servo_angle(14, angle)` uses that angle value to position the servo, each time increasing by 6 tenths of a degree, for 20 ms (1/50th of a second). This causes the servo to turn from 0 to 180 gradually.

```

for(angle = 0; angle <= 1800; angle += 6)
{
  servo_angle(14, angle);
  pause(20);
}

```

The loop that turns the servo back from 180 to 0 subtracts 12 from `angle` each time through. This makes it seem like it's turning from 180 to 0 twice as fast.

```

for(angle = 180; angle > 0; angle -= 12)
{
  servo_angle(14, 0);
  pause(20);
}

```

Your Turn – Adjusting the Velocities

You can change the += 6 and -= 12 to different values to customize the speed of the servo's sweep and return. There's a limit to how far the servo can turn in 1/50th of a second, and you can test to find it.

- Use SimpleIDE's Save Project As button, re-name the copy Servo-Velocities-YourTurn.
- Try changing += 6 to += 3 and -= 12 to -=8. Observe the changes; do they match what you expect?
- Try increasing the sweep step size a little bit at a time until you start to notice that the servo doesn't make it all the way to the end of its range of motion. That's the indicator that your program is asking the servo to turn faster than its motor can keep up with.

ACTIVITY #6: PUSHBUTTON-CONTROLLED SERVO

In this chapter, you have written programs that make the servo go through a pre-recorded set of motions, and you have also controlled the servo with the SimpleIDE Terminal. Now, let's program the Propeller to control the servo based on pushbutton inputs. In this activity you will:

- Build a circuit for a pushbutton servo control.
- Program the Propeller to control the servo based on those pushbutton inputs.

When you are done, you will be able to press and hold one button to get the Propeller to rotate the servo in one direction, and press and hold the other button to get the servo to rotate in the other direction. When no buttons are pressed, the servo will hold whatever position it moved to last.

Extra Parts for Pushbutton Servo Control

The same parts from the previous activities in this chapter are still used. In addition, you will need to gather the following parts for the pushbutton circuits:

- (2) Pushbuttons – normally open
- (2) Resistors – 10 k Ω (brown-black-orange)
- (2) Resistors – 220 Ω (red-red-brown)

(2) Jumper wires (red)

Adding the Pushbutton Control Circuit

Figure 4-20 shows the pushbutton circuits that you will use to control the servo.

- Add this circuit to the servo+LED circuit that you have been using up to this point. When you are done, your circuit should resemble Figure 4-21.

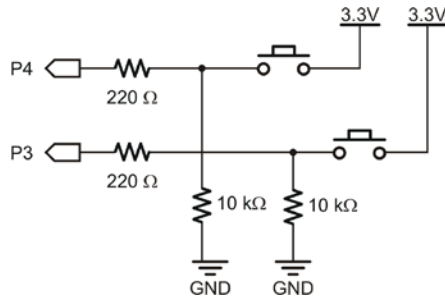


Figure 4-20
Pushbutton
Circuits for Servo
Control

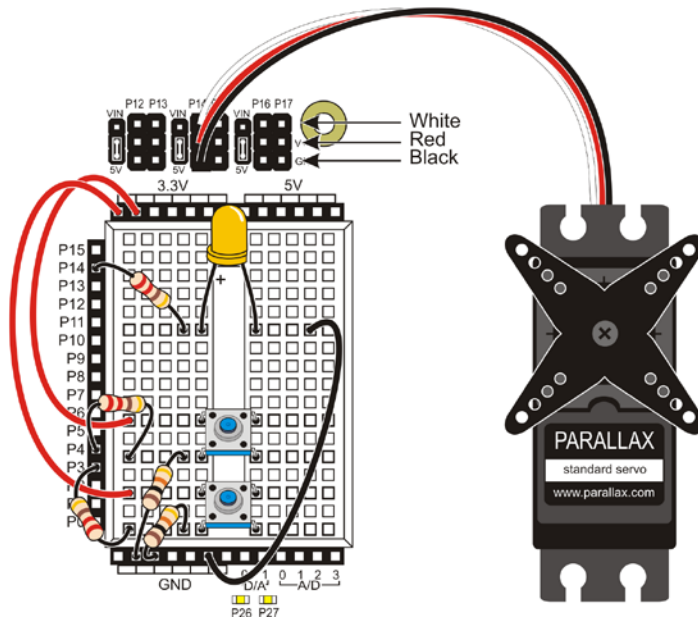


Figure 4-21
Propeller Activity
Board Servo,
LED and
Pushbutton
Circuits

- First just test your pushbutton circuit independent of servo control. Test the pushbutton connected to P3 using the original version of `Button-ReadState.c`. The program and the instructions on how to use it begin on page 73.
- Modify the program so that it reads P4.
- Run the modified program to test the pushbutton connected to P4.

Programming Pushbutton Servo Control

`if...` statements inside a `while` loop can be used to increase or decrease the servo's angle while the button is pressed. In addition to checking if the button is pressed, the program also needs `if...` statements to make sure they have not exceeded the servo's range of motion. The solution for this is recycled from Activity #4's Your Turn section.

Example Program: Servo-ButtonControl.c

This example program makes the servo's horn rotate counterclockwise when the P4 pushbutton is pressed. The servo's horn will keep rotating so long as the P4 pushbutton is held down and the value of `angle` is smaller than 1800. When the P3 pushbutton is pressed, the servo horn rotates clockwise. The servo also is limited in its clockwise motion because the `angle` variable is not allowed to go below 0. The SimpleIDE Terminal displays the value of `angle` while the program is running.

- Click SimpleIDE's New Project button, and save as `Servo-ButtonControl`.
- Enter `Servo-ButtonControl.c` into SimpleIDE.
- Click Run with Terminal.
- Verify that the servo turns counterclockwise (until 180-degrees) when you press and hold the pushbutton connected to P4.
- Verify that the servo turns clockwise (down to 0-degrees) when you press and hold the P3 pushbutton.

```
/* Servo-ButtonControl.c */

#include "simpletools.h"           // Include simple tools
#include "servo.h"

int main()                       // Main function
{
    int angle = 900;             // Servo angle variable
```

```

while(1) // Main loop
{
  if(input(4) == 1) // P4 button pressed?
    angle += 18; // ...increase angle

  if(input(3) == 1) // P3 button pressed?
    angle -= 18; // ...decrease angle

  if(angle > 1800) angle = 1800; // Limit angle
  if(angle < 0) angle = 0;

  print("%c angle = %d %c", // Display angle
        HOME, angle, CLREOL);

  servo_angle(14, angle); // Set servo to position

  pause(20); // Wait 1/50th second
}
}

```

How it Works – Servo-ButtonControl

At the start of the `main` function, an `int` variable named `angle` is declared and initialized to 900. This should start the servo at 90 degrees.

```
int angle = 900;
```

In the `while(1)` loop, if the P4 pushbutton is pressed, the number 18 gets added to the `angle` variable. If the P3 button is pressed, the number 18 gets subtracted from `angle`.

```

if(input(4) == 1)
  angle += 18;

if(input(3) == 1)
  angle -= 18;

```

This is the code from the Activity #4 Your Turn section that prevents the servo from turning outside its mechanical limits by changing the `angle` variable's value to 1800 if it tries to exceed it, or to 0 if it tries to drop below it.

```

if(angle > 1800) angle = 1800;
if(angle < 0) angle = 0;

```

With only 1/50th of a second pause between updates, this program keeps the `angle` variable information on a single line. Otherwise, it would scroll too fast and not be readable.

```
print("%c angle = %d %c",
      HOME, angle, CLREOL);
```

At this point, the `angle` variable has been adjusted if one of the buttons is pressed and its value limited to stay within the servo's mechanical limits. So this statement sets the servo connected to P14 to the position dictated by the `angle` variable's value, which is the position in 10ths of a degree from 0 to 1800.

```
servo_angle(14, angle);
```

The servo library only updates the servo position 50 times per second, so the program should wait for 20 ms (1/50th of a second) before allowing the `while` loop to repeat.

```
pause(20);
```

Your Turn – Speed and Limit Adjustments

You can change the 18s in these statements to larger values to make the servo respond more quickly, or smaller values to make the servo respond more slowly.

```
if(input(4) == 1)
    angle += 18;

if(input(3) == 1)
    angle -= 18;
```

You can also limit the servos' range of motion by decreasing the 1800 values (counterclockwise limit) and/or increasing the 0 values. Make sure to adjust both 1800s and both 0s.

```
if(angle > 1800) angle = 1800;
if(angle < 0) angle = 0;
```

- Click SimpleIDE's Save Project As button, and save as Servo-ButtonControl-YourTurn.
- Try it!

SUMMARY

This chapter introduced microcontrolled motion with the Parallax Standard Servo, including the following:

- What a servo is, and the parts of a servo.
- What servos are used for in a variety of industries and devices.
- Servo control signals that are required to set and hold a servo's position.
- How to safely supply power to a Parallax Standard Servo via a Propeller Activity Board.
- How to use the functions in the `servo.h` library to control the servo's position with functions such as `servo_angle` and `servo_stop`.
- How to use the SimpleIDE terminal and the `scan` function to send values to a program during run time.
- The difference between a variable's value and a variable's address.
- How to use `if(condition...)` statements to set limits in software, so the program does not attempt to push the servo beyond its mechanical limits.
- How to build a circuit that uses the Propeller microcontroller to control a servo with pushbuttons.

Questions

1. What are the five external parts on a servo? What are they used for?
2. Is an LED circuit required to make a servo work?
3. What command controls the angle the servo's horn gets turned to? What are its parameters?
4. What function can you use to control the amount of time that a servo holds a particular position?
5. How do you use the SimpleIDE Terminal to send messages to the Propeller? What function was used to make the Propeller receive messages from the SimpleIDE Terminal?

Exercises

1. Write code that positions the servo at 60 degrees for 3 seconds, then at 120 degrees for 2 seconds.
2. Write a code block that sweeps the value of `angle` controlling a servo's position from 700, to 800, then back to 700, in increments of (a) 1, (b) 4.

Project

1. Modify ServoVelocities so that the P3 pushbutton functions as a “kill switch”, causing the windshield wiper motion to cease. Hints: You can use code that says `if(condition) break;` to a code block to “break out” of it before all the loop’s repetitions are done. (1) Declare a variable named `button` before the `while` loop starts and initialize it to zero. (2) Read the P3 button inside each `for...` loop and save its state to a variable named `button`. Example: `button = input(3)`. (3) Add `if(button == 1) break;` to each `for...` loop. (4) Also add `if(button == 1) break;` after each `for...` loop so that you can get out of the `while` loop too. (5) If the code exits the `while` loop, it should execute a `servo_stop();` call before the `main` function runs out of commands and ends. This will stop the cog running the servo control code and release control of the servo so that you can manually turn it again.

Solutions

- Q1. Plug – connects servo to power and signal sources; 2) Cable – conducts power and signals from plug into the servo; 3) Horn – the moving part of the servo; 4) Screw – attaches servo’s horn to the output shaft; 5) Case – contains DC motor, gears, and control circuits.
 - Q2. No, the LED just helps us see what’s going on with the control signals.
 - Q3. The function is `servo_angle`, and its parameters are `pin` (I/O pin number) and `degreeTenths` (position).
 - Q4. The `pause` function.
 - Q5. Click the SimpleIDE Terminal and start typing. Use the `scan("%d", &variableName)` to pass the address of the variable you want `scan` to fill with the result you typed into the terminal.
- E1. A couple `if...` statements did the job in this chapter. Example: `if(angle > 1800) angle = 1800.`

```
servo_angle(14, 600);
pause(3000);
servo_angle(14, 1200);
pause(2000);
```

- E2. a) Increments of 1

```
for(angle = 700; angle <= 800; angle++)
```

```

{
servo_angle(14, angle);
pause(20);
}
for(angle = 800; angle > 700; angle--)
{
servo_angle(14, angle);
pause(20);
}

```

b) Increments of 4

```

for(angle = 700; angle <= 800; angle += 4)
{
servo_angle(14, angle);
pause(20);
}
for(angle = 800; angle > 700; angle -= 4)
{
servo_angle(14, angle);
pause(20);
}

```

P1. There are many possible solutions, here is one.

```

/* Servo-P1-Solution.c */

#include "simpletools.h" // Header includes
#include "servo.h"

int main() // main function
{
    int angle; // Declare angle variable
    int button = 0; // <- Add

    while(1) // Main loop
    {
        for(angle = 0; angle <= 1800; angle += 6) // 0 to 180, steps of 6
        {
            servo_angle(14, angle);
            pause(20);
            button = input(3); // <- Add
            if(button == 1) break; // <- Add
        }
        if(button == 1) break; // <- Add

        for(angle = 180; angle > 0; angle -= 12) // 180 to 0, steps of 3

```

```
{
  servo_angle(14, angle);
  pause(20);
  button = input(3);           // <- Add
  if(button == 1) break;      // <- Add
}
if(button == 1) break;       // <- Add

}

servo_stop();                // <- Add
}
```

Chapter 5: Write Multicore Code

Multicore isn't just for code tucked away in libraries like `servo.h`. You can also use it in your programs to make different processors (cogs) pay attention to different tasks at the same time. With multiprocessing, your program do things like have one processor focus on a repetitive task requiring high speed and precision, while another processor focuses on a different task without interruption. That different task might be waiting for input signals, repeat at a different rate, or whatever else is needed. The best part is that code for each task can exchange information by simply updating and checking certain variables.



Figure 5-1
Multiprocessing is like
teamwork

The previous chapter introduced you to multiprocessing with the terminal-controlled servo position example. The servo needed to receive precisely timed signals every 20 milliseconds. A library took care of that in another cog. Meanwhile, the cog running the `main` function could wait patiently for the next servo position to be typed in without having to repeatedly take breaks every 20 ms to send the next servo control signal.

In this chapter, you will write a variety of multicore programs that make more than one processor execute different parts of your program at the same time. The parts of your program that processors will execute are called functions, which are some of the most widely used C language building blocks. Functions tucked away in libraries that you have been using already include `high`, `low`, `pause`, `print`, etc., but now you'll write your own. This chapter also introduces global variables (you've been using local variables up until now), and explains the difference with a term called *variable scope*. It also shows how to add global variables to your programs, and what you need to do so that those variables can be accessed by functions running in different processors.

INTRODUCING THE FUNCTION

Up to now, the example programs have used functions that were already defined within some library. All your code needed to do was `#include` the function's library, such as `simpletools.h` and `servo.h`, and then call the function with arguments to pass to its parameters as needed.

Now, let's see how to make custom functions right in your program. Figure 5-2 shows an example of a function definition you can add to your program. This function contains two `print` statements that send messages to the SimpleIDE Terminal — these will let you know that code in the function is getting executed. The “curly” braces `{ }` contain the two statements that make up the function's code block.

The first line is the *function prototype*, and it typically contains three elements in this order: a return value, a function name, and a parameter list. This function's name is `hello`. In this case, `hello` does not send any information back when it is done executing, so the term `void` is used instead of a return value. The `hello` function does not need to receive any values (arguments) to do its job, so its parameter list is just `(void)` inside parentheses.

Figure 5-2: Function Definition for Hello Function

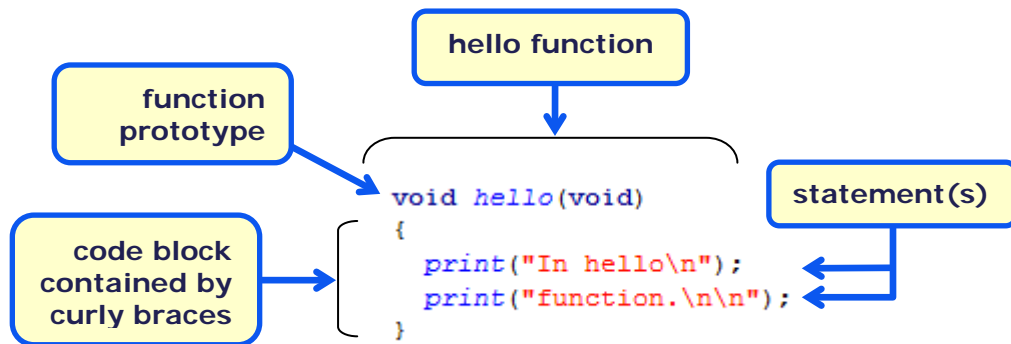
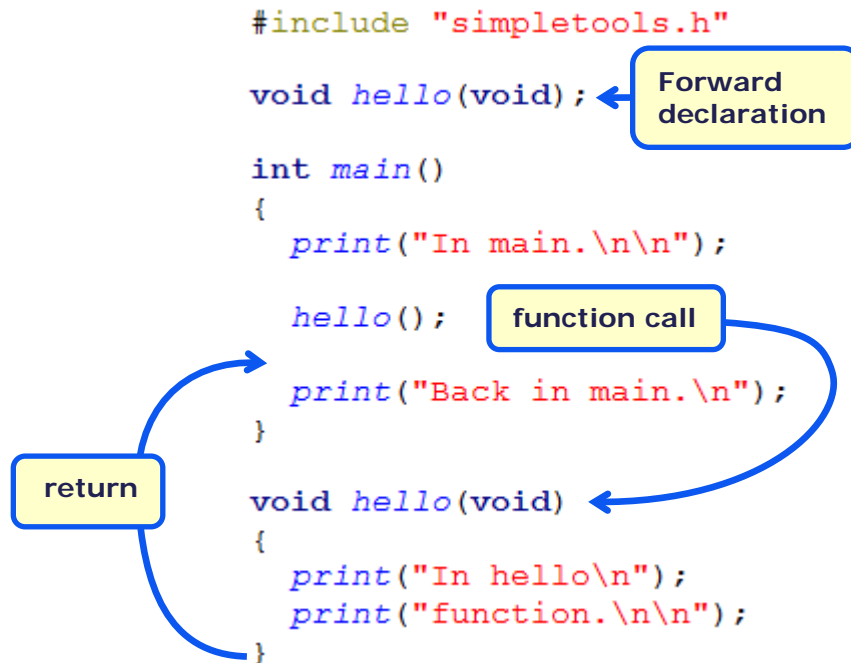


Figure 5-3 shows a code example with two functions, named `main` and `hello`. Yes, `main` is also a function, a special one since programs start executing with the first statement in `main`. The `hello` function was added below the `main` function.

Inside `main`, we have the `hello()` function call. It tells the program to go find the `hello` function, execute all of its statements, and come back (*return*) when done. After the function call is done, code execution resumes where it left off, at the statement that comes right after the `hello` call in `main`.

The *forward declaration* is a “heads-up” statement for the C compiler that it might find a call to a function named `hello` before it ever sees the function itself.

Figure 5-3: How to Call a Function in Your Code



ACTIVITY #1: TEST THE MULTI-HELLOFUNCTION

Judging from Figure 5-3, we should expect the program to print “In main”, then “In hello function” followed by “Back in main.” In other words, the SimpleIDE Terminal output should resemble Figure 5-4. Let’s check and see.

How it Works

After the `#include` for `simpletools`, and before the `main` function, we need a forward declaration for the `hello` function: `void hello(void);`. It's just like the first line in the `hello` function definition, but it's followed by a semicolon instead of the entire code block. It lets the C compiler know there will be calls to this custom function coming up.

The `void` before `hello` means this function does not return a value. The `(void)` after `hello` means this function does not require any parameters.

```
#include "simpletools.h"

void hello(void);
```

Inside the `main` function, we have a `print` statement that displays "In main.\n\n" in the SimpleIDE Terminal. The second statement is a `hello` function call. This tells the program, "Go find the function named `hello`, execute its code, and come back when done." After the `hello` function is done, the `main` function prints, "Back in main.\n" before running out of code and ending the program.

```
int main()
{
    print("In main.\n\n");

    hello();

    print("Back in main.\n");
}
```

When the `main` function got to the `hello` function call, it skipped to the function definition, executed the two statements in its code block, and then returned.

```
void hello(void)
{
    print("In hello\n");
    print("function.\n\n");
}
```



Forward Declarations – Get in the Habit

If you put all of your custom function definitions before the `main` function, forward declarations are not necessary. However, many programmers find code easier to follow when it begins with forward declarations before `main`, and function definitions after `main`.

You might forget to add a forward declaration and discover that your code still runs. Why? The PropGCC compiler is “forgiving” — it can figure out what to do if a forward declaration is missing from a single-core program*. But, forward declarations are *absolutely required* for running our upcoming multi-core Propeller C programs, so it is best to get into the habit of using them to keep your coding options open.

Compilers for other devices and other C-related programming languages might not be so forgiving if you forget forward declarations, so again, best to get in the habit now.

*Forgetting a forward declaration does generate a compiler warning. To view all warnings, see SimpleIDE Help.

Your Turn – Multiple Hello Calls

A key advantage to functions is that they allow you to create reusable blocks of code. Here is an example that calls `hello` a second time.

- Save a copy as `Multi-HelloFunction-YourTurn1`, and update the `main` function to match the example below. What do you think the code will do now?
- Click Run with Terminal. Were you right? If so, great! If not, look at the code carefully and make sure you’re clear on what’s happening before moving on.

```
int main()
{
    print("In main.\n\n");

    hello();
    hello();                // <- add

    print("Back in main.\n");
}
```

- You can even do things like add a loop to your `main` function that calls `hello` repeatedly. Save a new copy as `Multi-HelloFunction-YourTurn2` and try replacing the two `hello` calls with this loop:

```
for(int i = 0; i < 4; i++)
{
    hello();
}
```

ACTIVITY #2: PARAMETERS AND RETURN VALUES

Take a look at Figure 5-5. Here, a function named `add` has different features that the `hello` function above didn't have. You probably guessed that `add` adds two numbers together. But, how does it get those numbers? And what does it do with the answer?

In Figure 5-5, arrows show how two arguments from the `add` function call in `main` are passed to the actual `add` function's parameters: 2 is passed to `a`, and 3 is passed to `b`. Inside `add`, the first statement declares an `int` variable named `c`, and assigns it the result of `a + b`. Next is `return c`, which means that the function will send the value stored in `c` (5 in this example) back to the call in `main`. That result gets stored in a value named `sum`. After the `add` call, a `print` statement lets us see the value of the `sum` variable.

```
#include "simpletools.h"

int add(int a, int b);

int main()
{
    int sum = add(2, 3);
    print("sum = %d\n", sum);
}

int add(int a, int b)
{
    int c = a + b;
    return c;
}
```

Figure 5-5
Function with Parameters
and Return Value



Parameter options: Parameters can receive both values and expressions as arguments. For example, `int x = 2, y = 3; int sum = add(x, y)` is a valid function call too.

Return value tricks: In this example, the `add` function's return value was stored in the `sum` variable. Then, `sum` was used in the next instruction. However, you can place a function call right inside the instruction or expression where you'd like the return value to be used. For example, `if(add(a, b) > 5)...`, and `for(int i = 0; i < add(2, 3), i++)...` all contain valid function calls.

Let's make sure this program works as expected.

Example Program: Multi-TestFunction

- Create a New Project, name it Multi-TestFunction, and save it to My Projects.
- Enter the Multi-TestFunction.c code into SimpleIDE.
- Click SimpleIDE's Run with Terminal button.
- Verify that the output of the unmodified code matches Figure 5-6.
- Try passing different integer values, and see if the results are correct each time.

```
/* Multi-TestFunction.c */
#include "simpletools.h"
int add(int a, int b);
int main()
{
    int sum = add(2, 3);
    print("sum = %d\n", sum);
}
int add(int a, int b)
{
    int c = a + b;
    return c;
}
```

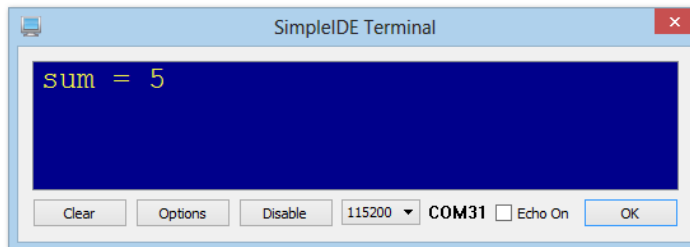


Figure 5-6
SimpleIDE
Terminal Output
from the Add
Function.

How It Works

Unlike the simple `hello` function in the last example, the `add` function requires two parameters and returns a value. In its function prototype, `int` to the left of `add` means it

returns an `int` value when it's done. The two-item parameter list to the right of `add` means it needs to receive two integer values as arguments from each function call.

```
#include "simpletools.h"

int add(int a, int b);
```

The `main` function starts with the function call `int sum = add(2, 3)`. The `add(2, 3)` part of this statement sends the values 2 and 3 to the `add` function. Then, the `add` function does the math and sends back the answer with `return c`. Back in the function call, the `int sum =` part stores the answer in a variable named `sum`. Then, the `main` function prints "sum = ", followed by the value of `sum`, and a newline character (just in case you want to add a statement that prints something on the next line).

```
int main()
{
    int sum = add(2, 3);

    print("sum = %d\n", sum);
}
```

When the `main` function executes `add(2, 3)`, it passes the value 2 to the `add` function's `a` parameter and 3 to its `b` parameter. Now the `add` function has two variables loaded with values, so it is ready to execute `int c = a + b` for a result of 5. The `return c` statement sends the value of 5 that `c` stores back to the function call in `main`.

```
int add(int a, int b)
{
    int c = a + b;
    return c;
}
```



Another way to think about a return value:

Before the call, we have:

```
int val = add(2, 3);
```

When `add` returns its value, it's helpful to imagine that the function call *becomes* that return value.

```
int val = 5;
```


Try This – One Program, More Functions

Let's try putting a `subtract` function in a copy of `Multi-TestFunction.c`.

- Save a copy of the `Multi-TestFunction` as `Multi-TestFunction-TryThis`.
- Make a second function below the `add` function, named `subtract`, with two `int` parameters `a` and `b`, and an `int` return value
- Make the `subtract` function's operation `int c = a - b`, that returns `c`.

```
int subtract(int a, int b)
{
    int c = a - b;
    return c;
}
```

- Update the forward declarations to include a `subtract` function prototype.

```
int subtract(int a, int b);
```

- In `main`, add a function call to `subtract` that stores the return value in an `int` variable named `difference`.

```
int difference = subtract(2, 3);
```

- Make another `print` statement to display `difference`.

```
print("val difference = %d\n", difference);
```

- Run the code and verify that it works as you expect. Debug, rinse, repeat!

Your Turn – A Function That Repeats

Do you remember the simple `hello` function from Activity #1? To make “Hello!” print again, you had to call the `hello` function again. A `for` loop inside `main` is one way to call a function repeatedly. Another way is to put the `for` loop inside the function itself, and add a parameter to specify how many times the loop should repeat. Let's try it.

- Save a copy of the `Multi-TestFunction` as `Multi-TestFunction-YourTurn`.
- Enter the `Multi-TestFunction-YourTurn` code into SimpleIDE.

- ❑ Click SimpleIDE's Run with Terminal button and verify that SimpleIDE Terminal displays six "Hello!" messages.

```

/* Multi-TestFunction-YourTurn.c */

#include "simpletools.h"

void hellos(int reps);

int main()
{
    hellos(6);
}

void hellos(int reps)
{
    for(int i = 0; i < reps; i++)
    {
        print("Hello!\n");
    }
}

```

ACTIVITY #3: VARIABLE SCOPE

C language has a feature called *variable scope* that allows you to control what sections of code can access a variable to check its value (read it) or modify it (write to it). Scope is determined by where the variable is declared. There are two things to think about: code blocks, and position within the block.

Up to now, all the example programs contained only *local variables*. A local variable can only be used inside the code block where it was declared, by code that comes after the declaration. Variables must be declared before they can be used, so they are often placed right after the opening curly brace of a code block. The code block could be for the `main` function, a custom function, or even for a conditional loop such as `while(1)`.

A *global variable* can be used by any code within the application. Global variables must be declared outside of any code block, including the `main` function. They are often placed right after any `#include` statements.

In this activity, you will experiment with variables of different scope. Upcoming activities will then use global variables to exchange information between functions running in different Propeller cores at the same time.

Local Scope Examples

In previous chapters, program variables were declared right after the opening curly brace of the `main()` function, above the rest of the code in `main()`. In those programs, the variables could be used in any code that came after them inside of `main()`, including inside other code blocks that were nested within `main()`. That's a pretty broad scope, though still local to `main()`.

You may have noticed an exception, in `for` loops, like this one:

```
for(int x = 1; x <= 10; x++)
{
    print("x = %d\n", x);
}
```

Here, an `int` variable `x` is declared right inside the `for` statement, and is then used again inside the `for` code block. This `x` variable is local to this `for` loop; as if it only exists while this instruction is being executed.

If you were to also declare another `int x`; elsewhere within `main`, the `for` loop itself would not use that value for `x`. The `x` local to this `for` loop does not read nor modify any variables with the same name that are declared outside of the loop.

It sounds confusing, but it's easier to see with an example.

Example Program: Multi-LocalScope

- Click SimpleIDE's New Project button. Name the project Multi-LocalScope, and save it to My Projects.
- Enter the Multi-LocalScope.c code into SimpleIDE.
- Click SimpleIDE's Run with Terminal button.
- Check your terminal output against Figure 5-7.

```
/* Multi-LocalScope.c */
#include "simpletools.h"                // Include simple tools

int main()                             // Main function
{
    int x = 19;                         // x local to main

    print("x within main = %d\n", x);  // print val of x local to main
}
```

```

pause(300); // pause for serial terminal

for(int x = 1; x <= 3; x++) // x local to for loop
{
    print("x within for loop = %d\n", x); // print val of x local to for
    pause(300); // pause for serial terminal
}

print("x within main = %d!\n", x); // print val of x local to main
}

```

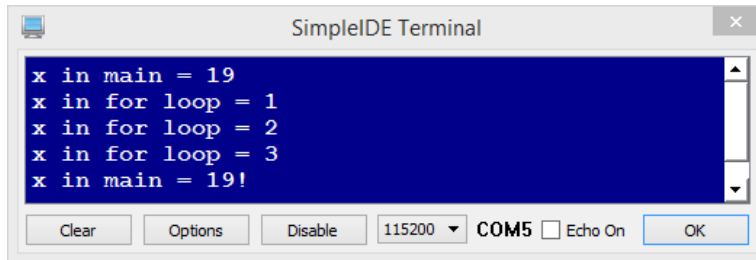


Figure 5-7
SimpleIDE
Terminal Output
for Local
Variables

How it Works

The variable declaration `int x = 19;` is right at the top of the `main` function's code block. So, it is local to the `main` function, and can be used anywhere within it. Next, a `print` statement displays the value of `x` in the SimpleIDE Terminal.

```

int main()
{
    int x = 19;

    print("x within main = %d\n", x);
    pause(300);
}

```

After a short pause, a `for` loop declares a new variable named `x` right inside its statement. Since this `x` was declared right here, it is local to the `for` loop. Each time through the loop, the value of this `x` is printed to the SimpleIDE terminal and incremented by 1.

```

for(int x = 1; x <= 3; x++)
{
    print("x within for loop = %d\n", x);
    pause(300);
}

```

When the `for` loop is finished, the next line of code re-prints the value of the original `x` that was declared in `main`. This `x` is still equal to 19, since the `for` loop declared and used its own local variable `x`, ignoring the other. The `x` declared in `main` is unaffected by the `for` loop.

```
    print("x within main = %d!\n", x);
}
```

Try This – Just One X

What do you think would happen if the `for` loop did not declare a new `int x` variable in its statement so that it was local to just itself? By deleting a single term in the program, you can find out.

- Save a copy of the program as Multi-LocalScope-TryThis.
- Delete just the `int` in `for(int x = 1; x <= 3; x++)`
- Run the modified program. Its SimpleIDE terminal output should look like Figure 5-8.

Now, you can see that because the `for` loop is no longer declaring its own `int x`, it is re-using the original `x` declared in `main`. The `for` statement does reset `x` to 1, and the loop increments it three times. So, when `x` is printed for the last time from the `main` function, its value is now 4.

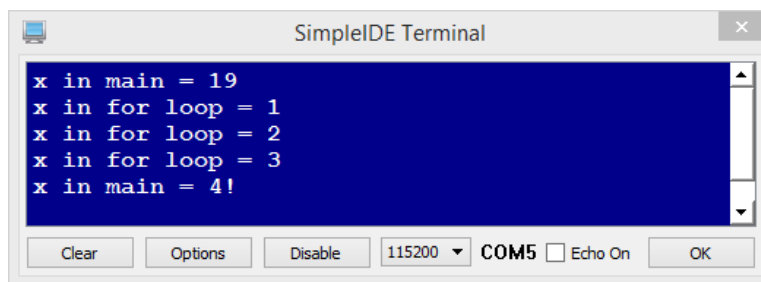


Figure 5-8
SimpleIDE
Terminal Output
for Local
Variables

These last two examples proved that, technically, a variable name can be re-used in a program as long as each instance has a different scope. However, these examples also made it clear that it can be very confusing to read and edit code where the same name is

re-used for different local variables, and very easy to make a mistake that can lead to unexpected outcomes.

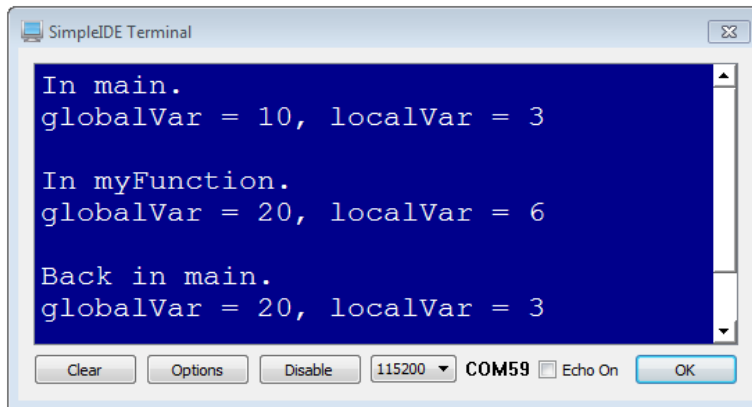
Many programmers prefer to give each variable in an application a unique name, regardless of its scope, to make code more readable. Still, it is quite common to re-use `i` (short for index) as a local variable in `for` loops.

Global Scope Examples

If you declare a variable outside of any functions, such as where the `#include` statements are, it will be *global* in scope. When a variable is global, all functions in the application can check its value and modify it.

Example Program: Multi-LocalVsGlobal

This next program underscores the difference between local and global variables by performing operations on variables with differing levels of scope in two different functions. All the functions are still running in the same cog.



```
In main.  
globalVar = 10, localVar = 3  
  
In myFunction.  
globalVar = 20, localVar = 6  
  
Back in main.  
globalVar = 20, localVar = 3
```

Figure 5-9
Local vs. Global
Output

- Click SimpleIDE's New Project button. Name the project Multi-LocalVsGlobal, and save it to My Projects.
- Enter the Multi-LocalVsGlobal.c code into SimpleIDE.
- Click SimpleIDE's Run with Terminal button.
- Check your terminal output against Figure 5-9.

```

/* Multi-LocalVsGlobal.c */

#include "simpletools.h"           // Library include

void myFunction();              // Forward declaration

int globalVar;                  // Global variable declaration

int main()                       // Main function
{
    globalVar = 10;              // Set global
    int localVar = 3;           // Declare & set local to main

    print("In main.\n");        // Display where we are
    print("globalVar = %d, localVar = %d\n\n", // Display variable values
          globalVar, localVar);

    myFunction();               // Call myFunction

    print("Back in main.\n");    // Display where we are again
    print("globalVar = %d, localVar = %d\n\n", // Display values yet again
          globalVar, localVar);
}

void myFunction()                // Function, arbitrarily named
{
    globalVar = 20;              // Modify global variable
    int localVar = 6;           // Declare & set new localVar
    print("In myFunction.\n");   // Display where we are
    print("globalVar = %d, localVar = %d\n\n", // Display variable values again
          globalVar, localVar);
}

```

How it Works

Right after including `simpletools` and the `myFunction` forward declaration, we have `int globalVar`. Since this declaration is outside of any function, it is global.

```

#include "simpletools.h"

void myFunction();

int globalVar;

```

The `main` function starts by setting `globalVar` to 10. Then, it declares and sets `localVar` to 3. Since `localVar` is declared inside a function, the `main` function in this case, code inside `main` can check and modify its value, but code in other functions

cannot. Next, two `print` statements display the “In Main...” message and the values of both variables.

```
int main()
{
    globalVar = 10;
    int localVar = 3;

    print("In main.\n");
    print("globalVar = %d, localVar = %d\n\n",
          globalVar, localVar);
}
```

The next thing that happens in `main` is the `myFunction()` call, so let's look at that code.

```
myFunction();
```

In `myFunction`, `globalVar` gets set to 20. A second local variable named `localVar` is declared—and so is only visible to this function—and set to 6. Two `print` statements prove that the `myFunction` code is getting executed by displaying the updated value of `globalVar` along with the value of this second `localVar`. Then, the function runs out of code and returns.

```
void myFunction()
{
    globalVar = 20;
    int localVar = 6;
    print("In myFunction.\n");
    print("globalVar = %d, localVar = %d\n\n",
          globalVar, localVar);
}
```

Back in `main` the values get printed again. This time, `globalVar` is 20 because it's a global variable that `myFunction` changed from 10 to 20. However, `localVar` is back to 3. That's because this `localVar` is part of the `main` function. The other instance of `localVar` in `myFunction` was set to 6, but it didn't change this instance because it was local to `myFunction`, not `main`.

```
print("Back in main.\n");
print("globalVar = %d, localVar = %d\n\n",
      globalVar, localVar);
```


Your Turn – Add a Global Variable and Operation

Global variables can interact with local variables. Let's add a second global variable to the project, and use it to store the value of `globalVar * localVar` inside `myFunction`.

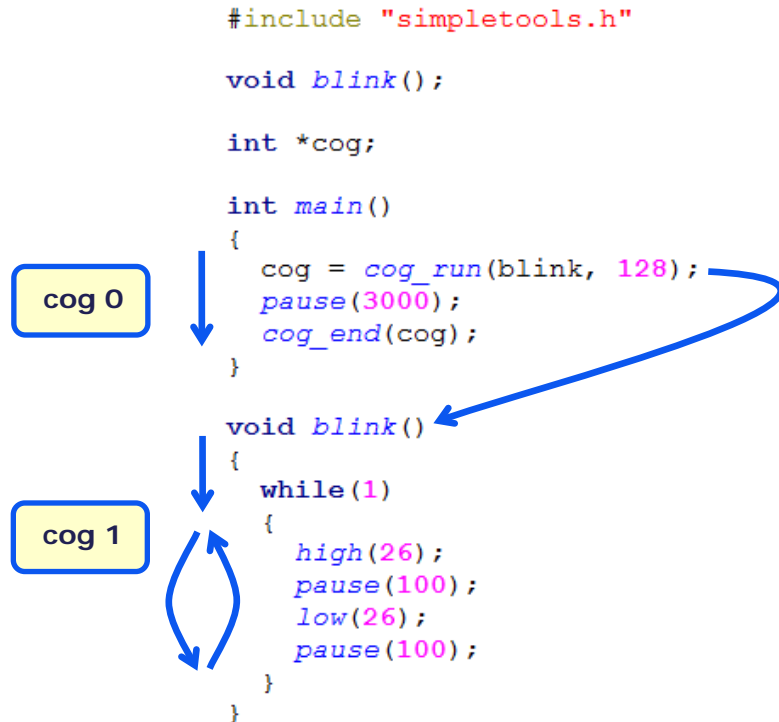
- Use SimpleIDE's Save Project As button, rename the project to Multi-LocalVsGlobal-YourTurn, and Save to the My Projects folder.
- Add a second global variable to the project, named `globalxLocal`.
- Inside `myFunction`, just above the `print` statements, add a statement that multiplies `globalVar` and `localVar`, and stores the result in `globalxLocal`.
- Expand the second `print` statement to display the value of `globalxLocal`.
- Click SimpleIDE's Run with Terminal button and verify the result; `globalxLocal` should equal 120.

ACTIVITY #4: RUN FUNCTIONS IN OTHER PROCESSORS (COGS)

The Propeller chip has eight processors, called *cogs* or sometimes *cores*. Each cog has its own number, 0 through 7. When a program starts, the `main` function automatically runs in cog 0, leaving seven other cogs available to run other functions in your program. In this activity, you will experiment with a program that uses another cog to run a function that blinks an LED. You will also expand the program to make a second cog blink a second LED at a different rate. The `main` function in cog 0 will also be put to work, so that three different processes are running simultaneously.

Figure 5-10 shows how two functions in the `simpletools` library, `cog_run` and `cog_end`, can be used to start a function in another cog, and stop it again. The first one uses `cog_run` to start the `blink` function in the next available cog. The second statement pauses for 3 seconds. While cog 0 is busy executing the `pause` function, cog 1 starts running the `blink` function, switching P26 on/off repeatedly. The third statement in the `main` function shuts down the cog running the `blink` function to make the LED stop blinking.

Figure 5-10: Function Example



Note that the `cog_run` function has two parameters:

1. The name of the function that will run in the new cog. (This gives the `cog_run` function the address in program memory where the named function resides.)
2. A number of `int` variables to set aside for the new cog to perform its computations

In Figure 5-10, the `blink` inside `cog = cog_run(blink, 128)` provides the `blink` function's address. Then, `128` is the number of `int` variables to set aside for the cog's computations. This block of memory `cog_run` creates is called *stack space*.

The `cog_run` function returns the memory address for where it set aside the stack space and recorded the ID number of the cog it launched. The example program set up a global *pointer variable* especially for storing this memory addresses with `int *cog`. The `*` in front of the variable name tells the C compiler that that variable “points to” an address in memory (instead of just storing a value).

The `cog = part of cog = cog_run(blink, 128)` copies the memory address `cog_run` returns to the `cog` pointer variable. At the end of the `main` function, `cog_end(cog)` takes the memory address stored by that `cog` pointer variable and uses it to stop the cog. This also frees the 128 `int` stack space for other uses.



***cog is a global variable.**

In this case, the `main` function used `cog_run` to run the `blink` function in another cog and `cog_end` to end it. Since `int *cog` is a global declaration, any function could use `cog_end(cog)` to end the `blink` function.

Test Cog-Launching Code

Let’s try the program from Figure 5-10.

Example Program: Multi-CogRun

- Click SimpleIDE’s New Project button. Set the File name to Multi-CogRun and Save.
- Enter the Multi-CogRun.c code into SimpleIDE.
- Click SimpleIDE’s Load RAM & Run button.
- Verify that the P26 LED blinks for 3 seconds, and then stops.

```

/* Multi-CogRun.c */

#include "simpletools.h"           // Library include

void blink();                   // Forward declaration

int *cog;                       // Pointer for cog data

int main()                      // Main function
{
    cog = cog_run(blink, 128);   // Run blink in other cog
    pause(3000);                // ...for 3 seconds
    cog_end(cog);               // then stop the cog
}

```

```

void blink()                                     // Blink function for other cog
{
  while(1)                                       // Endless loop for other cog
  {
    high(26);                                    // P26 LED on
    pause(100);                                  // ...for 0.1 seconds
    low(26);                                     // P26 LED off
    pause(100);                                  // ...for 0.1 seconds
  }
}

```

How it Works

In addition to now-familiar functions like **high**, **low**, and **pause**, simpletools library also has **cog_run** for starting functions in other cogs and **cog_end** for stopping them.

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

A forward declaration for the **blink** function is necessary, since **blink** is defined below the **main** function. This way the compiler knows to expect it before it sees the first reference to it in the code.

```
void blink();
```

Next, **int *cog** declares an **int** pointer variable named **cog**. Unlike regular **int** variables, **int** pointer variables store memory addresses instead of values. In this case, the **cog** pointer variable will store a memory address that gets returned from the **cog_run** function call. The **cog_end** function will use the address stored in **cog** to stop that cog.

```
int *cog;
```

Inside the **main** function, **cog = cog_run(blink, 128)** is what makes the **blink** function run in the next available cog. The first argument **cog_run** needs is the function's name without any parentheses next to it; this actually provides the starting memory address of the **blink** function. The second argument **cog_run** needs is a number of **int** variables to set aside as stack space for the new cog to use; here, 128 **int** sized pieces of memory are allocated. This is a recommended number for prototyping that you will see in many of this book's example programs.

```
int main()
{
```

```

    cog = cog_run(blink, 128);
    pause(3000);
    cog_end(cog);
}

```

Now, the `cog` = part of `cog = cog_run(blink, 128)` is put to use. The `cog_run` function call returns the starting address of a memory block that holds the ID number of the cog that was launched as well as the stack space. This address gets stored in the `cog` pointer variable. So now the `blink` function is running in a new cog, while the first cog is executing a 3 second delay from `pause(3000)`. After that, the `cog_end(cog)` function call uses the address stored in `cog` to shut down the cog running the `blink` function.

Functions launched by `cog_run` have three requirements: an empty parameter list, `void` return type, and infinite loop structure. The infinite loop prevents it from running out of code and shutting itself down without releasing its stack space for re-use. That's a job for `cog_end`.

```

void blink()
{
    while(1)
    {
        high(26);
        pause(100);
        low(26);
        pause(100);
    }
}

```

Recap and More Details for cog_run and cog_end

cog_run - For launching another processor (cog) and running a function that has a `void` return type and an empty parameter list. If the function does not self-terminate with a `cog_end`, the statements in its code block must be inside an infinite loop.

```

int *cogPointer;
...
cogPointer = cog_run(functionName, stackSize);

```

cogPointer is an optional pointer variable for storing the address where the cog's number and stack space are set aside. A program can use it to stop the cog and recover stack space later with `cog_end`. Note: If all the cogs are already in use, `cog_run` will return -1.

functionName is the name of the function that gets run in another cog. The function's name without parentheses returns the function's address in memory, which is what `cog_run` needs to make another cog start executing the function's code.

stackSize is the number of `int` variables to set aside for the function's local variables and function calls. Use 128 for prototyping. It's probably more than needed, but stack that is too small can cause program bugs.

In general, the number of `int` variables needed for **stackSize** increases by 1 for each local variable declared in **functionName**, 2 for each call to other functions, and 1 for each function parameter and return value. When you have finished expanding and refining a function running in another cog, try the program Cog Stack Usage Test.side from ...Learn\Examples\Multicore. After adding test code that exercises all of the features of the function(s) running in the other cog, this program can tell you how much stack space is actually used so you can reduce the size if desired.

cog_end – Uses `cogPointer` as a *process identifier* to stop a cog that was started by `cog_run`. This frees the cog and the stack space for other uses.

```
cog_end(cogPointer);
```



Other functions for starting and stopping cogs

The `propeller.h` library has additional functions for advanced ways to start and stop processes that run in other cogs. Libraries will often use these other functions to support assembly code and higher-speed code execution. Advanced tutorials that demonstrate the use of some of these functions are available on learn.parallax.com.

Try This – Add Another Function and Run it in Another Cog

Let's expand the example program so that it blinks the P27 LED at a different rate, using another cog. The process is fairly simple: just add a second function and pass it to `cog_run`. Your code will also need a second forward declaration for that function and a second cog pointer variable for shutting the cog back down.

- Save a copy as `Multi-CogRun-TryThis` in your My Projects folder.
- Add a second forward declaration for a `blink2` function, and declare a second pointer variable named `cog2`.

```

void blink();
void blink2();           //<-add

int *cog;
int *cog2;               //<-add

```

- ❑ Inside the `main` function, add a second `cog_run` function call start `blink2`, and a second `cog_end` function call to stop it.

```

int main()
{
    cog = cog_run(blink, 128);
    cog2 = cog_run(blink2, 128); //<-add
    pause(3000);
    cog_end(cog);
    cog_end(cog2);             //<-add
}

```

- ❑ Add the `blink2` function definition at the end of the program.

```

void blink2()           //<-add from here...
{
    while(1)
    {
        high(27);
        pause(223);
        low(27);
        pause(223);
    }
}                       //...to here

```

- ❑ Click SimpleIDE's Load EEPROM & Run button and verify that the P26 and P27 LEDs blink at different rates for 3 seconds.

Your Turn – Keep the First Cog Busy

The `main` function executed by cog 0 doesn't really need to sit around and do nothing while the other cogs are blinking LEDs. The `pause` call was just for the sake of example, so let's at least make it count while the other cogs blink lights.

- ❑ Use SimpleIDE's Save Project As button, and name it Multi-CogRun-YourTurn. Replace the `pause(3000)` statement with the code below.

- ❑ Click SimpleIDE's Run with Terminal button and verify that both lights blink at different rates while the `main` function displays a 3-second up-count in the SimpleIDE Terminal.

```

// pause(3000);                // <- remove
for(int i = 0; i < 12; i++)    // <- add
{
    print("i = %d\n", i);      // <- add
    pause(250);                // <- add
}                               // <- add

```

Now you have a glimpse of the power of the Propeller multiprocessing. With it, your programs can do many time-sensitive tasks at once, without conflict. Instead of just blinking LEDs, these cogs could be driving a servo, or playing audio files. The Propeller microcontroller's multicore architecture simplifies or solves many problems common to single-core microcontrollers. Next, we'll make the cores work together.

ACTIVITY #5 SHARING GLOBAL VARIABLES BETWEEN COGS

Picture a robot with two servo-driven arms, each controlled by its own cog, and a sensor to find the distance to an object. All three cogs might need to communicate with each other in order to determine the servo position needed for each arm to reach the object.

The previous activity didn't provide a way for one cog to influence or exchange information with another cog once it is running. For that, we use global variables, which were introduced in Activity #3. This activity shows how to use global variables exchange information between functions running in different cogs.

Global Variables Shared by Cogs Need to be Volatile

A global variable for cogs to share must be preceded with the keyword `volatile` to prevent *code optimization*.

```
volatile int globalVar;
```

The compiler can optimize code by removing unnecessary parts to make it execute faster and/or take less memory. For example, if the compiler sees a function that uses a variable but does not change it, it might remove code that re-check the variable's value before printing it.

In our multicore system, what the compiler might not see yet is that another function running in another cog could change that variable's value. The `volatile` modifier marks a variable as "subject to change" so the compiler won't try to optimize any code that uses it.

Example Program: Multi-InfoExchange

This next program declares a global `volatile int` variable named `t` (an abbreviation for time) that will allow the `main` function to control the `blink` function's LED on/off rate. The `main` function sets the value of `t`, which the `blink` function uses to set its `pause` times between `high/low` statements.

- Click SimpleIDE's New Project button. Name the project Multi-InfoExchange, and save to My Projects.
- Enter the Multi-InfoExchange.c code into SimpleIDE.
- Click SimpleIDE's Load RAM & Run button.
- Verify that the LED blinks at one rate for 2 seconds, and then at twice that rate for another two seconds, and then stops.

```

/* Multi-InfoExchange.c */

#include "simpletools.h"           // Library include

void blink();                    // Forward declaration

int *cog;                        // Pointer for cog data
volatile int t;                 // Declare t for both cogs

int main()                       // Main function
{
    t = 100;                     // Set value of t to 100
    cog = cog_run(blink, 128);   // Run blink in other cog
    pause(2000);                 // Let run for 2 s
    t = 50;                      // Update value of t
    pause(2000);                 // New rate for 2 s
    cog_end(cog);               // Stop the cog
}

void blink()                     // Function for other cog
{
    while(1)                     // Endless loop
    {
        high(26);                // LED on
        pause(t);                // ...for t ms
    }
}

```

```

    low(26);                // LED off
    pause(t);              // ...for t ms
}
}

```

How it Works

This program starts with declarations that are by now familiar: including the `simpletools` library, a forward declaration for the `blink` function, and a pointer variable declaration that `cog_run` and `cog_end` will use.

```

#include "simpletools.h"

void blink();

int *cog;

```

The `t` variable is also declared globally (outside any functions) so that one function can change its value, and a different function will be affected by that change. This global variable has to be `volatile` because different cogs are executing different functions that use it. Again, the `volatile` modifier prevents the C compiler from making optimizations that could cause one function running in one cog to miss a change to `t` made by another function running in another cog.

```

volatile int t;

```

The `main` function starts by setting the value of `t` to 100, and then runs the `blink` function in another cog. The `blink` function uses `t` as an argument in its `pause` calls, so the light will stay on/off for 100 ms initially. Meanwhile, the `main` function in cog 0 executes `pause(2000)`. After the 2-second delay, cog 0 changes `t` to 50. Since `t` is global, it's the same `t` value that the `blink` function uses in cog 1. So, this cuts the `blink` function's `pause` times in half, doubling the LED's on/off rate. Back in main, after another `pause(2000)`, the `main` function stops that cog 1 with `cog_end(cog)` so the light stops blinking.

```

int main()
{
    t = 100;
    int *cog = cog_run(blink, 128);
    pause(2000);
    t = 50;
}

```

```

    pause(2000);
    cog_end(cog);
}

```



When the main function runs out of commands, cog 0 also shuts down. With no cogs running, the Propeller enters a low power consumption mode and waits to be restarted. The Propeller can be restarted by loading a program, pressing/releasing the RST button, or turning the power off and back on.

This `blink` function uses the global, volatile variable `t` as the argument in its `pause` function calls. This allows functions running in other cogs to control the `pause` times, and therefore the LED blink rate, by changing the value of `t`.

```

void blink()
{
    while(1)
    {
        high(26);
        pause(t);
        low(26);
        pause(t);
    }
}

```

Try This – Make One Function Monitor Another’s Activity

We just used a volatile global variable to let a function in one cog control the behavior of a different function in a different cog. Global volatile variables can also be used to let one function monitor another function running in a different cog.

In this example, you’ll add a global variable named `reps` and make the `blink` function add 1 to it with each `while` loop repetition. Then, `main` can check the and display the value of `reps` to find out how many times the light has blinked, even though a different cog running a different function is modifying it, as shown in Figure 5-11.

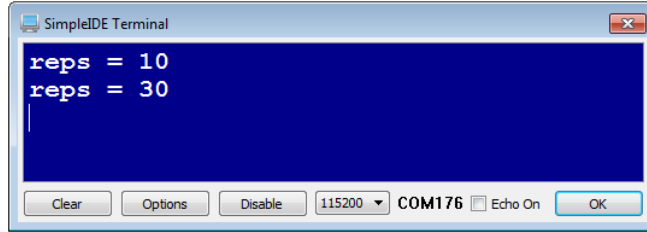


Figure 5-11
SimpleIDE
Terminal Output
Multi-
InfoExchange-
TryThis.

- Save As a copy of the project and name it Multi-InfoExchange-TryThis.
- Add a second volatile global variable named `reps`.

```
volatile int reps = 0;           // <-add
```

- Add two print statements to display the value of `reps`, one after each `pause(2000)`.

```
print("reps = %d\n", reps); // <-add
```

- Add a statement to post-increment the value of `reps` at the beginning of the `blink` function's `while(1)` loop.

```
void blink()
{
    while(1)
    {
        reps++;           // <-add
    }
}
```

- Examine the code, and consider how many times the light should blink with 100 ms pauses over 2 seconds, and then 50 ms pauses over 2 more seconds.
- Click SimpleIDE's Run with Terminal button and check the number of reps. Did it work as you expected?

ACTIVITY #6: SELF-TERMINATING COGS

Recall that functions started by `cog_run` must either consist of an infinite loop, or be self-terminating. Up to this point, the functions we've started with `cog_run` had infinite loops, and the same function that ran them also ended them. A function that runs in another cog can also end itself. This can be useful if some process only needs to run until

it has completed a list of important tasks. If the function that gets run can then tell its own cog to end, then the code in the function that launched it can just move on to other tasks without worrying about ending the process it launched.

Along with a self-terminating code example, this activity also demonstrates how to print messages from a function that another cog is running. If you ran Multi-InfoExchange-TryThis from Page 163, you already know the easiest way to display activity from another cog. Just share the info with a `volatile` variable and print it from the `main` function. The other option is to print it straight from the function the other cog is running. It's trickier than you might think because unexpected things happen when more than one cog controls the SimpleIDE Terminal's communication lines. Because of this, a function run by one cog has to close the connection with SimpleIDE terminal before the function running in another cog can open it.

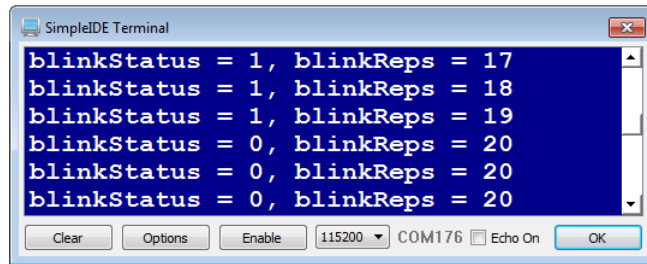
Making a Cog Self-Terminate

Making another cog self-terminate is pretty simple. Just add a `cog_end` call to the function that is running in another cog, and it will shut itself down when it's done. Yes, the function would automatically stop executing if it ran out of code, but that does not free up the cog or the stack space for re-use — `cog_end` does that.

Example Program: Cog Self-Terminates

The now familiar `blink` function in this example program has been modified to self-terminate after 20 repetitions (counting from 0 to 19). Before `blink` starts repeating, it sets a `blinkStatus` global variable to 1. It also sets `blinkStatus` to 0 just before it self-terminates. The `main` function keeps on displaying the `blinkStatus` and `blinkReps` variables. As you can see in Figure 5-12, `blinkStatus` changes to 0 after the 19th repetition. Since the cog also ends, the P26 light stops blinking and `blinkReps` stops increasing.

- Click SimpleIDE's New Project button. Then set the File name to Multi-CogEnd and Save.
- Enter the Multi-CogEnd.c code into SimpleIDE.
- Click SimpleIDE's Run with Terminal button.
- Make sure the `blink` function terminates the cog running it by verifying that: the light stops blinking, `blinkStatus` changes from 1 to 0, and `blinkReps` never makes it past 20.



```

SimpleIDE Terminal
blinkStatus = 1, blinkReps = 17
blinkStatus = 1, blinkReps = 18
blinkStatus = 1, blinkReps = 19
blinkStatus = 0, blinkReps = 20
blinkStatus = 0, blinkReps = 20
blinkStatus = 0, blinkReps = 20
Clear Options Enable 115200 COM176 Echo On OK

```

Figure 5-12
Main function in cog 0 indicates when blink function in cog 1 self-terminates.

```

/* Multi-CogEnd.c */
#include "simpletools.h" // Library include
void blink(); // Forward declaration
int *cog; // Pointer for cog data
volatile int blinkReps, blinkStatus; // Shared variables
int main() // Main function
{
  cog = cog_run(blink, 128); // Run blink in other cog
  while(1) // Endless loop
  {
    print("blinkStatus = %d, ", blinkStatus); // Display blinkStatus
    print("blinkReps = %d\n", blinkReps); // Display blinkReps
    pause(200); // Wait 1/5 s before repeat
  }
}
void blink() // Blink function for other cog
{
  blinkStatus = 1; // Set blinkStatus to 1
  blinkReps = 0; // Set blinkReps to 0
  while(blinkReps < 20) // Blink 20 repetitions
  {
    high(26); // P26 LED on
    pause(100); // ...for 0.1 seconds
    low(26); // P26 LED off
    pause(100); // ...for 0.1 seconds
    blinkReps++; // Add 1 to blinkReps
  }
  blinkStatus = 0; // Tell other cogs blink ending
  cog_end(cog); // Cog self terminates
}

```

How It Works

In addition to the now familiar `simpletools` library include, `blink` forward declaration and `*cog` pointer variable, we have two `volatile` variables named `blinkReps` and `blinkStatus`. The `blink` function will modify these values and the `main` function will print them in SimpleIDE Terminal.

```
#include "simpletools.h"

void blink();

int *cog;

volatile int blinkReps, blinkStatus;
```

As usual, the first thing the `main` function does is use `cog_run` to run the `blink` function in another cog. Then, it goes into an endless `while` loop that repeatedly displays values the `blink` function modifies.

```
int main()
{
    cog = cog_run(blink, 128);

    while(1)
    {
        print("blinkStatus = %d, ", blinkStatus);
        print("blinkReps = %d\n", blinkReps);
        pause(200);
    }
}
```

This `blink` function repeats 20-times, then self-terminates. The first thing the `blink` function does is set `blinkStatus` to 1 to let other functions in other cogs know that it's running. Then, it sets `blinkReps` to 0, and enters a loop that repeats until `blinkReps` reaches 20. Since the last statement in the loop adds 1 to `blinkReps`, it'll reach 20 after 20 repetitions. After finishing the loop, it sets `blinkStatus` back to 0 to let other functions in other cogs know it's done. Then, it self-terminates with `cog_end(cog)`.

```
void blink()
{
    blinkStatus = 1;
    blinkReps = 0;
    while(blinkReps < 20)
```

```

    {
        high(26);
        pause(100);
        low(26);
        pause(100);
        blinkReps++;
    }
    blinkStatus = 0;
    cog_end(cog);
}

```



Wait a minute, `*cog` is used in functions run by two different cogs.

Why isn't it `volatile`? It's an exception to the rule. When this `int` pointer variable was tested as a return value for `cog_run` and parameter for `cog_end`, it is not optimized out.

The only tricky situation might happen if a `while` loop monitors its value. For example, `while(cog > 0)` might wait endlessly in one cog even though another changed it to 0. If you suspect that has happened in your code, you can use `volatile int *cog` to fix that problem. The code would run, but the C compiler will display a warning that the `volatile` qualifier was discarded. To see such warnings, click the Show Build Status button near the bottom-center of the SimpleIDE window.

ACTIVITY #7: PRINTING AND TERMINATING FROM A LAUNCHED COG

If you ran `Multi-InfoExchange-TryThis`, you saw that is easy to display activity from another cog: just share the info with a `volatile` variable and print it from the `main` function. The other option is to print it straight from the function running the other cog. It's trickier than you might think, because unexpected things happen when more than one cog controls the SimpleIDE Terminal's communication lines. Because of this, the `main` function must close the connection with SimpleIDE terminal before a function running in another cog can open it.

The `simpletools` library includes another library, called `simpletext`. The `simpletext` library has `print`, `scan`, and a variety of other functions for communicating with the SimpleIDE Terminal and other devices. It has a `simpleterm_close()` function for stopping terminal communication in one cog, and a `simpleterm_open()` function for starting it in another.



The Propeller can easily hold multiple serial communication sessions with multiple devices. The tricky part is having multiple cogs take turns talking with a single device, in this case the SimpleIDE Terminal. First, cog 0 has to release the P30 pin that transmits messages to the computer's RX line before cog 1 can take over and control it.

Example Program: Multi-CogPrint

Figure 5-13 shows how the Multi-CogPrint example program starts by printing a message from `main` (run by cog 0). Then, it prints a number of messages from `blink` (run by cog 1). After `blink` is done counting from 0 to 4 (and 5 light blinks), it starts printing from `main` (cog 0) again.

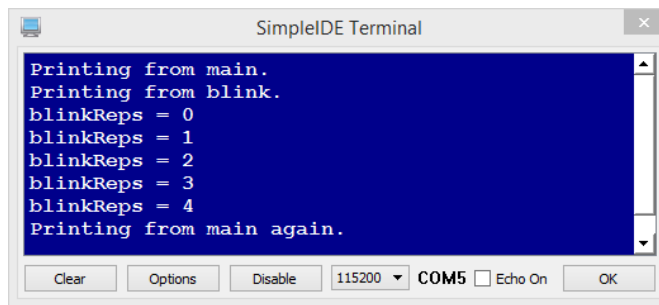


Figure 5-13
Example of printing from different cogs.

- Enter the Multi-CogPrint.c code into SimpleIDE.
- Click SimpleIDE's Run with Terminal button.
- Verify that the terminal behaves as shown in Figure 5-13, counting while the light blinks.

```

/* Multi-CogPrint.c */

#include "simpletools.h"           // Library include

void blink();                    // Forward declaration

int *cog;                        // Pointer for cog data

volatile int blinkReps, blinkStatus; // Shared global variables

int main()                       // Main function
{
    print("Printing from main.\n"); // Message from main
    simpleterm_close();           // Close terminal COM in cog 0
    cog = cog_run(blink, 128);    // Run blink in other cog
}

```

```

while(blinkStatus == 0);           // Wait for cog 1 to start
while(blinkStatus == 1);           // Wait for cog 1 to finish
simpleterm_open();                  // Safe to reopen terminal COM
print("Printing from main again.\n"); // Message from main
}

void blink()                        // Blink function for other cog
{
    blinkStatus = 1;                // Set blinkStatus to 1
    simpleterm_open();              // Open terminal COM in cog 1
    print("Printing from blink.\n"); // Messages from blink
    blinkReps = 0;                  // Set blinkReps to 0
    while(blinkReps < 5)            // Blink 5 repetitions
    {
        print("blinkReps = %d\n", blinkReps); // Display blinkReps
        high(26);                       // P26 LED on
        pause(100);                      // ...for 0.1 seconds
        low(26);                         // P26 LED off
        pause(100);                      // ...for 0.1 seconds
        blinkReps++;                    // Add 1 to blinkReps
    }
    simpleterm_close();              // Close terminal COM in cog 1
    blinkStatus = 0;                 // Tell other cogs blink ending
    cog_end(cog);                    // Cog self terminates
}

```

How it Works

The program starts with the familiar forward declaration for the `blink` function, and a pointer variable named `cog` for launching it. Next come two volatile global variables for sharing data between cogs: `blinkReps` and `blinkStatus`.

```

void blink();

int *cog;

volatile int blinkReps, blinkStatus;

```

The `main` function begins with a `print` statement, from the `main` function running in cog 0. Next comes `simpleterm_close`, which releases the serial terminal connection so it can be opened from a function running in a different cog, such as `blink`. Next the `cog_run` call starts the `blink` function in cog 1.

```

print("Printing from main.\n");
simpleterm_close();
cog = cog_run(blink, 128);

```

The next two instructions in `main` are conditional loops that are controlled by the value of `blinkStatus`, which is being changed by the `blink` function running in another cog. The first loop, `while(blinkStatus == 0);` translates to “stay here and keep checking the value of `blinkStatus` as long as it is equal to zero.” Once `blinkStatus` changes from 0 to 1, code execution moves on to the next line: `while(blinkStatus == 1)` and code execution loops here until `blinkStatus` changes back to zero. Only then does `simpleTerm_open` re-establish the terminal connection to cog 0, just in time for a final `print` statement.

```

    while(blinkStatus == 0);
    while(blinkStatus == 1);
    simpleterm_open();
    print("Printing from main again.\n");
}

```

Since the rule is to only allow one cog to send messages to SimpleIDE terminal at any given time, the code needs to make sure that cog 0 does not try to re-open the serial terminal while the `blink` function is still using it in cog 1. That’s where the `blinkStatus` variable comes in.

The first thing the `blink` function does is set `blinkStatus` to 1, which makes the `main` function stay in that first conditional loop. Then, `blink` uses `simpleterm_open` to open the connection with the SimpleIDE terminal in this cog. This allows the text to be seen in the `print` command that follows.

```

void blink()
{
    blinkStatus = 1;
    simpleterm_open();
    print("Printing from blink.\n");
    blinkReps = 0;
    while(blinkReps < 5)
    {
        print("blinkReps = %d\n", blinkReps);
        high(26);
        pause(100);
        low(26);
        pause(100);
        blinkReps++;
    }
}

```

After `blinkReps` gets to 5 and the loop is finished, the `blink` function uses `simpleterm_close` to release control over the serial connection. Only then does it set `blinkStatus` back to 0, which allows the program execution back in `main` to exit its second `while(1)` loop. The last thing the `blink` function does is shut itself down with `cog_end(cog)`. By using `simpleterm_close()` and `cog_end(cog)`, this cog has properly released all of its resources for re-use.

```
simpleterm_close();
blinkStatus = 0;
cog_end(cog);
}
```



A Semaphore Variable

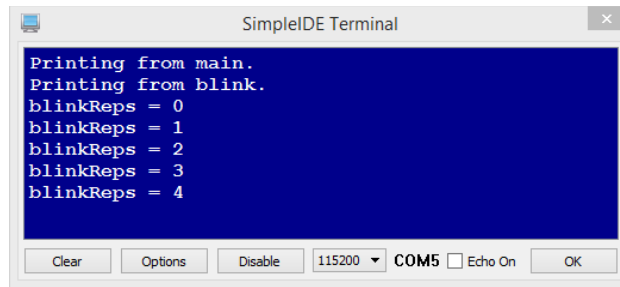
The `blinkStatus` variable is an example of a *semaphore*. A semaphore variable is used to prevent more than one processor from using a given resource at the same time.

Try This – Test Volatile

With one small change to `Multi-CogPrint.c`, you can examine a bug that happens by forgetting to use `volatile` when declaring global variables that cogs will share.

- Click SimpleIDE's Save Project As button, and rename the project Multi-CogPrint-TryThis.
- Remove the `volatile` modifier from the `volatile int blinkReps, blinkStatus;` statement.
- Click the Run with Terminal button.
- Find the item missing from the display. Can you guess what happened?

Figure 5-14 shows what's missing. Compare it to Figure 5-13 and you'll see that it doesn't say, "Printing from main again."

**Figure 5-14**

Forgetting `volatile` causes the program to get stuck

The reason it doesn't say "Printing from main again." is because the `main` function gets stuck in the `while` loops that wait for `blinkStatus` to change from 0 to 1 and back to 0 again.

```
while(blinkStatus == 0);
while(blinkStatus == 1);
```

The first `while` loop waits for the `blink` function to change `blinkStatus` from 0 to 1. The C compiler doesn't know that `blink` is running in another processor. Because the variable is no longer `volatile`, the compiler thinks that `main` would have to call the `blink` function for that variable to change. So, it removed code to repeatedly recheck the value of `blinkStatus` from `while(blinkStatus == 0);`. As a result, the `main` function will think `blinkStatus` is still 0 even after the `blink` function running in another cog changes it to 1.

That's why it's important to declare variables that are used by more than one function in more than one cog as `volatile`.

SUMMARY

This chapter introduced some common C language programming techniques, including function writing and setting variable scope. It then applied those concepts in programs that utilized the Propeller microcontroller's multiprocessing design to make different processors (cogs) execute code in different functions at the same time.

Key concepts:

- A function and its components: definition, return type, name, parameter list, and code block with statements.

- Function forward declaration, call and return.
- Function parameter passing and return value.
- Using a function named `cog_run` to run a function in another cog by passing function pointer and stack size as parameters. Using a function named `cog_end` to end the cog's execution of code in that function.
- Running more than one additional function in more than one available cog.
- Variable scope: local vs. global variables.
- Using global variables with the `volatile` keyword to make functions in different cogs monitor and control each other.
- Stopping a cog from the `main` routine, and also from the launched cog.
- Printing from multiple cogs by opening and closing their access to the terminal.

Questions

1. In this function prototype, what is its name, parameters, and return type?
`float addfloat(float a, float b);`
2. What is a forward declaration? Where does it go in programs?
3. What's the difference between a function call and return?
4. What's the difference between a parameter and a return value?
5. What function can your code use to launch a cog, and what information does it need? What information does it return and how can your code use it?
6. What kind of variable do you need for cogs to exchange information?
7. If a variable is declared inside a function, what is its scope?

Exercises

1. Write a function named `blinker` for launching into another cog, that allows another function to determine which LED blinks along with both high and low times and monitors the number of times the light has blinked. Assume your global variables are `pin`, `tHigh`, `tLow`, and `reps`.
2. Write the variable declarations and function prototype for `blinker`.
3. Write a call to launch `blinker`. Stay safe, set the stack to 128. Use a pointer `int` variable named `myCog`, and write a call to end `blinker` using `myCog`.

Project

1. Write an application that allows you to configure the `blinker` function from the `main` function. Have your application initialize `tHigh` to 50 and `tLow` to 200. It should ask for the LED pin once, and then repeatedly ask for `tHigh` and `tLow`.

Solutions

- Q1. Name is `addfloat`, return type is `float`, and parameters are `float a` and `float b`.
- Q2. A forward declaration tells the C compiler to expect a function with that prototype later in the program. It is normally added before any executable functions.
- Q3. A function call tells the program to find the function, execute its code, and come back when done. The return is simply the part where the code “returns” to the function call and continues executing code from there.
- Q4. A parameter gets passed to a function by a function call. A return value gets passed back by the function.
- Q5. `cog_run` can launch a cog given the address of the function to launch `functionName`, and a number of `int` size slots to set aside for the cog’s stack space. `cog_run` returns a pointer to the place in memory where it stores the cog number and stack space. This value can be used later by `cog_end` stop the process to recover the cog and stack space for other uses.
- Q6. A global variable that has been declared `volatile`.
- Q7. Local.

E1.

```
void blinker()
{
    while(1)
    {
        reps++;
        high(pin);
        pause(tHigh);
        low(pin);
        pause(tLow);
    }
}
```

E2. `volatile int pin, tHigh, tLow, reps;`

E3.

```
int *myCog;
...
myCog = cog_run(blinker, 64);
```

```
...
cog_end(myCog);
```

P1.

```
/* Multi-P1-Solution.c */

#include "simpletools.h"           // Library include

void blinker();                 // Forward declaration

int myCog;

volatile int pin, tHigh, tLow, reps; // Cog share variables

int main()                      // Main function
{
    print("Enter pin: ");       // Get pin
    scan("%d", &pin);

    tHigh = 50;                // Initialize t values
    tLow = 200;

    myCog = cog_run(blinker, 64); // Run blink in other cog

    int tHighTemp, tLowTemp;    // Temporary variables

    while(1)                   // Main loop
    {
        print("Enter tHigh: "); // Get high time
        scan("%d", &tHighTemp);
        print("Enter tLow: "); // Get low time
        scan("%d", &tLowTemp);

        tHigh = tHighTemp;     // Update blinker cog.
        tLow = tLowTemp;
    }
}

void blinker()                  // Function for other cog
{
    while(1)                   // Endless loop
    {
        reps++;
        high(pin);             // LED on
        pause(tHigh);          // ...for tHigh ms
        low(pin);              // LED off
        pause(tLow);           // ...for t ms
    }
}
```


Chapter 6: Measure Voltage and Position

Control knobs are used in all kinds of equipment. Think of adjustable lighting: twist a knob in one direction and the lights get brighter, twist it in the other direction, and the lights get dimmer. Think of machines that have control knobs for fine-tuning the position of cutting blades and guiding surfaces. Think of audio equipment, where turning a knob adjusts how music and voices sound. Can you see any more examples from where you are right now?

Figure 6-1 shows a control knob on a speaker that adjusts volume. Turning the knob adjusts a circuit inside the speaker, which in turn changes the audio volume.



Figure 6-1
Volume Adjustment on a Speaker

THE VARIABLE RESISTOR – A POTENTIOMETER

The device under many control knobs is a variable resistor called a *potentiometer*, often abbreviated as a “pot.” They are also used inside equipment where you may not see them, such as joysticks and even inside the servo you used in Chapter 4. Figure 6-2 shows a picture of some common potentiometers. Notice that they all have three pins.

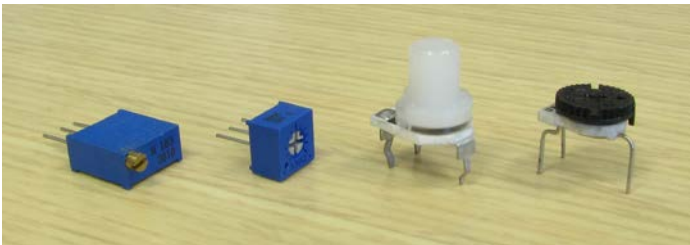


Figure 6-2
A Few Potentiometer Examples

Figure 6-3 shows the schematic symbol and part drawing of the potentiometer you will use in this chapter. Terminals A and B are connected to a 10 k Ω resistive element. Terminal W is called the *wiper terminal*, and it is connected to a wire that touches the resistive element somewhere between its ends.

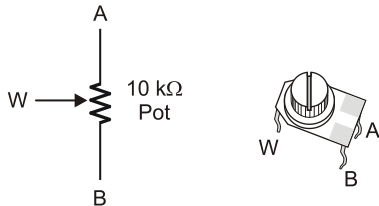


Figure 6-3
Potentiometer Schematic Symbol
and Part Drawing

Figure 6-4 shows how the wiper on a potentiometer works. As you adjust the control knob on top of the potentiometer, the wiper terminal contacts the resistive element at different places. Turning the knob clockwise moves the wiper closer to terminal A. This decreases the resistance between the wiper and terminal A, and increases the resistance between the wiper and terminal B. Turning the knob counterclockwise decreases the resistance between the wiper and terminal B, and increases the resistance between the wiper and terminal A.

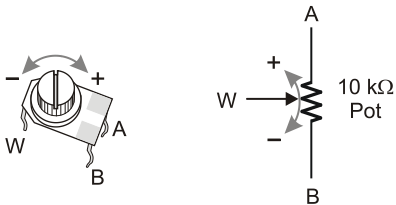


Figure 6-4
Adjusting the Potentiometer's Wiper
Terminal

ACTIVITY #1: SET VOLTAGES WITH TWO RESISTORS

A potentiometer works like two resistors in a row with a wire 'tap' between them. Turning the control knob essentially changes the value each of the resistors. So, let's use actual resistors and a wire to get a better idea of how the circuit inside a potentiometer works.

Voltage Divider Circuit

When voltage is applied to two resistors in series (connected end-to-end), the values of the resistors determine the voltage that appears between them. The circuit for setting voltage like this is called a *voltage divider*. The equation for finding the voltage at point V_O between the two resistors is shown in Figure 2-10, along with the circuit.

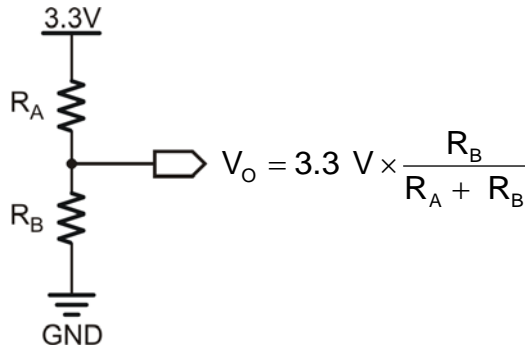


Figure 6-5

Voltage Divider Circuit and Equation, using 3.3 V supply on the Propeller Activity Board

This equation is really easy to use. Let's say you've got two 1 k Ω resistors; that would mean $R_A = 1000$ and $R_B = 1000$. So, let's substitute the 1000 for R_A and R_B and see what happens.

$$\begin{aligned} V_O &= 3.3 \text{ V} \times \frac{1000}{1000 + 1000} \\ &= 3.3 \text{ V} \times 0.5 \\ &= 1.65 \text{ V} \end{aligned}$$

In this case, voltage V_O between the resistors is 1.65 V, which is half the voltage applied across both.

So, what would happen if you made R_B 2 k Ω and left R_A at 1 k Ω ?

$$\begin{aligned} V_O &= 3.3 \text{ V} \times \frac{2000}{1000 + 2000} \\ &= 3.3 \text{ V} \times 0.66\dots \\ &= 2.2 \text{ V} \end{aligned}$$

What would happen if you swap resistors so $R_A = 2 \text{ k}\Omega$ and $R_B = 1 \text{ k}\Omega$?

$$\begin{aligned} V_O &= 3.3 \text{ V} \times \frac{1000}{2000 + 1000} \\ &= 3.3 \text{ V} \times 0.33\dots \\ &= 1.1 \text{ V} \end{aligned}$$

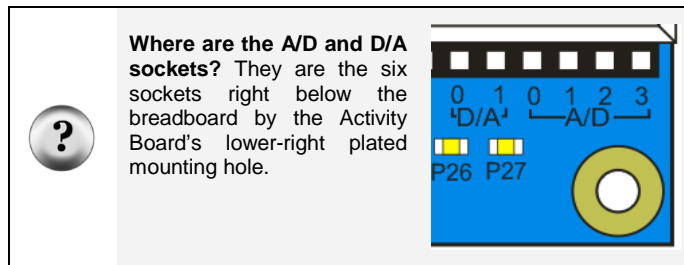
Now imagine substituting the potentiometer for the 2-resistor circuit, with terminal A connected to 3.3 volts and terminal B connected to GND. The wiper divides the potentiometer's resistive material into two sections, which you can think of as R_A and R_B . Moving the wiper would change the resistors' values, making one resistor larger and one smaller, and thus changing the value at V_O (though $R_A + R_B$ would always equal 10 k Ω).

Voltage Divider Parts

- (2) Resistors – 1 k Ω (brown-black-red)
- (1) Resistor – 2 k Ω (red-black-red)

First Voltage Divider Circuit

The Propeller Activity board has four sockets for measuring voltage, labeled A/D0, A/D1, A/D2, and A/D3.



A/D stands for *analog to digital*. Like most natural phenomena, actual voltage values vary continuously. However, in the electronics world, digital values tend to take discrete steps. The A/D converter chip on the Activity Board converts the analog voltage value received into a digital measurement that the Propeller chip can work with: a number of 4096^{ths} of 5 V. The A/D converter rounds down to the nearest 4096th. For example, if it receives a voltage somewhere between 1351/4096 and 1352/4096 of 5 V, it will round

down to an even 1351 before passing the value to the Propeller microcontroller. These digital measurements are called *quantized* values, meaning rounded to discrete steps.

- ❑ Connect the voltage divider shown in Figure 6-6.

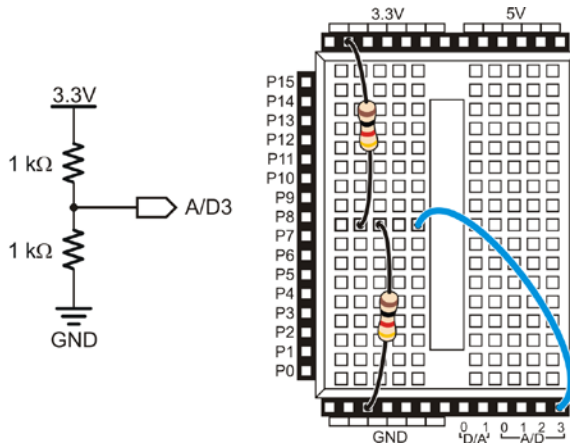


Figure 6-6
Voltage Divider
Schematic and
Wiring Diagram

Example Program: Volts-DividerVoltage

We will be using a Simple Library named `abvolts` that was written specifically for the A/D and D/A hardware on the Propeller Activity Board. The functions in `abvolts` can take the quantized value from the A/D chip and convert it to an easy-to-read voltage value; for example, 1351 becomes 1.649 V. Let's try it, verifying our previous 1.65 V calculation for the voltage divider between two 1 kΩ resistors in series.

Don't expect the output to be exactly 1.65 V! Yours may be a bit off, as is the example in Figure 6-7 shows. Back in Chapter 2, we introduced the tolerance color band on the resistor, with a gold band indicating that it's good to 5%. Since 5% of 1000 is 50, a 1 kΩ resistor with a 5% tolerance could actually measure between 950 and 1050 Ω. Putting these numbers back into our voltage divider equation shows that our measurement could be as low as 1.57 V or as high as 1.73 V.

Other contributors to inexact values include the tolerance of the 5 V voltage regulator on your Propeller Activity Board, and the rounding operation that the A/D converter performs.

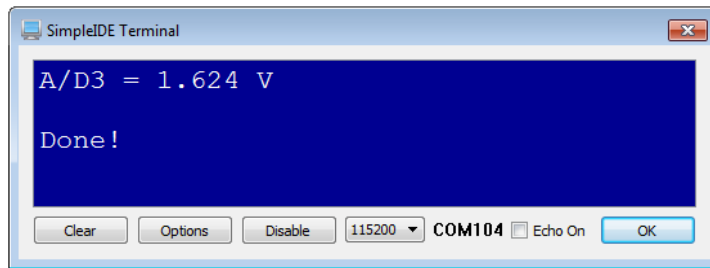


Figure 6-7
Measured
Voltage

- Click SimpleIDE's New Project button, save the project as Volts-DividerVoltage and save it to My Projects.
- Enter the Volts-DividerVoltage.c code into SimpleIDE.
- Reconnect power to your Activity Board.
- Click SimpleIDE's Run with Terminal button.
- Verify that your measurement is close to 1.65 V, plus or minus 0.15 V.

```

/* Volts-DividerVoltage.c */

#include "simpletools.h"           // Library includes
#include "abvolts.h"             // Must include to use abvolts

int main()                       // Main function
{
    float volts = ad_volts(3);    // Get A/D3 volts

    print("A/D3 = %1.3f V\n", volts); // Display result

    print("\nDone!");           // Announce program done
}

```

How Volts-DividerVoltage Works

In addition to `simpletools.h`, a second `#include` statement adds the library `abvolts.h`. It gives the program access to the `ad_volts` function.

```

#include "simpletools.h"
#include "abvolts.h"

```

Inside `main`, the first function call is to `ad_volts`. This function returns a float variable value, and its parameter requires an A/D socket number, 0–3.

```

int main()

```

```
{
  float volts = ad_volts(3);
```

This call makes the Propeller fetch the quantized voltage value from the AD3 socket, which is in terms of 4096^{ths} of 5 V (even though our circuit is connected to 3.3V). The function returns a floating-point decimal value in volts, which is why the function call began with `float volts`.

The value of `volts` is then sent for display on SimpleIDE Terminal with `print("A/D3 = %1.3f V\n", volts)`. You may not have seen the `%1.3f` formatting flag before. It's a variation of `%f` (display floating point flag) that allows you to specify the number of digits to the left and right of the decimal point. In this case, it displays the floating point `volts` variable with 1 digit to the left and 3 to the right.

```
print("A/D3 = %1.3f V\n", volts);
```

Since this is the first program we've run in a while that hasn't used a loop, the "Done!" message provides a cue not to expect any more output.

```
print("\nDone!");
}
```



Leading with zeroes or spaces, your choice. The example above only needed to print one digit to the left of the decimal, and so it used `%1.3f`. For larger number ranges, you can specify more digits, and whether to pad smaller values with spaces or zeroes. For example, `%3.2f` in a `print` statement accommodates three digits to the left of the decimal point, and it will pad a value like 1.23 with two leading spaces before the 1. If you instead want it to pad with leading zeroes, use `%03.2f`. This will print the value 1.23 like this: 001.23.

Your Turn – Different Voltage Dividers

Okay, so we've verified that two resistors of the same size will give you half the voltage at V_O . Next, let's verify the 1.1 V and 2.2 V dividers.

- Modify the circuits using the schematics in Figure 6-8, with the aid of the wiring diagrams if needed. Remember to turn off power when changing circuits.
- Use the Run with Terminal button to re-run Volts-DividerVoltage and verify each circuit's voltage.

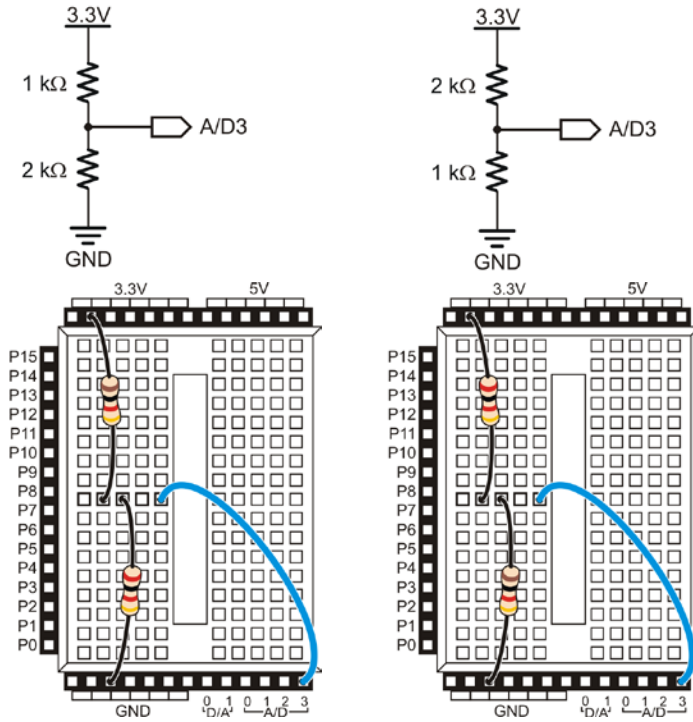


Figure 6-8
Two More Voltage
Divider Circuits

ACTIVITY #2: READ THE POSITION WITH THE PROPELLER

Figure 6-9 shows a conceptual drawing inside of the potentiometer, as if the control knob were transparent. A semicircular resistive element connects to the A and B terminals. The wiper is a contact that swivels with the knob while maintaining electrical contact with a second lead connected to the W terminal. Each time you turn the knob to a new position, you make the wiper touch a new point on the resistive element. The wiper contact creates two resistors in series: one from B to W and the other from W to A. So, if you connect 3.3 V to B, and GND to A, you can connect W to A/D3, and cause the voltage divider to vary as you twist the knob. Let's try it.

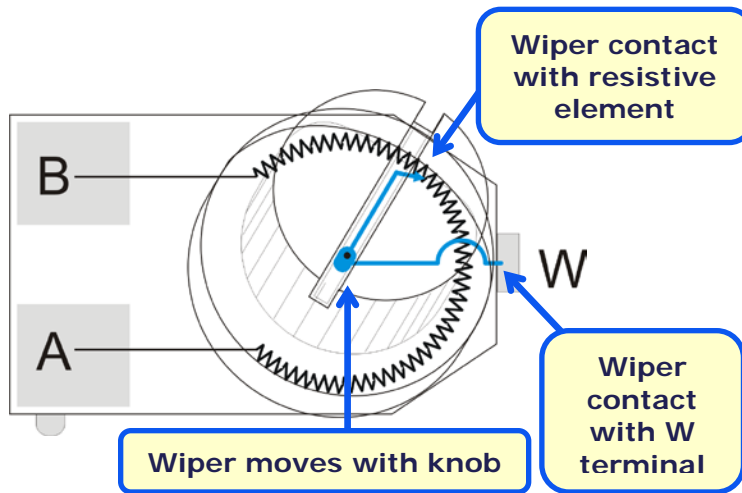


Figure 6-9
 Concept
 Drawing —
 Inside the
 Potentiometer

Potentiometer Parts

- (1) Potentiometer – 10 k Ω
- (3) Jumper wires – red, black, and blue

Potentiometer Circuit

Figure 6-10 shows a schematic and wiring diagram for a potentiometer voltage output circuit. This circuit should allow you to turn the knob from its clockwise limit to its counterclockwise limit, for voltage measurements ranging from about 0 V to just less than 3.3 V.

- Build the circuit shown in Figure 6-10.

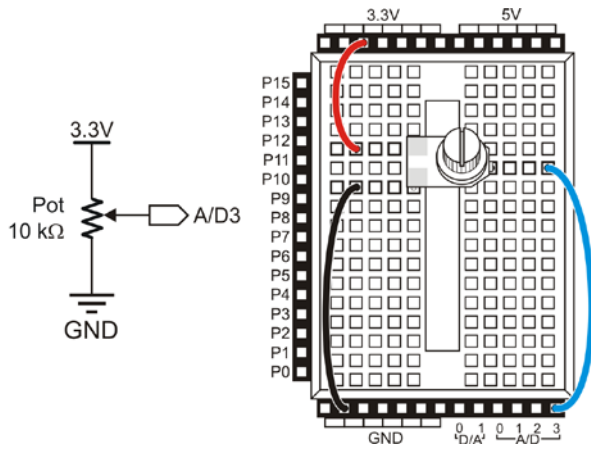


Figure 6-10
Potentiometer
Schematic and Wiring
Diagram

Potentiometer Test Code

The voltage measurement will display on the same line and refresh 5 times per second, similar to Figure 6-11. As you twist the potentiometer's knob, make sure to apply some downward pressure to make it maintain contact with the breadboard sockets.

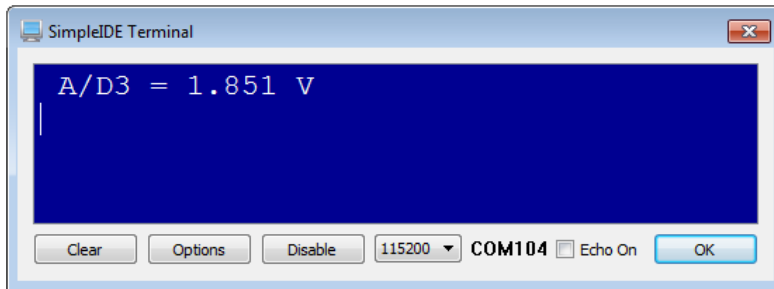


Figure 6-11
Potentiometer
Voltage Display

Example Program: Volts-Monitor

- Click SimpleIDE's New Project button, name the project Volts-Monitor, and save it to My Projects.
- Enter the Volts-Monitor.c code into SimpleIDE.
- Click SimpleIDE's Run with Terminal button.

- ❑ Turn the control knob all the way clockwise and verify that the voltage is close to 0 V, since the current is going through the full length of the potentiometer's resistive element.
- ❑ Gradually turn the knob counterclockwise, and monitor the voltage. Does it gradually increase to almost 3.3 V at about the time it reaches its counterclockwise limit? There is very little resistive element between the wiper and the A lead which goes to 3.3 volts



Make sure to apply a little downward pressure to keep the potentiometer seated on the breadboard as you twist its knob. (Using pliers to make a very, very gentle 1/4 turn twist in the thin part of each lead will help the potentiometer stay in the breadboard sockets, but do this at your own risk to your potentiometer!)

```

/* Volts-Monitor.c */

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

int main()                       // Main function
{
    float volts;

    while(1)
    {
        volts = ad_volts(3);      // Get A/D3 volts
        print("%c A/D3 = %1.3f V ", // Display result
              HOME, volts);
        pause(200);             // 200 ms pause
    }
}

```

How it Works

The `main` routine starts by declaring a floating-point variable named `volts`.

```

int main()
{
    float volts;

```

Each time through the `while(1)` loop, `volts = ad_volts(3)` measures and stores the voltage applied to the A/D3 socket, this time by the potentiometer, in `volts`. Next, the statement `print("%c A/D3 = %1.3f V ", HOME, volts)` sends the cursor to the SimpleIDE Terminal's top-left home position with the first `%c` and `HOME`. Then, `%1.3f`

and `volts` displays the `volts` variable with 1 character to the left of the decimal point, and three to the right.

```
while(1)
{
    volts = ad_volts(3);
    print("%c A/D3 = %1.3f V",
          HOME,      volts);
    pause(200);
}
```



Why isn't there a `CLREOL` at the end? We used to need `CLREOL` when overprinting the same line while displaying a number with a variable number of digits. That was because a 2 digit number displayed after a 3 digit number would not overprint the last digit. `CLREOL` used to clear all the text to the right, but it just isn't needed when the number of digits displayed never changes.

`%1.3f` displays 1 digit, a decimal point, and 3 more digits, every time.

Try This – Display Actual A/D Values

As mentioned earlier, the Activity Board's A/D converter chip outputs measured voltage as a number of 4096^{ths} of 5 V. This next example uses a function named `ad_in` to get those raw measurements. Since your potentiometer is only wired to display up to 3.3 V, we can expect values from 0 to about 2703. That's because $3.3 \times 4096 / 5 = 2703$.

```
int volts;                                // <-change here

while(1)
{
    volts = ad_in(3);                      // <-change here
    print("%c A/D3 = %4d 4096ths of 5 V ", // <-change here
          HOME,      volts);
    pause(200);
}
```

- Use SimpleIDE's Save Project As button to save a copy of the project in Your My Projects folder. Name it Volts-Monitor-TryThis.
- Modify it as shown above to get the raw A/D converter value and display it.
- Verify that your measurement range is now 0...2703 (instead of 0.0...3.29).

Your Turn - Casting a Variable

The `ad_volts` function receives an integer value (`int`) from the ADC, but returns the voltage as a floating-point decimal value (`float`). How does it do that?

In C language, you can *cast* a value as it is copied from one type of variable (such as `int`) to another type of variable (such as `float`) as needed for operations. The statement `float fvolts = (float) volts` casts the value stored by the `int` variable `volts` to the `float` type while copying it to `fvolts`. Then, the value can be printed with the floating-point formatter `%1.3f`.

```
float fvolts = (float) volts;
fvolts = fvolts * 5.0 / 4096.0;
print("\n fvolts = %1.3f V", fvolts);
```

- Save another copy of your program as Volts-Monitor-YourTurn.
- Declare a float variable named `fvolts`.
- Add the above code to the `while` loop so that it displays both the integer and floating point measurements.

ACTIVITY #3: CALIBRATE D/A OUTPUTS

The D/A sockets are the counterparts to the A/D sockets — they are for setting voltages instead of measuring them. Each D/A socket has an LED indicator that gets brighter with higher voltages and dimmer with lower voltages. In this activity, you will calibrate your board's D/A voltage outputs, write programs to set their voltages, and measure and compare calibrated and un-calibrated voltage output levels.

Additional Parts

(2) Jumper Wires

D/A Calibration Setup

Figure 6-12 shows where to add the jumper wires. Make sure to use one jumper wire to connect D/A0 to A/D0 and the other to connect D/A1 to A/D1.

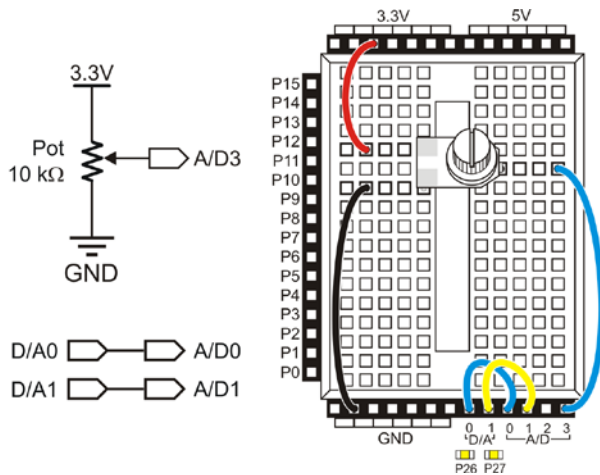


Figure 6-12
Pot Circuit with Jumper
wires added: D/A0 to
A/D0 and DA/1 to AD/1

D/A Calibration

The `abvolts` library includes functions such as `da_out` and `da_volts` to make D/A0 and D/A1 maintain a given voltage level between 0 V to just less than 3.3 V. This is in contrast to A/D inputs that can measure from 0 to just less than 5 V. Second, output voltages are set in terms of 256ths of 3.3 V. The A/D converter reports measurements as 4096ths of 5 V. Third, a one-time calibration is needed for best results with the `da_volts` function. In contrast, the A/D has built-in voltage references, so it does not need calibration.

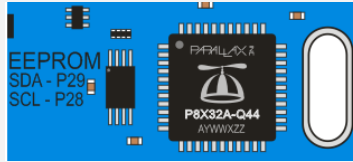
The one-time calibration uses a function named `da_setupScale` to measure D/A voltage outputs with the A/D inputs. The function uses differences between the expected and measured output levels to set up a scale factor for improving the accuracy of the D/A voltage outputs. The `da_setupScale` function saves these scale factors in a portion of the Activity Board's EEPROM that is reserved for the `abvolts` library. After the calibration, programs can call a function named `ab_useScale` to fetch the scale factors from EEPROM and apply them to improve `da_volts` outputs.

What is EEPROM Memory?

EEPROM stands for Electrically Erasable Programmable Read-Only Memory. It keeps its values even when you turn power off; turn it back on and all the information is still there. In contrast, RAM (Random Access Memory) loses all its values when you shut down power, or even when you press and release the Activity Board's RST button.

How does the Propeller use EEPROM?

The Propeller Activity Board has a 64 KB EEPROM. It's the little black chip just to the left of the Propeller. The upper-case K means it can store the nearest power of 2, which is $2^{16} = 65,536$ bytes.



Half of this EEPROM (the first or "lower" 32 KB = 32,768 bytes) is dedicated to storing program images when you use SimpleIDE's Load EEPROM & Run button. After a reset, the Propeller will wake up and detect that a computer is not trying to load a program, so it goes and gets the program from EEPROM memory.

How do libraries use EEPROM?

Our example Propeller programs get stored in lower EEPROM, leaving the "upper" 32,768th through 65,535th bytes for data storage. The Propeller C Tutorials' Simple Libraries store Activity Board related calibration data using the highest addresses, starting at the 65,535th byte and working downward as libraries are added. Examples include compass, abdrive (for the Propeller ActivityBot), and abvolts calibration values. The abvolts library was added to the Simple Libraries most recently, so it occupies the lowest addresses, from the 63,400th byte to the 63,416th bytes.

How can you use EEPROM?

You will also use EEPROM to store data later in this tutorial. To make sure your values don't roll over any calibration data, we'll use the 32,768th byte—the lowest byte in upper EEPROM—and work our way upward. We'll restrict the amount of data so that it doesn't roll over the 63,400th byte. For storing more data than that, we could switch to SD cards for data storage. In the future, it is good practice to check each library's documentation for upper EEPROM usage to prevent conflicts.

Example Program: Volts-CalibratedA

Running the next program should give you a message with scale factors resembling those shown in Figure 6-13, with values very close to 1.0. This message indicates that your scale factors have been determined and saved to the Activity Board's upper EEPROM memory, for use in later programs.

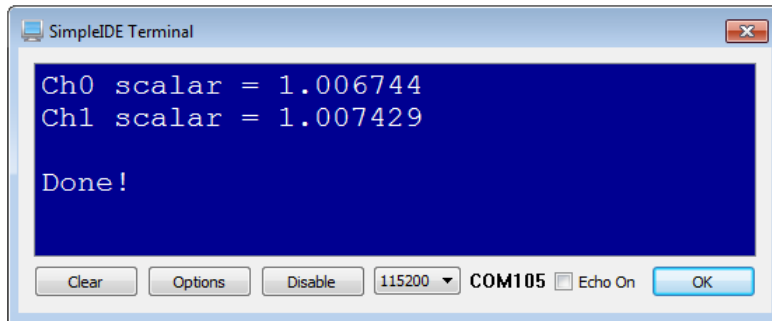


Figure 6-13
Scale Factors
from
da_setupScale
Function

If you see the “Error!” message in Figure 6-14 instead, it might mean that there’s a wiring mistake. It could also mean that the USB port isn’t supplying enough power, which can happen with an un-powered USB hub. It can also happen with pre-USB 2.0 ports. Plugging in an external power supply can help rule out USB port supply problems.

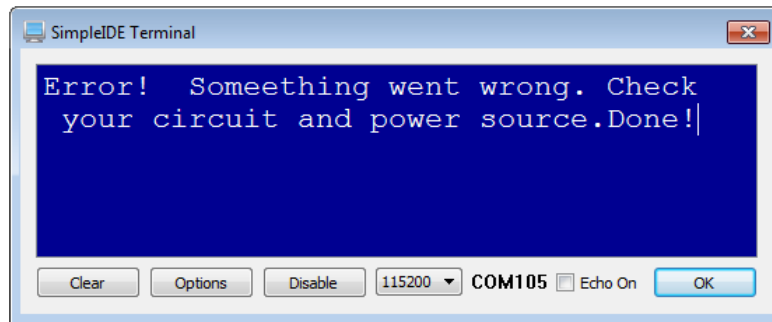


Figure 6-14
Error Message
from
da_setupScale
Function

- Click SimpleIDE’s New Project button, name the project Volts-CalibrateDA, and save to My Projects.
- Enter the Volts-CalibrateDA.c code into SimpleIDE.
- Reconnect power to your Activity Board (PWR switch to 1 and USB cable connected).
- Click SimpleIDE’s Run with Terminal button.
- If you get the message displaying the scale factors, you’re ready to move on.
- If you get the error message, double-check your wiring. If your wiring is correct, try plugging in an external power supply to the Activity Board’s 6-9 V power jack. Power supply options are in Figure 4-4 on page 106.


```

/* Volts-CalibrateDA.c */

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

int main()                       // Main function
{
    da_setupScale();             // Calibration function

    print("Done!");             // Done message
}

```

How it Works

Instead of a dedicated D/A chip, the Activity Board relies on some simple circuits and the Propeller microcontroller's signaling ability to set voltages. The `da_volts` function causes the Propeller to send rapid sequences of high/low signals to D/A conversion circuits on the Activity Board.

Like the resistors, the various parts in the D/A circuits have tolerances that affect its ability to supply exactly 3.3 V when a Propeller I/O pin sends it a high signal, hence the need for calibration. For example, if the high output of these conversion circuits is only 3.2 V instead of 3.3 V, all the D/A conversions will be 3.2/3.3 of what they should be. The `da_volts` function could correct this by multiplying whatever voltage it's supposed to supply by $3.3 / 3.2$, which could be called a *scale factor* or *scalar*.

When your code calls the `da_setupScale` function, it figures out the scale factor that is needed to correct the `da_volts` output on your particular board. It does this by sending high signals to P26 and P27, which are the I/O pins that send signals to the D/A0 and D/A1 sockets. Since you connected D/A0 to A/D0 and D/A1 to A/D1 with wires, `da_setupScale` measures the voltages and divides them into 3.3 V to calculate the scale factors, which get stored in the Activity Board's EEPROM memory.

```

int main()
{
    da_setupScale();

    print("Done!");
}

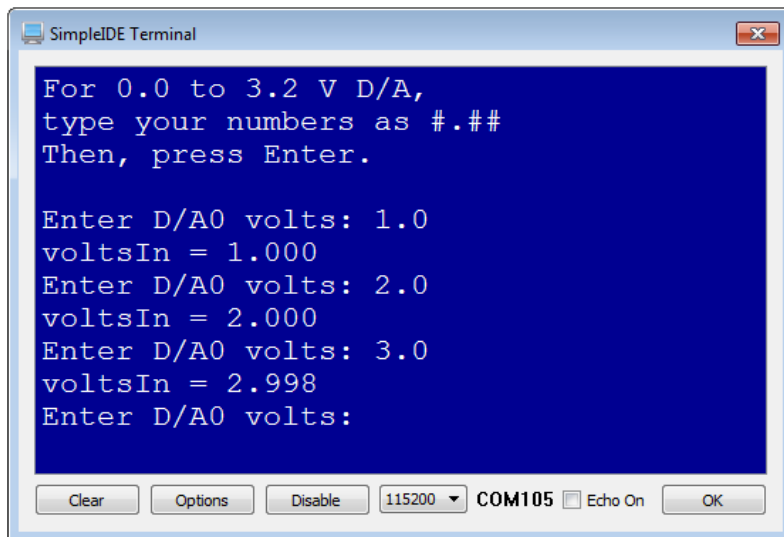
```

After doing this, you can call `da_useScale` at the beginning of any program that includes the `abvolts` library. Then, `da_useScale` will fetch those scale factors from

upper EEPROM and copy them to variables that `da_volts` automatically uses to correct its output.

Test the D/A with the A/D

This next program tests the scale correction by prompting you to type in voltage values, like those shown in Figure 6-15. The measured values should be very close to the requested values.



```

SimpleIDE Terminal
For 0.0 to 3.2 V D/A,
type your numbers as #.##
Then, press Enter.

Enter D/A0 volts: 1.0
voltsIn = 1.000
Enter D/A0 volts: 2.0
voltsIn = 2.000
Enter D/A0 volts: 3.0
voltsIn = 2.998
Enter D/A0 volts:
  
```

Figure 6-15
Voltage
Measurements
with useScale

Example Program: Volts-DAConversion

- Click SimpleIDE's New Project button, name the project Volts-DAConversion, and save it in My Projects.
- Enter the Volts-DAConversion.c code into SimpleIDE.
- Click SimpleIDE's Run with Terminal button.
- Try entering 1.0, 2.0, and 3.0 as test values.
- Verify that the SimpleIDE Terminal displays `voltsIn` values that are very close to what you typed.
- Also, monitor the P26 LED. It should get brighter after you enter a larger voltages and dimmer as you enter smaller voltages.

```

/* Volts-DAConversion.c */

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

int main()                       // Main function
{
    da_useScale();               // Get scale factors

    print("For 0.0 to 3.2 V D/A,\n"); // User instructions
    print("type your numbers as #.##\n");
    print("Then, press Enter.\n\n");

    float voltsOut, voltsIn;     // Voltage variables

    while(1)
    {
        print("Enter D/A0 volts: "); // Get voltage
        scan("%f\n", &voltsOut);

        da_volts(0, voltsOut);     // Set volts

        voltsIn = ad_volts(0);     // Measure volts

        print("voltsIn = %1.3f\n", voltsIn); // Display measurement
    }
}

```

How it Works

The `da_useScale` function tells the `abvolts` library to use the scale factors saved earlier by `da_setupScale`.

```
da_useScale();
```

These are just instructions for how to enter the D/A values and what range to use.

```

print("For 0.0 to 3.2 V D/A,\n");
print("type your numbers as #.##\n");
print("Then, press Enter.\n\n");

```

Next, floating point variables for D/A (`voltsOut`) and A/D (`voltsIn`) are declared.

```
float voltsOut, voltsIn;
```

At the beginning of the `while` loop, a `print` statement prompts you to enter a voltage. Then, a `scan` statement fetches the value you entered into the SimpleIDE Terminal and stores it in `voltsOut`. Note that the `scan` statement uses the `%f` modifier to make sure what you entered is captured as floating point value.

```
while(1)
{
    print("Enter D/A0 volts: ");
    scan("%f\n", &voltsOut);
```

The `da_volts(0, voltsOut)` statement sets the D/A0 pin to the floating point value of `voltsOut`. If it were instead `da_volts(1, voltsOut)`, it would set the D/A1 channel.

```
da_volts(0, voltsOut);
```

The `voltsIn = ad_volts(0)` statement measures the volts that D/A0 applies to A/D0 with the wire you connected at the start of this activity. The last `print` statement displays the measured value, for you to compare with the value you entered.

```
voltsIn = ad_volts(0);

print("voltsIn = %1.3f\n", voltsIn);
}
```

Your Turn – Try it Without `da_useScale`

Without `da_useScale`, you may notice slightly larger differences between your requested and measured voltages. The largest difference in error will typically be noticeable at 3.0 V.

- Click SimpleIDE's Save Project As button, rename the project Volts-DAConversion-YourTurn, and save it in My Projects.
- Comment out the `da_useScale` function call (add `//` to its left), then re-run your code.
- Repeat your test with 1.00, 2.00, and 3.00 V.
- You are now done with the calibration, so you can remove the jumper wires that connect D/A0 to A/D0 and D/A1 to A/D1.

ACTIVITY #4: POTENTIOMETER CONTROLLED LED

At this point, we have two new ingredients: measuring knob position (input) and controlling light brightness (output). Let's put them together with a program that measures knob position and uses it to control light brightness.

Example Program: Volts-ControlLED

- Click SimpleIDE's New Project button, set the File name to Volts-ControlLED and Save.
- Enter the Volts-ControlLED.c code into SimpleIDE.
- Remove all jumpers between the D/A and A/D sockets, but keep the potentiometer circuit of Figure 6-10. The output circuit is the built-in P26 LED.
- Reconnect power to your Activity Board (PWR switch to 1 and USB cable connected).
- Click SimpleIDE's Run with Terminal button.
- Turn the pot all the way clockwise to turn off the LED.
- Gradually turn the pot counterclockwise. The light should get brighter as you turn it further toward its counterclockwise limit.

```

/* Volts-ControlLED.c */

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

int main()                       // Main function
{
    float volts;

    print("Use knob to control P26 light."); // User prompt

    while(1)
    {
        volts = ad_volts(3);      // Measure potentiometer
        da_volts(0, volts);      // Set volts & light
    }
}

```

How it Works

As was done before, the `main` function begins by declaring the floating-point variable `volts`.

```

int main()
{

```

```
float volts;
```

All the `while` loop has to do is repeatedly check the potentiometer's wiper voltage with `ad_volts(3)`, and then feed that result to `da_volts(0, volts)`. That sets the voltage at D/A0 as well as the P26 LED brightness.

```
while(1)
{
    volts = ad_volts(3);
    da_volts(0, volts);
}
```

Note that `da_useScale` was not used here. There isn't really any point in this application because human detection of led brightness is relative. Nobody's going to look at the LED and say, "Hey, that's a hundredth of a volt too dim!"

Your Turn – Control Both LEDs

Just for fun, you can add a single line of code to make the potentiometer control the P27 LED as well. Can you see what you would need to do to make the P27 LED do the opposite of the P26 LED as you turn the potentiometer's control knob?

- Save the project as Volts-ControlLED-YourTurn.
- Add a second `da_volts` function call below the first:

```
da_volts(1, (3.3-volts));
```

- Run the program and twist the potentiometer's control knob again. The P27 LED should get brighter as the P26 LED gets dimmer, and vice versa.

ACTIVITY #5: MEASURE INPUT, SCALE VALUE, SET OUTPUT

The last activity was an example of using an input device to control an output device, with the Propeller microcontroller in the middle making it happen. It was a relatively straightforward example, since we were measuring a varying-voltage input to control a voltage output. Furthermore, the `abvolts` library and on-board circuits did a lot of the work for us.

But, what if you want to use the potentiometer—or some other analog input device—with something that requires a different kind of control signal? To do that, you often need to use a little math to write a program that takes your input device’s measurements and translates them into meaningful values for your output device. Here we’ll provide you with those math tools, so you can put them to use in Activity 6.

Remember how the `ad_in` function reports measurements as a number of 4096ths of 5 V? There’s an equivalent `da_out` function for setting voltage output, and its `daVal` parameter requires a number of 256ths of 3.3 V. We are going to use `ad_in` and `da_out` to repeat the last activity, sharpening those coding and math tools along the way.

From the `ad_in` function you’re going to have an input range of 0...2703 for 0 to 3.3 V. The code will need to scale the input value to the `da_out` function’s output range of 0...256 for 0 to 3.3 V.

- Before continuing, grab a pencil and paper and see if you can write some code that you think will convert a measurement in the A/D’s range to an output in the D/A’s range.

This is an example of a $y = mx$ problem. The value x is the raw A/D measurement from `ad_in`. The value m is what we need to multiply it by to get y values that fall in the right range for the parameter `da_out` uses to set the D/A output. We need to solve for the value of m , using two corresponding x and y values. Since we know that 2703 corresponds to a 3.3 V input and 256 corresponds to a 3.3 V output, we can use them to solve for m .

$$y = mx$$

$$\frac{y}{x} = \frac{mx}{x}$$

$$m = \frac{y}{x}$$

$$m = \frac{256}{2703}$$

Let’s try a piece of code that uses the value of m to make the conversion. It lets you twist the knob to control LED brightness and displays the x input and y output values. Note that if you verify this display with a calculator, you’ll get about 9.47 instead of the 9

shown in Figure 6-16. Keep in mind that `int` calculations always round division results downward.

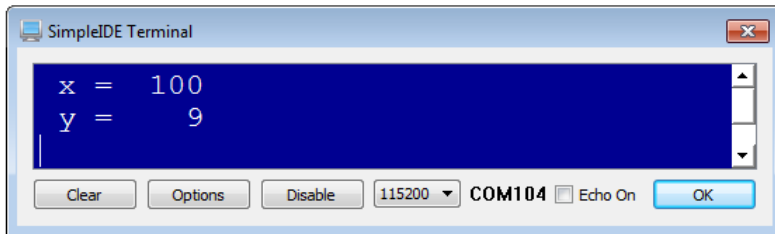


Figure 6-16
Scale
Verification
Display

- Use SimpleIDE's Save Project As button to save a copy of Volts-ControlLED. Name it Volts-ControlScaled, and save it in My Projects.
- Delete the `float` volts variable declaration, and replace it with two `int` variables, `x` and `y`.
- Remove the `print` statement above the `while(1)` loop.
- Update the `while(1)` loop to match the one below.

```
while(1)
{
  x = ad_in (3);           // Measure potentiometer
  print("%c x = %4d\n ",   // Print input value (x)
        HOME, x);

  y = x * 256 / 2703;     // Scale value
  print(" y = %4d\n ", y); // Print output value (y)

  da_out(0, y);          // Set volts & light

  pause(200);           // Slow data for terminal
}
```

- Double-check your updates to the `%` flags in the `print` functions. We are now working with 4-digit `int` values, so instead of using `%1.3f`, we are using `%4d`.



Leading spaces or zeroes, again it is your choice. `%4d` displays leading spaces in decimal integers with less than 4 digits so that it occupies a total of four characters. If you want to display leading zeroes instead, use `%04d`.

- Use Run with Terminal and verify that you have light control.
- Monitor the SimpleIDE terminal to view the **x** input and scaled **y** output values.

Operations Rules for INT Variables

Be aware that with `int` variables, operations do not always give the same results as with algebra. The four rules to keep in mind are:

- 1) Integer division always rounds down. So $4 / 10 = 0$, and $12 / 10 = 1$.
- 2) Operators have precedence. `*`, `/`, and `%` have higher precedence than `+` and `-`, meaning that a statement will execute all the `*`, `/`, and `%` operators first, and then go back and finish the `+` and `-` operators.
- 3) Parentheses can override precedence. `z = (a + b) * x;` will add `a` to `b` before multiplying the result by `x`.
- 4) Operators at the same level of precedence get executed from left to right.



Integer Remainders and the % Operator — In integer calculations, $4 / 10$ is really 0 with a remainder of 4, and $12 / 10$ is really 1 with a remainder of 2. For the remainder, you can use the modulus `%` operator: $4 \% 10 = 4$, and $12 \% 10 = 2$.

Try This – Int Order of Operations

Given `int x, y`; the statement `y = x * 256 / 2703` applies the above rules for `int` variable operations correctly. Because of Rule 1, the statement has to start with a value that can be larger than 2703 before dividing. Rule 2 means that the `*` operator gets executed first because it's leftmost. So `x` gets multiplied by 256 so it can be larger than 2703. Then the `/` operator gets executed second, for a meaningful result.

- Enter `Volts-ControlScaled-TryThis1` into SimpleIDE.

```
/* Volts-ControlScaled-TryThis1.c */
#include "simpletools.h"           // Include simple tools

int main()                       // Main function
{
    int x = 10;
    int y;

    y = x * 256 / 2703;
}
```

```
print("x = %d\n", x);
print("y = %d\n", y);
}
```

- Try a few different values for x, between and including 10 and 2703, and then Run with Terminal. You will see the corresponding scaled value of y in the SimpleIDE terminal.

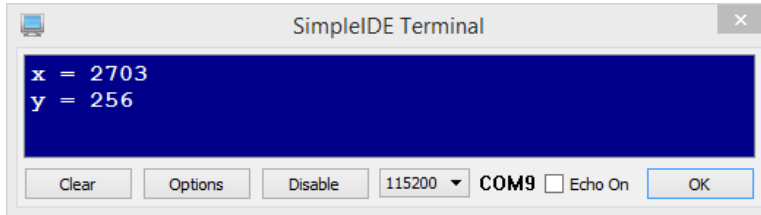


Figure 6-17
Modified “x”
Value Scale
Display

- Use SimpleIDE’s Save Project As button to save another copy as Volts-ControlScaled-TryThis2.
- Change the formula to $y = 256 / 2703 * x$.
- Retry different values for x, including 10 and 2073. Now, what do you see in the SimpleIDE Terminal?

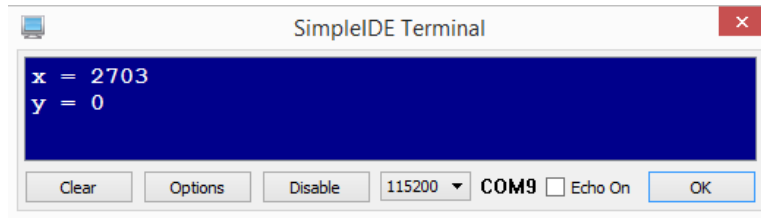


Figure 6-18
Scale Display
After Modifying
Operator Order

Since operators are evaluated from left to right, $256 / 2703$ gets executed first, and that result is always 0. So y ends up always being $0 * x$, for a result of 0 no matter what.

ACTIVITY #6: POTENTIOMETER CONTROLLED SERVO

As mentioned in Chapter 4, a hobby servo is a device that controls position, and you can find them in just about any radio controlled (RC) car, boat or plane. In RC cars, the servo holds the steering to control how sharply the car turns. In an RC boat, it holds the rudder

in position for turns. RC planes typically have several servos that position the different flaps to control the plane's motion. In RC vehicles with gas powered engines, another servo moves the engine's throttle lever to control how fast the engine runs. An example of an RC airplane and its radio controller are shown in Figure 6-19. The hobbyist "flies" the airplane by manipulating thumb joysticks on the radio controller which, via the radio link, causes the servos on the plane to control the positions of the RC plane's elevator flaps and rudder. When there is external force like air pressure against the servo it will actively work to hold the position it was sent.



Figure 6-19
Model Airplane and
Radio Controller

Thumb joysticks like the one in Figure 6-20 are commonly found in both RC and video game controllers. Each joystick typically has two potentiometers that allow the electronics inside the controller to report the joystick's position. One potentiometer rotates with the joystick's horizontal motion (left/right), and the other rotates with the joystick's vertical motion (forward/backward). In the case of the RC controller, a microcontroller inside monitors each potentiometer's output voltage and uses as radio to relay that information to the plane, where an onboard controller receives those radio signals and converts them to signals that control the various servos.

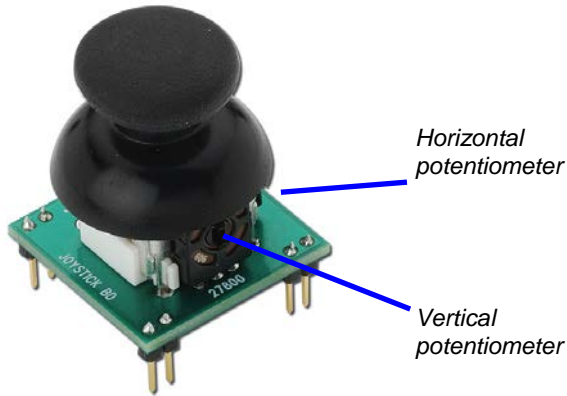


Figure 6-20
Potentiometers Inside
the Parallax Thumb
Joystick Module

In this activity, you will use your potentiometer, which is similar to the ones found in thumb joysticks, to control a servo's position. As you turn the potentiometer's knob, the servo's horn will mirror this motion.



With four A/D inputs, your Propeller Activity Board application could easily monitor two joysticks (horizontal and vertical for each).

Potentiometer Controlled Servo Parts

- (1) Potentiometer – 10 k Ω
- (1) Parallax Standard Servo
- (1) 2.1 mm, center positive plug supply option from Chapter 4, Activity #1.
- (1) Jumper wire (black)
- (1) Potentiometer – 10 k Ω
- (3) Jumper wires – 1 red, 1 black, 1 blue

Potentiometer and Servo Circuits

This activity will use two circuits that you have already built individually: the potentiometer circuit from the activity you just finished and the servo circuit from Chapter 4.

Circuit Safety First! Before connecting the servo, set the Activity Board's PWR switch to 0. Make sure to plug in the servo the right direction, with the white wire nearest the top edge of the board. The USB cable will not power a servo – you must use supply external power supply (See Figure 4-4 on page 92 for power supply options.) When you turn power back on, make sure to set the PWR switch to 2.

- ❑ Leave your potentiometer A/D circuit from Activity #2 on your prototyping area.
- ❑ Add your servo circuit from Chapter 4, Activity #1 as shown in Figure 6-21. Remember to connect your external power supply to the Activity Board's 6-9 V jack! (See Figure 4-4 on page 106 for power supply options.)

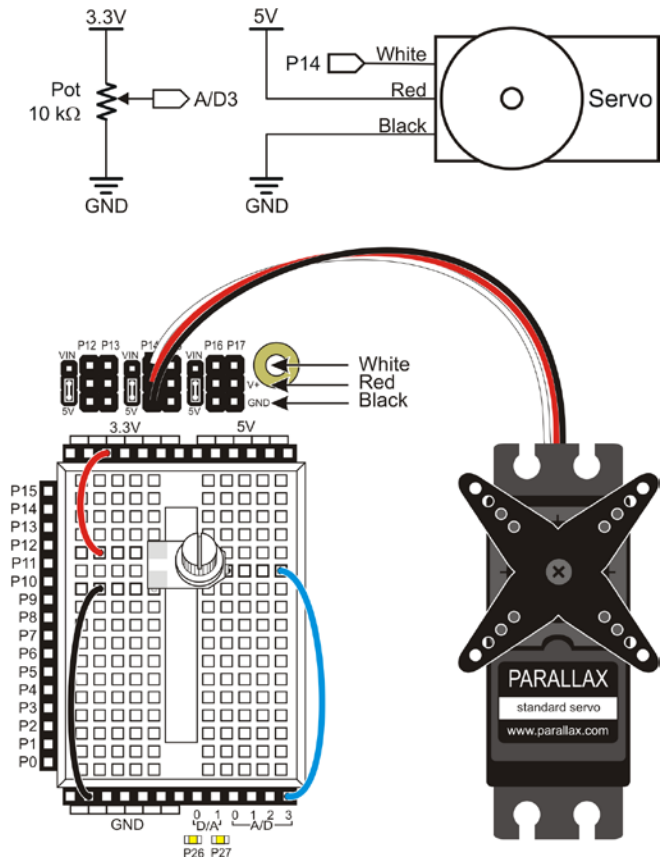


Figure 6-21
Pot & Servo Circuits

Potentiometer Servo Control

Here, we can take the potentiometer's output (approximately 0 to 2703) and scale it to the `servo_angle` function's input range of (0 to 1800). We'll need a new value of `m` for this:

$$y = mx$$

$$m = \frac{y}{x}$$

$$m = \frac{1800}{2703}$$

Since we'll be using int variables, we have to make sure to use `y = x * m(numerator) / m(denominator)`. The result is:

```
x = ad_in(3);
y = x * 1800 / 2703;
```

Example Program: Volts-ServoControl

- ❑ Enter and run this program, then twist the potentiometer's knob and verify that the servo's movements echo the potentiometer's movements. (Make sure to push the pot onto the breadboard to maintain electrical contact. If you don't, the servo might seem twitchy or jittery.)

```
/* Volts-ServoControl.c */
#include "simpletools.h"           // Library includes
#include "abvolts.h"
#include "servo.h"

int main()                       // Main function
{
    print("Twist knob to control servo."); // User prompt

    while(1)
    {
        int x = ad_in(3);        // Measure potentiometer

        int y = x * 1800 / 2703; // Scale value

        servo_angle(14, y);     // Set degreeTenths to y
    }
}
```

How it Works

Since this code controls a servo, it needs access to be able to access the servo functions. So remember to add the servo library with `#include "servo.h"`.

```
#include "simpletools.h"
#include "abvoltz.h"
#include "servo.h"
```

Inside the `while(1)` loop, the code assigns the raw A/D pot voltage measurement to a new variable named `x` with `int x = ad_in(3)`. Then it takes the `x` value, which is in the 0...2703 range, and calculates a scaled `y` value that fits in the 0...1800 range with `int y = x * 1800 / 2703`. Then `servo_angle(14, y)` uses this value to control the servo's position.

```
int x = ad_in(3);

int y = x * 1800 / 2703;

servo_angle(14, y);
```

Try This – Scale and Offset

Let's say we want the full range of potentiometer motion to only move the servo from 45° to 135°. Now, we have a $y = mx + b$ problem. In this case, solving for `b` is pretty easy because we know that `y` should be 450 when `x` is 0. (Keep in mind that $m \times 0 = 0$.)

$$y = mx + b$$

$$450 = m \times 0 + b$$

$$b = 450$$

Next, solve for `m` with known values of `y` and `x`, like `x = 2703` (for max 3.3 V input voltage) and `y = 1350` (for 135 degrees on the servo).

$$y = mx + 450$$

$$1350 = m \times 2703 + 450$$

$$m = \frac{1350 - 450}{2703}$$

$$m = \frac{900}{2703}$$

This means the equation the code needs to implement is:

$$y = \frac{900}{2703}x + 450$$

Remember that integer values need to be multiplied by the numerator first, before dividing by the denominator. So the code we think will make the scale is:

```
int y = x * 900 / 2703 + 450;
```

Let's test that with the Terminal before adding potentiometer code:

- Click New Project, name the project Volts-ServoControl-TryThis, and save it to My Projects.
- Type this code into SimpleIDE.

```
/* Volts-ServoControl-TryThis.c */  
  
#include "simpletools.h" // Library includes  
  
int main() // Main function  
{  
    print("Enter values in 0...2703 range\n");  
    print("Verify results in 450...1350 range\n\n");  
  
    while(1)  
    {  
        print("Enter value: ");  
        int x;  
        scan("%d", &x);  
  
        int y = x * 900 / 2703 + 450;  
        print("y = %d\n\n", y);  
    }  
}
```


- ❑ Click Run with Terminal, and use some test values (like in Figure 6-22) to verify that the code correctly scales and offsets the expected A/D3 input values.

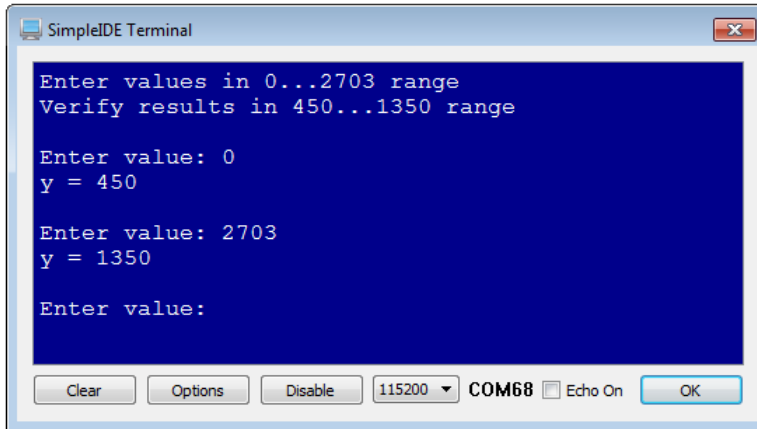


Figure 6-22
Scale and Offset
in SimpleIDE
Terminal

*Test Calculations
before Running*

Your Turn – Servo Motion Scale and Offset

Now that we know $\text{int } y = x * 900 / 2703 + 450$ works, let's test it with our pot-controlled servo.

- ❑ Use Open Project to open Volts-ServoControl.
- ❑ Use Save Project As to save a copy named Volts-ServoControl-YourTurn in My Projects.
- ❑ Change $\text{int } y = x * 1800 / 2703$; to $\text{int } y = x * 900 / 2703 + 450$; Click the Load RAM & Run button.
- ❑ Test to make sure the servo's output range only turns from 45° to 135° when you turn the knob over its full range.

ACTIVITY #7: POTENTIOMETER CONTROLLING OTHER COG

Let's use the potentiometer input values to control the LED blink rate from another cog. When the pot is turned to its clockwise limit, the fastest blink rate will have 25 ms pauses. When it's turned to its counterclockwise limit, the slowest rate will have 275 ms

pauses. Let's first apply your scale and offset calculation and coding skills to fit this output range to the potentiometers 0...2703 input range.

- Apply the math from the Your Turn you just completed to determine a statement that calculates a `y` output in the 25...275 range that corresponds to an `x` input in the 0...2703 range. Correct answer options include `int y = x * (275 - 25) / 2703 + 25` as well as `int y = x * 250 / 2703 + 25`.

Pot Input Controls Output in Other Process

We have already written code that controls LED blink rate in another cog using the Activity Board's built-in P26 LED. It was in the Multi-InfoExchange project from Chapter 5, Activity #5. The program's `main` function would set the value of a volatile global variable named `t`. It also had a function named `blink`, which used `t` to set blink rate running in another cog.

This program can serve as a starting point for potentiometer-controlled blink rate. Instead of using statements like `t = 100` and `t = 50` to set the blink rate, the `main` function can instead measure the potentiometer, apply scale and offset, and then use the result to set the value of `t` for the `blink` function.

- Go back to Chapter 5, Activity #5 and examine `Multi-InfoExchange.c`.
- Think about how you would modify the `main` function to enable potentiometer control of the blink rate.
- Would you need to make other modifications to the file? Maybe an extra `#include` and some changes to the comments?

Example Program: Volts-Multicore

The Volts-Multicore program is a modified version of Multi-InfoExchange from Chapter 5, Activity #5 that controls blink rate in another cog based on potentiometer measurements.

- Click SimpleIDE's New Project button, set the File name to Volts-Multicore and then Save.
- Enter the following Volts-Multicore.c code into SimpleIDE.
- Click SimpleIDE's Load RAM & Run button.

- Turn the potentiometer's knob, and verify that it now controls the rate at which the LED light blinks.

```

/* Volts-Multicore.c */

#include "simpletools.h"           // Library include
#include "abvolts.h"

void blink();                    // Forward declaration

volatile int t;                  // Declare t for both cogs

int main()                       // Main function
{
    print("Adjust pot knob, and verify \n"); // User prompts
    print("blink rate control.\n");

    // Initialize t before running
    // blink in other cog.
    int x = ad_in(3);            // Check pot
    int y = x * 250 / 2703 + 25; // Scale + offset
    t = y;                       // Change blink's pause time

    cog_run(blink, 128);         // Run blink in other cog

    while(1)                     // Main loop
    {
        x = ad_in(3);            // Check pot
        y = x * 250 / 2703 + 25; // Scale + offset
        t = y;                   // Update blink's pause time
    }
}

void blink()                     // Function for other cog
{
    while(1)                     // Endless loop
    {
        high(26);                // LED on
        pause(t);                 // ...for t ms
        low(26);                 // LED off
        pause(t);                 // ...for t ms
    }
}

```

How it Works

This application needs the `simpletools` library for access to its `high`, `low`, `pause`, and `cog_run` functions. It also needs the `abvolts` library for access to its `ad_in` function.

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

Since the `blink` function is below `main`, but there's a reference to it in `main`, the forward declaration `void blink()` is required. A global variable named `t` is declared as volatile so that functions running in different cogs use it to exchange information.

```
void blink();

volatile int t;
```

The `main` function starts with a couple of `print` statements prompting to test the pot's control of the blink rate.

```
int main()
{
    print("Adjust pot knob, and verify \n");
    print("blink rate control.\n");
```

The `blink` function needs to start with a value of `t`, so these three lines check the pot, apply scale and offset, and copy the result to `t`. After that, it's safe to run the `blink` function in another cog with `cog_run(blink, 20)`.

```
    int x = ad_in(3);
    int y = x * 250 / 2703 + 25;
    t = y;

    cog_run(blink, 20);
```

The `while(1)` loop repeatedly checks the pot, scales its output, and copies it to the shared `t` variable for control of the `blink` function's LED on/off rate.

```
    while(1)
    {
        x = ad_in(3);
        y = x * 250 / 2703 + 25;
        t = y;
    }
}
```

This is the same LED `blink` function from Multi-InfoExchange. It expects to have its global `pin` and `t` variables set before it gets launched in another cog. As it repeats itself, code in another cog can change the value of the global `t` variable, and this cog's blink rate will change.

```
void blink()
{
  while(1)
  {
    high(26);
    pause(t);
    low(26);
    pause(t);
  }
}
```

Your Turn – Reduce Duplicate Code, Add a Function

This code is repeated twice, once in the initialization and again in the `while` loop.

```
int x = ad_in(3);
int y = x * 250 / 2703 + 25;
t = y;
```

Instead of having all that code repeating itself twice, why not just have a function that your code calls twice? Here is an example of a function that can do the job. This one is not for launching into another cog; it's just for reading and scaling the pot.

```
int potScaled(int channel)
{
  int x = ad_in(channel);
  int y = x * 250 / 2703 + 25;
  return y;
}
```

A call to that function might look like this:

```
t = potScaled(3);
```

After adding the function, that call can replace the three lines that read the pot, scale, and set the `t` variable.

Don't forget the forward declaration above `main`:

```
int potScaled(int channel);
```

- Continuing with `Volts-Multicore.c`, click SimpleIDE's Save Project As button, set the File name to `Volts-Multicore-YourTurn`. Make sure to save in My Projects.
- Add the `potScaled` function below the `main` function.
- Add the forward declaration above the `main` function.
- Find the two groups of 3 commands that look like this:

```
int x = ad_in(3);
int y = x * 250 / 2703 + 25;
t = y;
```

...and replace them with the `t = potScaled(3)` function call.

- Click SimpleIDE's Load RAM & Run button.
- Verify that it still works correctly.

SUMMARY



Why isn't `x` declared at the start of `main`? Just as a variable can be local to a function, it can also be local to a code block. The `x` variable is only needed within the `while` loop, so it is declared at the start of the `while` loop. As a general rule, it's best to minimize the scope of local variables your program uses. There are some cases where you'll need to increase the scope by declaring the variable earlier. For example, if the `while` loop was not endless, and your code needed to retain the value of `x` after the loop finishes, you would have to declare `x` above the `while` loop. In that case it would be:

```
int x;                // Variable for A/D input
while(1)
{
    x = ad_in(3);
```

This chapter used the potentiometer as a knob in activities that both measured and set voltages. Along the way, it introduced the following:

- Potentiometer schematic and part drawing, and explanation of its terminals.
- Potentiometer theory of operation, and some of its uses.
- Voltage divider circuits and equation.

- Analog to digital (A/D) conversion for measuring voltages.
- Digital to analog (D/A) conversion for setting voltages.
- Part tolerance contributions to measurement errors.
- Solving $y = mx$ and $y = mx + b$ for m and b .
- Scaling values from an input range to their corresponding values in a different output range.
- Using casts to copy values between variables of different types.
- Operator precedence and order.
- Moving scalar operations into a function that can be re-used.

Questions

1. How many terminals does a potentiometer have and what are they named?
2. What parts does a voltage divider circuit have, and how are they connected?
3. What does A/D stand for? What does D/A stand for?
4. What does the `%1.2f` flag do in a `print` statement?
5. How many voltage measurement sockets does the Propeller Activity Board have?
6. What library contains the functions `ad_volts` and `da_volts`?
7. Which has higher precedence, `+` or `*`? How does that affect which gets executed first in a statement?
8. What's the difference between an analog and digital value?
9. What is the rounding rule for the `/` operator when applied to `int` variables?
10. Will this work? `int i; float f; i = 6.0 * f;` Explain.
11. What does a thumb joystick use to detect its position?
12. What's a good way to clean up repeated blocks of identical code in your program?
13. How would you position a potentiometer's knob in a 3.3 volt circuit to make its output close to 1.65 V?

Exercises

1. Calculate the voltage divider for $R_A = 1,000$ and $R_B = 10,000$ with a 3.3 V supply.
2. Calculate the voltage divider for $R_A = 10,000$ and $R_B = 10,000$ with a 3.3 V supply.
3. Calculate the voltage divider for $R_A = 10,000$ and $R_B = 1,000$ with a 3.3 V supply.
4. Copy the value of a `float` variable named `f` to an `int` variable named `i`.

5. Copy the value of an `int` variable named `i` to a float variable named `f`.
6. Write a statement to measure the volts applied to A/D2.
7. Write a statement to apply 2.9 V with D/A1.
8. Write a statement to get the raw A/D value from A/D1.
9. Write a line of code that scales an input in the 0 to 100 range to an output in the 0 to 50 range. Assume your input and output variables are `int x` and `int y`.
10. Write a line of code that takes an input in the 0 to 120 range and scales it to the 30 to 90 range. Assume your input and output variables are `int x` and `int y`.

Projects

1. Modify Volts-Multicore from Activity #5 so that its knob-position monitoring all happens in another cog. Then, add code to the `main` function that prints the pot voltage and the number of seconds elapsed since the application started every second. Hints: You will want to make `x` global and volatile. Also, look for and remove any instances of `int x` in functions. (Yes, you can have a global variable and a local variable with the same name, and it can cause problems.) Use `while(t == 0);` to wait for the potentiometer reading function in the other cog to store a value in `t` before allowing `cog_run(blink, 20)` to execute.
2. Add a button to the potentiometer servo controller. When you press and hold the button, it reverses the direction of servo control. So, instead of turning the same direction with the knob, it turns the opposite direction. When you release the button, rotation direction should return to normal. Hints: For opposite direction, subtract the servo setting from 1800. Use the technique introduced in the Activity #4's Try This –Scale and Offset to test your solution first. That way, you can be sure it works before trying it with your servo hardware.

Solutions

- Q1. 3 terminals: A, B, and W (or wiper).
- Q2. A voltage divider circuit has two resistors connected in series. Generally there is a connection to measure the voltage at the point between the two resistors.
- Q3. A/D stands for analog to digital, D/A stands for digital to analog.
- Q4. It displays the corresponding floating point value in the `print` statements parameter list with 1 digit to the left of the decimal point and two to the right.
- Q5. Four A/D sockets numbered 0 to 3.
- Q6. The `abvolts` library.
- Q7. `*` has higher precedence, so it gets executed before `+`.

- Q8. An analog value varies continuously. A digital value has some form of discrete step.
- Q9. There can be no fractional part in the result, so it always rounds down to the next integer.
- Q10. No, the `int` variable on the left cannot be assigned the right-side value which is a `float`. Correct answers would be:
- `int i; float f; i = (int)(6.0 * f)`
 - `float f1; float f2; f1 = 6.0 * f2`
- Q11. The two outputs of the whole unit are the wiper terminal voltages of its two potentiometers.
- Q12. Move the code block to a function and call it from the various places that used to have the redundant block.
- Q13. Position it roughly in the middle of its range of motion.
- E1. The equation for this voltage divider is $R1$ divided by $(R1 + R2)$ where $R1$ is the resistor towards 3.3 V. $3.3 \text{ V} \times 10,000 / (1,000 + 10,000) = 3.0 \text{ V}$.
- E2. $3.3 \text{ V} \times 10,000 / (10,000 + 10,000) = 1.65 \text{ V}$. (Noticing a pattern with equal resistors yet?)
- E3. $3.3 \text{ V} \times 1,000 / (1,000 + 10,000) = 0.3 \text{ V}$. (How about a pattern for swapping unequal resistors?)
- E4. Solution: `i = (int) f;`
- E5. Solution: `f = (float) i;`
- E6. Solution: `float volts = ad_volts(2);`
- E7. Solution: `da_volts(1, 2.9);`
- E8. Solution: `int myVar = ad_in(1);`
- E9. $y = x / 2$
- E10. $y = x * (90 - 30) / 120 + 30$

P1. Example solution:

```

/* Volts-P1-Solution.c */

#include "simpletools.h"           // Library include
#include "abvolts.h"

void blink();                    // Forward declaration
void potentiometer();           // <- add
volatile int pin, t, x;         // Variables for cogs go share

int main()                       // Main function
{

```

```

cog_run(potentiometer, 20);           // potentiometer() to other cog

pin = 26;                             // Set up blink cog vars

// Wait for other cog to store value in t. Blink will need it.
while(t == 0);

cog_run(blink, 40);                   // blink() to other cog

// Seconds & pot display
int seconds = 0;                      // Seconds variable

while(1)                               // New main loop
{
    pause(1000);                       // Wait 1 second
    seconds++;                          // Add 1 to seconds
    print("Seconds = %d\n", seconds);   // Display seconds
    print("Pot = %d\n", x);            // Display pot measurement
}

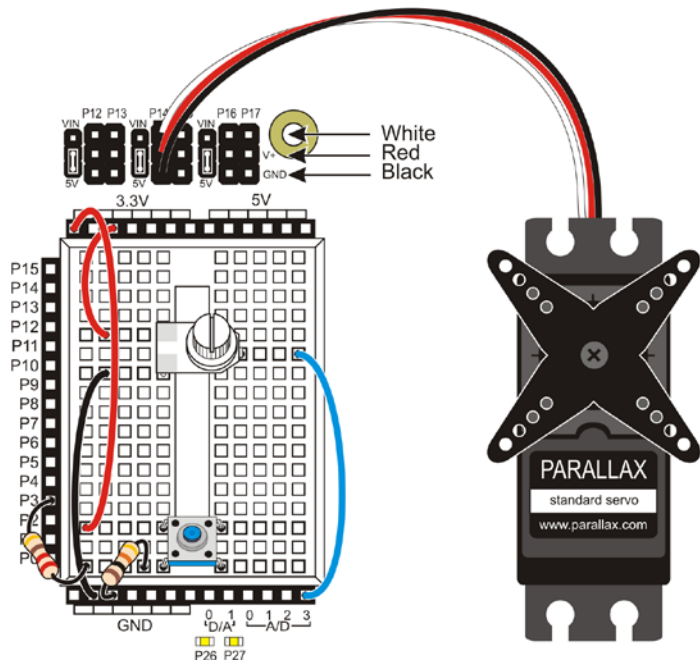
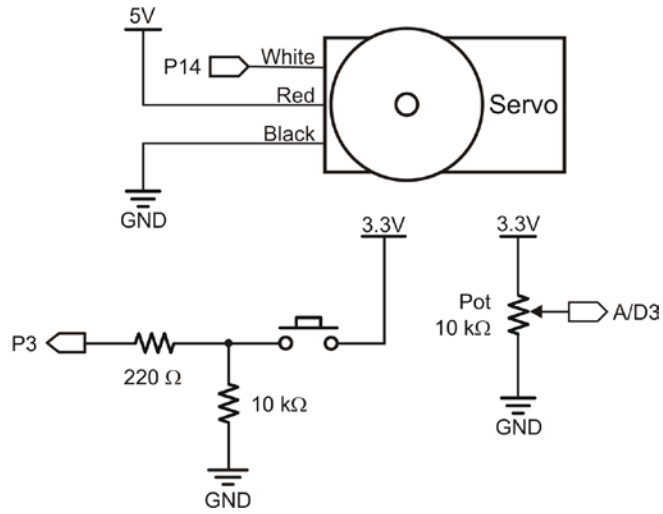
void potentiometer()
{
    x = ad_in(3);                      // Initialize shared variables
    int y = x * 250 / 2703 + 25;
    t = y;

    while(1)                           // Main loop
    {
        x = ad_in(3);                  // Check pot
        int y = x * 250 / 2703 + 25;   // Scale + offset
        t = y;                         // Change blink's pause time
    }
}

void blink()                           // Blink function for other cog
{
    while(1)                             // Blink loop
    {
        high(pin);                    // LED on
        pause(t);                      // ...for t ms
        low(pin);                      // LED off
        pause(t);                      // ...for t ms
    }
}

```

P2. Build circuits shown.



Test the potentiometer and pushbutton with SimpleIDE Terminal first.

```

/* Volts-P2-Solution1.c */

#include "simpletools.h"           // Library includes
#include "abvolts.h"
#include "servo.h"

int main()                       // Main function
{
    print("Twist knob to control value.\n"); // User prompt
    print("Monitor terminal for servo value.\n\n");

    while(1)
    {
        int x = ad_in(3);        // Measure potentiometer

        int y = x * 1800 / 2703; // Scale value same direction

        if(input(3) == 1)
        {
            y = 1800 - y;        // Scale opposite direction
        }

        // servo_angle(14, y);   // Set degreeTenths to y
        print("y = %d\n", y);
        pause(500);
    }
}

```

Then try it again with the servo.

```

/* Volts-P2-Solution2.c */

#include "simpletools.h"           // Library includes
#include "abvolts.h"
#include "servo.h"

int main()                       // Main function
{
    print("Twist knob to control servo."); // User prompt

    while(1)
    {
        int x = ad_in(3);        // Measure potentiometer

        int y = x * 1800 / 2703; // Scale value same direction

        if(input(3) == 1)

```

```
{  
  y = 1800 - y; // Scale opposite direction  
}  
servo_angle(14, y); // Set degreeTenths to y  
}
```