# Arduino Cookbook

## Recipes to Begin, Expand, and Enhance Your Projects



Michael Margolis,
Brian Jepson
& Nicholas Robert Weldin

# Arduino Cookbook

THIRD EDITION

Recipes to Begin, Expand, and Enhance Your Projects

**Michael Margolis, Brian Jepson, and Nicholas Robert Weldin**

**O'REILLY®**

Beijing · Boston · Farnham · Sebastopol · Tokyo

**Arduino Cookbook**

by Michael Margolis, Brian Jepson, and Nicholas Robert Weldin Copyright © 2020 Michael Margolis, Nicholas Robert Weldin, and Brian Jepson. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Rachel Roumeliotis
- Development Editor: Jeff Bleiel
- Production Editor: Deborah Baker
- Copyeditor: Kim Cofer
- Proofreader: Josh Olejarz
- Indexer: Sue Klefstad
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- March 2011: First Edition
- December 2011: Second Edition
- April 2020: Third Edition

**Revision History for the Third Edition**

- 2020-04-16: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491903520* for release details.

# Preface

This book was written by Michael Margolis and Brian Jepson with Nick Weldin to help you explore the amazing things you can do with Arduino.

Arduino is a family of microcontrollers (tiny computers) and a software creation environment that makes it easy for you to create programs (called *sketches*) that can interact with the physical world. Things you make with Arduino can sense and respond to touch, sound, position, heat, and light. This type of technology, often referred to as *physical computing*, is used in all kinds of things from smartphones to automobile electronics systems. Arduino makes it possible for anyone with an interest—even people with no programming or electronics experience—to use this rich and complex technology.

# Who This Book Is For

This book is aimed at readers interested in using computer technology to interact with the environment. It is for people who want to quickly find the solution to hardware and software problems. The recipes provide the information you need to accomplish a broad range of tasks. It also has details to help you customize solutions to meet your specific needs. There is insufficient space in this book to cover general theoretical background, so links to external references are provided throughout the book. See "What Was Left Out" for some general references for those with no programming or electronics experience.

If you have no programming experience—perhaps you have a great idea for an interactive project but don't have the skills to develop it—this book will help you learn how to write code that works, using examples that cover over 200 common tasks. Absolute beginners may want to consult a beginner's book such as *Getting Started with Arduino* (Make Community), by Massimo Banzi and Michael Shiloh.

If you have some programming experience but are new to Arduino, the book will help you become productive quickly by demonstrating how to implement specific Arduino capabilities for your project.

People already using Arduino should find the content helpful for quickly learning new techniques, which are explained using practical examples. This will help you to embark on more complex projects by showing you how to solve problems and use capabilities that may be new to you.

Experienced C/C++ programmers will find examples of how to use the low-level AVR resources (interrupts, timers, I2C, Ethernet, etc.) to build applications using the Arduino environment.

# How This Book Is Organized

The book contains information that covers the broad range of Arduino's capabilities, from basic concepts and common tasks to advanced technology. Each technique is explained in a recipe that shows you how to implement a specific capability. You do not need to read the content in sequence. Where a recipe uses a technique covered in another recipe, the content in the other recipe is referenced rather than repeating details in multiple places.

Chapter 1, *Getting Started*
> Introduces the Arduino environment and provides help on getting the Arduino development environment and hardware installed and working. This chapter introduces some of the most popular new boards. The next couple of chapters introduce Arduino software development.

Chapter 2, *Arduino Programming*
> Covers essential software concepts and tasks.

Chapter 3, *Mathematical Operations*
> Shows how to make use of the most common mathematical functions.

Chapter 4, *Serial Communications*
> Describes how to get Arduino to connect and communicate with your computer and other devices. Serial is the most common method for Arduino input and output, and this capability is used in many of the recipes throughout the book.

Chapter 5, *Simple Digital and Analog Input*
> Introduces a range of basic techniques for reading digital and analog signals.

Chapter 6, *Getting Input from Sensors*
> Builds on concepts in the preceding chapter with recipes that explain how to use devices that enable Arduino to sense touch, sound, position, heat, and light.

Chapter 7, *Visual Output*

Covers controlling light. Recipes cover switching on one or many LEDs and controlling brightness and color. This chapter explains how you can drive bar graphs and numeric LED displays, as well as create patterns and animations with LED arrays. In addition, the chapter provides a general introduction to digital and analog output for those who are new to this.

Chapter 8, *Physical Output*

Explains how you can make things move by controlling motors with Arduino. A wide range of motor types is covered: solenoids, servo motors, DC motors, and stepper motors.

Chapter 9, *Audio Output*

Shows how to generate sound with Arduino via output devices such as a speaker. It covers playing simple tones and melodies and playing WAV files and MIDI.

Chapter 10, *Remotely Controlling External Devices*

Describes techniques that can be used to interact with almost any device that uses some form of remote controller, including TV, audio equipment, cameras, garage doors, appliances, and toys. It builds on techniques used in previous chapters for connecting Arduino to devices and modules.

Chapter 11, *Using Displays*

Covers interfacing text and graphical LCD displays. The chapter shows how you can connect these devices to display text, scroll or highlight words, and create special symbols and characters.

Chapter 12, *Using Time and Dates*

Covers built-in Arduino time-related functions and introduces many additional techniques for handling time delays, time measurement, and real-world times and dates.

Chapter 13, *Communicating Using I2C and SPI*

Covers the Inter-Integrated Circuit (I2C) and Serial Peripheral Interface (SPI) standards. These standards provide simple ways for digital information to be transferred between sensors and Arduino. This chapter shows how to use I2C and SPI to connect to common devices. It

also shows how to connect two or more Arduino boards, using I2C for multiboard applications.

Chapter 14, *Simple Wireless Communication*

Covers wireless communication with XBee, Bluetooth, and other wireless modules. This chapter provides examples ranging from simple wireless serial port replacements to mesh networks connecting multiple boards to multiple sensors.

Chapter 15, *WiFi and Ethernet*

Describes the many ways you can use Arduino with the internet. It has examples that demonstrate how to build and use web clients and servers and shows how to use the most common internet communication protocols with Arduino. This chapter also includes recipes that will help you connect Arduino to the Internet of Things.

Chapter 16, *Using, Modifying, and Creating Libraries*

Arduino software libraries are a standard way of adding functionality to the Arduino environment. This chapter explains how to use and modify software libraries. It also provides guidance on how to create your own libraries.

Chapter 17, *Advanced Coding and Memory Handling*

Covers advanced programming techniques, and the topics here are more technical than the other recipes in this book because they cover things that are usually concealed by the friendly Arduino wrapper. The techniques in this chapter can be used to make a sketch more efficient—they can help improve performance and reduce the code size of your sketches.

Chapter 18, *Using the Controller Chip Hardware*

Shows how to access and use hardware functions that are not fully exposed through the documented Arduino language. It covers low-level usage of the hardware input/output registers, timers, and interrupts.

Appendix A, *Electronic Components*

Provides an overview of the components used throughout the book.

Appendix B, *Using Schematic Diagrams and Datasheets*

Explains how to use schematic diagrams and datasheets.

Appendix C, *Building and Connecting the Circuit*
> Provides a brief introduction to using a breadboard, connecting and using external power supplies and batteries, and using capacitors for decoupling.

Appendix D, *Tips on Troubleshooting Software Problems*
> Provides tips on fixing compile and runtime problems.

Appendix E, *Tips on Troubleshooting Hardware Problems*
> Covers problems with electronic circuits.

Appendix F, *Digital and Analog Pins*
> Provides tables indicating functionality provided by the pins on standard Arduino boards.

Appendix G, *ASCII and Extended Character Sets*
> Provides tables showing ASCII characters.

# What Was Left Out

There isn't room in this book to cover electronics theory and practice, although guidance is provided for building the circuits used in the recipes. For more detail, readers may want to refer to material that is widely available on the internet or to books such as the following:

- *Make: Electronics,* Second Edition, by Charles Platt (Make Community)
- *Getting Started in Electronics* by Forrest M. Mims, III (Master Publishing)
- *Physical Computing* by Tom Igoe (Cengage)
- *Practical Electronics for Inventors,* Fourth Edition, by Paul Scherz and Simon Monk (McGraw-Hill)
- *The Art of Electronics* by Paul Horowitz and Winfield Hill (Cambridge University Press)

This cookbook explains how to write code to accomplish specific tasks, but it is not an introduction to programming C or C++ (the languages that the

Arduino development environment is built upon). Relevant programming concepts are briefly explained, but there is insufficient room to cover the details. If you want to learn more about C and C++, you may want to refer to one of the following books:

- *Head First C: A Brain-Friendly Guide* by David Griffiths and Dawn Griffiths (O'Reilly)

- *A Book on C* by Al Kelley and Ira Pohl (Addison-Wesley)

- *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie (Prentice Hall); a favorite, although not really a beginner's book, this is the book that has taught many people C programming

- *Expert C Programming: Deep C Secrets* by Peter van der Linden (Prentice Hall); an advanced though somewhat dated book, this book is entertaining at the same time it provides insights into why C is the way it is


# Code Style (About the Code)

The code used throughout this book has been tailored to clearly illustrate the topic covered in each recipe. As a consequence, some common coding shortcuts have been avoided, particularly in the early chapters. Experienced C programmers often use rich but terse expressions that are efficient but can be a little difficult for beginners to read. For example, the early chapters increment variables using explicit expressions that are easy for nonprogrammers to read:

```
result = result + 1; // increment the count
```

rather than the following, commonly used by experienced programmers, that does the same thing:

```
result++;  // increment using the post-increment operator
```

Feel free to substitute your preferred style. Beginners should be reassured that there is no benefit in performance or code size in using the terse form.

Some programming expressions are so common that they are used in their terse form. For example, the loop expressions are written as follows:

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

This is equivalent to the following:

```
int i;
for(i=0; i < 4; i = i+1)
```

See Chapter 2 for more details on these and other expressions used throughout the book.

Good programming practice involves ensuring that values used are valid (garbage in equals garbage out) by checking them before using them in calculations. However, to keep the code focused on the recipe topic, very little error-checking code has been included.

# Arduino Platform Release Notes

This edition has been updated and tested with Arduino 1.8.x. The downloadable code has been updated for this edition, and is posted in two repositories; one for the all the Arduino Sketches, and another for all the Processing Sketches.

This book's website, *https://oreil.ly/Arduino_Cookbook_3*, has a link to an errata page. Errata give readers a way to let us know about typos, errors, and other problems with the book. Posted errata will be visible on the page immediately, and we'll confirm them after checking them out. O'Reilly can also fix errata in future printings of the book and on the O'Reilly learning platform, making for a better reader experience pretty quickly.

If you have problems making examples work, see Appendix D, which covers troubleshooting software problems. The Arduino forum is a good

place to post a question if you need more help: *https://forum.arduino.cc*.

If you like—or don't like—this book, by all means, please let people know. Amazon reviews are one popular way to share your happiness or other comments. You can also leave reviews for the book on the O'Reilly online learning platform.

# Notes on the Third Edition

A lot has changed since the second edition: a proliferation of new boards, lots more processing power, memory, communications capabilities, and form factor. Although this book has grown in size through each edition, it is impossible to cover in depth everything all readers may wish to do. The focus of this edition is to ensure the content is up to date and to provide an overview of the rich capabilities made available to the Arduino community since the previous edition, to help you get started with this amazing technology.

Note that if you are using earlier releases of Arduino than that covered here you can still download code from the second and first editions of this book. To download this example code, visit *http://examples.oreilly.com/9780596802486* and *http://examples.oreilly.com/0636920022244*.

# Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data

types, environment variables, statements, and keywords.

**`Constant width bold`**

Shows commands or other text that should be typed literally by the user.

*`Constant width italic`*

Shows text that should be replaced with user-supplied values or by values determined by context.

---

**TIP**

This element signifies a tip or suggestion.

---

**NOTE**

This element signifies a general note.

---

**WARNING**

This element indicates a warning or caution.

---

# Using Code Examples

If you have a technical question or a problem using the code examples, please send an email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting

example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Arduino Cookbook*, Third Edition, by Michael Margolis, Brian Jepson, and Nicholas Robert Weldin (O'Reilly). Copyright 2020 Michael Margolis, Nicholas Robert Weldin, and Brian Jepson, 978-1-491-90352-0."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

# O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

# How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North

- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and more information about our books and courses, see our *website*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

# Acknowledgments for the Second Edition (Michael Margolis)

Nick Weldin's contribution was invaluable for the completion of this book. It was 90% written when Nick came on board—and without his skill and enthusiasm, it would still be 90% written. His hands-on experience running Arduino workshops for all levels of users enabled us to make the advice in this book practical for our broad range of readers. Thank you, Nick, for your knowledge and genial, collaborative nature.

Simon St. Laurent was the editor at O'Reilly who first expressed interest in this book. And in the end, he is the man who pulled it together. His support and encouragement kept us inspired as we sifted our way through the volumes of material necessary to do the subject justice.

Brian Jepson helped me get started with the writing of this book. His vast knowledge of all things Arduino and his concern and expertise for communicating about technology in plain English set a high standard. He was an ideal guiding hand for shaping the book and making technology readily accessible for readers. We also have Brian to thank for the new XBee content in Chapter 14.

Liquidware, Adafruit, MakerBot Industries, Mindkits, Oomlout, and SK Pang.

Nick would like to thank everyone who was involved with Tinker London, particularly Alexandra, Peter, Brock Craft, Daniel Soltis, and all the people who assisted on workshops over the years.

Nick's final thanks go to his family, Jeanie, Emily, and Finn, who agreed to let him do this over their summer holiday, and of course, much longer after that than they originally thought, and to his parents, Frank and Eva, for bringing him up to take things apart.

Last but not least, I express thanks to the following people:

Joshua Noble for introducing me to O'Reilly. His book *Programming Interactivity* is highly recommended for those interested in broadening their knowledge in interactive computing.

Robert Lacy-Thompson for offering advice early on with the first edition.

Mark Margolis for his support and help as a sounding board in the book's conception and development.

I thank my parents for helping me to see that the creative arts and technology were not distinctive entities and that, when combined, they can lead to extraordinary results.

And finally, this book would not have been started or finished without the support of my wife, Barbara Faden. My grateful appreciation to her for keeping me motivated and for her careful reading and contributions to the manuscript.

# Acknowledgments for the Third Edition (Brian Jepson)

A hearty thanks to Michael Margolis, the lead author of this book, and Jeff Bleiel, our editor for this edition. They trusted me to take the lead on this book and to bring this new edition to you. I appreciate their trust and confidence and I hope that they are as happy with the results as I am. On a

# Getting Started

## 1.0 Introduction

The Arduino environment has been designed to be easy to use for beginners who have no software or electronics experience. With Arduino, you can build objects that can respond to and/or control light, sound, touch, and movement. Arduino has been used to create an amazing variety of things, including musical instruments, robots, light sculptures, games, interactive furniture, and even interactive clothing.

Arduino is used in many educational programs around the world, particularly by designers and artists who want to easily create prototypes but do not need a deep understanding of the technical details behind their creations. Because it is designed to be used by nontechnical people, the software includes plenty of example code to demonstrate how to use the Arduino board's various facilities.

Though it is easy to use, Arduino's underlying hardware works at the same level of sophistication that engineers employ to build embedded devices. People already working with microcontrollers are also attracted to Arduino because of its agile development capabilities and its facility for quick implementation of ideas.

Arduino is best known for its hardware, but you also need software to program that hardware. Both the hardware and the software are called "Arduino." The hardware (Arduino and Arduino-compatible boards) is inexpensive to buy, or you can build your own (the hardware designs are open source). The software is free, open source, and cross-platform. The combination enables you to create projects that sense and control the physical world.

In addition, there is an active and supportive Arduino community that is accessible worldwide through the Arduino forums, tutorials, and project hub. These sites offer learning resources, project development examples,

and solutions to problems that can provide inspiration and assistance as you pursue your own projects.

## Arduino Software and Sketches

Software programs, called *sketches,* are created on a computer using the Arduino integrated development environment (IDE). The IDE enables you to write and edit code and convert this code into instructions that Arduino hardware understands. The IDE also transfers those instructions, in the form of compiled code, to the Arduino board (a process called *uploading*).

> ### TIP
>
> You may be used to referring to software source code as a "program" or just "code." In the Arduino community, source code that contains computer instructions for controlling Arduino functionality is referred to as a *sketch*. The word *sketch* will be used throughout this book to refer to Arduino program code.

The recipes in this chapter will get you started by explaining how to set up the development environment and how to compile and run an example sketch.

The Blink sketch, which is preinstalled on most Arduino boards and compatibles, is used as an example for recipes in this chapter, though the last recipe in the chapter goes further by adding sound and collecting input through some additional hardware, not just blinking the light built into the board. Chapter 2 covers how to structure a sketch for Arduino and provides an introduction to programming.
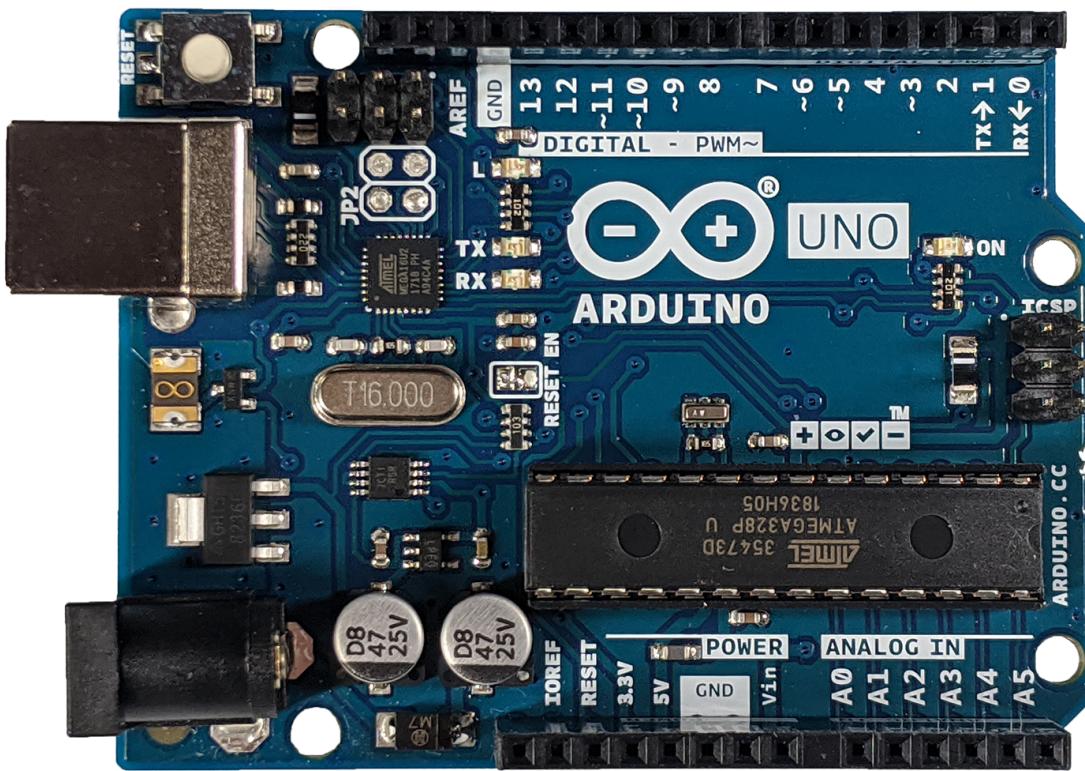
> ### NOTE
>
> If you already know your way around Arduino basics, feel free to jump forward to later chapters. If you're a first-time Arduino user, patience in these early recipes will pay off with smoother results later.

## Arduino Hardware

The Arduino board is where the code you write is executed. The board can only control and respond to electricity, so you'll attach specific components to it that enable it to interact with the real world. These components can be sensors, which convert some aspect of the physical world to electricity so that the board can sense it, or actuators, which get electricity from the board and convert it into something that changes the world. Examples of sensors include switches, accelerometers, and ultrasonic distance sensors. Actuators are things like lights and LEDs, speakers, motors, and displays.

There are a variety of official boards that you can use with Arduino software and a wide range of Arduino-compatible boards produced by companies and individual members of the community. In addition to all the boards on the market, you'll even find Arduino-compatible controllers inside everything from 3D printers to robots. Some of these Arduino-compatible boards and products are also compatible with other programming environments such as MicroPython or CircuitPython.

The most popular boards contain a USB connector that is used to provide power and connectivity for uploading your software onto the board. Figure 1-1 shows a basic board that most people start with, the Arduino Uno. It is powered by an 8-bit processor, the ATmega328P, which has 2 kilobytes of SRAM (static random-access memory, used to store program variables), 32 kilobytes of flash memory for storing your sketches, and runs at 16 MHz. A second chip handles USB connectivity.

*Basic board: the Arduino Uno*

The Arduino Leonardo board uses the same *form factor* (the layout of the board and its connector pins) as the Uno, but uses a different processor, the ATmega32U4, which runs your sketches and also takes care of USB connectivity. It is slightly cheaper than the Uno, and also offers some interesting features, such as the ability to emulate various USB devices such as mice and keyboards. The Arduino-compatible Teensy and Teensy++ boards from PJRC (*http://www.pjrc.com/teensy*) are also capable of emulating USB devices.

Another board with a similar pin layout and even faster processor is the Arduino Zero. Unlike the Arduino Uno and Leonardo, it cannot tolerate input pin voltages higher than 3.3 volts. The Arduino Zero has a 32-bit processor running at 48 MHz and has 32 kilobytes of RAM and 256 kilobytes of flash storage. Adafruit's Metro M0 Express and SparkFun's RedBoard Turbo come in the same form factor as the Arduino Zero and

also offer compatibility with multiple environments, including the Arduino IDE and CircuitPython.

**ARDUINO AND USB**

The Arduino Uno has a second microcontroller onboard to handle all USB communication; the small surface-mount chip (the ATmega16U2, ATmega8U2 in older versions of the Uno) is located near the USB socket on the board. The Arduino Leonardo has only one chip, the ATmega32U4, which runs your code and handles USB communications. You can reprogram the Leonardo to emulate USB devices (see Recipe 18.14).

Older Arduino boards, and some of the Arduino-compatible boards, use a chip from the company FTDI that provides a hardware USB solution for connection to the serial port of your computer. Some of the cheaper clones that you will encounter on eBay or Amazon may use a chip that performs a similar function, such as the CH340. You will probably need to install a driver to use CH340-based boards.

There's another class of USB-enabled Arduino-compatible boards you may encounter, which have no dedicated chip to handle USB communication. Instead, these boards use a technique called *bit-banging,* in which software running on the board manipulates I/O pins to send and receive USB signals. These boards, which include the popular original Adafruit Trinket, may not work well with modern computers, though you may have luck with an older computer. (Adafruit has released the Adafruit Trinket M0, which has native USB, and as a bonus, is much faster than its predecessor.)

Finally, you may find Arduino-compatible boards that have no USB connection whatsoever. Instead, they offer only serial pins that you cannot directly connect to a computer without a special adapter. See "Serial Hardware" for a list of some available adapters.

If you want a board for learning that will run the majority of sketches in this book, the Uno is a great choice. If you want more performance than the Uno, but still want to use the Uno form factor, then consider the Zero, or a similar board such as the Metro M0 Express or RedBoard Turbo. The MKR and Nano 33 series of boards also offer excellent performance, but in a smaller form factor than the Uno.

## Caution Needed with Some 3.3-Volt Boards

Many of the newer boards operate on 3.3 volts rather than the 5 volts used by older boards such as the Uno. Some such boards can be permanently damaged if an input or output pin receives 5 volts, even for a fraction of a second, so check the documentation for your board to see if it is tolerant of 5 volts before wiring things up when there is a risk of pin levels higher than 3.3 volts. Most 3.3-volt boards are powered by a 5-volt power supply (for example, through the USB port), but a voltage regulator converts it to 3.3 volts before it reaches the board's 3.3-volt electronics. This means that it is not unusual to see a 5-volt power supply pin on a board whose input and output pins are *not* 5-volt tolerant.

Arduino boards come in other form factors, which means that the pins on such boards have a different layout and aren't compatible with shields designed for the Uno. The MKR1010 is an Arduino board that uses a much smaller form factor. Its pins are designed for 3.3V I/O (it is *not* 5V tolerant) and like the Zero, it uses an ARM chip. However, the MKR1010 also includes WiFi and and a circuit to run from and recharge a LIPO battery. Although the MKR family of boards is not compatible with shields designed for the Uno, Arduino offers a selection of add-on boards for the MKR form factor called *carriers*.

### EXTEND ARDUINO WITH SHIELDS

Arduino boards can be enhanced by add-ons called *shields*, which you connect by stacking them on top with their pins connected to all the headers of the Arduino. Different models of Arduino and certain Arduino compatibles may have their own add-ons similar to, but not compatible with, shields. This is because some models of boards use a different form factor than the most common Arduino, the Uno. For example, the Arduino MKR is physically much smaller than the Uno. MKR add-on boards are also called shields even though they use a form factor that is incompatible with the Uno. Adafruit has a huge collection of *Featherwing* add-on boards for its Feather line of development boards, which are compatible with Arduino development software. Featherwing add-on boards are not compatible with other hardware form factors such as the Uno and MKR boards.

You can get boards as small as a postage stamp, such as the Adafruit Trinket M0; larger boards that have more connection options and more powerful processors, such as the Arduino Mega and Arduino Due; and

boards tailored for specific applications, such as the Arduino LilyPad for wearable applications, the Arduino Nano 33 IoT for wireless projects, and the Arduino Nano Every for embedded applications (standalone projects that are often battery-operated).

Other third-party Arduino-compatible boards are also available, including the following:

Bare Bones Board (BBB)
Low-cost Bare Bones Boards are available with or without USB capability from Modern Device, and from Educato in a shield-compatible version.

Adafruit Industries
Adafruit has a vast collection of Arduino and Arduino-compatible boards and accessories (modules and components).

SparkFun
SparkFun has lots of Arduino and Arduino-compatible accessories.

Seeed Studio
Seeed Studio sells Arduino and Arduino-compatible boards as well as many accessories. It also offers a flexible expansion system for Arduino and other embedded boards called Grove, which uses a modular connector system for sensors and actuators.

Teensy and Teensy++
These tiny but extremely versatile boards are available from PJRC.

Wikipedia has an exhaustive list of Arduino-compatible boards. You can also find an overview of Arduino boards on the Arduino site.

# 1.1 Installing the Integrated Development Environment (IDE)

**Problem**

You want to install the Arduino development environment on your computer.

## Solution

Download the Arduino software for Windows, MacOS, or Linux. Here are notes on installing the software on these platforms:

Windows

If you are running Windows 10, you can use the Microsoft Store to install Arduino without needing admin privileges. But for earlier versions of Windows you'll need admin privileges to double-click and run Windows installer when it's downloaded. Alternatively, you can download the Windows ZIP file, and unzip it to any convenient directory that you have write access to.

Unzipping the file will create a folder named *Arduino-<nn>* (where *<nn>* is the version number of the Arduino release you downloaded). The directory contains the executable file (named *Arduino.exe*), along with other files and folders.

> **NOTE**
>
> The first time you run Arduino on Windows, you may see a dialog that says "Windows Defender Firewall has blocked some features of this app," specifying `javaw.exe` as the source of the warning. The Arduino IDE is a Java-based application, which is why the warning comes from the Java program instead of *Arduino.exe*. This is used by boards that support the ability to upload sketches over a local network. If you plan to use this kind of board, you should use this dialog to permit *javaw.exe* access to the network.

When you plug in the board, it automatically associates the installed driver with the board (this may take a little time). If this process fails, or you installed Arduino using the ZIP file, then go to the Arduino Guide page, click the link for your board from the list there, and follow the instructions.

If you are using an earlier board (any board that uses FTDI drivers) and are online, you can let Windows search for drivers and they will install automatically. If you don't have internet access, or are using Windows XP, you should specify the location of the drivers. Use the file selector

to navigate to the *drivers\FTDI USB Drivers* directory, located in the directory where you unzipped the Arduino files. When this driver has installed, the Found New Hardware Wizard will reappear, saying a new serial port has been found. Follow the same process again.

> **TIP**
>
> You may need to go through the sequence of steps to install the drivers twice to ensure that the software is able to communicate with the board.

macOS

The Arduino download for the Mac is a ZIP file. If the ZIP file does not extract automatically after you download it, locate the download and double-click it to extract the application. Move the application to somewhere convenient—the *Applications* folder is a sensible place. Double-click the application. The splash screen will appear, followed by the main program window.

Current Arduino boards such as the Uno can be used without additional drivers. When you first plug the board in, a notification may pop up saying a new network port has been found; you can dismiss this. If you are using earlier boards that need FTDI drivers, you can get these from FTDI.
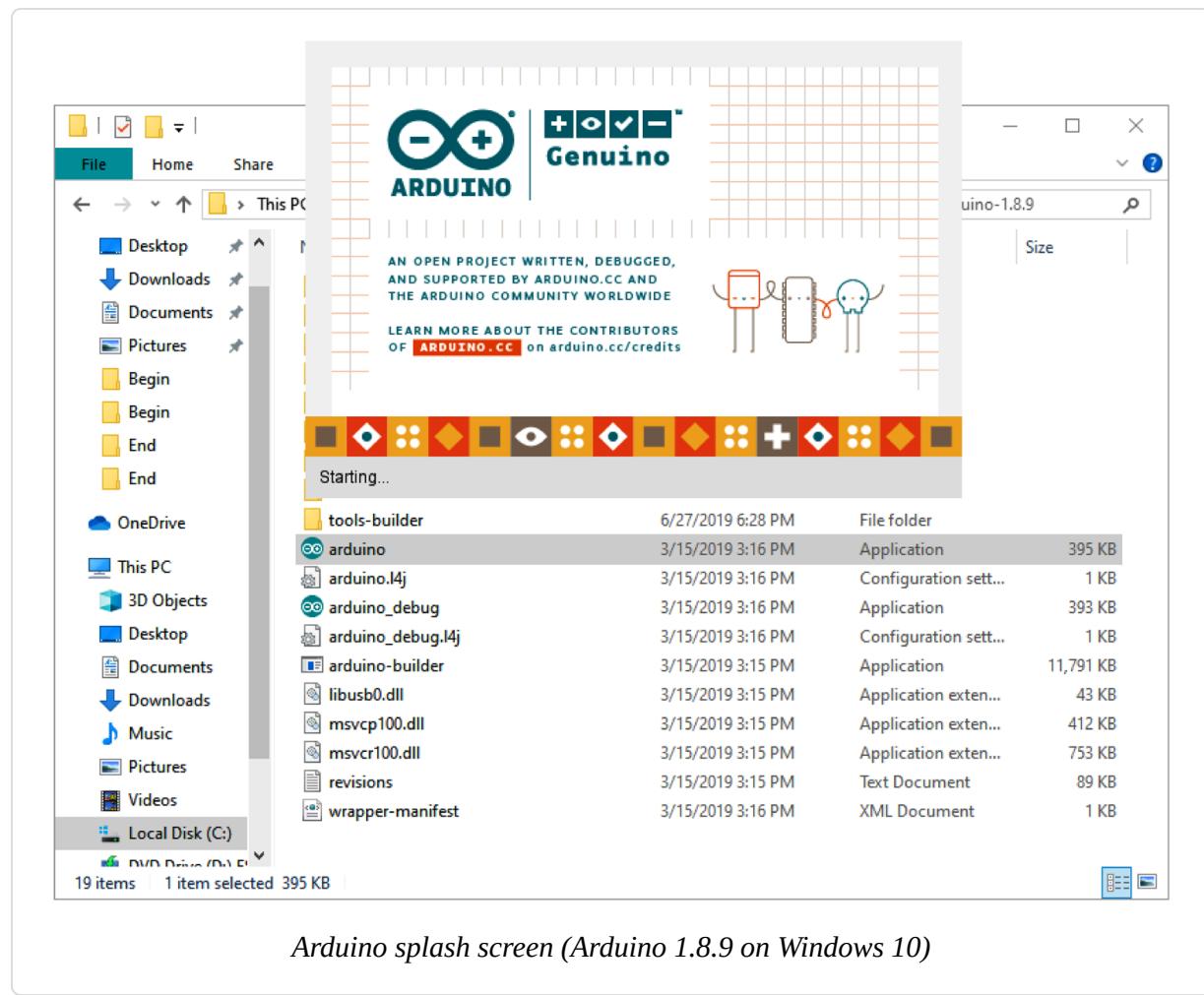
Linux

Linux versions are increasingly available from your distribution's package manager, but these versions are often not the most current release, so it is best to download the version from *http://arduino.cc/download*. It is available for 32 or 64 bit, and offers an ARM version that can be used on the Raspberry Pi and other Linux ARM boards. Most distributions use a standard driver that is already installed, and usually have FTDI support as well. See this Arduino Linux page for instructions on installing Arduino on Linux. You will

need to follow those instructions to run the install script, and you may need to do so to permit your user account to access the serial port.
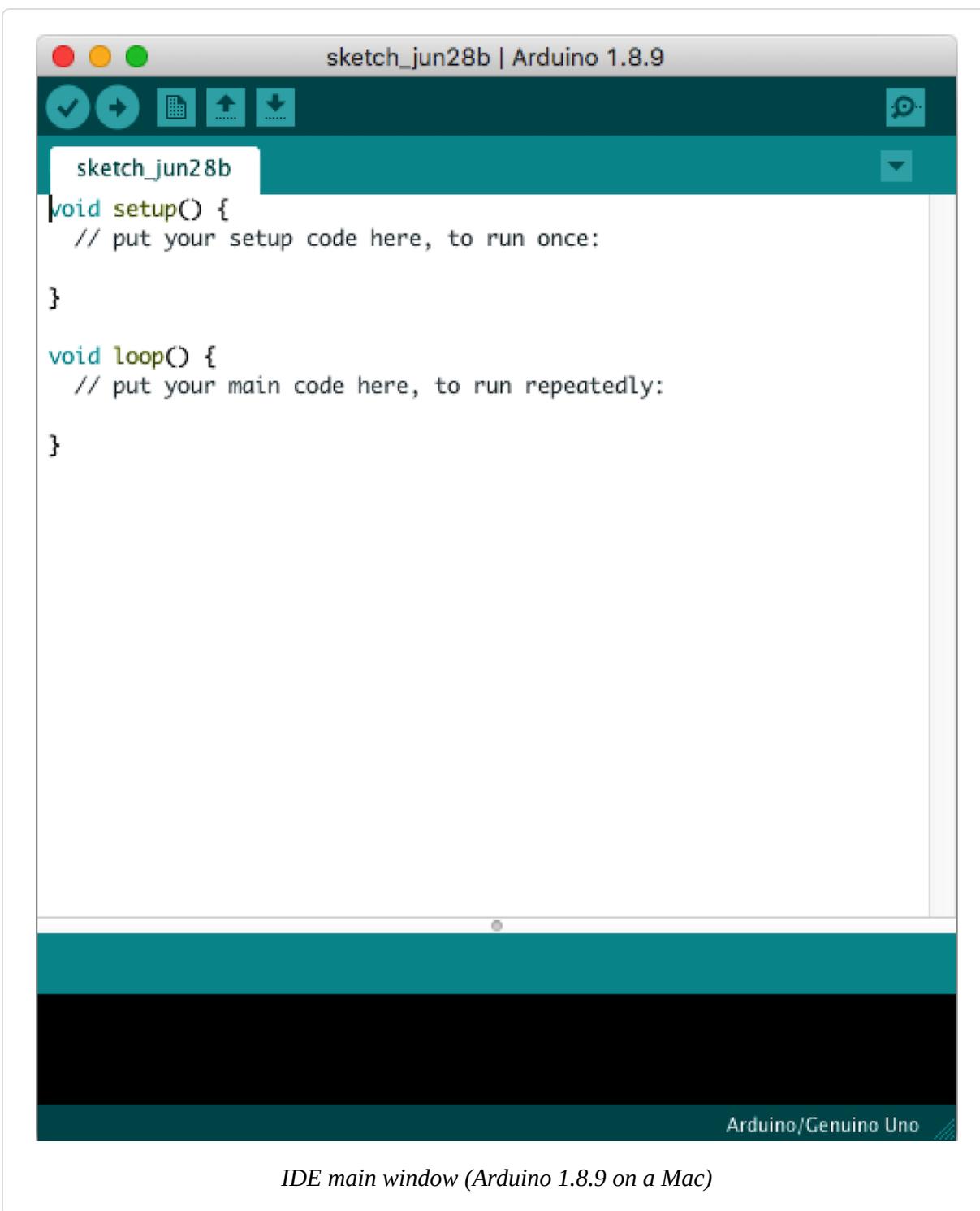
For Arduino-compatible boards that is not made by Arduino, you may need to install support files using the Boards Manager (see Recipe 1.7). You should also check the specific board's documentation for any additional steps needed.

After you've installed Arduino, double-click its icon, and the splash screen should appear (see Figure 1-2).



*Arduino splash screen (Arduino 1.8.9 on Windows 10)*

The initial splash screen is followed by the main program window (see Figure 1-3). Be patient, as it can take some time for the software to load. You can find the icon for Arduino on the Start menu (Windows), the

*Applications* folder (macOS), or possibly on the desktop. On Linux, you may need to run the Arduino executable from the Terminal shell.



*IDE main window (Arduino 1.8.9 on a Mac)*

## Discussion

If the software fails to start, check the troubleshooting section of the Arduino website for help solving installation problems.

## See Also

Online guides for getting started with Arduino are available at for Windows, for macOS, and for Linux.

The Arduino Pro IDE is a development environment for Arduino that's aimed at the needs of professional users. At the time of this writing, it was in an early state. See the Arduino Pro IDE GitHub repo.

The Arduino CLI is a command-line tool for compiling and uploading sketches. You can also use it in place of the Library and Boards Manager. See the Arduino CLI GitHub repo.

There is an online editing environment called Arduino Create. In order to use this you will need to create an account and download a plug-in that enables the website to communicate with the board to upload code. It has cloud storage where your sketches are saved and provides facilities for sharing code. At the time this book was written, Arduino Create was a fairly new, still-evolving service. If you would like the ability to create Arduino sketches without having to install a development environment on your computer, then have a look at Arduino Create.

If you are using a Chromebook, Arduino Create's Chrome App requires a monthly subscription of US$1. It has a time-limited trial so you can try it out. There is another alternative to compiling and uploading Arduino code from a Chromebook: Codebender is a web-based IDE like Arduino Create, but it also supports a number of third-party Arduino-compatible boards. Pricing plans are available for classrooms and schools as well. See this Codebender page.

# 1.2 Setting Up the Arduino Board
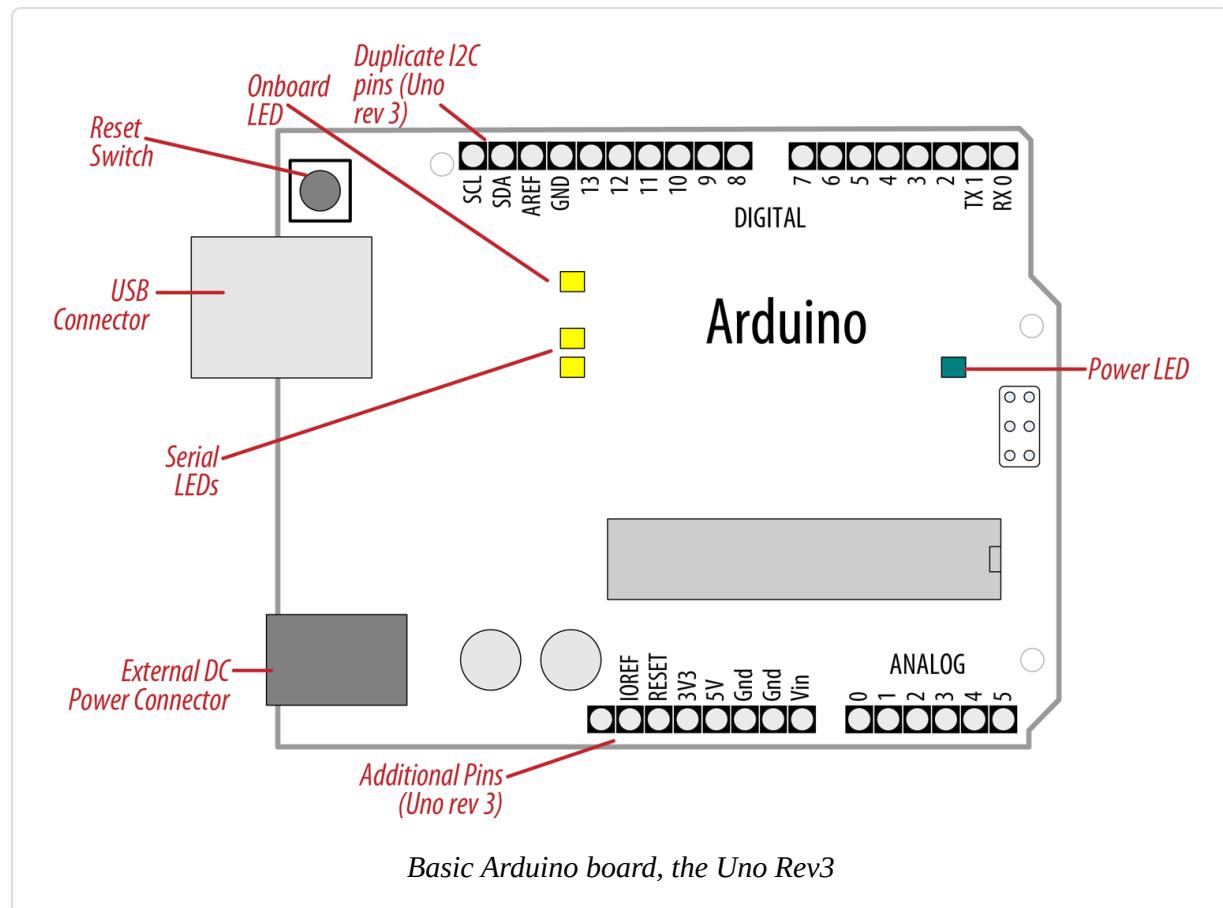
**Problem**

You want to power up a new board and verify that it is working.

## Solution

Plug the board into a USB port on your computer and check that the LED power indicator on the board illuminates. Most Arduino boards have an LED power indicator that stays on whenever the board is powered.

The onboard LED (labeled in Figure 1-4) should flash on and off when the board is powered up (most boards come from the factory preloaded with software to flash the LED as a simple check that the board is working).



*Basic Arduino board, the Uno Rev3*

## Discussion

If the power LED does not illuminate when the board is connected to your computer, the board is probably not receiving power (try a different USB socket or cable).

The flashing LED is being controlled by code running on the board (new boards are preloaded with the Blink example sketch). If the onboard LED is flashing, the sketch is running correctly, which means the chip on the board is working. If the power LED is on but the onboard LED (usually labeled L) is not flashing, it could be that the factory code is not on the chip; follow the instructions in Recipe 1.3 to load the Blink sketch onto the board to verify that the board is working. If you are not using a standard board, it may not have an onboard LED, so check the documentation for details of your board.

The Leonardo and Zero-class boards (Arduino Zero, Adafruit Metro M0, SparkFun RedBoard Turbo) have the same footprint as the Uno (its headers are in the same position, enabling shields to be attached). They are significantly different in other respects. The Leonardo has an 8-bit chip like the Uno, but because it doesn't have a separate chip for handling USB communications, the Leonardo only accepts program uploads immediately after the board has been reset. You'll see the Leonardo's onboard LED pulse

gently while it's waiting for an upload. The Leonardo is 5V tolerant. The Zero has a 32-bit ARM chip, with more memory for storing your program and running it. There is also a pin that provides a digital-to-analog converter (DAC), which means you can get a varying analog voltage from it. This can be used to generate audio signals at much higher quality than an Uno. The Zero is not 5V tolerant, nor are the similar boards from Adafruit (Metro M0 Express) or SparkFun (RedBoard Turbo).

The MKR1010 uses the same chip as the Zero (and like the Zero, is not 5V tolerant), but in a smaller form factor. It also includes WiFi, so it is able to connect to the internet through a WiFi network. The MKR form factor does not support shields that are designed for the Uno pin layout.

All the 32-bit boards have more pins that support interrupts than most of the 8-bit boards, which are useful for applications that must quickly detect signal changes (see Recipe 18.2). The one 8-bit exception to this is the Arduino Uno WiFi Rev2, which supports interrupts on any of its digital pins.

## See Also

Online guides for getting started with Arduino are available at for Windows, for macOS, and for Linux. See the Arduino Guide for board-specific instructions.

A troubleshooting guide can be found on the Arduino site.

# 1.3 Using the Integrated Development Environment to Prepare an Arduino Sketch

## Problem
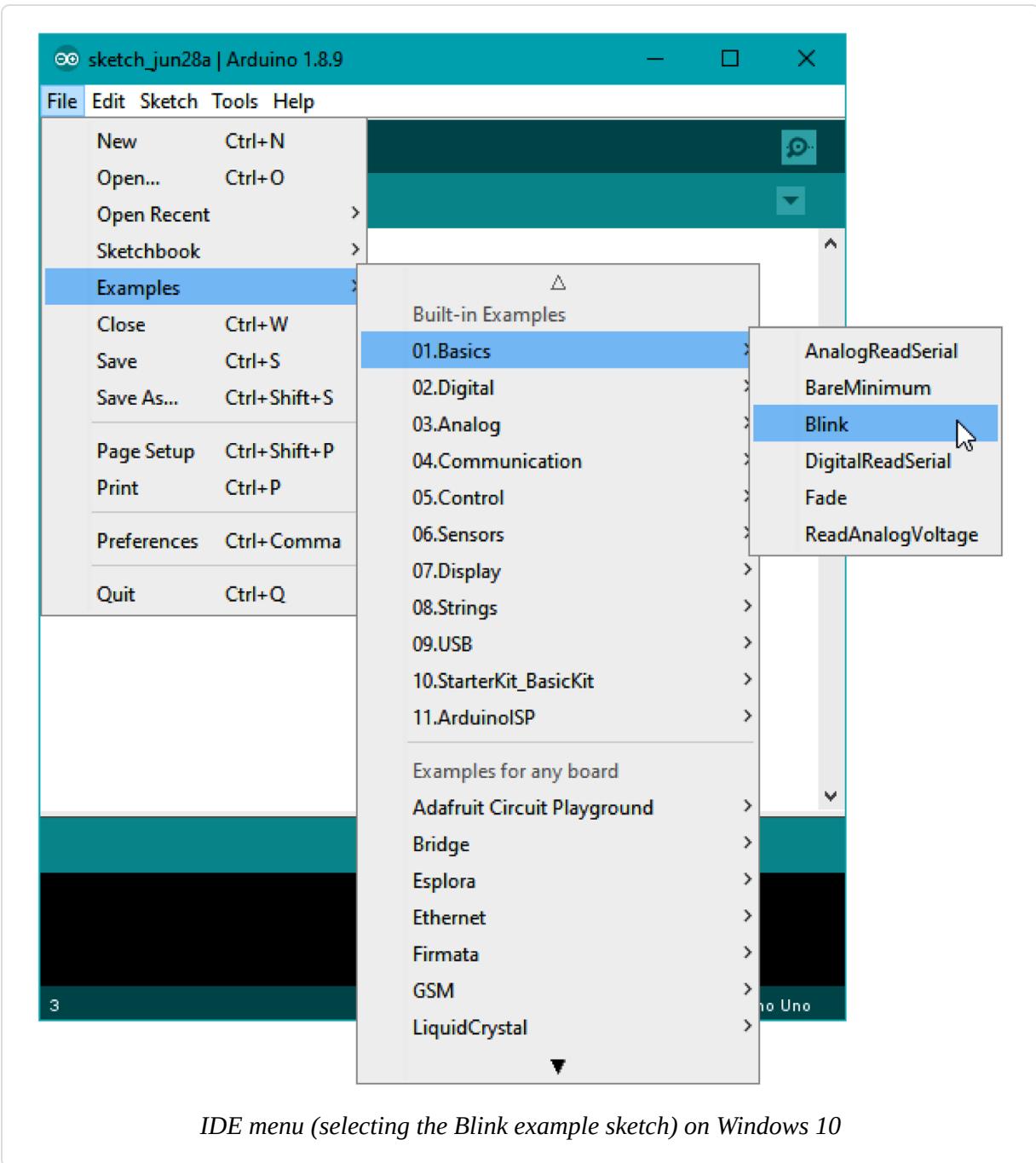
You want to familiarize yourself with Arduino's compilation process and understand both the status and error messages it can produce.

## Solution

Source code for Arduino is called a *sketch*. The process that takes a sketch and converts it into a form that will work on the board is called *compilation*. Use the Arduino IDE to create, open, and modify sketches that define what the board will do. You can use buttons along the top of the IDE to perform these actions (shown in Figure 1-6), or you can use the menus or keyboard shortcuts (visible in Figure 1-5).

The Sketch Editor area is where you view and edit code for a sketch. It supports common text-editing shortcuts such as Ctrl-F (⌘+F on a Mac) for find, Ctrl-Z (⌘+Z on a Mac) for undo, Ctrl-C (⌘+C on a Mac) to copy highlighted text, and Ctrl-V (⌘+V on a Mac) to paste highlighted text.

Figure 1-5 shows how to load the Blink sketch (the sketch that comes preloaded on a new Arduino board).

*IDE menu (selecting the Blink example sketch) on Windows 10*

After you have started the IDE, go to the File→Examples menu and select 01. Basics→Blink, as shown in Figure 1-5. The code for blinking the built-in LED will be displayed in the Sketch Editor window (refer to Figure 1-6).
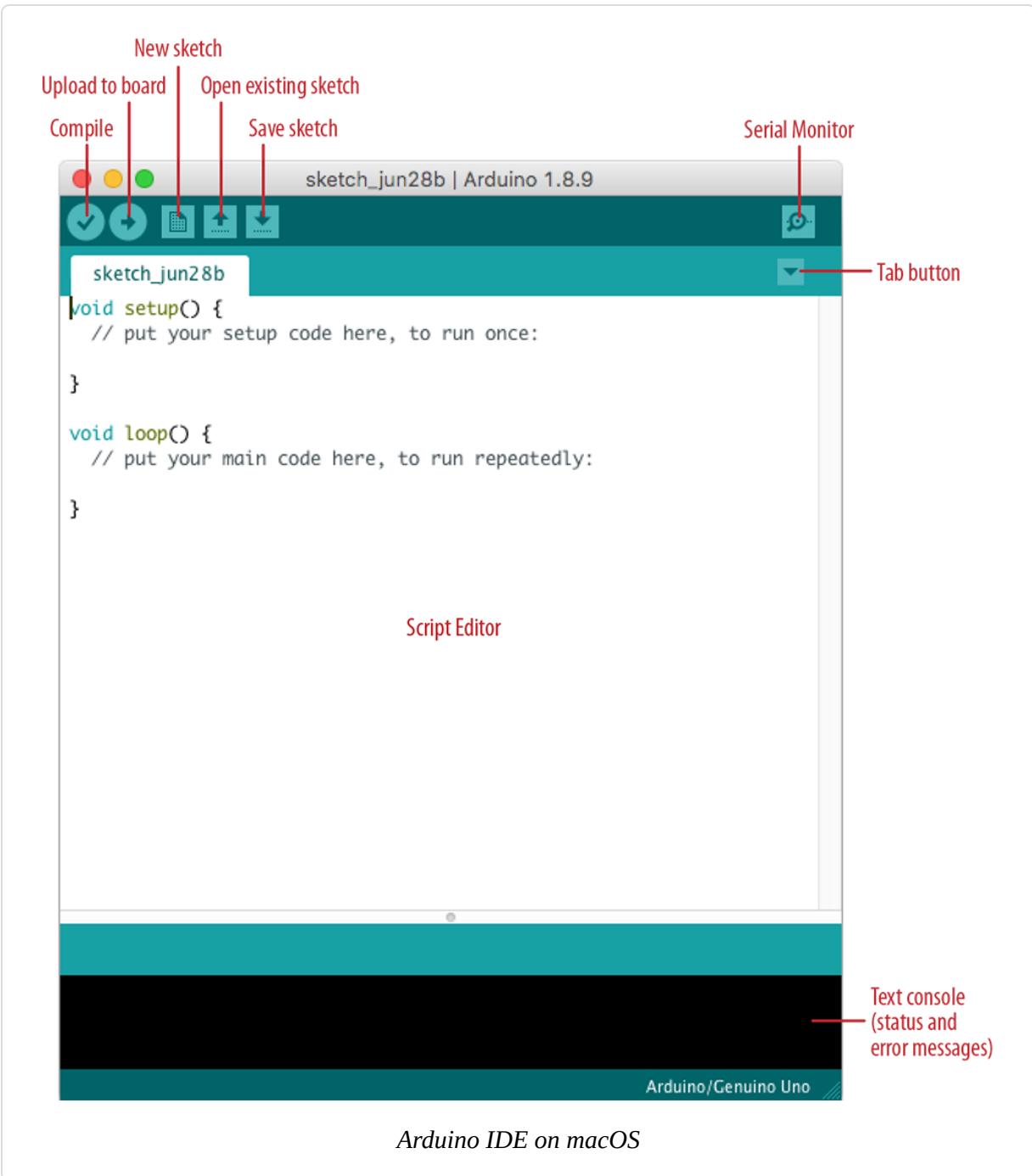
Before you can send the code to the board, it needs to be converted into instructions that can be read and executed by the Arduino controller chip; this is called *compiling*. To do this, click the compile button (the top-left

button with a checkmark inside), or select Sketch→Verify/Compile (Ctrl-R; ⌘+R on a Mac).

You should see a message that reads "Compiling sketch..." and a progress bar in the message area below the text-editing window. After a second or two, a message that reads "Done Compiling" will appear. The black console area will contain the following additional message:

```
Sketch uses 930 bytes (2%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for
local variables. Maximum is 2048 bytes.
```

The exact message may differ depending on your board and Arduino version; it is telling you the size of the sketch and the maximum size that your board can accept.

*Arduino IDE on macOS*

## Discussion

The IDE uses a number of command-line tools behind the scenes to compile a sketch. For more information on this, see Recipe 17.1.

The final message telling you the size of the sketch indicates how much program space is needed to store the controller instructions on the board. If the size of the compiled sketch is greater than the available memory on the board, the following error message is displayed:

```
Sketch too big; see
http://www.arduino.cc/en/Guide/Troubleshooting#size for tips on reducing it.
```

If this happens, you need to make your sketch smaller to be able to put it on the board, or get a board with higher flash memory capacity. If your global variables are using too much memory, you'll see a different error instead:

```
Not enough memory; see http://www.arduino.cc/en/Guide/Troubleshooting#size
for tips on reducing your footprint.
```

In that case, you'll need to go through your code and reduce the amount of memory that you are allocating to global variables, or get a board with a higher SRAM (dynamic memory) capacity.

---

**TIP**

To prevent you from accidentally overwriting the example code, the Arduino IDE does not allow you to save changes to the built-in example sketches. You must rename them using the File → Save As menu option. You can save sketches you write yourself with the Save button (see Recipe 1.5).

---

If there are errors in the code, the compiler will print one or more error messages in the console window. These messages can help identify the error—see Appendix D on software errors for troubleshooting tips.

The compiler can also generate warnings if it decides there are some peculiarities about your sketch that could cause problems. These can be very helpful to avoid problems that could trip you up later. You can configure your warning level by opening File → Preferences (Windows or Linux) or Arduino → Preferences (macOS) and setting Compiler Warnings to None, Default, More, or All. Despite the name, Arduino defaults to None. We suggest you set this to Default or More.

> **NOTE**
>
> Code uploaded onto the board cannot be downloaded back onto your computer. Make sure you save your sketch code on your computer. You cannot save changes that you've made to the example files; you need to use Save As and give the changed file another name.

## See Also

Recipe 1.5 shows an example sketch. Appendix D has tips on troubleshooting software problems.

# 1.4 Uploading and Running the Blink Sketch

## Problem

You want to transfer your compiled sketch to the Arduino board and see it working.

## Solution

Connect your Arduino board to your computer using the USB cable. Load the Blink sketch into the IDE by choosing File→Examples and selecting 01. Basics→Blink.

Next, select Tools→Board from the drop-down menu and select the name of the board you have connected (if it is the standard Uno board, it is probably one of the first entries in the board list).

Now select Tools→Serial Port. You will get a drop-down list of available serial ports on your computer. Each machine will have a different combination of serial ports, depending on what other devices you have used with your computer.

On Windows, they will be listed as numbered COM entries. If there is only one entry, select it. If there are multiple entries, your board will probably be the last entry.

On the Mac, if your board is an Uno it will be listed as:

```
/dev/cu.usbmodem-XXXXXXX(Arduino/Genuino Uno)
```

If you have an older board, it will be listed as follows:

```
/dev/tty.usbserial-XXXXXXX
/dev/cu.usbserial-XXXXXXX
```

Each board will have different values for *XXXXXXX*. Select either entry.

On Linux, if your board is an Uno it will probably be listed as:

```
/dev/ttyACMX(Arduino/Genuino Uno)
```

If you have an older board, it may be listed as follows:

```
/dev/ttyUSB-X
```

*X* is usually 0, but you will see 1, 2, etc. if you have multiple boards connected at once. Select the entry that corresponds to your Arduino.

---

**TIP**

If you have so many entries in the Port menu that you can't figure out which one goes to your Arduino, try this: look at the menu with the Arduino unplugged from your computer, then plug the Arduino in and look for the menu option that wasn't there before. Another approach is to select the ports one by one, and try uploading until you see the lights on the board flicker to indicate that the code is uploading.

---

Click the upload button (in Figure 1-6, it's the second button from the left), or choose Sketch→Upload (Ctrl-U; ⌘+U on a Mac).

The IDE will compile the code, as in Recipe 1.3. After the software is compiled, it is uploaded to the board. If this is a fresh-out-of-the-box Arduino that's preloaded with the Blink sketch, you will see the onboard LED (labeled as Onboard LED in Figure 1-4) stop blinking. When the upload begins, two LEDs (labeled as Serial LEDs in Figure 1-4) near the onboard LED should flicker for a couple of seconds as the code uploads.

The onboard LED should then start flashing as the code runs. The location of the onboard LED differs across some Arduino models, such as the Leonardo, MKR boards, and third-party Arduino clones.

## Discussion

For the IDE to send the compiled code to the board, the board needs to be plugged into the computer, and you need to tell the IDE which board and serial port you are using.

When an upload starts, whatever sketch is running on the board is stopped (if you were running the Blink sketch, the LED will stop flashing). The new sketch is uploaded to the board, replacing the previous sketch. The new sketch will start running when the upload has successfully completed.

> **NOTE**
>
> Some older Arduino boards and compatibles do not automatically interrupt the running sketch to initiate upload. In this case, you need to press the Reset button on the board just after the software reports that it is done compiling (when you see the message about the size of the sketch). It may take a few attempts to get the timing right between the end of the compilation and pressing the Reset button.

The IDE will display an error message if the upload is not successful. Problems are usually due to the wrong board or serial port being selected or the board not being plugged in. The currently selected board and serial port are displayed in the status bar at the bottom of the Arduino window.

## See Also

For more, see the Arduino troubleshooting page.

# 1.5 Creating and Saving a Sketch

## Problem

You want to create a sketch and save it to your computer.

## Solution

To open an editor window ready for a new sketch, launch the IDE (see Recipe 1.3), go to the File menu, and select New. Delete the boilerplate code that is in the Sketch Editor window, and paste the following code in its place (it's similar to the Blink sketch, but the blinks last twice as long):

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

void loop()
{
  digitalWrite(LED_BUILTIN, HIGH); // set the LED on
  delay(2000);                     // wait for two seconds
  digitalWrite(LED_BUILTIN, LOW);  // set the LED off
  delay(2000);                     // wait for two seconds
}
```

Compile the code by clicking the compile button (the top-left button with a triangle inside), or select Sketch → Verify/Compile (see Recipe 1.3).

Upload the code by clicking the upload button, or choose File → Upload to I/O board (see Recipe 1.4). After uploading, the LED should blink, with each flash lasting two seconds.

You can save this sketch to your computer by clicking the Save button, or select File → Save. You can save a sketch using a new name by selecting the Save As menu option. A dialog box will open where you can enter the filename.

## Discussion

When you save a file in the IDE, a standard dialog box for the operating system will open. It suggests that you save the sketch to a folder called *Arduino* in your *My Documents* folder (or your *Documents* folder on a

Mac). You can replace the default sketch name with a meaningful name that reflects the purpose of your sketch. Click Save to save the file.

> **NOTE**
>
> The default name is the word *sketch* followed by the current date. Sequential letters starting from *a* are used to distinguish sketches created on the same day. Replacing the default name with something meaningful helps you to identify the purpose of a sketch when you come back to it later.

If you use characters that the IDE does not allow (e.g., the space character), the IDE will automatically replace these with valid characters.

Arduino sketches are saved as plain-text files with the extension *.ino*. Older versions of the IDE used the *.pde* extension, also used by Processing. They are automatically saved in a folder with the same name as the sketch.

You can save your sketches to any folder, but if you use the default folder (the *Arduino* folder in your *Documents* folder; *~/Arduino* on Linux) your sketches will appear in the Sketchbook menu of the Arduino software and be easier to locate.

If you have edited one of the built-in examples, you will not be able to save the changed file into the examples folder, so you will be prompted to save it into a different folder.

After you have made changes, you will see a dialog box asking if you want to save the sketch when a sketch is closed. The § symbol following the name of the sketch in the top bar of the IDE window indicates that the sketch code has changes that have not yet been saved on the computer. This symbol is removed when you save the sketch.

As you develop and modify a sketch, you will want a way to keep track of changes. The easiest way to do this is to use the Git version control system (see this Atlassian Git Tutorial page for installation information). Git is typically accessed using a command-line interface (there are graphical clients available as well). The basic workflow for putting a sketch under version control in Git is:

- Figure out which folder your sketch resides in. You can find this using Sketch → Show Sketch Folder. This will open the sketch folder in your computer's file manager.

- Open a command line (on Windows, Command Prompt; on Linux or macOS, open a Terminal). Use the `cd` command to change to the directory where your sketch is located. For example, if you saved a sketch called Blink in the default sketch folder location, you'd be able to change to that directory with the following on macOS, Linux, and Windows, respectively:

  ```
  $ cd ~/Documents/Arduino/Blink
  $ cd ~/Arduino/Blink
  > cd %USERPROFILE%\Documents\Arduino\Blink
  ```

- Initialize the Git repository with the `git init` command.

- Add the Sketch file to Git with `git add Blink.ino` (replace `Blink.ino` with the name of your sketch). If you add any additional files to your sketch folder, you'll need to add them with the `git add` *filename* command.

- After you have made substantial changes, type **git commit -a -m "your comment here"**. Replace "your comment here" with something that describes the change you made.

After you've committed a change to Git, you can use `git log` to see a history of your changes. Each one of those changes will have a *commit hash* associated with it:

```
commit 87e962e54fe46d9e2a00575f7f0d1db6b900662a (HEAD -> master)
Author: Brian Jepson <bjepson@gmail.com>
Date:   Tue Jan 14 20:58:56 2020 -0500

    made massive improvements

commit 0ae1a1bcb0cd245ca9427352fc3298d6ccb91cef (HEAD -> master)
Author: Brian Jepson <bjepson@gmail.com>
Date:   Tue Jan 14 20:56:45 2020 -0500
```

```
    your comment here
```

With these hashes, you can work with older versions of files (you don't need the full hash, just enough of it to differentiate between versions). You can restore an old version with `git checkout` *hash filename*, as in `git checkout 0ae1 Blink.ino`. You can compare versions with `git diff` *firsthash..secondhash*, as in `git diff 0ae1..7018`. See *https://git-scm.com/doc* for complete documentation on Git.

Frequent compiling as you modify or add code is a good way to check for errors. It will be easier to find and fix any errors because they will usually be associated with what you have just written.

---

**NOTE**

Once a sketch has been uploaded onto the board there is no way to download it back to your computer. Make sure you save any changes to your sketches that you want to keep.

---

If you try to save a sketch file that is not in a folder with the same name as the sketch, the IDE will inform you that this can't be opened as is and suggest you click OK to create the folder for the sketch with the same name.

---

**NOTE**

Sketches must be located in a folder with the same name as the sketch. The IDE will create the folder automatically when you save a new sketch.

Sketches made with older versions of Arduino software have a different file extension (*.pde*). The IDE will open them, and when you save the sketch it will create a file with the new extension (*.ino*). Code written for early versions of the IDE may not be able to compile in version 1.0. Most of the changes to get old code running are easy to do.

---

# 1.6 An Easy First Arduino Project
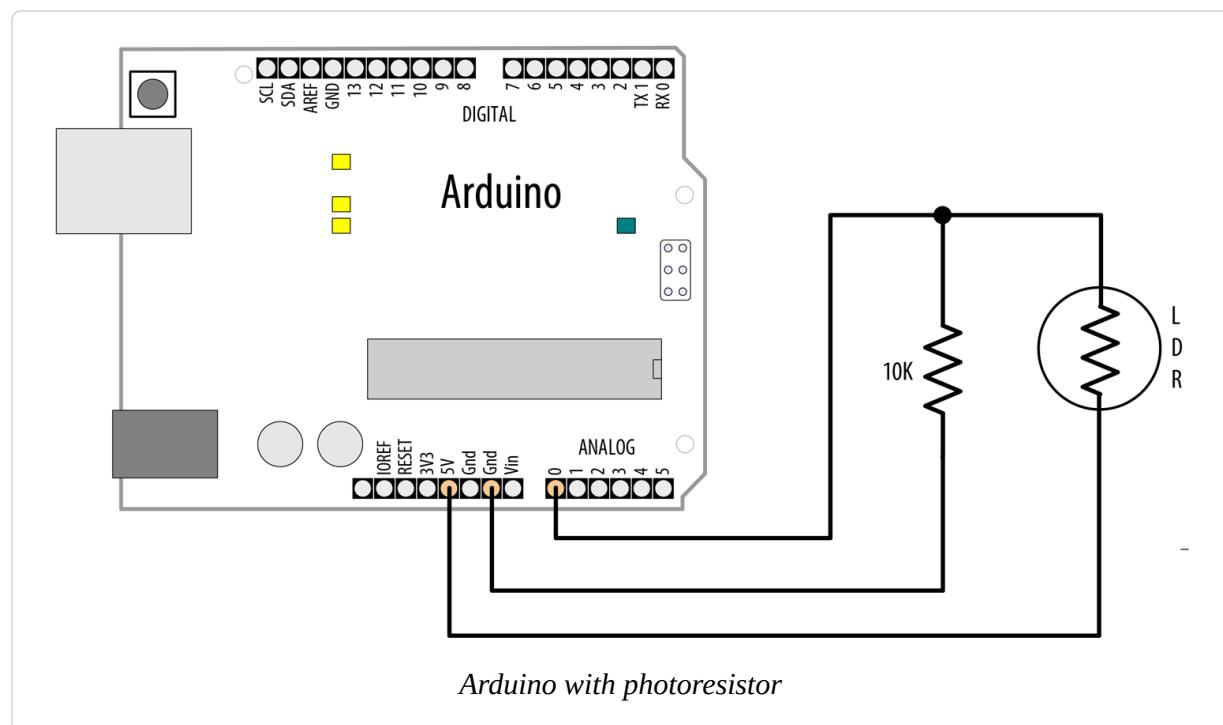
## Problem

You want to get started with a project that is easy to build and fun to use.

## Solution

This recipe provides a taste of some of the techniques that are covered in detail in later chapters.

The sketch is based on the LED blinking code from the previous recipe, but instead of using a fixed delay, the rate is determined by a light-sensitive sensor called a photoresistor or light-dependent resistor (LDR for short, see Recipe 6.3). Wire the photoresistor as shown in Figure 1-7.



*Arduino with photoresistor*

If you are not familiar with building a circuit from a schematic, see Appendix B for step-by-step illustrations on how to make this circuit on a breadboard.

> **NOTE**
>
> Photoresistors contain a compound (cadmium sulfide) that is a hazardous substance. You can use a phototransistor if you live in a jurisdiction where it is difficult to obtain a photoresistor, or if you simply prefer to not use a photoresistor. A phototransistor has a long lead and a short lead, much like an LED. You can wire it exactly as shown in the figure, but you must connect the long lead to 5V and the short lead to the resistor and pin 0. Be sure to buy a phototransistor such as Adafruit part number 2831 that can sense visible light so you can test it with a common light source.

The following sketch reads the light level of a photoresistor connected to analog pin 0. The light level striking the photoresistor will change the blink delay of the internal onboard LED:

```
/*
 * Blink with photoresistor sketch
 */
const int sensorPin = A0;          // connect sensor to analog input 0

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);  // enable output on the led pin
}

void loop()
{
  int delayval = analogRead(sensorPin); // read the analog input
  digitalWrite(LED_BUILTIN, HIGH);      // set the LED on
  delay(delayval);                      // delay is dependent on light level
  digitalWrite(LED_BUILTIN, LOW);       // set the LED off
  delay(delayval);
}
```

The code in this recipe and throughout this book uses the `const int` expression to provide meaningful names (`sensorPin`) for constants instead of numbers (`0`). See Recipe 17.5 for more on the use of constants.

## Discussion

The value of the resistor shown in Figure 1-7 depends on the range of your photoresistor: you will want a resistor that is in the ballpark of the

maximum (dark) resistance of your photoresistor (you can find this by covering the photoresistor while you measure its resistance on a multimeter). So if your photoresistor measures 10K ohms in darkness, use a 10K resistor. If you are using a phototransistor, you will generally be OK with a value between 1K and 10K. The light level on the sensor will change the voltage level on analog pin 0. The `analogRead` command (see Chapter 6) provides a value that ranges from around 200 when the sensor is dark to 800 or so when it is very bright (the sensitivity will vary depending on the type of photoresistor and resistor you use, and whether you use a phototransistor in place of the photoresistor). The analog reading determines the duration of the LED on and off times, so the blink delay increases with light intensity.

You can scale the blink rate by using the Arduino `map` function as follows:

```
/*
 * Blink with photoresistor (scaled) sketch
 */
const int sensorPin = A0;     // connect sensor to analog input 0

// low and high values for the sensor readings; you may need to adjust these
const int low  = 200;
const int high = 800;

// The next two lines set the min and max delay between blinks.
const int minDuration = 100;  // minimum wait between blinks
const int maxDuration = 1000; // maximum wait between blinks

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);  // enable output on the LED pin
}

void loop()
{
  int delayval = analogRead(sensorPin);     // read the analog input

  // the next line scales the delay value between the min and max values
  delayval = map(delayval, low, high, minDuration, maxDuration);
  delayval = constrain(delayval, minDuration, maxDuration);

  digitalWrite(LED_BUILTIN, HIGH); // set the LED on
```

```
    delay(delayval);                // delay is dependent on light level
    digitalWrite(LED_BUILTIN, LOW);  // set the LED off
    delay(delayval);
}
```

> **NOTE**
>
> If you're not seeing any change in values as you adjust the light, you will need to play with the values for low and high. If you are using a phototransistor and aren't getting changes in the blink rate, try a value of 10 for low.

Recipe 5.7 provides more details on using the map function to scale values. Recipe 3.5 has details on using the constrain function to ensure values do not exceed a given range. If, for some reason, your delay value is outside the range between low and high, map will return a value outside the range between minDuration and maxDuration. If you call constrain after map as shown in the sketch, you will avoid the problem of out-of-range values.

If you want to view the value of the delayval variable on your computer, you can print this to the Arduino Serial Monitor as shown in the revised loop code that follows. The sketch will display the delay value in the Serial Monitor. You open the Serial Monitor window in the Arduino IDE by clicking the icon on the right of the top bar (see Chapter 4 for more on using the Serial Monitor):

```
/*
 * Blink sketch with photoresistor (scaled with serial output)
 */
const int sensorPin = A0;     // connect sensor to analog input 0

// Low and high values for the sensor readings. You may need to adjust these.
const int low  = 200;
const int high = 800;

// the next two lines set the min and max delay between blinks
const int minDuration = 100;  // minimum wait between blinks
const int maxDuration = 1000; // maximum wait between blinks

void setup()
```
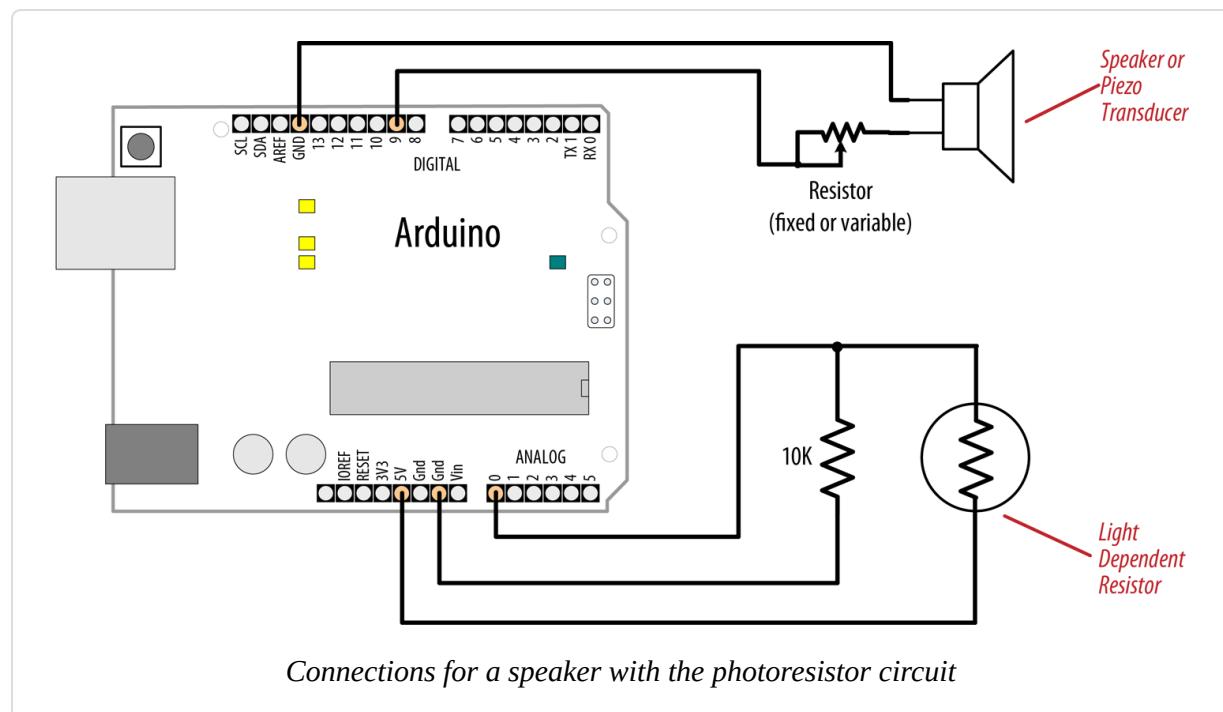
```
{
  pinMode(LED_BUILTIN, OUTPUT);  // enable output on the led pin
  Serial.begin(9600);           // initialize Serial
}

void loop()
{
  int delayval = analogRead(sensorPin);    // read the analog input
  // the next line scales the delay value between the min and max values
  delayval = map(delayval, low, high, minDuration, maxDuration);
  delayval = constrain(delayval, minDuration, maxDuration);

  Serial.println(delayval);         // print delay value to serial monitor
  digitalWrite(LED_BUILTIN, HIGH);  // set the LED on
  delay(delayval);                  // delay is dependent on light level
  digitalWrite(LED_BUILTIN, LOW);   // set the LED off
  delay(delayval);
}
```

You can use the sensor to control the pitch of a sound by connecting a small speaker to the pin, as shown in Figure 1-8.



*Connections for a speaker with the photoresistor circuit*

You will need to increase the on/off rate on the pin to a frequency in the audio spectrum. This is achieved, as shown here, by decreasing the min and max durations:

```
/*
 * Speaker sketch with photoresistor
 */
const int outputPin = 9;  // Speaker connected to digital pin 9
const int sensorPin = A0; // connect sensor to analog input 0

const int low  = 200;
const int high = 800;

const int minDuration = 1;  // 1 ms on, 1 ms off (500 Hz)
const int maxDuration = 10; // 10 ms on, 10 ms off (50 Hz)

void setup()
{
  pinMode(outputPin, OUTPUT); // enable output on the led pin
}

void loop()
{
  int sensorReading = analogRead(sensorPin); // read the analog input
  int delayval = map(sensorReading, low, high, minDuration, maxDuration);
  delayval = constrain(delayval, minDuration, maxDuration);

  digitalWrite(outputPin, HIGH); // set the pin on
  delay(delayval);               // delay is dependent on light level
  digitalWrite(outputPin, LOW);  // set the pin off
  delay(delayval);
}
```

## See Also

For a full discussion on audio output with Arduino, see Chapter 9.
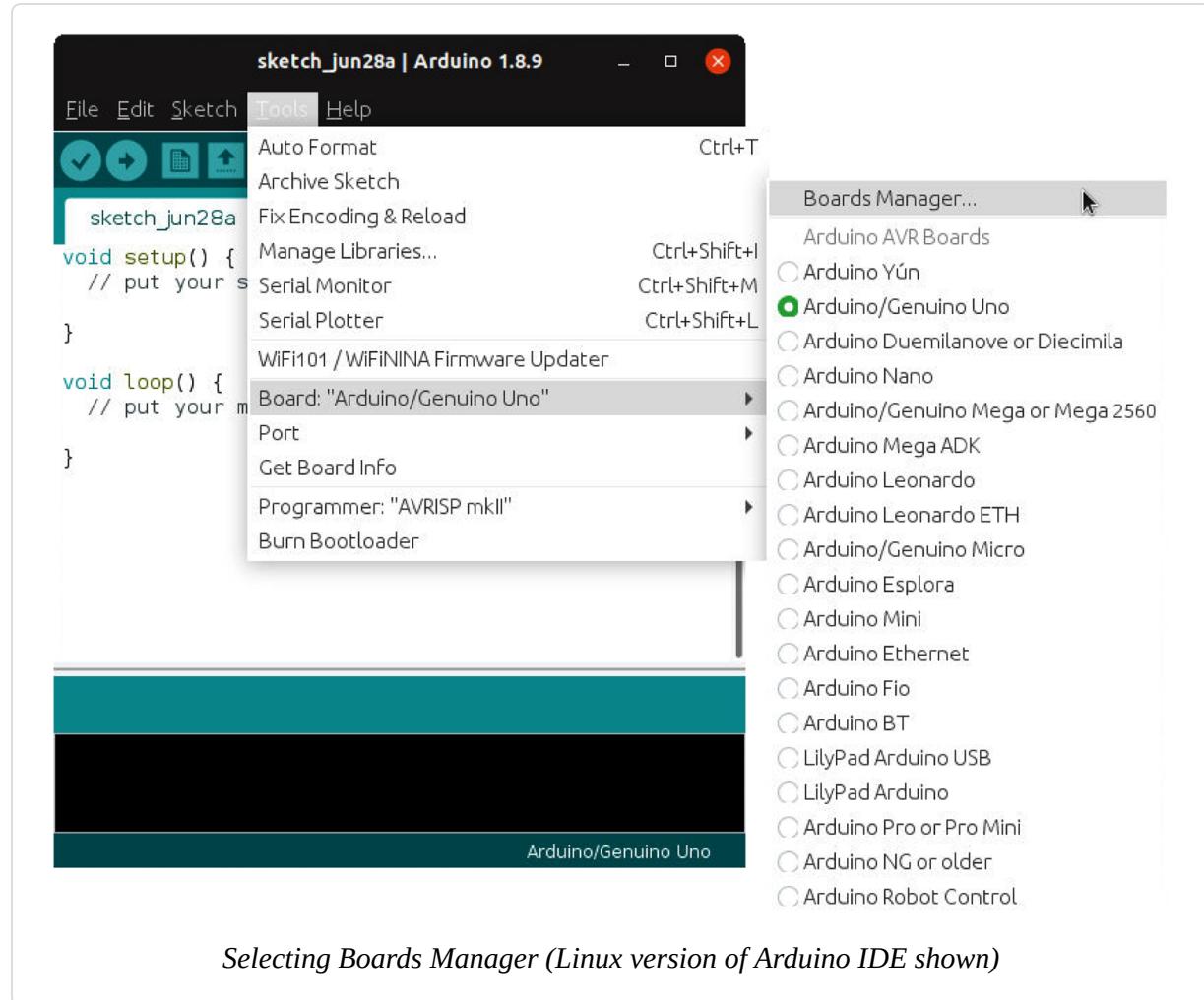
# 1.7 Using Arduino with Boards Not Included in the Standard Distribution

## Problem

You want to use a board such as the Arduino MKR 1010, but it does not appear in the boards menu.

## Solution

To use the MKR 1010 with Arduino, you need to add its details to the Arduino software you have already downloaded. To do this go to Tools → Board → Boards Manager (Figure 1-9).



*Selecting Boards Manager (Linux version of Arduino IDE shown)*
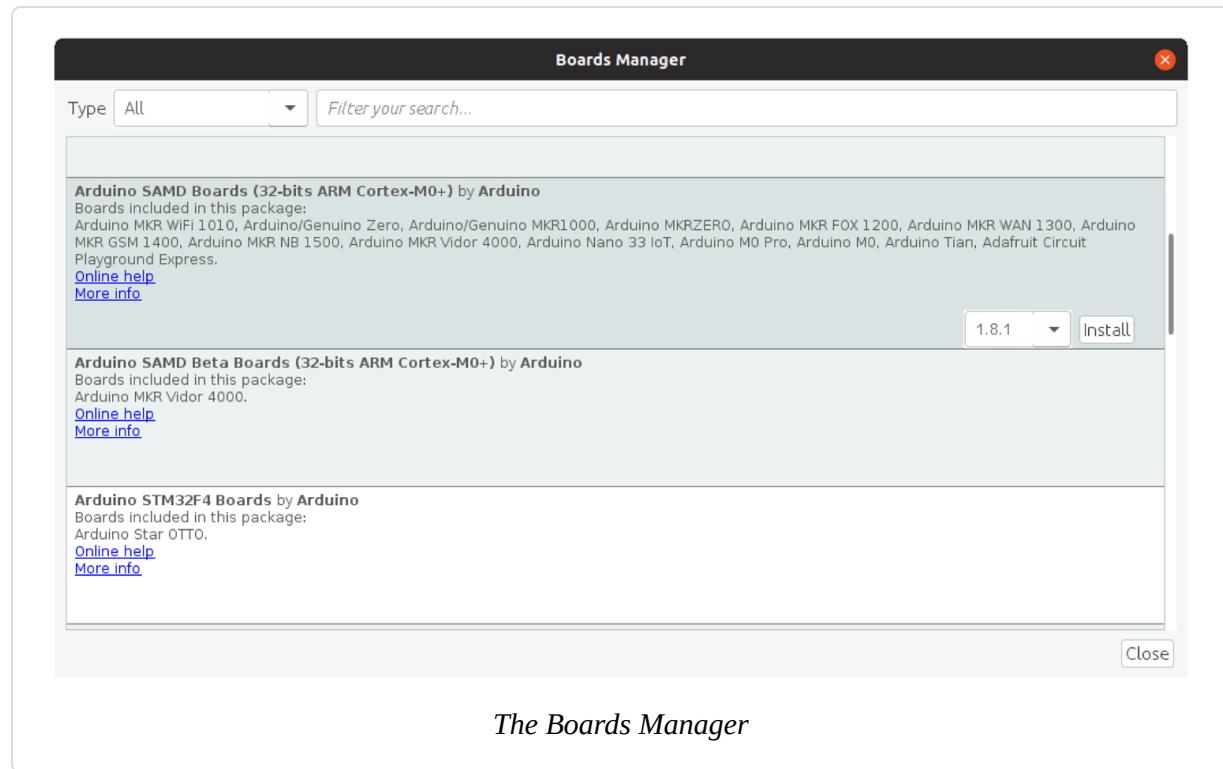
As this window opens, the list of board definitions available online will be checked to ensure you have the latest versions available, so wait till this has finished.

## Adding Other Boards to the Boards Menu

The procedure described here is similar for other boards you may want to add to the boards menu. Check the documentation for your board to find the location of the definition files.
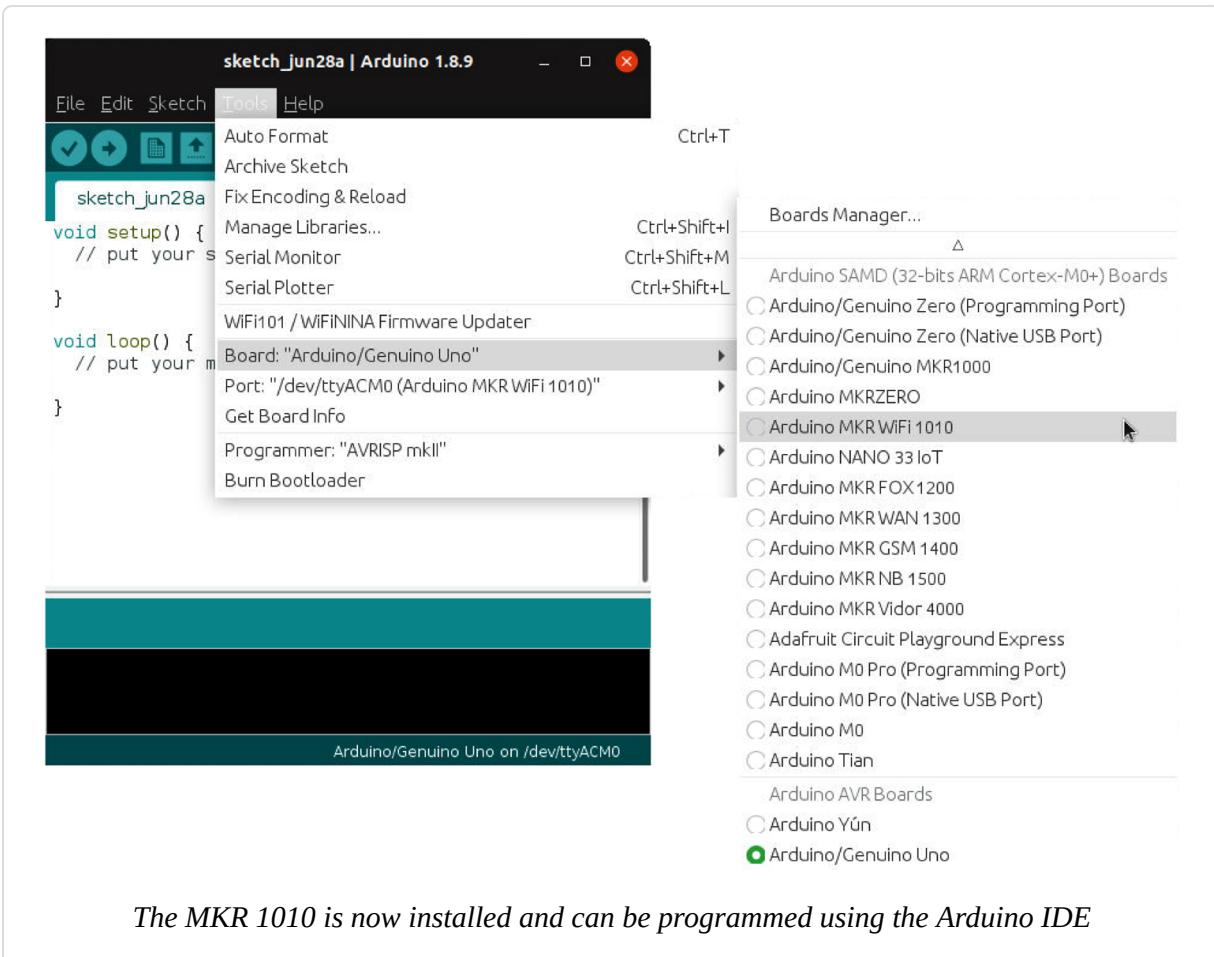
The window that opens (Figure 1-10) shows you the board definitions that are already installed and ones that are available to download.



*The Boards Manager*

To find the MKR 1010 you can scroll down the list, or type its name in the filter box. For the MKR 1010, you'll need to select the Arduino SAMD Boards entry from the list. Once you have selected it, click install and it will be downloaded and added to the Arduino IDE. This may take some time.

Once it has finished you can add other boards, or click Close to finish using the Boards Manager. If you open Tools → Board, you should now have the option of selecting the MKR 1010 as shown in Figure 1-11.

*The MKR 1010 is now installed and can be programmed using the Arduino IDE*

## Discussion
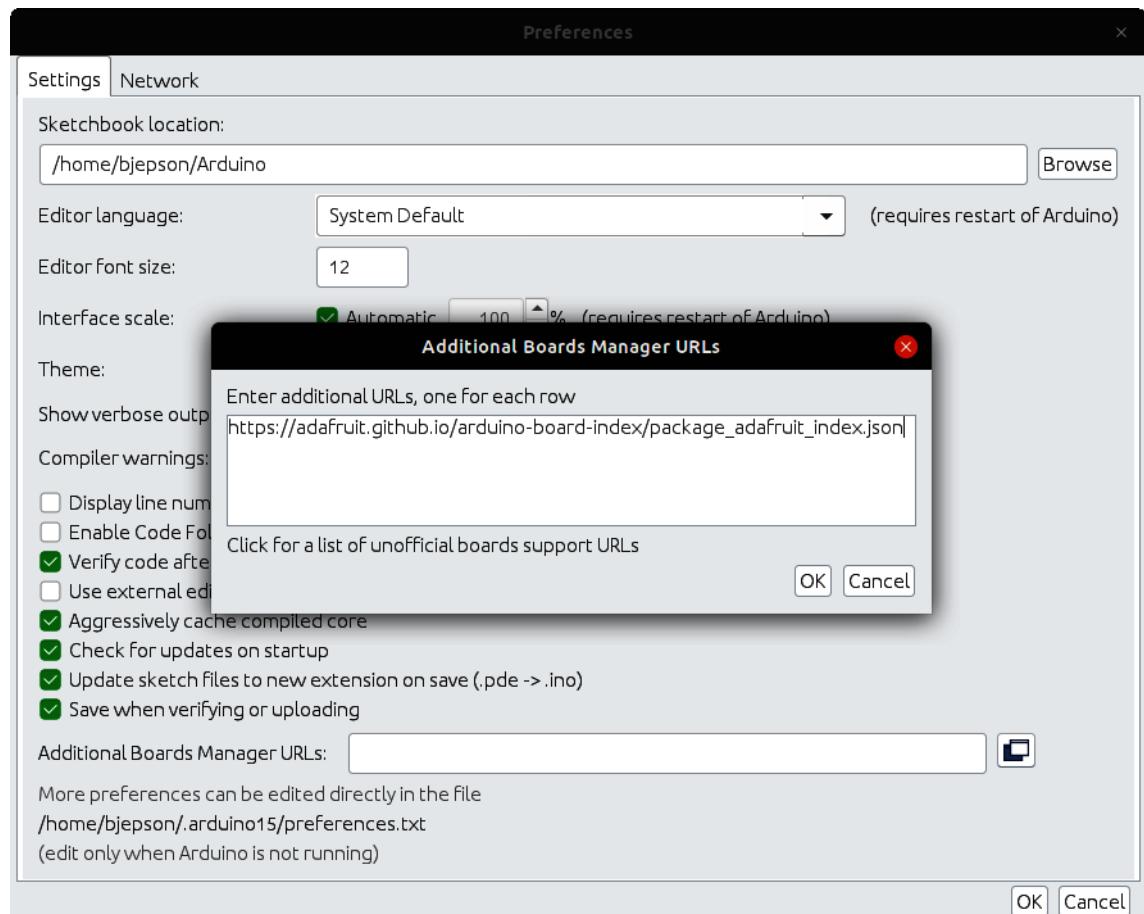
The files that you download when you do this describe how to map the programming concepts in Arduino that connect to specific bits of hardware in the board's microcontroller chip, to where that hardware is located in a specific chip or family of chips.

Once you have added the description for a particular chip, you will often be able to work with a family of boards that use that chip. For example, adding support for the MKR 1010 board also provides support for the Arduino Zero as both boards use the same microcontroller chip.

To facilitate support for the growing number of Arduino and Arduino-compatible boards, the Arduino IDE added a Boards Manager in release 1.6. The Boards Manager was developed to enable people to easily add and remove board details from their installation. It also enables you to update

the board support files if newer versions are available, or choose the version you use if you need to use a particular one. The Arduino IDE no longer includes the description files for all the Arduino boards, so even if you download the latest IDE you may not get the descriptions for the board you have.

The Boards Manager also enables third parties to add the details of their boards to the system. If their board descriptions are available online in the correct format, you can add the location as one of the places for Boards Manager to use to populate the list it produces. This means those files will also get checked whenever the Boards Manager updates its details, so you get notified of updates and can use the same mechanism to update them once they are installed. To do this, go to Arduino→Preferences and click the icon to the right of the Additional Boards Manager URLs field, and the Additional Boards Manager URL dialog will appear as shown in Figure 1-12.

*Preferences after clicking the icon to the right of the Additional Boards Manager URLs entry*

If the people who made the board provide a URL to add to Arduino, paste it into the "additional URLs" dialog box (on a separate line if there are any other entries). If there isn't an explicit URL, click the text below the box to go to the web page that maintains a list of unofficial Arduino board description URLs and see if you can find a link there.

If you want to use a Teensy board, you need to download a separate installer program from the Teensy website. It is important that you use a Teensy installer that has support for the IDE version that you are using. A compatible version is usually produced within a week or two of a new Arduino release.

## See Also
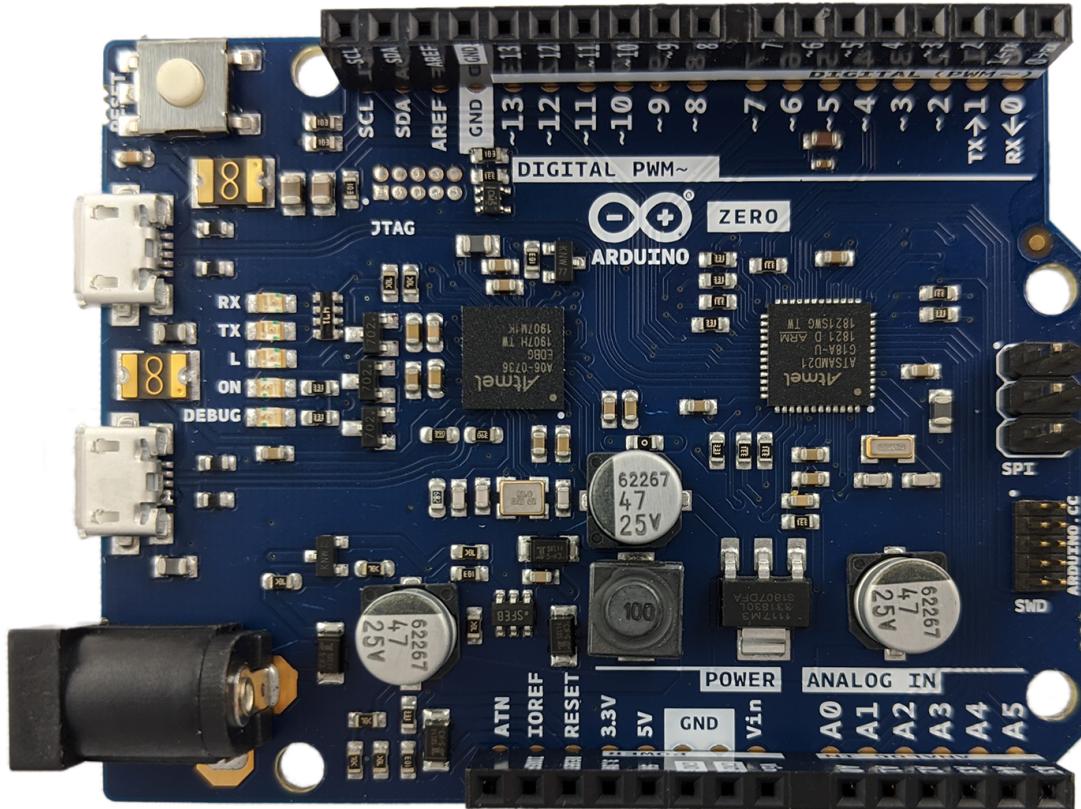
# 1.8 Using a 32-Bit Arduino (or Compatible)

## Problem

You want 32-bit performance in the Uno form factor.

## Solution

The Arduino Zero (Figure 1-13) has the familiar pin layout of the Uno but has much more memory and a faster processor. If you have trouble obtaining a Zero, Adafruit's Metro M0 Express and SparkFun's RedBoard Turbo are compatible alternatives.



*The Arduino/Genuino Zero board*

Despite the similar physical layout of the pins, there are a number of differences. What distinguishes these boards from the Uno is that they use a 32-bit ARM chip, the Microchip SAMD21. The following sketch, similar to the previous recipe, highlights some significant differences between the ARM-based boards and the Uno:

---

**NOTE**

If you're not hearing any change in values as you adjust the light, you will need to play with the values for `low` and `high`. If you are using a phototransistor and aren't getting changes in the blink rate, try a value of 10 for `low`. If your ambient light is from a fluorescent or LED source, you may hear a distinct warbling to the sound due to flicker in such sources that are visually imperceptible.

---

```
/*
 * Zero wave sketch
 */
const int outputPin = A0; // headphones connected to analog 0
const int sensorPin = A1; // connect sensor to analog input 1

const int low  = 200;
const int high = 800;

const int sampleCount = 16;  // number of samples used to render one cycle

const int minDur = 1000000/(sampleCount*500); // period in uS for 500 Hz
const int maxDur = 1000000/(sampleCount*50);  // period for 50 Hz

// table of values for 16 samples of one sine wave cycle
static int sinewave[sampleCount] = {
  0x1FF,0x2B6,0x355,0x3C7,0x3FF,0x3C7,0x355,0x2B6,
  0x1FF,0x148,0x0A9,0x037,0x000,0x037,0x0A9,0x148
};

void setup()
{
  analogWriteResolution(10); // set the Arduino DAC resolution to maximum
}

void loop()
{
  int sensorReading = analogRead(sensorPin);    // read the analog input
  int duration = map(sensorReading, low, high, minDur, maxDur);
```
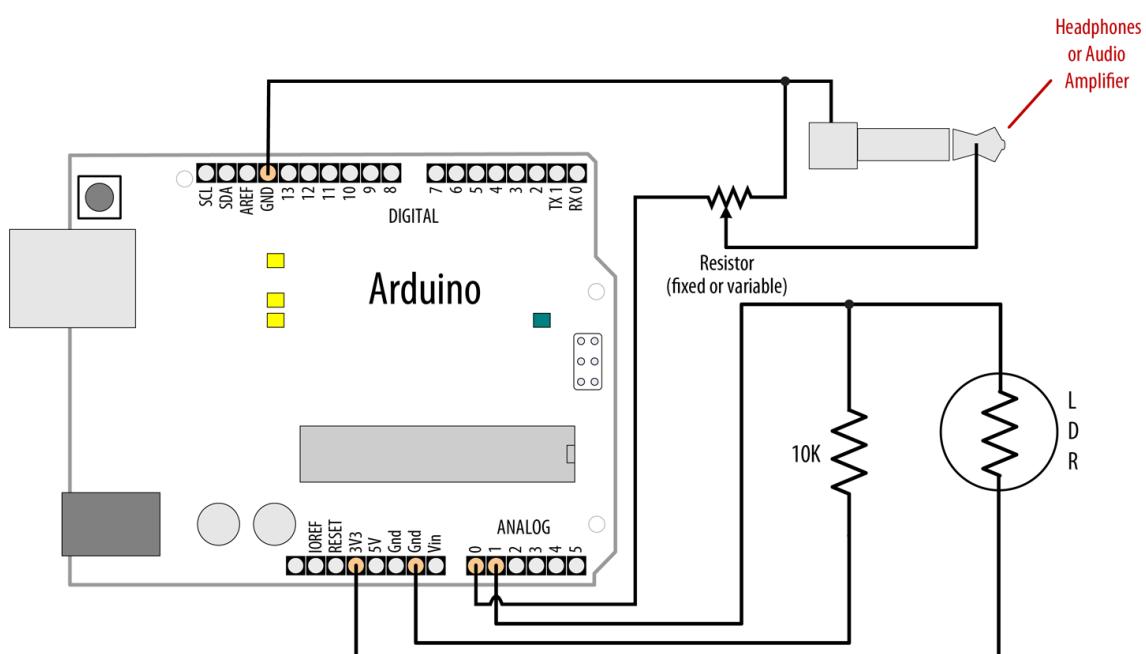
```
    duration = constrain(duration, minDur, maxDur);
    duration = constrain(duration, minDur, maxDur);

    for(int sample=0; sample < sampleCount; sample++) {
      analogWrite(outputPin, sinewave[sample]);
      delayMicroseconds(duration);
    }
  }
}
```

Before you can load sketches on the Zero, Adafruit Metro M0 or M4, or SparkFun RedBoard, open the Arduino Boards Manager and install the appropriate package (see Recipe 1.7). If you are using an Adafruit or SparkFun board, you'll need to add its board manager URL to the Arduino IDE first. See Adafruit or SparkFun for details. After you've installed support for your SAMD board, use the Tools menu to configure the Arduino IDE to use that board and set the correct serial port for connecting to it. Connect a resistor, potentiometer, and photoresistor (also known as a light-dependent resistor) as shown in Figure 1-14. Next, upload the code using the Arduino IDE.



*Connections for audio output with the photoresistor circuit for the Zero board*

## Discussion

Although the wiring may appear similar to Figure 1-8 at first glance, the sensor input and audio output use different pins. These boards have a digital-to-analog converter (DAC) that can create more realistic audio than the binary output of standard digital pins. However, the DAC is only available on analog pin 0 so the sensor input is here connected to analog pin 1.

Another difference that may not be obvious from the figure is that these boards can only drive up to 7 mA on a pin, compared to 40 mA on the Uno. And because the pin voltage ranges from 0 to 3.3 volts, compared to the 0- to 5-volt range of the Uno, the maximum power delivered to a pin is almost 10 times less than the Uno. For that reason, the output pins should be connected to headphones or an amplifier input as they will not drive a speaker directly.

The sketch uses a lookup table of 16 samples per sine wave cycle, however these boards are fast enough to handle much higher resolutions, and you can increase the number of samples to improve the purity of the signal.

## See Also

Arduino Zero quick start guide

More on audio with Arduino in Chapter 9

# Arduino Programming

## 2.0 Introduction

Though much of an Arduino project will involve integrating the Arduino board with supporting hardware, you need to be able to tell the board what to do with the rest of your project. This chapter introduces core elements of Arduino programming, shows nonprogrammers how to use common language constructs, and provides an overview of the language syntax for readers who are not familiar with C or C++, the language that Arduino uses.

Since making the examples interesting requires making Arduino do something, the recipes use physical capabilities of the board that are explained in detail in later chapters. If any of the code in this chapter is not clear, feel free to jump forward, particularly to Chapter 4 for more on serial output and Chapter 5 for more on using digital and analog pins. You don't need to understand all the code in the examples, though, to see how to perform the specific capabilities that are the focus of the recipes. Here are some of the more common functions used in the examples that are covered in the next few chapters:

```
Serial.println(value);
```

Prints the value to the Arduino IDE's Serial Monitor so you can view Arduino's output on your computer; see Recipe 4.1.

```
pinMode(pin, mode);
```

Configures a digital pin to read (input) or write (output) a digital value; see the introduction to Chapter 5.

```
digitalRead(pin);
```

Reads a digital value (`HIGH` or `LOW`) on a pin set for input; see Recipe 5.1.

```
digitalWrite(pin, value);
```

Writes the digital value (`HIGH` or `LOW`) to a pin set for output; see Recipe 5.1.

# 2.1 A Typical Arduino Sketch

**Problem**

You want to understand the fundamental structure of an Arduino program. We'll show this structure in the following sketch, which programs an Arduino to continually flash an LED light.

**Solution**

Programs for Arduino are usually referred to as sketches; the first users were artists and designers, and *sketch* highlights the quick-and-easy way to have an idea realized. The terms *sketch* and *program* are interchangeable. Sketches contain code—the instructions the board will carry out. Code that needs to run only once (such as to set up the board for your application) must be placed in the `setup` function. Code to be run continuously after the initial setup has finished goes into the `loop` function. Here is a typical sketch:

```
// The setup() method runs once, when the sketch starts
void setup()
{
  pinMode(LED_BUILTIN, OUTPUT); // initialize the onboard LED as an output
}

// the loop() method runs over and over again,
void loop()
{
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on
  delay(1000);                       // wait a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off
  delay(1000);                       // wait a second
}
```

When the Arduino IDE finishes uploading the code, and every time you power on the board after you've uploaded this code, it starts at the top of

the sketch and carries out the instructions sequentially. It runs the code in `setup` once and then goes through the code in `loop`. When it gets to the end of `loop` (marked by the closing bracket, `}`) it calls the `loop` function again, and does so over and over again until you disconnect power or reset the board.

## Discussion

This example continuously flashes an LED by writing `HIGH` and `LOW` outputs to a pin. See Chapter 5 to learn more about using Arduino pins. When the sketch begins, the code in `setup` sets the pin mode (so it's capable of lighting an LED). After the code in `setup` is completed, the code in `loop` is repeatedly called (to flash the LED) for as long as the Arduino board is powered on.

You don't need to know this to write Arduino sketches, but experienced C/C++ programmers may wonder where the expected `main()` entry point function has gone. It's there, but it's hidden under the covers by the Arduino build environment. The build process creates an intermediate file that includes the sketch code and the following additional statements. Here's what the `main` function looks like for 8-bit boards (32-bit boards are similar):

```c
int main( void )
{
  init();

  initVariant();

#if defined(USBCON)
  USBDevice.attach();
#endif

  setup();

  for (;;)
  {
    loop();
    if (serialEventRun) serialEventRun();
```

```
    }

    return 0;
}
```

The first thing that happens is a call to an `init()` function that initializes the Arduino hardware. After that, `initVariant()` gets called. This is a rarely used hook to give makers of Arduino-compatible boards a way to invoke their own custom initialization routines. If the microcontroller on the board has dedicated USB hardware, `main` will prepare (attach) it for use.

Next, your sketch's `setup()` function is called. Finally, your `loop()` function is called over and over. Because the `for` loop never terminates, the `return` statement is never executed.

> **NOTE**
>
> Right after each call to `loop`, the `main` function will call `serialEventRun` if it's supported on your board (it's not available on boards that are based on the ATmega32U4 such as the Leonardo). This allows you to add a special function called `serialEvent` in your sketch that will be called whenever data is available on the serial port (see Recipe 4.3).

## See Also

Recipe 1.4 explains how to upload a sketch to the Arduino board.

Chapter 17 and the Arduino CLI sketch build process page provide more on the build process.

# 2.2 Using Simple Primitive Types (Variables)

## Problem

Arduino has different types of variables to efficiently represent values. You want to know how to select and use these Arduino data types.

## Solution

Although the `int` (short for *integer*) data type is the most common choice for the numeric values encountered in Arduino applications, you can use Tables 2-1 and 2-2 to determine the data type that fits the range of values your application expects. Table 2-1 shows data types for 8-bit boards, and Table 2-2 shows data types for 32-bit boards.

*Arduino data types for 8-bit boards such as the Uno*

| Numeric types | Bytes | Range | Use |
|---|---|---|---|
| `int` | 2 | –32768 to 32767 | Represents positive and negative integer values. |
| `unsigned int` | 2 | 0 to 65535 | Represents only positive values; otherwise, similar to `int`. |
| `long` | 4 | –2147483648 to 2147483647 | Represents a very large range of positive and negative values. |
| `unsigned long` | 4 | 4294967295 | Represents a very large range of positive values. |
| `float` | 4 | 3.4028235E+38 to –3.4028235E+38 | Represents numbers with fractions; use to approximate real-world measurements. |
| `double` | 4 | Same as `float` | In Arduino, `double` is just another name for `float`. |
| `bool` | 1 | `false` (0) or `true` (1) | Represents true and false values. |
| `char` | 1 | –128 to 127 | Represents a single character. Can also represent a signed numeric value between –128 and 127. |
| `byte` | 1 | 0 to 255 | Similar to `char`, but for unsigned values. |

| Other types | Use |
|---|---|
| `String` | Represents a sequence of characters typically used to contain text. |
| `void` | Used only in function declarations where no value is returned. |

*Arduino data types for 32-bit boards such as the Zero and 101*

| Numeric types | Bytes | Range | Use |
|---|---|---|---|
| `short int` | 2 | –32768 to 32767 | Same as `int` on 8-bit boards. |
| `unsigned short int` | 2 | 0 to 65535 | Same as `unsigned int` on 8-bit boards. |
| `int` | 4 | –2147483648 to 2147483647 | Represents positive and negative integer values. |
| `unsigned int` | 4 | 0 to 4294967295 | Represents only positive values; otherwise, similar to `int`. |
| `long` | 4 | –2147483648 to 2147483647 | Same as `int`. |
| `unsigned long` | 4 | 4294967295 | Same as `unsigned int`. |
| `float` | 4 | ±3.4028235E+38 | Represents numbers with fractions; use to approximate real-world measurements. |
| `double` | 8 | ±1.7976931348623158E+308 | 32-bit boards have much greater range and precision than 8-bit boards. |
| `bool` | 1 | `false` (0) or `true` (1) | Represents true and false values. |
| `char` | 1 | –128 to 127 | Represents a single character. Can also represent a signed value between –128 and 127. |
| `byte` | 1 | 0 to 255 | Similar to `char`, but for unsigned values. |

| Other types | Use |
|---|---|
| `String` | Represents a sequence of characters typically used to contain text. |
| `void` | Used only in function declarations where no value is returned. |

# Discussion

Except in situations where maximum performance or memory efficiency is required, variables declared using `int` will be suitable for numeric values if the values do not exceed the range (shown in Table 2-1) and if you don't need to work with fractional values. Most of the official Arduino example code declares numeric variables as `int`. But sometimes you do need to choose a type that specifically suits your application. This is especially important if you are calling library functions that return values other than `int`. Take, for example, the `millis` function shown in Recipe 2.10 and other recipes. It returns an `unsigned long` value. If you use an `int` on an 8-bit board to store the results of that function, you won't get a warning, but you will get the wrong results because an `int` is not large enough to hold the maximum value of a `long`. Instead, after you reach 32,767, it will roll over to -32,768. If you were to try to stuff a `long` into an `unsigned int`, you'll roll over to zero after you pass the maximum value for an `unsigned int` (65,535).

Sometimes you need negative numbers and sometimes you don't, so numeric types come in two varieties: `signed` and `unsigned`. `unsigned` values are always positive. Variables without the keyword `unsigned` in front are signed so that they can represent negative and positive values. One reason to use `unsigned` values is when the range of `signed` values will not fit the range of the variable (an unsigned variable has twice the capacity of a signed variable). Another reason programmers choose to use `unsigned` types is to clearly indicate to people reading the code that the value expected will never be a negative number.

On a 32-bit board an `int` requires twice as many bytes as on an 8-bit board, however, memory is ample on 32-bit boards, so most code for 8-bit will run on 32-bit boards. A rare exception is code that assumes that `ints` will always be represented in memory using 2 bytes, something well-written code and libraries should not do.

`bool` (boolean) types have two possible values: `true` or `false`. They are commonly used to store values that represent a yes/no condition. You may also see `bool` types used in place of the built-in constants `HIGH` and `LOW`,

which are used to modify (with `digitalWrite()`) or determine (with `digitalRead()`) the state of a digital I/O pin. For example, the statement `digitalWrite(LED_BUILTIN, HIGH);` will transmit power to the pin that the built-in LED is connected to. Using `LOW` instead of `HIGH` will turn off the power. You can use `true` or `false` in place of `HIGH` or `LOW`, and you are likely to find examples of this in code you find online. You will also see examples where 1 and 0 are used (1 is equivalent to true and 0 is equivalent to false). However, it is a bad habit to make assumptions about the underlying value of a constant, so you should always use the constants `HIGH` and `LOW`. It is extremely unlikely that you would ever come across an Arduino variant where `HIGH` was equal to `false`. But there are many other constants you will come across, and most of them do not have such an explicit and obvious relationship to their underlying values.

## See Also

The Arduino reference provides details on data types.

# 2.3 Using Floating-Point Numbers

## Problem

Floating-point numbers are used for values expressed with decimal points (this is the way to represent fractional values). You want to calculate and compare these values in your sketch.

## Solution

The following code shows how to declare floating-point variables, illustrates problems you can encounter when comparing floating-point values, and demonstrates how to overcome them:

```
/*
 * Floating-point example
 * This sketch initialized a float value to 1.1
 * It repeatedly reduces the value by 0.1 until the value is 0
```

```
  */

float value = 1.1;

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

void loop()
{
  value = value - 0.1;   // reduce value by 0.1 each time through the loop
  if( value == 0)
  {
     Serial.println("The value is exactly zero");
  }
  else if(almostEqual(value, 0))
  {
    Serial.print("The value ");
    Serial.print(value,7); // print to 7 decimal places
    Serial.println(" is almost equal to zero, restarting countdown");
    value = 1.1;
  }
  else
  {
    Serial.println(value);
  }
  delay(250);
}

// returns true if the difference between a and b is small
bool almostEqual(float a, float b)
{
  const float DELTA = .00001; // max difference to be almost equal
  if (a == 0) return fabs(b) <= DELTA;
  if (b == 0) return fabs(a) <= DELTA;
  return fabs((a - b) / max(fabs(a), fabs(b))) <= DELTA;
}
```

## Discussion

Floating-point math is not exact, and values returned can have a small approximation error. The error occurs because floating-point values cover a huge range, so the internal representation of the value can only hold an

approximation. Because of this, you need to test if the values are within a range of tolerance rather than exactly equal.

The Serial Monitor output from this sketch is as follows:

```
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
The value -0.0000001 is almost equal to zero, restarting countdown
1.00
0.90
```

The output starts over from the beginning (1.00) and continues the countdown.

You may expect the code to print "The value is exactly zero" after value is 0.1 and then 0.1 is subtracted from it. But value never equals exactly zero; it gets very close, but that is not good enough to pass the test: if (value == 0). This is because the only memory-efficient way that floating-point numbers can contain the huge range in values they can represent is by storing an approximation of the number.

The solution to this is to check if a variable is close to the desired value, as shown in this recipe's Solution.

The almostEqual function tests if the variable value is within a margin of the desired target and returns true if so. The acceptable range is set with the constant DELTA; you can change this to smaller or larger values as required. The function named fabs (short for *floating-point absolute value*) returns the absolute value of a floating-point variable, and this is used to test the difference between the given parameters.

Before the `almostEqual` function compares the difference between `a` and `b` to `DELTA`, it scales that difference by the maximum value of either `a` or `b`. This is necessary to account for the fact that the precision of floating-point values varies by their magnitude. In fact, because this code compares a value to 0, this expression is not necessary because the logic in the preceding two lines takes over when either `a` or `b` is 0. Table 2-3 shows what would happen at different orders of magnitude for pairs of Start and Comparison values. Equal At shows the value reached by the starting value when they are considered equal. Unscaled Difference shows the difference between `a` and `b` when `almostEqual` determines they are almost equal. Scaled Difference shows the difference that the final line in `almostEqual` uses to make that determination. As you can see, by the time you get up to 100, the unscaled value exceeds the `DELTA` of 0.00001.

*Counting down in floating point*

| Start | Comparison | Equal at | Unscaled difference | Scaled difference |
| --- | --- | --- | --- | --- |
| 11.1 | 10 | 9.9999962 | 0.0000038 | 0.0000004 |
| 101.1 | 100 | 100.0000153 | 0.0000153 | 0.0000002 |
| 1001.1 | 1000 | 1000.0002441 | 0.0002441 | 0.0000002 |

**NOTE**

Floating point approximates numbers because it only uses 32 bits to hold all values within a huge range. Eight bits are used for the decimal multiplier (the exponent), and that leaves 24 bits for the sign and value—only enough for seven significant decimal digits.

**WARNING**

Although `float` and `double` are exactly the same on Arduino Uno, `double`s do have a higher precision on 32-bit boards and many other platforms. If you are importing code that uses `float` and `double` from another platform, check that there is sufficient precision for your application.

## See Also

The Arduino reference for `float`

# 2.4 Working with Groups of Values

## Problem

You want to create and use a group of values (called *arrays*). The arrays may be a simple list or they could have two or more dimensions. You want to know how to determine the size of the array and how to access the elements in the array.

## Solution
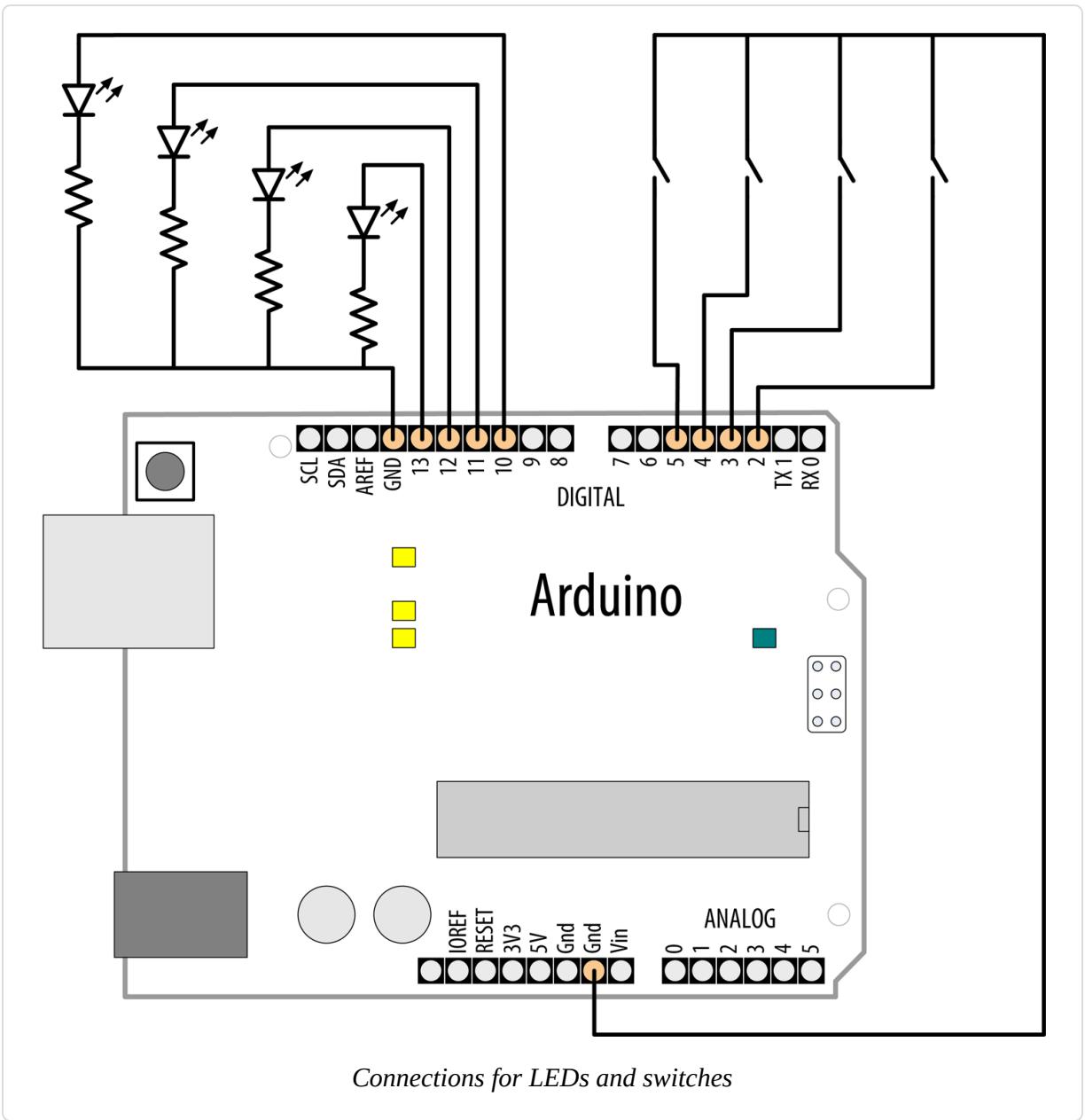
This sketch creates two arrays—an array of integers for pins connected to switches and an array of pins connected to LEDs, as shown in Figure 2-1:

```
/*
 array sketch
 an array of switches controls an array of LEDs
 see Chapter 5 for more on using switches
 see Chapter 7 for information on LEDs
 */

int inputPins[] = {2, 3, 4, 5}; // create an array of pins for switch inputs

int ledPins[] = {10, 11, 12, 13}; // create array of output pins for LEDs

void setup()
{
  for (int index = 0; index < 4; index++)
  {
    pinMode(ledPins[index], OUTPUT);        // declare LED as output
    pinMode(inputPins[index], INPUT_PULLUP); // declare as input
  }
}

void loop() {
  for (int index = 0; index < 4; index++)
  {
    int val = digitalRead(inputPins[index]);  // read input value
    if (val == LOW)                           // check if the switch is
```

```
  pressed
    {
      digitalWrite(ledPins[index], HIGH); // LED on if switch is pressed
    }
    else
    {
      digitalWrite(ledPins[index], LOW);  // turn LED off
    }
  }
}
```

> ### NOTE
>
> If you're familiar with Arduino's INPUT mode, you may be used to wiring the button up with a pull-down resistor that connects the input pin to ground. But with the INPUT_PULLUP mode, you don't need this resistor in your circuit, because this mode enables Arduino's internal pull-up resistors. The difference with the INPUT_PULLUP mode is that when the button is pressed, digitalRead returns LOW rather than HIGH.

*Connections for LEDs and switches*

## Discussion

Arrays are collections of consecutive variables of the same type. Each variable in the collection is called an *element*. The number of elements is called the *size* of the array.

The Solution demonstrates a common use of arrays in Arduino code: storing a collection of pins. Here the pins connect to switches and LEDs (a

topic covered in more detail in Chapter 5). The important parts of this example are the declaration of the array and access to the array elements.

The following line of code declares (creates) an array of integers with four elements and initializes each element. The first element is set equal to 2, the second to 3, and so on:

```
int inputPins[] = {2,3,4,5};
```

If you don't initialize values when you declare an array (for example, when the values will only be available when the sketch is running), you must set each element individually. You can declare the array as follows:

```
int inputPins[4];
```

If you declare the array outside of a function, this declares an array of four elements with the initial value of each element set to zero. (If you declare it inside of a function, such as with `setup()` or `loop()`, the elements will be set to seemingly random values.) The number within the square brackets (`[]`) is the size, and this sets the number of elements. This array has a size of four and can hold, at most, four integer values. The size can be omitted if the array declaration contains initializers (as shown in the first example) because the compiler figures out how big to make the array by counting the number of initializers.

---

**TIP**

Because Arduino's programming environment accepts C or C++ syntax, it is governed by the conventions of those languages. In C and C++, arrays that are declared globally (outside of a function) but are not initialized will have their elements initialized to 0. If they are declared within a function and not initialized, their elements will be undefined, and will probably contain whatever happens to be sitting inside the memory that the array element points to. Uninitialized variables (such as `int i;`) may often be set to zero, but there is no guarantee of this, so it's always important to initialize variables before you try to use their values.

---

The first element of the array is *arrayname*[0]:

```
int firstElement = inputPins[0];  // this is the first element

inputPins[0] = 2; // set the value of the first element to 2
```

The last element is one less than the size, so for a four-element array, the last element is element 3:

```
int lastElement = inputPins[3]; // this is the last element
```

It may seem odd that an array with a size of four has the last element accessed using `array[3]`, but because the first element is `array[0]`, the four elements are:

```
inputPins[0],inputPins[1],inputPins[2],inputPins[3]
```

In the previous sketch, the four elements are accessed using a `for` loop:

```
for (int index = 0; index < 4; index++)
{
  pinMode(ledPins[index], OUTPUT);         // declare LED as output
  pinMode(inputPins[index], INPUT_PULLUP); // declare as input
}
```

This loop will step through the variable `index` with values starting at 0 and ending at 3. It is a common mistake to accidentally access an element that is beyond the actual size of the array. This is a bug that can have many different symptoms, and care must be taken to avoid it. One way to keep your loops under control is to set the size of an array by using a constant as follows:

```
const int PIN_COUNT = 4;  // define a constant for the number of elements
int inputPins[PIN_COUNT] = {2,3,4,5};
int ledPins[PIN_COUNT] = {10, 11, 12, 13};

/* ... */

for(int index = 0; index < PIN_COUNT; index++)
{
  pinMode(ledPins[index], OUTPUT);
```

```
    pinMode(inputPins[index], INPUT_PULLUP);
  }
```

> **WARNING**
>
> The compiler will not report an error if you accidentally try to store or read beyond the size of the array, but it's likely that your sketch will mysteriously crash if you do. You must be careful that you only access elements that are within the bounds you have set. Using a constant to set the size of an array and in code referring to its elements helps your code stay within the bounds of the array.

Another use of arrays is to hold a string of text characters. In Arduino code, these are called *character strings* (*strings* for short). A character string consists of one or more characters, followed by the null character (the value 0) to indicate the end of the string.

> **NOTE**
>
> The null at the end of a character string is not the same as the character 0. The null has an ASCII value of 0, whereas 0 has an ASCII value of 48.

Methods to use strings are covered in Recipes 2.5 and 2.6.

## See Also

Recipe 5.2; Recipe 7.1

# 2.5 Using Arduino String Functionality

## Problem

You want to manipulate text. You need to copy it, add bits together, and determine the number of characters.

## Solution

The previous recipe mentioned how arrays of characters can be used to store text: these character arrays are usually called strings. Arduino has a `String` object that adds rich functionality for storing and manipulating text. Note that the "S" in the `String` object's name is uppercase.

---

**TIP**

The word *String* with an uppercase *S* refers to the Arduino text capability provided by the Arduino String library. The word *string* with a lowercase *s* refers to the group of characters rather than the Arduino `String` functionality.

---

This recipe demonstrates how to use Arduino `String`s.

Load the following sketch onto your board, and open the Serial Monitor to view the results:

```
/*
  Basic_Strings sketch
 */

String text1 = "This text";
String text2 = " has more characters";
String text3;  // to be assigned within the sketch

void setup()
{
  Serial.begin(9600);
  while(!Serial); // Wait for serial port (Leonardo, 32-bit boards)

  Serial.print("text1 is ");
  Serial.print(text1.length());
  Serial.println(" characters long.");

  Serial.print("text2 is ");
  Serial.print(text2.length());
  Serial.println(" characters long.");

  text1.concat(text2);
  Serial.println("text1 now contains: ");
  Serial.println(text1);
}
```

```
void loop()
{
}
```

## Discussion

This sketch creates three variables of type `String`, called `text1`, `text2`, and `text3`. Variables of type `String` have built-in capabilities for manipulating text. The statement `text1.length()` returns (provides the value of) the length (number of characters) in the string `text1`.

`text1.concat(text2)` combines the contents of strings; in this case, it appends the contents of `text2` to the end of `text1` (`concat` is short for *concatenate*).

The Serial Monitor will display the following:

```
text1 is 9 characters long.
text2 is 20 characters long.
text1 now contains:
This text has more characters
```

Another way to combine strings is to use the string addition operator. Add these two lines to the end of the `setup` code:

```
text3 = text1 + " and more";
    Serial.println(text3);
```

The new code will result in the Serial Monitor adding the following line to the end of the display:

```
This text has more characters and more
```

You can use the `indexOf` and `lastIndexOf` functions to find an instance of a particular character in a string. Like arrays, Arduino strings are indexed beginning with 0.

> **TIP**
>
> You will come across Arduino sketches that use arrays of characters or pointers to a sequence of characters rather than the `String` type. See Recipe 2.6 for more on using arrays of characters without the help of the Arduino `String` functionality. See Recipe 17.4 for instructions on storing string literals in flash memory rather than Arduino's main working RAM memory.

If you see a line such as the following:

```
char oldString[] = "this is a character array";
```

the code is using C-style character arrays (see Recipe 2.6). If the declaration looks like this:

```
String newString = "this is a string object";
```

the code uses Arduino `String`s. To convert a C-style character array to an Arduino `String`, just assign the contents of the array to the `String` object:

```
char oldString[] = "I want this character array in a String object";
String newString = oldString;
```

To use any of the functions listed in Table 2-4, you need to invoke them upon an existing string object, as in this example:

```
int len = myString.length();
```

## *Brief overview of Arduino String functions*

| Function | What it does |
| --- | --- |
| `charAt(n)` | Returns the *n*th character of the `String` |
| `compareTo(S2)` | Compares the `String` to the given `String` S2 |
| `concat(S2)` | Returns a new `String` that is the combination of the `String` and S2 |
| `endsWith(S2)` | Returns true if the `String` ends with the characters of S2 |
| `equals(S2)` | Returns true if the `String` is an exact match for S2 (case-sensitive) |
| `equalsIgnoreCase(S2)` | Same as `equals` but is not case-sensitive |
| `getBytes(buffer,len)` | Copies `len`(gth) characters into the supplied byte buffer |
| `indexOf(S)` | Returns the index of the supplied `String` (or character) or –1 if not found |
| `lastIndexOf(S)` | Same as `indexOf` but starts from the end of the `String` |
| `length()` | Returns the number of characters in the `String` |
| `remove(index)` | Removes the character in the `String` at the given index |
| `remove(index, count)` | Removes the specified number of characters from the `String` starting at the given index |
| `replace(A,B)` | Replaces all instances of `String` (or character) A with B |
| `reserve(count)` | Sets aside (allocates) the specified number of bytes to make subsequent `String` operations more efficient |
| `setCharAt(index,c)` | Stores the character c in the `String` at the given index |
| `startsWith(S2)` | Returns true if the `String` starts with the characters of S2 |
| `substring(index)` | Returns a `String` with the characters starting from index to the end of the `String` |
| `substring(index,to)` | Same as above, but the substring ends at the character location before the `to` position |

| Function | What it does |
|---|---|
| `toCharArray(buffer,len)` | Copies up to len characters of the `String` to the supplied buffer |

| | |
|---|---|
| `toFloat()` | Returns the floating-point value of the numeric digits in the `String` |
| `toInt()` | Returns the integer value of the numeric digits in the `String` |
| `toLowerCase()` | Returns a `String` with all characters converted to lowercase |
| `toUpperCase()` | Returns a `String` with all characters converted to uppercase |
| `trim()` | Returns a `String` with all leading and trailing whitespace removed |

See the Arduino reference pages for more about the usage and variants for these functions.

## Choosing between Arduino Strings and C character arrays

Arduino's built-in `String` data type is easier to use than C character arrays, but this is achieved through complex code in the String library, which makes more demands on your Arduino, and is, by nature, more prone to problems.

The `String` data type is so flexible because it makes use of dynamic memory allocation. That is, when you create or modify a `String`, Arduino requests a new region of memory from the C library, and when you're done using a `String`, Arduino needs to release that memory. This usually works smoothly, but 8-bit Arduino boards have so little working RAM (2K on the Arduino Uno) that even small memory leaks can have a big impact on your sketch. A memory leak occurs when, through a bug in a library or incorrect usage of it, memory that you allocate is not released. When this happens, the memory available to Arduino will slowly decrease (until you reboot the Arduino). A related issue is memory fragmentation: as you repeatedly allocate and release memory, Arduino will have successively smaller

contiguous blocks of free memory, which could cause a `String` allocation to fail even if there's otherwise sufficient RAM.

Even if there are no memory leaks, it's complicated to write code to check if a `String` request failed due to insufficient memory (the `String` functions mimic those in Processing, but unlike that platform, Arduino does not have runtime error exception handling). Running out of dynamic memory is a bug that can be very difficult to track down because the sketch can run without problems for days or weeks before it starts misbehaving due to insufficient memory.

If you use C character arrays, you are in control of memory usage: you're allocating a fixed (static) amount of memory at compile time so you don't get memory leaks. Your Arduino sketch will have the same amount of memory available to it all the time it's running. And if you do try to allocate more memory than is available, finding the cause is easier because there are tools that tell you how much static memory you have allocated (see the reference to `avr-objdump` in Recipe 17.1).

However, with C character arrays, it's easier for you to have another problem: C will not prevent you from modifying memory beyond the bounds of the array. So if you allocate an array as `myString[4]`, and assign `myString[4] = 'A'` (remember, `myString[3]` is the end of the array), nothing will stop you from doing this. But who knows what piece of memory `myString[4]` refers to? And who knows whether assigning `'A'` to that memory location will cause you a problem? Most likely, it will cause your sketch to misbehave.

So, Arduino's built-in String library, by virtue of using dynamic memory, runs the risk of eating up your available memory. C's character arrays require care on your part to ensure that you do not exceed the bounds of the arrays you use. So use Arduino's built-in String library if you need rich-text handling capability and you won't be creating and modifying `String`s over and over again. If you need to create and modify them in a loop that is constantly repeating, you're better off allocating a large C character array

and writing your code carefully so you don't write past the bounds of that array.

Another instance where you may prefer C character arrays over Arduino `String`s is in large sketches that need most of the available RAM or flash. The Arduino `StringToInt` example code uses almost 2 KB more flash than the code using a C character array and `atoi` to convert to an `int`. The Arduino `String` version also needs a bit more RAM to store allocation information in addition to the actual string.

If you do suspect that the String library, or any other library that makes use of dynamically allocated memory, might be leaking memory, you can determine how much memory is free at any given time; see Recipe 17.2. Check the amount of RAM when your sketch starts, and monitor it to see whether it's decreasing over time. If you suspect a problem with the String library, search the list of open bugs for "String."

## See Also

The Arduino distribution provides `String` example sketches (File → Examples → Strings).

The `String` reference page

Tutorials for the String library

# 2.6 Using C Character Strings

## Problem

You want to understand how to use raw character strings: you want to know how to create a string, find its length, and compare, copy, or append strings. The core C language does not support the Arduino-style `String` capability, so you want to understand code from other platforms written to operate with primitive character arrays.

## Solution

Arrays of characters are sometimes called *character strings* (or simply *strings* for short). Recipe 2.4 describes Arduino arrays in general. This recipe describes functions that operate on character strings. If you have done any C or C++ programming, you may be used to adding `#include <string.h>` to your code in order to get access to these functions. The Arduino IDE does this for you under the hood, so you don't need the `#include`.

You declare strings like this:

```
char stringA[8]; // declare a string of up to 7 chars plus terminating null
char stringB[8]  = "Arduino"; // as above and initialize the string to
"Arduino"
char stringC[16] = "Arduino"; // as above, but string has room to grow
char stringD[ ]  = "Arduino"; // the compiler inits string and calculates size
```

Use `strlen` (short for *string length*) to determine the number of characters before the terminating null:

```
int length = strlen(string);  // return the number of characters in the string
```

`length` will be 0 for `stringA` and 7 for the other strings shown in the preceding code. The null that indicates the end of the string is not counted by `strlen`.

Use `strcpy` (short for *string copy*) to copy one string to another:

```
strcpy(destination, source);   // copy string source to destination
```

Use `strncpy` (like `strcpy`, but with a limit) to limit the number of characters to copy (useful to prevent writing more characters than the destination string can hold). You can see this used in Recipe 2.7:

```
// copy up to 6 characters from source to destination
strncpy(destination, source, 6);
```

Use `strcat` (short for *string concatenate*) to append one string to the end of another:

```
// append source string to the end of the destination string
strcat(destination, source);
```

> **NOTE**
>
> Always make sure there is enough room in the destination when copying or concatenating strings. Don't forget to allow room for the terminating null.

Use `strcmp` (short for *string compare*) to compare two strings. You can see this used in Recipe 2.7:

```
if(strcmp(str, "Arduino") == 0)
{
  // do something if the variable str is equal to "Arduino"
}
```

## Discussion

Text is represented in the Arduino environment using an array of characters called strings. A string consists of a number of characters followed by a null (the value 0). The null is not displayed, but it is needed to indicate the end of the string to the software.

## See Also

The `str*` functions described in this recipe are part of C's `string.h` library. See one of the many online C/C++ reference pages, such as cplusplus.com and the C++ Referencepage.

# 2.7 Splitting Comma-Separated Text into Groups

## Problem

You have a string that contains two or more pieces of data separated by commas (or any other separator). You want to split the string so that you can use each individual part.

## Solution

This sketch prints the text found between each comma:

```
/*
 * SplitSplit sketch
 * split a comma-separated string
 */

String  text = "Peter,Paul,Mary";  // an example string
String  message = text; // holds text not yet split
int      commaPosition;  // the position of the next comma in the string

void setup()
{
  Serial.begin(9600);
  while(!Serial); // Wait for serial port (Leonardo, 32-bit boards)

  Serial.println(message); // show the source string
  do
  {
    commaPosition = message.indexOf(',');
    if(commaPosition != -1)
    {
      Serial.println( message.substring(0,commaPosition));
      message = message.substring(commaPosition+1, message.length());
    }
    else
    { // here after the last comma is found
      if(message.length() > 0)
        Serial.println(message);  // if there is text after the last comma,
                                  // print it
    }
  }
  while(commaPosition >=0);
}

void loop()
```

```
    {
    }
```

The Serial Monitor will display the following:

```
Peter,Paul,Mary
Peter
Paul
Mary
```

## Discussion

This sketch uses `String` functions to extract text from between commas. The following code:

```
commaPosition = message.indexOf(',');
```

sets the variable `commaPosition` to the position of the first comma in the `String` named `message` (it will be set to –1 if no comma is found). If there is a comma, the `substring` function is used to print the text from the beginning of the string up to, but excluding, the comma. The text that was printed, and its trailing comma, are removed from `message` in this line:

```
message = message.substring(commaPosition+1, message.length());
```

`substring` returns a string starting from `commaPosition+1` (the position just after the first comma) up to the length of the message. This results in that message containing only the text following the first comma. This is repeated until no more commas are found (`commaPosition` will be equal to –1).

If you are an experienced programmer, you can also use the low-level functions that are part of the standard C library. The following sketch has similar functionality to the preceding one using Arduino strings:

```
/*
 * strtok sketch
 * split a comma-separated string
```

```
   */

   const int  MAX_STRING_LEN = 20; // set this to the largest string
                                   // you will process

   char stringList[] = "Peter,Paul,Mary"; // an example string

   char stringBuffer[MAX_STRING_LEN+1]; // static buffer for computation/output

   void setup()
   {
     Serial.begin(9600);
     while(!Serial); // Wait for serial port (Leonardo, 32-bit boards)

     char *str;
     char *p;
     strncpy(stringBuffer, stringList, MAX_STRING_LEN); // copy source string
     Serial.println(stringBuffer);                      // show the source string

     for( str = strtok_r(stringBuffer, ",", &p);        // split using comma
          str;                                          // while str is not null
          str = strtok_r(NULL, ",", &p)                 // get subsequent tokens
        )
     {
       Serial.println(str);
     }
   }

   void loop()
   {
   }
```

---

**TIP**

Although you can use pointers with Arduino, it's generally discouraged in sketches because it makes it harder for beginners to understand your code. In practice, you will rarely see pointers or any advanced C functionality in example sketches. For more information on recommended Arduino coding style, see the Arduino Style Guide.

---

The core functionality comes from the function named `strtok_r` (the name of the version of `strtok` that comes with the Arduino compiler). The first time you call `strtok_r`, you pass it the string you want to tokenize

(separate into individual values). But `strtok_r` overwrites the characters in this string each time it finds a new token, so it's best to pass a copy of the string as shown in this example. Each call that follows uses a `NULL` to tell the function that it should move on to the next token. In this example, each token is printed to the serial port. `*p` is a pointer that `strtok_r` uses to keep track of the string it's working on. You declare it as `*p` but you pass it into the `strtok_r` function as `&p`.

If your tokens consist only of numbers, see Recipe 4.5. This shows how to extract numeric values separated by commas in a stream of serial characters.

## See Also

See AVR Libc home page for more on C string functions such as `strtok_r` and `strcmp`.

Recipe 2.5

See Man7.org for an online reference to the C/C++ functions `strtok_r` and `strcmp`.

# 2.8 Converting a Number to a String

## Problem

You need to convert a number to a string, perhaps to show the number on an LCD or other display.

## Solution

The `String` variable will convert numbers to strings of characters. You can use literal values, or the contents of a variable. For example, the following code will work:

```
String myNumber = String(1234);
```

As will this:

```
int value = 127;
String myReadout = "The reading was ";
myReadout.concat(value);
```

Or this:

```
int value = 127;
String myReadout = "The reading was ";
myReadout += value;
```

## Discussion

If you are converting a number to display as text on an LCD or serial device, the simplest solution is to use the conversion capability built into the LCD and Serial libraries (see Recipe 4.2). But perhaps you are using a device that does not have this built-in support (see Chapter 13) or you want to manipulate the number as a string in your sketch.

The Arduino `String` class automatically converts numerical values when they are assigned to a `String` variable. You can combine (concatenate) numeric values at the end of a string using the `concat` function or the string + operator.

> **TIP**
>
> The + operator is used with number types as well as strings, but it behaves differently with each.

The following code results in `number` having a value of `13`:

```
int number = 12;
    number += 1;
```

With a `String`, as shown here:

```
String textNumber = "12";
    textNumber += 1;
```

`textNumber` is the text string `"121"`.

Prior to the introduction of the `String` class, it was common to find Arduino code using the `itoa` or `ltoa` function. The names come from "integer to ASCII" (`itoa`) and "long to ASCII" (`ltoa`). The `String` version described earlier is easier to use, but the following can be used if you prefer working with C character arrays as described in .

`itoa` and `ltoa` take three parameters: the value to convert, the buffer that will hold the output string, and the number base (10 for a decimal number, 16 for hex, and 2 for binary).

The following sketch illustrates how to convert numeric values using `ltoa`:

```
/*
 * NumberToString
 * Creates a string from a given number
 */

char buffer[12];   // long data type has 11 characters (including the
                   // minus sign) and a terminating null

void setup()
{
  Serial.begin(9600);
  while(!Serial);

  long value = 12345;
  ltoa(value, buffer, 10);

  Serial.print( value);
  Serial.print(" has  ");
  Serial.print(strlen(buffer));
  Serial.println(" digits");

  value = 123456789;
  ltoa(value, buffer, 10);

  Serial.print( value);
  Serial.print(" has  ");
  Serial.print(strlen(buffer));
```

```
    Serial.println(" digits");
}

void loop()
{
}
```

Your buffer must be large enough to hold the maximum number of characters in the string. For 16-bit base 10 (decimal) integers, that is seven characters (five digits, a possible minus sign, and a terminating 0 that always signifies the end of a string); 32-bit long integers need 12-character buffers (10 digits, the minus sign, and the terminating 0). No warning is given if you exceed the buffer size; this is a bug that can cause all kinds of strange symptoms, because the overflow will corrupt some other part of memory that may be used by your program. The easiest way to handle this is to always use a 12-character buffer and always use `ltoa` because this will work on both 16-bit and 32-bit values.

# 2.9 Converting a String to a Number

## Problem

You need to convert a string to a number. Perhaps you have received a value as a string over a communication link and you need to use this as an integer or floating-point value.

## Solution

There are a number of ways to solve this. If the string is received as serial stream data, it can be converted using the `parseInt` function. See the Discussion section of this recipe or Recipe 4.3 for examples of how to do this using the serial port.

Another approach to converting text strings representing numbers is to use the C language conversion function called `atoi` (for `int` variables) or `atol` (for `long` variables).

This sketch terminates the incoming digits on any character that is not a digit (or if the buffer is full). After you upload the sketch, open the Serial Monitor and type some numeric characters, then press Enter or Return. For this to work, though, you'll need to enable the newline option in the Serial Monitor or type some nondigit characters before you press Enter or Return:

```
/*
 * StringToNumber
 * Creates a number from a string
 */

int   blinkDelay;      // blink rate determined by this variable
char strValue[6];      // must be big enough to hold all the digits and the
                       // 0 that terminates the string
int index = 0;         // the index into the array storing the received digits

void setup()
{
 Serial.begin(9600);
 pinMode(LED_BUILTIN, OUTPUT); // enable LED pin as output
}

void loop()
{
  if( Serial.available())
  {
    char ch = Serial.read();
    if(index < 5 && isDigit(ch) ){
      strValue[index++] = ch; // add the ASCII character to the string;
    }
    else
    {
      // here when buffer full or on the first nondigit
      strValue[index] = 0;         // terminate the string with a 0
      blinkDelay = atoi(strValue);  // use atoi to convert the string to an
int
      index = 0;
    }
  }
  blink();
}

void blink()
{
    digitalWrite(LED_BUILTIN, HIGH);
```

```
    delay(blinkDelay/2);  // wait for half the blink period
    digitalWrite(LED_BUILTIN, LOW);
    delay(blinkDelay/2);  // wait for the other half
}
```

## Discussion

The obscurely named `atoi` (for ASCII to `int`) and `atol` (for ASCII to `long`) functions convert a string into integers or long integers. To use them, you have to receive and store the entire string in a character array before you can call the conversion function. The code creates a character array named `strValue` that can hold up to five digits (it's declared as `char strValue[6]` because there must be room for the terminating null). It fills this array with digits from `Serial.read` until it gets the first character that is not a valid digit. The array is terminated with a null and the `atoi` function is called to convert the character array into the variable `blinkDelay`.

A function called `blink` is called that uses the value stored in `blinkDelay`.

As mentioned in the warning in Recipe 2.4, you must be careful not to exceed the bounds of the array. If you are not sure how to do that, see the Discussion section of that recipe.

Arduino also offers the `parseInt` function that can be used to get integer values from `Serial` and `Ethernet` (or any object that derives from the `Stream` class). The following fragment will convert sequences of numeric digits into numbers. It is similar to the solution but does not need a buffer (and does not limit the number of digits to five):

```
void loop()
{
  if( Serial.available())
  {
    int newValue = Serial.parseInt();
    if (newValue != 0) {
        blinkDelay = newValue;
        Serial.print("New delay: ");
        Serial.println(blinkDelay);
    }
```

```
    }
    blink();
  }
```

> ### NOTE
>
> Stream-parsing methods such as `parseInt` use a timeout to return control to your sketch if data does not arrive within the desired interval. The default timeout is one second but this can be changed by calling the `setTimeout` method:
>
> ```
> Serial.setTimeout(1000 * 60); // wait up to one minute
> ```
>
> `parseInt` (and all other stream methods) will return whatever value was obtained prior to the timeout if no delimiter was received. The return value will consist of whatever values were collected; if no digits were received, the return value will be zero.

## See Also

Documentation for `atoi` can be found at the AVR Libc site.

There are many online C/C++ reference pages covering these low-level functions, such as cplusplus or CPP Reference.

See Recipes 4.3 and 4.5 for more about using `parseInt` with `Serial`.

# 2.10 Structuring Your Code into Functional Blocks

### Problem

You want to know how to add functions to a sketch, and understand how to plan the overall structure of a sketch.

### Solution

Functions are used to organize the actions performed by your sketch into functional blocks. Functions package functionality into well-defined *inputs* (information given to a function) and *outputs* (information provided by a function) that make it easier to structure, maintain, and reuse your code.

You are already familiar with the two functions that are in every Arduino sketch: `setup` and `loop`. You create a function by declaring its *return type* (the information it provides), its name, and any optional parameters (values) that the function will receive when it is called.

Here is a simple function that just blinks an LED. It has no parameters and doesn't return anything (the `void` preceding the function indicates that nothing will be returned):

```
// blink an LED once
void blink1()
{
   digitalWrite(LED_BUILTIN, HIGH); // turn the LED on
   delay(500);                      // wait 500 milliseconds
   digitalWrite(LED_BUILTIN, LOW);  // turn the LED off
   delay(500);                      // wait 500 milliseconds
}
```

The following version has a parameter (the integer named `count`) that determines how many times the LED will flash:

```
// blink an LED the number of times given in the count parameter
void blink2(int count)
{
  while(count > 0 )  // repeat until count is no longer greater than zero
  {
    digitalWrite(LED_BUILTIN, HIGH); // turn the LED on
    delay(500);                      // wait 500 milliseconds
    digitalWrite(LED_BUILTIN, LOW);  // turn the LED off
```

```
    delay(500);                          // wait 500 milliseconds
    count = count -1; // decrement count
  }
}
```

---

**NOTE**

Experienced programmers will note that both functions could be named `blink` because the compiler will differentiate them by the type of values used for the parameter. This behavior is called function overloading. The Arduino print function discussed in Recipe 4.2 is a common example. Another example of overloading is in the discussion of Recipe 4.6.

---

That version checks to see if the value of `count` is `0`. If not, it blinks the LED and then reduces the value of `count` by one. This will be repeated until `count` is no longer greater than `0`.

---

**TIP**

A *parameter* is sometimes referred to as an *argument* in some documentation. For practical purposes, you can treat these terms as meaning the same thing.

---

Here is an example sketch with a function that takes a parameter and returns a value. The parameter determines the length of the LED on and off times (in milliseconds). The function continues to flash the LED until a button is pressed, and the number of times the LED flashed is returned from the function. This sketch uses the same wiring as the pull-up sketch from Recipe 5.2:

```
/*
  blink3 sketch
  Demonstrates calling a function with a parameter and returning a value.

  The LED flashes when the program starts and stops when a switch connected
  to digital pin 2 is pressed.
  The program prints the number of times that the LED flashes.
*/
```

```
const int inputPin = 2;           // input pin for the switch

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(inputPin, INPUT);
  digitalWrite(inputPin,HIGH); // use internal pull-up resistor (Recipe 5.2)
  Serial.begin(9600);
}

void loop(){
  Serial.println("Press and hold the switch to stop blinking");
  int count = blink3(250); // blink the LED 250 ms on and 250 ms off
  Serial.print("The number of times the switch blinked was ");
  Serial.println(count);
  while(digitalRead(inputPin) == LOW)
  {
    // do nothing until they let go of the button
  }
}

// blink an LED using the given delay period
// return the number of times the LED flashed
int blink3(int period)
{
  int blinkCount = 0;

  while(digitalRead(inputPin) == HIGH) // repeat until switch is pressed
                                       // (it will go low when pressed)
  {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(period);
    digitalWrite(LED_BUILTIN, LOW);
    delay(period);
    blinkCount = blinkCount + 1; // increment the count
  }
  // here when inputPin is no longer HIGH (means the switch is pressed)
  return blinkCount;  // this value will be returned
}
```

---

### NOTE

A function declaration is a *prototype*—a specification of the name, the types of values that may be passed to the function, and the function's return type. The Arduino build process creates the declarations for you under the covers, so you do not need to follow the standard C requirement of declaring the function separately.

## Discussion

The code in this recipe's Solution illustrates the three forms of function call that you will come across. `blink1` has no parameter and no return value. Its form is:

```
void blink1()
{
  // implementation code goes here...
}
```

`blink2` takes a single parameter but does not return a value:

```
void blink2(int count)
{
  // implementation code goes here...
}
```

`blink3` has a single parameter and returns a value:

```
int blink3(int period)
{
  int result = 0;
  // implementation code goes here...
  return result; // this value will be returned
}
```

The data type that precedes the function name indicates the return type (or no return type if `void`). When *declaring the function* (writing out the code that defines the function and its action), you do not put a semicolon following the parenthesis at the end. When you *use* (call) the function, you do need a semicolon at the end of the line that calls the function.

Most of the functions you come across will be some variation on these forms.

The data type identifier in front of the declaration tells the compiler (and reminds the programmer) what data type the function returns. In the case of `blink1` and `blink2`, `void` indicates that it returns no value. In the case of

`blink3`, `int` indicates that it returns an integer. When creating functions, choose the return type appropriate to the action the function performs.

> **NOTE**
>
> It is recommended that you give your functions meaningful names, and it is a common practice to combine words by capitalizing the first letter of each word, except for the first word. Use whatever style you prefer, but it helps others who read your code if you keep your naming style consistent.

The `blink2` function has a parameter called `count` (when the function is called, `count` is given the value that is passed to the function). The `blink3` function is different in that it is given a parameter called `period`.

The body of the function (the code within the curly brackets) performs the action you want—for `blink1`, it blinks the LED on and then off. For `blink2`, it iterates through a `while` loop the number of times specified by `count`, blinking the LED each time through. For `blink3`, it flashes the LED until you press a button, and then it returns a value to the calling function: the number of times that the LED blinked before you pressed the button.

## See Also

The Arduino function reference page

# 2.11 Returning More than One Value from a Function

## Problem

You want to return two or more values from a function. Recipe 2.10 provided examples for the most common form of a function, one that returns just one value or none at all. But sometimes you need to modify or return more than one value.

## Solution

There are various ways to solve this. The easiest to understand is to have the function change some global variables and not actually return anything from the function:

```
/*
  swap sketch
  demonstrates changing two values using global variables
 */

int x; // x and y are global variables
int y;

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

void loop(){
  x = random(10); // pick some random numbers
  y = random(10);

  Serial.print("The value of x and y before swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);
  swap();

  Serial.print("The value of x and y after swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);Serial.println();


  delay(1000);
}

// swap the two global values
void swap()
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

The swap function changes two values by using global variables. Global variables are easy to understand (global variables are values that are

accessible everywhere and anything can change them), but they are avoided by experienced programmers because it's easy to inadvertently modify the value of a variable or to have a function stop working because you changed the name or type of a global variable elsewhere in the sketch.

A safer and more elegant solution is to pass references to the values you want to change and let the function use the references to modify the values. This is done as follows:

```
/*
  functionReferences sketch
  demonstrates returning more than one value by passing references
 */


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

void loop(){
  int x = random(10); // pick some random numbers
  int y = random(10);

  Serial.print("The value of x and y before swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);
  swapRef(x,y);

  Serial.print("The value of x and y after swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);Serial.println();

  delay(1000);
}

// swap the two given values
void swapRef(int &value1, int &value2)
{
  int temp;
  temp = value1;
  value1 = value2;
  value2 = temp;
}
```

Finally, another option is to use a C *structure,* which can contain multiple *fields,* allowing you to pass and return all kinds of data:

```
/*
  struct sketch
  demonstrates returning more than one value by using a struct
 */

struct Pair {
  int a, b;
};

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

void loop() {
  int x = random(10); // pick some random numbers
  int y = random(10);
  struct Pair mypair = {random(10), random(10)};

  Serial.print("The value of x and y before swapping are: ");
  Serial.print(mypair.a); Serial.print(","); Serial.println(mypair.b);
  mypair = swap(mypair);

  Serial.print("The value of x and y after swapping are: ");
  Serial.print(mypair.a); Serial.print(",");
       Serial.println(mypair.b);Serial.println();

  delay(1000);
}

// swap the two given values
Pair swap(Pair pair)
{
  int temp;
  temp = pair.a;
  pair.a = pair.b;
  pair.b = temp;
  return pair;
}
```

## Discussion

The `swapRef` function is similar to the functions with parameters described in Recipe 2.10, but the ampersand (&) symbol indicates that the parameters are *references*. This means changes in values within the function will also change the value of the variable that is given when the function is called.

You can see how this works by first running the code in this recipe's Solution and verifying that the parameters are swapped. Then modify the code by removing the two ampersands in the function definition.

The changed line should look like this:

```
void swapRef(int value1, int value2)
```

Running the code shows that the values are not swapped—changes made within the function are local to the function and are lost when the function returns.

The `swapPair` function uses a C language feature called a `struct` (or *structure*). A structure contains any number of primitive types or pointers. The amount of memory reserved for a `struct` is equivalent to the size of its elements (on an 8-bit Arduino, a `Pair` would take up four bytes, eight on a 32-bit board). If you are familiar with object-oriented programming, it may be tempting to think of `structs` as similar to classes, but `structs` are nothing more than the data they contain.

# 2.12 Taking Actions Based on Conditions

## Problem

You want to execute a block of code only if a particular condition is true. For example, you may want to light an LED if a switch is pressed or if an analog value is greater than some threshold.

## Solution

The following code uses the wiring shown in :

```
/*
 * Pushbutton sketch
 * a switch connected to digital pin 2 lights the built-in LED
 */

const int inputPin = 2;          // choose the input pin (for a pushbutton)
```

```
void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);     // declare LED pin as output
  pinMode(inputPin, INPUT_PULLUP);  // declare pushbutton pin as input
}

void loop()
{
  int val = digitalRead(inputPin);  // read input value
  if (val == LOW)                   // Input is LOW when the button is pressed
  {
    digitalWrite(LED_BUILTIN, HIGH); // turn LED on if switch is pressed
  }
}
```

## Discussion

The `if` statement is used to test the value of `digitalRead`. An `if` statement must have a test within the parentheses that can only be true or false. In the example in this recipe's Solution, it's `val == LOW`, and the code block following the `if` statement is only executed if the expression is true. A code block consists of all code within the curly brackets (or if you don't use curly brackets, the block is just the next executable statement terminated by a semicolon).

If you want to do one thing if a statement is true and another if it is false, use the `if...else` statement:

```
/*
 * Pushbutton sketch
 * a switch connected to pin 2 lights the built-in LED
 */

const int inputPin = 2;          // choose the input pin (for a pushbutton)

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);    // declare LED pin as output
  pinMode(inputPin, INPUT_PULLUP); // declare pushbutton pin as input
}

void loop()
```

```
{
  int val = digitalRead(inputPin);    // read input value
  if (val == LOW)                      // Input is LOW when the button is
pressed
  {
    // do this if val is LOW
    digitalWrite(LED_BUILTIN, HIGH); // turn LED on if switch is pressed
  }
  else
  {
    // else do this if val is not LOW
    digitalWrite(LED_BUILTIN, LOW);  // turn LED off
  }
}
```

## See Also

See the discussion on Boolean types in Recipe 2.2.

# 2.13 Repeating a Sequence of Statements

## Problem

You want to repeat a block of statements while an expression is true.

## Solution

A `while` loop repeats one or more instructions while an expression is true:

```
/*
 * Repeat
 * blinks while a condition is true
 */

const int sensorPin = A0; // analog input 0

void setup()
{
  Serial.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT); // enable LED pin as output
}

void loop()
```

```
{
  while(analogRead(sensorPin) > 100)
  {
    blink();      // call a function to turn an LED on and off
    Serial.print(".");
  }
  Serial.println(analogRead(sensorPin)); // this is not executed until after
                                          // the while loop finishes!!!
}

void blink()
{
  digitalWrite(LED_BUILTIN, HIGH);
  delay(100);
  digitalWrite(LED_BUILTIN, LOW);
  delay(100);
}
```

This code will execute the statements in the block within the curly brackets ({}) while the value from `analogRead` is greater than 100. This could be used to flash an LED as an alarm while some value exceeded a threshold. The LED is off when the sensor value is 100 or less; it flashes continuously when the value is greater than 100.

## Discussion

Curly brackets define the extent of the code block to be executed in a loop. If curly brackets are not used, only the first line of code will be repeated in the loop, so you should use this style sparingly (or not at all):

```
while(analogRead(sensorPin) > 100)
  blink();    // line immediately following the loop expression is executed
  Serial.print("."); // this is executed only after the while loop finishes!
```

The `do...while` loop is similar to the `while` loop, but the instructions in the code block are executed before the condition is checked. Use this form when you must have the code executed at least once, even if the expression is false:

```
do
{
    blink();  // call a function to turn an LED on and off
    Serial.print(".");
}
while (analogRead(sensorPin) > 100);
```

The preceding code will flash the LED at least once and will keep flashing it as long as the value read from a sensor is greater than 100. If the value is not greater than 100, the LED will only flash once. This code could be used in a battery-charging circuit, if it were called once every 10 seconds or so: a single flash shows that the circuit is active, whereas continuous flashing indicates the battery is charged.

> ### WARNING
>
> Only the code within a `while` or `do` loop will run until the conditions permit exit. If your sketch needs to break out of a loop in response to some other condition such as a timeout, sensor state, or other input, you can use `break`:
>
> ```
> while(analogRead(sensorPin) > 100)
> {
>     blink();
>     Serial.print(".");
>     if(Serial.available())
>     {
>       while(Serial.available())
>       {
>           // consume any pending serial input
>           Serial.read();
>       }
>       break; // any serial input breaks out of while loop
>     }
> }
> ```

# See Also

Chapters 4 and 5

# 2.14 Repeating Statements with a Counter

## Problem

You want to repeat one or more statements a certain number of times. The `for` loop is similar to the `while` loop, but you have more control over the starting and ending conditions.

## Solution

This sketch counts from zero to three by printing the value of the variable `i` in a `for` loop:

```
/*
   ForLoop sketch
   demonstrates for loop
*/

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

void loop(){
  Serial.println("for(int i=0; i < 4; i++)");
  for(int i=0; i < 4; i++)
  {
    Serial.println(i);
  }
  delay(1000);
}
```

The Serial Monitor output from this is as follows (it will be displayed over and over):

```
for(int i=0; i < 4; i++)
0
1
2
3
```

## Discussion

A `for` loop consists of three parts: initialization, conditional test, and iteration (a statement that is executed at the end of every pass through the loop). Each part is separated by a semicolon. In the code in this recipe's Solution, `int i=0;` initializes the variable `i` to `0`; `i < 4;` tests the variable to see if it's less than 4; and `i++` increments `i`.

A `for` loop can use an existing variable, or it can create a variable for exclusive use inside the loop. This version uses the value of the variable `j` created earlier in the sketch:

```
int j;

  Serial.println("for(j=0; j < 4; j++)");
  for(j=0; j < 4; j++ )
  {
    Serial.println(j);
  }
```

This is almost the same as the earlier example, but it does not have the `int` keyword in the initialization part because the variable `j` was already defined. The output of this version is similar to the output of the earlier version:

```
for(j=0; j < 4; j++)
0
1
2
3
```

You control when the loop stops in the conditional test. The previous example tests whether the loop variable is less than 4 and will terminate when the condition is no longer true.

---

**TIP**

If your loop variable starts at 0 and you want it to repeat four times, your conditional statement should test for a value less than 4. The loop repeats while the condition is true, and there are four values that are less than 4 with a loop starting at 0.

The following code tests if the value of the loop variable is less than or equal to 4. It will print the digits from 0 to 4:

```
Serial.println("for(int i=0; i <= 4; i++)");
  for(int i=0; i <= 4; i++)
  {
    Serial.println(i);
  }
```

The third part of a `for` loop is the iterator statement that gets executed at the end of each pass through the loop. This can be any valid C/C++ statement. The following increases the value of `i` by two on each pass:

```
Serial.println("for(int i=0; i < 4; i+= 2)");
  for(int i=0; i < 4; i+=2)  {
    Serial.println(i);
  }
```

That expression only prints the values 0 and 2.

The *iterator* expression can be used to cause the loop to count from high to low, in this case from 3 to 0:

```
Serial.println("for(int i=3; i >= 0 ; i--)");
  for(int i=3; i >= 0 ; i--)
  {
    Serial.println(i);
  }
```

Like the other parts of a `for` loop, the iterator expression can be left blank (you must always have the two semicolons separating the three parts even if they are blank).

This version only increments `i` when an input pin is high. The `for` loop does not change the value of `i`; it is only changed by the `if` statement after `Serial.print`—you'll need to define `inPin` and set it to `INPUT` with `pinMode()`:

```
pinMode(2, INPUT_PULLUP); // this goes in setup()

  /* ... */

  Serial.println("for(int i=0; i < 4; )");
  for(int i=0; i < 4; )
  {
    Serial.println(i);
    if(digitalRead(2) == LOW) {
      i++;  // only increment the value if the button is pressed
    }
  }
```

## See Also

The Arduino reference for the `for` statement

# 2.15 Breaking Out of Loops

## Problem

You want to terminate a loop early based on some condition you are testing.

## Solution

Use the `break` statement as shown in the following sketch:

```
/*
 * break sketch
 * Demonstrates the use of the break statement
 */

const int switchPin = 2; // digital input 2

void setup()
{
  Serial.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT); // enable LED pin as output
  pinMode(switchPin, INPUT_PULLUP); // enable button pin as input
}

void loop()
{
```

```
  while(true) // endless loop
  {
    if(digitalRead(switchPin) == LOW)
    {
      break; // exit the loop if the switch is pressed
    }
    blink(); // call a function to turn an LED on and off
  }
}

void blink()
{
  digitalWrite(LED_BUILTIN, HIGH);
  delay(100);
  digitalWrite(LED_BUILTIN, LOW);
  delay(100);
}
```

## Discussion

This code is similar to the one using `while` loops, but it uses the `break` statement to exit the loop if a digital pin goes high. For example, if a switch is connected on the pin as shown in Recipe 5.2, the loop will exit and the LED will stop flashing even though the condition in the `while` loop is true.

## See Also

The Arduino reference for the `break` statement

An *interrupt* is a facility built into the microcontroller hardware that allows you to take action more or less immediately when a pin state changes. See Recipe 18.2 for more details.

# 2.16 Taking a Variety of Actions Based on a Single Variable

## Problem

You need to do different things depending on some value. You could use multiple `if` and `else if` statements, but the code soon gets complex and

difficult to understand or modify. Additionally, you may want to test for a range of values.

## Solution

The `switch` statement provides for the selection of a number of alternatives. It is functionally similar to multiple `if`/`else if` statements but is more concise:

```
/*
 * SwitchCase sketch
 * example showing switch statement by switching on chars from the serial port
 *
 * sending the character 1 blinks the LED once, sending 2 blinks twice
 * sending + turns the LED on, sending - turns it off
 * any other character prints a message to the Serial Monitor
 */

void setup()
{
  Serial.begin(9600); // Initialize serial port to send and
                      // receive at 9600 baud
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop()
{
  if ( Serial.available()) // Check to see if at least one
                           // character is available
  {
    char ch = Serial.read();
    switch(ch)
    {
    case '1':
      blink();
      break;
    case '2':
      blink();
      blink();
      break;
    case '+':
      digitalWrite(LED_BUILTIN, HIGH);
      break;
    case '-':
```

```
        digitalWrite(LED_BUILTIN, LOW);
        break;
      case '\n': // newline, safe to ignore
        break;
      case '\r': // carriage return, safe to ignore
        break;
      default:
        Serial.print(ch);
        Serial.println(" was received but not expected");
        break;
      }
    }
  }

  void blink()
  {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(500);
    digitalWrite(LED_BUILTIN, LOW);
    delay(500);
  }
```

## Discussion

The `switch` statement evaluates the variable `ch` received from the serial port and branches to the label that matches its value. The labels must be numeric constants. Because the `char` data type is represented as a numeric value (see Recipe 2.2), it is permitted. However, you can't use strings in a `case` statement, and no two labels can have the same value. If you don't have a `break` statement following each expression, the execution will *fall through* into the statement:

```
  case '1':
      blink();    // no break statement before the next label
    case '2':
      blink();    // case '1' will continue here
      blink();
      break;      // break statement will exit the switch expression
```

If the `break` statement at the end of `case '1':` was removed (as shown in the preceding code), when `ch` is equal to the character `1` the `blink` function

will be called three times. Accidentally forgetting the `break` is a common mistake. Intentionally leaving out the `break` is sometimes handy; it can be confusing to others reading your code, so it's a good practice to clearly indicate your intentions with comments in the code.

> **TIP**
>
> If your `switch` statement is misbehaving, check to ensure that you have not forgotten the `break` statements.

The `default:` label is used to catch values that don't match any of the `case` labels. If there is no `default` label, the `switch` expression will not do anything if there is no match.

## See Also

The Arduino reference for the `switch` and `case` statements

# 2.17 Comparing Character and Numeric Values

## Problem

You want to determine the relationship between values.

## Solution

Compare integer values using the relational operators shown in Table 2-5.

*Relational and equality operators*

| Operator | Test for | Example |
|----------|----------|---------|
| `==` | Equal to | `2 == 3 // evaluates to false` |
| `!=` | Not equal to | `2 != 3 // evaluates to true` |

| Operator | Test for | Example |
| --- | --- | --- |
| > | Greater than | `2 > 3 // evaluates to false` |
| < | Less than | `2 < 3 // evaluates to true` |
| >= | Greater than or equal to | `2 >= 3 // evaluates to false` |
| <= | Less than or equal to | `2 <= 3 // evaluates to true` |

The following sketch demonstrates the results of using the comparison operators:

```
/*
 * RelationalExpressions sketch
 * demonstrates comparing values
 */

int i = 1;  // some values to start with
int j = 2;

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

void loop(){
  Serial.print("i = ");
  Serial.print(i);
  Serial.print(" and j = ");
  Serial.println(j);

  if(i < j)
    Serial.println(" i is less than j");
  if(i <= j)
    Serial.println(" i is less than or equal to j");
  if(i != j)
    Serial.println(" i is not equal to j");
  if(i == j)
    Serial.println(" i is equal to j");
  if(i >= j)
    Serial.println(" i is greater than or equal to j");
  if(i > j)
    Serial.println(" i is greater than j");

  Serial.println();
```

```
  i = i + 1;
  if(i > j + 1)
  {
    delay(10000);  // long delay after i is no longer close to j
  }
  else
  {
    delay(1000);   // short delay
  }
}
```

Here is the output:

```
i = 1 and j = 2
 i is less than j
 i is less than or equal to j
 i is not equal to j

i = 2 and j = 2
 i is less than or equal to j
 i is equal to j
 i is greater than or equal to j

i = 3 and j = 2
 i is not equal to j
 i is greater than or equal to j
 i is greater than j
```

## Discussion

Note that the equality operator is the double equals sign, **==**. One of the most common programming mistakes is to confuse this with the assignment operator, which uses a single equals sign.

The following expression will compare the value of **i** to 3. The programmer intended this:

```
if(i == 3)  // test if i equals 3
```

But suppose they put this in the sketch:

```
if(i = 3)  // single equals sign used by mistake!!!!
```

This will always return `true`, because `i` will be set to `3`, so they will be equal when compared.

You can also perform these sorts of comparisons on character values because they are represented as numeric values (see Recipe 2.2). This will evaluate to true:

```
if ('b' > 'a')
```

As will this, because the numeric value of `'a'` is 97 in the ASCII character set that Arduino uses:

```
if ('a' == 97)
```

However, strings cannot be directly compared to numeric values:

```
String word1 = String("Hello");
char word2[] = "World";
if (word1 > 'G') // This will not compile
{
  Serial.println("word1 > G");
}
if (word2 >= 'W') // This also will not compile
{
  Serial.println("word2 >= W");
}
```

But you can always compare a number or char against a single character from a string:

```
if (word1.charAt(0) > 'G')
{
  Serial.println("word1[0] > G");
}
if (word2[0] >= 'W')
{
  Serial.println("word2[0] >= W");
}
```

## See Also

The Arduino reference for conditional and comparison operators

The Arduino ASCII chart reference

# 2.18 Comparing Strings

**Problem**

You want to see if two character strings are identical.

## Solution

There is a function to compare strings, called `strcmp` (short for *string compare*). Here is a fragment showing its use:

```
char string1[ ] = "left";
  char string2[ ] = "right";

  if(strcmp(string1, string2) == 0)
  {
    Serial.println("strings are equal");
  }
```

## Discussion

`strcmp` returns the value 0 if the strings are equal and a value greater than zero if the first character that does not match has a greater value in the first string than in the second. It returns a value less than zero if the first nonmatching character in the first string is less than in the second. Usually you only want to know if they are equal, and although the test for zero may seem unintuitive at first, you'll soon get used to it.

Bear in mind that strings of unequal length will not be evaluated as equal even if the shorter string is contained in the longer one. So:

```
if (strcmp("left", "leftcenter") == 0)  // this will evaluate to false
```

You can compare strings up to a given number of characters by using the `strncmp` function. You give `strncmp` the maximum number of characters to

compare and it will stop comparing after that many characters:

```
if (strncmp("left", "leftcenter", 4) == 0)  // this will evaluate to true
```

Unlike character strings, Arduino `String`s can be directly compared as follows:

```
String stringOne = String("this");
if (stringOne == "this")
{
  Serial.println("this will be true");
}
if (stringOne == "that")
{
  Serial.println("this will be false");
}
```

For more see the tutorial on Arduino string comparison.

## See Also

More information on `strcmp` is available at cplusplus.

See Recipe 2.5 for an introduction to the Arduino `String`.

# 2.19 Performing Logical Comparisons

## Problem

You want to evaluate the logical relationship between two or more expressions. For example, you want to take a different action depending on the conditions of an `if` statement.

## Solution

Use the logical operators as outlined in Table 2-6.

*Logical operators*

| Symbol | Function | Comments |
| --- | --- | --- |
| && | Logical And | Evaluates as `true` if the conditions on both sides of the && operator are true |
| || | Logical Or | Evaluates as `true` if the condition on at least one side of the || operator is true |
| ! | Not | Evaluates as `true` if the expression is false, and `false` if the expression is true |

## Discussion

Logical operators return `true` or `false` values based on the logical relationship. The examples that follow assume you have sensors wired to digital pins 2 and 3 as discussed in Chapter 5.

The logical And operator `&&` will return `true` if both its two operands are true, and `false` otherwise:

```
if( digitalRead(2) && digitalRead(3) )
  blink(); // blink if both pins are HIGH
```

The logical Or operator `||` will return `true` if either of its two operands are true, and `false` if both operands are false:

```
if( digitalRead(2) || digitalRead(3) )
  blink(); // blink if either pin is HIGH
```

The Not operator `!` has only one operand, whose value is inverted—it results in `false` if its operand is true and `true` if its operand is false:

```
if( !digitalRead(2)  )
  blink(); // blink if the pin is not HIGH
```

# 2.20 Performing Bitwise Operations

**Problem**

You want to set or clear certain bits in a value.

## Solution

Use the bit operators as outlined in Table 2-7. The `0b` prefix indicates binary representation of numbers and is used to disambiguate between the decimal and binary numbers in the table.

*Bit operators*

| Symbol | Function | Result | Example |
|---|---|---|---|
| & | Bitwise And | Sets bits in each place to 1 if both bits are 1; otherwise, bits are set to 0. | `3 & 1` equals 1<br><br>(`0b11 & 0b01` equals `0b01`) |
| \| | Bitwise Or | Sets bits in each place to 1 if either bit is 1. | `3 \| 1` equals 3<br><br>(`0b11 \| 0b01` equals `0b11`) |
| ^ | Bitwise Exclusive Or | Sets bits in each place to 1 only if one of the two bits is 1. | `3 ^ 1` equals 2<br><br>(`0b11 ^ 0b01` equals `0b10`) |
| ~ | Bitwise Negation | Inverts the value of each bit. The result depends on the number of bits in the data type. | `~1` equals 254<br><br>(`~00000001` equals `11111110`) |

Here is a sketch that demonstrates the example values shown in Table 2-7:

```
/*
 * bits sketch
 * demonstrates bitwise operators
 */

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

void loop(){
```

```
    Serial.print("3 & 1 equals "); // bitwise And 3 and 1
    Serial.print(3 & 1);            // print the result
    Serial.print(" decimal, or in binary: ");
    Serial.println(3 & 1 , BIN);    // print the binary representation of the
result

    Serial.print("3 | 1 equals "); // bitwise Or 3 and 1
    Serial.print(3 | 1 );
    Serial.print(" decimal, or in binary: ");
    Serial.println(3 | 1 , BIN);    // print the binary representation of the
result

    Serial.print("3 ^ 1 equals "); // bitwise exclusive or 3 and 1
    Serial.print(3 ^ 1);  Serial.print(" decimal, or in binary: ");
    Serial.println(3 ^ 1 , BIN);    // print the binary representation of the
result


    byte byteVal = 1;
    int intVal = 1;

    byteVal = ~byteVal;  // do the bitwise negate
    intVal = ~intVal;

    Serial.print("~byteVal (1) equals "); // bitwise negate an 8-bit value
    Serial.println(byteVal, BIN);  // print the binary representation of the
result
    Serial.print("~intVal (1) equals "); // bitwise negate a 16-bit value
    Serial.println(intVal, BIN);    // print the binary representation of the
result

    delay(10000);
}
```

This is what is displayed on the Serial Monitor:

```
3 & 1 equals 1 decimal, or in binary: 1
3 | 1 equals 3 decimal, or in binary: 11
3 ^ 1 equals 2 decimal, or in binary: 10
~byteVal (1) equals 11111110
~intVal (1) equals 1111111111111111111111111111110
```

## Discussion

Bitwise operators are used to set or test bits. When you "And" or "Or" two values, the operator works on each individual bit. It is easier to see how this works by looking at the binary representation of the values.

The decimal integer 3 is binary 11, and the decimal integer 1 is binary 1. Bitwise And operates on each bit. The rightmost bits are both 1, so the result of And-ing these is 1. Moving to the left, the next bits are 1 and 0; And-ing these results in 0. All the remaining bits are 0, so the bitwise result of these will be 0. In other words, for each bit position where there is a 1 in both places, the result will have a 1; otherwise, it will have a 0. So, `11 & 01` equals 1. Binary numbers are often written with leading zeros because it makes it easier to evaluate the effect of bitwise operations at a glance, but the leading zeros are not significant.

Tables 2-8, 2-9, and 2-10 should help to clarify the bitwise And, Or, and Exclusive Or values.

*Bitwise And*

| Bit 1 | Bit 2 | Bit 1 and Bit 2 |
|-------|-------|-----------------|
| 0     | 0     | 0               |
| 0     | 1     | 0               |
| 1     | 0     | 0               |
| 1     | 1     | 1               |

*Bitwise Or*

| Bit 1 | Bit 2 | Bit 1 or Bit 2 |
|-------|-------|----------------|
| 0     | 0     | 0              |
| 0     | 1     | 1              |
| 1     | 0     | 1              |
| 1     | 1     | 1              |

| Bit 1 | Bit 2 | Bit 1 ^ Bit 2 |
|-------|-------|---------------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

All the bitwise expressions operate on two values, except for the negation operator. This simply flips each bit, so 0 becomes 1 and 1 becomes 0. In the example, the `byte` (8-bit) value 00000001 becomes 11111110. The `int` value has 16 bits, so when each is flipped, the result is 15 ones followed by a single zero.

## See Also

The Arduino reference for the bitwise And, Or, and Exclusive Or operators

# 2.21 Combining Operations and Assignment

## Problem

You want to understand and use compound operators. It is not uncommon to see published code that uses expressions that do more than one thing in a single statement. You want to understand `a += b`, `a >>= b`, and `a &= b`.

## Solution

Table 2-11 shows the compound assignment operators and their equivalent full expression.

*Compound operators*

| Operator | Example | Equivalent expression |
|----------|---------|-----------------------|

| Operator | Example | Equivalent expression |
|---|---|---|
| += | value += 5; | value = value + 5; // add 5 to value |
| -= | value -= 4; | value = value - 4; // subtract 4 from value |
| *= | value *= 3; | value = value * 3; // multiply value by 3 |
| /= | value /= 2; | value = value / 2; // divide value by 2 |
| >>= | value >>= 2; | value = value >> 2; // shift value right two places |
| <<= | value <<= 2; | value = value << 2; // shift value left two places |
| &= | mask &= 2; | mask = mask & 2; // binary-and mask with 2 |
| \|= | mask \|= 2; | mask = mask \| 2; // binary-or mask with 2 |

## Discussion

These compound statements are no more efficient at runtime than the equivalent full expression, and if you are new to programming, using the full expression is clearer. Experienced coders often use the shorter form, so it is helpful to be able to recognize the expressions when you run across them.

## See Also

See the Arduino Reference home page for an index to the reference pages for compound operators.

# Mathematical Operations

### 3.0 Introduction

Almost every sketch uses mathematical operations to manipulate the value of variables. This chapter provides a brief overview of the most common mathematical operations. If you are already familiar with C or C++, you may be tempted to skip this chapter, but we suggest you review it because there are some idioms used by Arduino programmers that you may encounter even if you don't use them yourself (such as the use of `bitSet` to change the value of a bit). If you are new to C and C++, see one of the C reference books mentioned in the Preface.

## 3.1 Adding, Subtracting, Multiplying, and Dividing

### Problem

You want to perform simple math on values in your sketch. You want to control the order in which the operations are performed and you may need to handle different variable types.

### Solution

Use the following code:

```
int myValue;
myValue = 1 + 2;  // addition
myValue = 3 - 2;  // subtraction
myValue = 3 * 2;  // multiplication
myValue = 3 / 2;  // division (the result is 1)
```

### Discussion

Addition, subtraction, and multiplication for integers work much as you expect.

Integer division truncates the fractional remainder in the division example shown in this recipe's Solution; `myValue` will equal 1 after the division (see Recipe 2.3 if your application requires fractional results):

```
int value =   1 + 2 * 3 + 4;
```

Compound statements, such as the preceding statement, may appear ambiguous, but the *precedence* (order) of every operator is well defined. Multiplication and division have a higher precedence than addition and subtraction, so the result will be 11. It's advisable to use parentheses in your code to make the desired calculation precedence clear. `int value = 1 + (2 * 3) + 4;` produces the same result but is easier to read.

Use parentheses if you need to alter the precedence, as in this example:

```
int value =   ((1 + 2) * 3) + 4;
```

The result will be 13. The expression in the inner parentheses is calculated first, so 1 gets added to 2, this then gets multiplied by 3, and finally is added to 4, yielding 13.

You'll need to make sure your result will not exceed the maximum size of the destination variable, because the Arduino IDE will not warn you about that, unless you enable warnings in File → Preferences. See Recipe 2.2. However, even if you use the correct type, you can still overflow the size of the destination variable. Consider this code:

```
// 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day
long seconds_per_day = 60 * 60 * 24;
```

In theory, that should be fine because the result is 86,400, which can fit in a `long` data type. But the value that's really stored in `seconds_per_day` is 20,864. 86,400 is enough to overflow an integer twice (86,400 − 32,768 * 2 = 20,864). The overflow happens because the Arduino IDE's C compiler

sees an arithmetic expression composed of integers, and doesn't know any better. You must tell the compiler that it should treat the whole expression like a `long` by appending L to the first value that is evaluated in the expression:

```
long seconds_per_day = 60L * 60 * 24;
```

If, for some reason, you are using parentheses, remember that innermost parentheses are evaluated first, so this will overflow:

```
long seconds_per_day_plus_one = 1L + 60 * (60 * 24);
```

However, this will run correctly:

```
long seconds_per_day_plus_one = 1 + 60 * (60L * 24);
```

> **NOTE**
>
> Floating-point arithmetic is subject to all the imprecisions described in Recipe 2.3. For example, the following code, which divides 36.3 by 3 and prints the result to 10 decimal places, would display `12.0999994277`:
>
> ```
> Serial.println(36.3/3, 10);
> ```
>
> See Recipe 3.3 for a trick you can use to display an accurate calculation.

## See Also

Recipe 2.2; Recipe 2.3

# 3.2 Incrementing and Decrementing Values

## Problem

You want to increase or decrease the value of a variable.

## Solution

Use the following code:

```
int myValue = 0;

myValue = myValue + 1;   // this adds one to the variable myValue
myValue += 1;            // this does the same as the above

myValue = myValue - 1;   // this subtracts one from the variable myValue
myValue -= 1;            // this does the same as the above

myValue = myValue + 5;   // this adds five to the variable myValue
myValue += 5;            // this does the same as the above
```

## Discussion

Increasing and decreasing the values of variables is one of the most common programming tasks, and Arduino has operators to make this easy. Increasing a value by one is called *incrementing*, and decreasing it by one is called *decrementing*. The longhand way to do this is as follows:

```
myValue = myValue + 1;   // this adds one to the variable myValue
```

But you can also combine the increment and decrement operators with the assign operator, like this:

```
myValue += 1;            // this does the same as the above
```

If you are incrementing or decrementing a value by 1, you can use the abbreviated increment and decrement operators ++ or --:

```
myValue++;               // this does the same as the above
```

When the increment or decrement operators appear after a variable, they are known as the *post-increment* or *post-decrement* operators because they perform their operation after the variable is evaluated. If they appear before the identifier (*pre-increment* or *pre-decrement*), they modify the value before the variable is evaluated:

```
int myVal = 1;
Serial.println(myVal++); // prints 1
Serial.println(myVal);   // prints 2
Serial.println(++myVal); // prints 3
Serial.println(myVal);   // prints 3
```

## See Also

Recipe 3.1

# 3.3 Finding the Remainder After Dividing Two Values

## Problem

You want to find the remainder after you divide two integer values.

## Solution

Use the % symbol (the modulus operator) to get the remainder:

```
int myValue0 =  20 % 10;  // get the modulus(remainder) of 20 divided by 10
int myValue1 =  21 % 10;  // get the modulus(remainder) of 21 divided by 10
```

myValue0 equals 0 (20 divided by 10 has a remainder of 0). myValue1 equals 1 (21 divided by 10 has a remainder of 1).

## Discussion

The modulus operator is surprisingly useful, particularly when you want to see if a value is a multiple of a number. For example, the code in this recipe's Solution can be enhanced to detect when a value is a multiple of 10:

```
for (int myValue = 0; myValue <= 100; myValue += 5)
{
  if (myValue % 10 == 0)
  {
    Serial.println("The value is a multiple of 10");
```

```
    }
  }
```

The preceding code takes the modulus of the `myValue` variable and compares the result to zero (see Recipe 2.17). If the result is zero, a message is printed saying the value is a multiple of 10.

Here is a similar example, but by using 2 with the modulus operator, the result can be used to check if a value is odd or even:

```
for (int myValue = 0; myValue <= 10; myValue++)
{
  if (myValue % 2 == 0)
  {
    Serial.println("The value is even");
  }
  else
  {
    Serial.println("The value is odd");
  }
}
```

This example calculates the hour on a 24-hour clock for any given number of hours offset:

```
void printOffsetHour( int hourNow, int offsetHours)
{
    Serial.println((hourNow + offsetHours) % 24);
}
```

You can also use the modulus operator to help simulate floating-point operations. For example, consider the problem described in Recipe 3.1 where dividing 36.3 by 3 yields `12.0999994277` rather than the expected 12.1. You can multiply the two values by 10, then perform the division as an integer operation to get the integer part:

```
int int_part = 363/30; // result: 12
```

Next, you can calculate the remainder, multiply it by 100, then divide by the divisor to get the fractional part:

```
int remainder = 363 % 30; // result: 3
int fractional_part = remainder * 100 / 30;
```

Finally, print the integer and fractional part separated by a period (full stop) to get 12.10:

```
Serial.print(int_part); Serial.print("."); Serial.println(fractional_part);
```

## See Also

The Arduino reference for % (the modulus operator)

# 3.4 Determining the Absolute Value

## Problem

You want to get the absolute value of a number.

## Solution

abs(x) computes the absolute value of x. The following example takes the absolute value of the difference between readings on two analog input ports (see Chapter 5 for more on analogRead()):

```
int x = analogRead(A0);
int y = analogRead(A1);

if (abs(x-y) > 10)
{
  Serial.println("The analog values differ by more than 10");
}
```

## Discussion

abs(x-y); returns the absolute value of the difference between x and y. It is used for integer (and long integer) values. To return the absolute value of floating-point values, see Recipe 2.3.

## See Also

The Arduino reference for `abs`

# 3.5 Constraining a Number to a Range of Values

### Problem

You want to ensure that a value is always within some lower and upper limit.

### Solution

`constrain(x, min, max)` returns a value that is within the bounds of `min` and `max`:

```
int myConstrainedValue  = constrain(myValue, 100, 200);
```

### Discussion

`myConstrainedValue` is set to a value that will always be greater than or equal to 100 and less than or equal to 200. If `myValue` is less than 100, the result will be 100; if it is more than 200, it will be set to 200.

Table 3-1 shows some example output values using a `min` of 100 and a `max` of 200.

*Output from constrain with min = 100 and max = 200*

| myValue (the input value) | constrain(myValue, 100, 200) |
| --- | --- |
| 99 | 100 |
| 100 | 100 |
| 150 | 150 |

| myValue (the input value) | constrain(myValue, 100, 200) |
|---|---|
| 200 | 200 |
| 201 | 200 |

## See Also

Recipe 3.6

# 3.6 Finding the Minimum or Maximum of Some Values

## Problem

You want to find the minimum or maximum of two or more values.

## Solution

`min(x,y)` returns the smaller of two numbers. `max(x,y)` returns the larger of two numbers:

```
int myValue = analogRead(A0);
int myMinValue = min(myValue, 200);   // myMinValue will be the smaller of
                                      // myVal or 200

int myMaxValue = max(myValue, 100);   // myMaxValue will be the larger of
                                      // myVal or 100
```

## Discussion

Table 3-2 shows some example output values using a `min` of 200. The table shows that the output is the same as the input (`myValue`) until the value becomes greater than 200.

*Output from min(myValue, 200)*

| myValue (the input value) | min(myValue, 200) |
|---|---|

| myValue (the input value) | min(myValue, 200) |
| --- | --- |
| 99 | 99 |
| 100 | 100 |
| 150 | 150 |
| 200 | 200 |
| 201 | 200 |

Table 3-3 shows the output using a max of 100. The table shows that the output is the same as the input (myValue) when the value is greater than or equal to 100.

*Output from max(myValue, 100)*

| myValue (the input value) | max(myValue, 100) |
| --- | --- |
| 99 | 100 |
| 100 | 100 |
| 150 | 150 |
| 200 | 200 |
| 201 | 201 |

Use min when you want to limit the upper bound. That may be counterintuitive, but by returning the smaller of the input value and the minimum value, the output from min will never be higher than the minimum value (200 in the example).

Similarly, use max to limit the lower bound. The output from max will never be lower than the maximum value (100 in the example).

If you want to find the min or max value from more than two values, you can cascade the values as follows:

```
// myMinValue will be the smaller of the three analog readings:
int myMinValue = min(analogRead(0), min(analogRead(1), analogRead(2)));
```

In this example, the minimum value is found for analog ports 1 and 2, and then the minimum of that and port 0. This can be extended for as many items as you need, but take care to position the parentheses correctly. The following example gets the maximum of four values:

```
int myMaxValue = max(analogRead(0),
                     max(analogRead(1),
                         max(analogRead(2), analogRead(3))));
```

## See Also

Recipe 3.5

# 3.7 Raising a Number to a Power

## Problem

You want to raise a number to a power.

## Solution

`pow(x, y)` returns the value of `x` raised to the power of `y`:

```
int myValue =  pow(3,2);
```

This calculates $3^2$, so `myValue` will equal 9.

## Discussion

The `pow` function can operate on integer or floating-point values and it returns the result as a floating-point value:

```
Serial.println(pow(3,2)); // this prints 9.00
int z = pow(3,2);
Serial.println(z);        // this prints 9
```

The first output is `9.00` and the second is `9`; they are not exactly the same because the first `print` displays the output as a floating-point number and the second treats the value as an integer before printing, and therefore displays without the decimal point. If you use the `pow` function, you may want to read Recipe 2.3 to understand the difference between these and integer values.

Here is an example of raising a number to a fractional power:

```
float s = pow(2, 1.0 / 12); // the twelfth root of two
```

The twelfth root of two is the same as 2 to the power of 0.083333. The resultant value, `s`, is 1.05946 (this is the ratio of the frequency of two adjacent notes on a piano).

# 3.8 Taking the Square Root

## Problem

You want to calculate the square root of a number.

## Solution

The `sqrt(x)` function returns the square root of `x`:

```
Serial.println( sqrt(9) ); // this prints 3.00
```

## Discussion

The `sqrt` function returns a floating-point number (see the `pow` function discussed in Recipe 3.7).

# 3.9 Rounding Floating-Point Numbers Up and Down

## Problem

You want the next smallest or largest integer value of a floating-point number (`floor` or `ceil`).

## Solution

`floor(x)` returns the largest integral value that is not greater than `x`. `ceil(x)` returns the smallest integer that is not less than `x`.

## Discussion

These functions are used for rounding floating-point numbers; use `floor(x)` to get the largest integer that is not greater than `x`. Use `ceil` to get the smallest integer that is greater than `x`.

Here is some example output using `floor`:

```
Serial.println( floor(1) );    // this prints  1.00
  Serial.println( floor(1.1) );  // this prints  1.00
  Serial.println( floor(0) );    // this prints  0.00
  Serial.println( floor(.1) );   // this prints  0.00
  Serial.println( floor(-1) );   // this prints -1.00
  Serial.println( floor(-1.1) ); // this prints -2.00
```

Here is some example output using `ceil`:

```
Serial.println( ceil(1) );     // this prints  1.00
  Serial.println( ceil(1.1) );   // this prints  2.00
  Serial.println( ceil(0) );     // this prints  0.00
  Serial.println( ceil(.1) );    // this prints  1.00
  Serial.println( ceil(-1) );    // this prints -1.00
  Serial.println( ceil(-1.1) );  // this prints -1.00
```

You can round to the nearest integer as follows:

```
int result = round(1.1);
```

> **NOTE**
>
> You can truncate a floating-point number by *casting* (converting) to an `int`, but this does not round correctly. Negative numbers such as –1.9 should round down to –2, but when cast to an `int` they are rounded up to –1. The same problem exists with positive numbers: 1.9 should round up to 2 but will round down to 1. Use `floor`, `ceil`, and `round` to get the correct results.

# 3.10 Using Trigonometric Functions

## Problem

You want to get the sine, cosine, or tangent of an angle given in radians or degrees.

## Solution

`sin(x)` returns the sine of angle `x`. `cos(x)` returns the cosine of angle `x`. `tan(x)` returns the tangent of angle `x`.

## Discussion

Angles are specified in radians and the result is a floating-point number (see Recipe 2.3). The following example illustrates the trig functions:

```
float deg = 30;                  // angle in degrees
  float rad  = deg * PI / 180;    // convert to radians
  Serial.println(rad);            // print the radians
  Serial.println(sin(rad), 5);    // print the sine
  Serial.println(cos(rad), 5);    // print the cosine
```

This converts the angle into radians and prints the sine and cosine. Here is the output with annotation added:

```
0.52     30 degrees is 0.5235988 radians, println only shows two decimal places
0.50000  sine of 30 degrees is .5000000, displayed here to 5 decimal places
0.86603  cosine is .8660254, which rounds up to 0.86603 at 5 decimal places
```

Although the sketch calculates these values using the full precision of floating-point numbers, the `Serial.print` and `Serial.println` routines show the values of floating-point numbers to two decimal places by default, but you can specify a precision as the second argument (5 in the case of sine and cosine in this example) as discussed in Recipe 2.3.

The conversion from radians to degrees and back again is textbook trigonometry. `PI` is the familiar constant for π (3.14159265...). `PI` and `180` are both constants, and Arduino provides some precalculated constants you can use to perform degree/radian conversions:

```
rad = deg * DEG_TO_RAD;  // a way to convert degrees to radians
deg = rad * RAD_TO_DEG;  // a way to convert radians to degrees
```

Using `deg * DEG_TO_RAD` looks more efficient than `deg * PI / 180`, but it's not, since the Arduino compiler is smart enough to recognize that `PI / 180` is a constant (the value will never change), so it substitutes the result of dividing `PI` by 180, which happens to be the same value as the constant `DEG_TO_RAD` (0.017453292519...). Use whichever approach you prefer.

### See Also

The Arduino references for `sin`, `cos`, and `tan`

# 3.11 Generating Random Numbers

### Problem

You want to get a random number, either ranging from zero up to a specified maximum or constrained between a minimum and maximum value you provide.

### Solution

Use the `random` function to return a random number. Calling `random` with a single parameter sets the upper bound; the values returned will range from

zero to one less than the upper bound:

```
int minr = 50;
int maxr = 100;
long randnum = random(maxr);        // random number between 0 and maxr -1
```

Calling `random` with two parameters sets the lower and upper bounds; the values returned will range from the lower bound (inclusive) to one less than the upper bound:

```
long randnum = random(minr, maxr); // random number between minr and maxr -1
```

## Discussion

Although there appears to be no obvious pattern to the numbers returned, the values are not truly random. Exactly the same sequence will repeat each time the sketch starts. In many applications, this does not matter. But if you need a different sequence each time your sketch starts, use the function `randomSeed(seed)` with a different seed value each time (if you use the same seed value, you'll get the same sequence). This function starts the random number generator at some arbitrary place based on the seed parameter you pass:

```
randomSeed(1234);  // change the starting sequence of random numbers
```

Here is an example that uses the different forms of random number generation available on Arduino:

```
// Random
// demonstrates generating random numbers

int randNumber;

void setup()
{
  Serial.begin(9600);
  while(!Serial);

  // Print random numbers with no seed value
```

```
  Serial.println("Print 20 random numbers between 0 and 9");
  for(int i=0; i < 20; i++)
  {
    randNumber = random(10);
    Serial.print(randNumber);
    Serial.print(" ");
  }
  Serial.println();
  Serial.println("Print 20 random numbers between 2 and 9");
  for(int i=0; i < 20; i++)
  {
    randNumber = random(2,10);
    Serial.print(randNumber);
    Serial.print(" ");
  }

  // Print random numbers with the same seed value each time
  randomSeed(1234);
  Serial.println();
  Serial.println("Print 20 random numbers between 0 and 9 after constant seed
");
  for(int i=0; i < 20; i++)
  {
    randNumber = random(10);
    Serial.print(randNumber);
    Serial.print(" ");
  }

  // Print random numbers with a different seed value each time
  randomSeed(analogRead(0));  // read from an analog port with nothing
connected
  Serial.println();
  Serial.println("Print 20 random numbers between 0 and 9 after floating seed
");
  for(int i=0; i < 20; i++)
  {
    randNumber = random(10);
    Serial.print(randNumber);
    Serial.print(" ");
  }
  Serial.println();
  Serial.println();
}

void loop()
{
}
```

Here is the output from this code as run on an Uno (you may get different results on different architectures):

```
Print 20 random numbers between 0 and 9
7 9 3 8 0 2 4 8 3 9 0 5 2 2 7 3 7 9 0 2
Print 20 random numbers between 2 and 9
9 3 7 7 2 7 5 8 2 9 3 4 2 5 4 3 5 7 5 7
Print 20 random numbers between 0 and 9 after constant seed
8 2 8 7 1 8 0 3 6 5 9 0 3 4 3 1 2 3 9 4
Print 20 random numbers between 0 and 9 after floating seed
0 9 7 4 4 7 7 4 4 9 1 6 0 2 3 1 5 9 1 1
```

If you press the reset button on your Arduino to restart the sketch, the first three lines of random numbers will be unchanged. (You may need to close and reopen the Serial Monitor after you press the reset button.) Only the last line changes each time the sketch starts, because it sets the seed to a different value by reading it from an unconnected analog input port as a seed to the `randomSeed` function. If you are using analog port 0 for something else, change the argument for `analogRead` to an unused analog port.

In general, the preceding example is the start and end of the options you have available for random number generation on an Arduino without external hardware. It may seem that an unconnected analog input port is a good, or at least acceptable, way to seed your random number generator. However, the analog-to-digital converter on most Arduino boards will return at most a 10-bit value, which can only hold 1,024 different values. This is far too small a range of values to seed a random number generator for strong random numbers. Additionally, a floating analog pin is not quite as random as you might think it would be. It is likely to exhibit somewhat consistent patterns, and can certainly be influenced by anyone who can get within proximity of your Arduino.

It would be difficult to generate truly random numbers on an Arduino, but like most computers, you can generate cryptographically strong pseudorandom numbers, or random numbers that are "random enough" to be suitable for use in cryptographic applications. Some Arduino boards, such as the Arduino WiFi Rev2, MKR Vidor 4000, and MKR WiFi

1000/1010 include the Atmel ECC508 or ECC608 crypto chip that has hardware support for cryptographic functions, including a strong random number generator. You can access it by installing the ArduinoECCX08 library using Arduino's Library Manager (see Recipe 16.2 for instructions on installing libraries). For strong random number generation on any Arduino, check out Rhys Weatherley's Crypto library, in particular the RNG class.

## See Also

Arduino references for `random` and `randomSeed`

# 3.12 Setting and Reading Bits

## Problem

You want to read or set a particular bit in a numeric variable.

## Solution

Use the following functions:

`bitSet(x, bitPosition)`

Sets (writes a 1 to) the given `bitPosition` of variable `x`

`bitClear(x, bitPosition)`

Clears (writes a 0 to) the given `bitPosition` of variable `x`

`bitRead(x, bitPosition)`

Returns the value (as 0 or 1) of the bit at the given `bitPosition` of variable `x`

`bitWrite(x, bitPosition, value)`

Sets the given value (as 0 or 1) of the bit at the given `bitPosition` of variable `x`

```
bit(bitPosition)
```

Returns the value of the given bit position: `bit(0)` is 1, `bit(1)` is 2, `bit(2)` is 4, and so on

In all these functions, `bitPosition 0` is the least significant (rightmost) bit.

Here is a sketch that uses these functions to manipulate the bits of an 8-bit variable called `flags`. It uses each of the eight bits as an independent flag that can be toggled on and off:

```cpp
// bitFunctions
// demonstrates using the bit functions

byte flags = 0;

// these examples set, clear, or read bits in a variable called flags

// bitSet example
void setFlag(int flagNumber)
{
   bitSet(flags, flagNumber);
}

// bitClear example
void clearFlag(int flagNumber)
{
   bitClear(flags, flagNumber);
}

// bitPosition example

int getFlag(int flagNumber)
{
   return bitRead(flags, flagNumber);
}

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

void loop()
{
```

```
    flags = 0;    // clear all flags
    showFlags();
    setFlag(2);   // set some flags
    setFlag(5);
    showFlags();
    clearFlag(2);
    showFlags();

    delay(10000); // wait a very long time
}

// reports flags that are set
void showFlags()
{
    for(int flag=0; flag < 8; flag++)
    {
      if (getFlag(flag) == true)
          Serial.print("* bit set for flag ");      else
          Serial.print("bit clear for flag ");

      Serial.println(flag);
    }
    Serial.println();
}
```

This code will print the following every 10 seconds:

```
bit clear for flag 0
bit clear for flag 1
bit clear for flag 2
bit clear for flag 3
bit clear for flag 4
bit clear for flag 5
bit clear for flag 6
bit clear for flag 7

bit clear for flag 0
bit clear for flag 1
* bit set for flag 2
bit clear for flag 3
bit clear for flag 4
* bit set for flag 5
bit clear for flag 6
bit clear for flag 7

bit clear for flag 0
```

```
bit clear for flag 1
bit clear for flag 2
bit clear for flag 3
bit clear for flag 4
* bit set for flag 5
bit clear for flag 6
bit clear for flag 7
```

## Discussion

Reading and setting bits is a common task, and many of the Arduino libraries use this functionality. One of the more common uses of bit operations is to efficiently store and retrieve binary values (on/off, true/false, 1/0, high/low, etc.).

> **TIP**
>
> Arduino defines the constants `true` and `HIGH` as 1 and `false` and `LOW` as 0.

The state of eight switches can be packed into a single 8-bit value instead of requiring eight bytes or integers. The example in this recipe's Solution shows how eight values can be individually set or cleared in a single byte.

The term *flag* is a programming term for values that store the state of some aspect of a program. In this sketch, the flag bits are read using `bitRead`, and they are set or cleared using `bitSet` or `bitClear`. These functions take two parameters: the first is the value to read or write (`flags` in this example), and the second is the bit position indicating where the read or write should take place. Bit position 0 is the least significant (rightmost) bit; position 1 is the second position from the right, and so on. So:

```
bitRead(2, 1); // returns 1 : 2 is binary 10 and bit in position 1 is 1
bitRead(4, 1); // returns 0 : 4 is binary 100 and bit in position 1 is 0
```

There is also a function called `bit` that returns the value of each bit position:

```
bit(0)  is equal to 1;
bit(1)  is equal to 2;
bit(2)  is equal to 4;
...
bit(7)  is equal to 128
```

## See Also

The Arduino references for bit and byte functions:

- `lowByte`
- `highByte`
- `bitRead`
- `bitWrite`
- `bitSet`
- `bitClear`
- `bit`

# 3.13 Shifting Bits

## Problem

You need to perform bit operations that shift bits left or right in a `byte`, `int`, or `long`.

## Solution

Use the **<<** (bit-shift left) and **>>** (bit-shift right) operators to shift the bits of a value.

## Discussion

This fragment sets variable x equal to 6. It shifts the bits left by one and prints the new value (12). Then that value is shifted right two places (and in this example becomes equal to 3):

```
int x = 6;
x = x << 1;  // 6  shifted left once is 12
Serial.println(x);
x = x >> 2;   // 12 shifted right twice is 3
Serial.println(x);
```

Here is how this works: 6 shifted left one place equals 12, because the decimal number 6 is 0110 in binary. When the digits are shifted left, the value becomes 1100 (decimal 12). Shifting 1100 right two places becomes 0011 (decimal 3). You may notice that shifting a number left by *n* places is the same as multiplying the value by 2 raised to the power of *n*. Shifting a number right by *n* places is the same as dividing the value by 2 raised to the power of *n*. In other words, the following pairs of expressions are the same:

- x << 1 is the same as x * 2.
- x << 2 is the same as x * 4.
- x << 3 is the same as x * 8.
- x >> 1 is the same as x / 2.
- x >> 2 is the same as x / 4.
- x >> 3 is the same as x / 8.

The Arduino controller chip can shift bits more efficiently than it can multiply and divide, and you may come across code that uses the bit shift to multiply and divide:

```
int c = (a << 1) + (b >> 2); //add (a times 2) plus ( b divided by 4)
```

The expression (a << 1) + (b >> 2); does not look much like (a * 2) + (b / 4);, but both expressions do the same thing. Indeed, the Arduino compiler is smart enough to recognize that multiplying an integer by a constant that is a power of two is identical to a shift and will produce the same machine code as the version using shift. The source code using arithmetic operators is easier for humans to read, so it is preferred when the intent is to multiply and divide.

## See Also

Arduino references for bit and byte functions: `lowByte`, `highByte`, `bitRead`, `bitWrite`, `bitSet`, `bitClear`, and `bit` (see Recipe 3.12)

# 3.14 Extracting High and Low Bytes in an int or long

### Problem

You want to extract the high byte or low byte of an integer; for example, when sending integer values as bytes on a serial or other communication line.

### Solution

Use `lowByte(i)` to get the least significant byte from an integer. Use `highByte(i)` to get the most significant byte from an integer.

The following sketch converts an integer value into low and high bytes:

```
/*
 * ByteOperators sketch
 */

int intValue = 258; // 258 in hexadecimal notation is 0x102

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

void loop()
{
  int loWord,hiWord;
  byte loByte, hiByte;

  hiByte = highByte(intValue);
  loByte = lowByte(intValue);

  Serial.println(intValue, DEC);
  Serial.println(intValue, HEX);
```

```
    Serial.println(loByte, DEC);
    Serial.println(hiByte, DEC);

    delay(10000); // wait a very long time
}
```

## Discussion

The example sketch prints `intValue` followed by the low byte and high byte (with annotations added):

```
258   the integer value to be converted
102   the value in hexadecimal notation
2     the low byte
1     the high byte
```

To extract the byte values from a `long`, the 32-bit long value first gets broken into two 16-bit *words* that can then be converted into bytes as shown in the earlier code. At the time of this writing, the standard Arduino library did not have a function to perform this operation on a `long`, but you can add the following lines to your sketch to provide this:

```
#define highWord(w) ((w) >> 16)
#define lowWord(w) ((w) & 0xffff)
```

These are *macro expressions*: `highWord` performs a 16-bit shift operation to produce a 16-bit value, and `lowWord` masks the lower 16 bits using the bitwise And operator (see Recipe 2.20).

---

**TIP**

The number of bits in an `int` varies on different platforms. On Arduino it is 16 bits, but in other environments it is 32 bits. The term *word* as used here refers to a 16-bit value.

---

This code converts the 32-bit value 16909060 (hexadecimal 0x1020304) to its 16-bit constituent high and low values:

```
long longValue = 16909060;
  int loWord = lowWord(longValue);
  int hiWord = highWord(longValue);
  Serial.println(loWord, DEC);
  Serial.println(hiWord, DEC);
```

This prints the following values:

```
772   772 is 0x0304 in hexadecimal
258   258 is 0x0102 in hexadecimal
```

Note that 772 in decimal is 0x0304 in hexadecimal, which is the low-order word (16 bits) of the `longValue 0x10020304`. You may recognize 258 from the first part of this recipe as the value produced by combining a high byte of 1 and a low byte of 2 (0x0102 in hexadecimal).

## See Also

Arduino references for bit and byte functions: `lowByte`, `highByte`, `bitRead`, `bitWrite`, `bitSet`, `bitClear`, and `bit` (see Recipe 3.12)

# 3.15 Forming an int or long from High and Low Bytes

### Problem

You want to create a 16-bit (`int`) or 32-bit (`long`) integer value from individual bytes; for example, when receiving integers as individual bytes over a serial communication link. This is the inverse operation of Recipe 3.14.

### Solution

Use the `word(h,l)` function to convert two bytes into a single Arduino integer. Here is the code from Recipe 3.14 expanded to convert the individual high and low bytes back into an integer:

```
/*
 * Forming an int or long with byte operations sketch
 */

int intValue = 0x102;   // 258

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

void loop()
{
  int aWord;
  byte loByte, hiByte;

  hiByte = highByte(intValue);
  loByte = lowByte(intValue);

  Serial.println(intValue, DEC);
  Serial.println(loByte, DEC);
  Serial.println(hiByte, DEC);

  aWord = word(hiByte, loByte); // convert the bytes back into a word
  Serial.println(aWord, DEC);
  delay(10000); // wait a very long time
}
```

## Discussion

The word(high,low) expression assembles a high and low byte into a 16-bit value. The code in this recipe's Solution takes the low and high bytes formed as shown in Recipe 3.14 and assembles them back into a word. The output is the integer value, the low byte, the high byte, and the bytes converted back to an integer value:

```
258
2
1
258
```

Arduino does not have a function to convert a 32-bit long value into two 16-bit words (at the time of this writing), but you can add your own

`makeLong()` capability by adding the following line to the top of your sketch:

```
#define makeLong(hi, low)  ((hi) << 16 & (low))
```

This defines a command that will shift the high value 16 bits to the left and add it to the low value:

```
#define makeLong(hi, low)  (((long) hi) << 16 | (low))
#define highWord(w) ((w) >> 16)
#define lowWord(w) ((w) & 0xffff)

// declare a value to test
long longValue = 0x1020304;  // in decimal: 16909060
                             // in binary : 00000001 00000010 00000011
00000100

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

void loop()
{
  int loWord,hiWord;

  Serial.println(longValue,DEC);  // this prints  16909060
  loWord = lowWord(longValue);     // convert long to two words
  hiWord = highWord(longValue);
  Serial.println(loWord,DEC);      // print the value 772
  Serial.println(hiWord,DEC);      // print the value 258
  longValue = makeLong(hiWord, loWord);  // convert the words back to a long
  Serial.println(longValue,DEC);  // this again prints  16909060

  delay(10000); // wait a very long time
}
```

The output is:

```
16909060
772
258
16909060
```

> **NOTE**
>
> The term *word* refers to 16 bits when compiling for 8-bit boards such as the Uno; when compiling for 32-bit boards a word is 32 bits and a half word is 16 bits. Although the underlying architecture is different, the preceding code will return high and low 16-bit values on both 8-bit and 32-bit boards.

## See Also

The Arduino references for bit and byte functions: `lowByte`, `highByte`, `bitRead`, `bitWrite`, `bitSet`, `bitClear`, and `bit` (see Recipe 3.12)

# Serial Communications
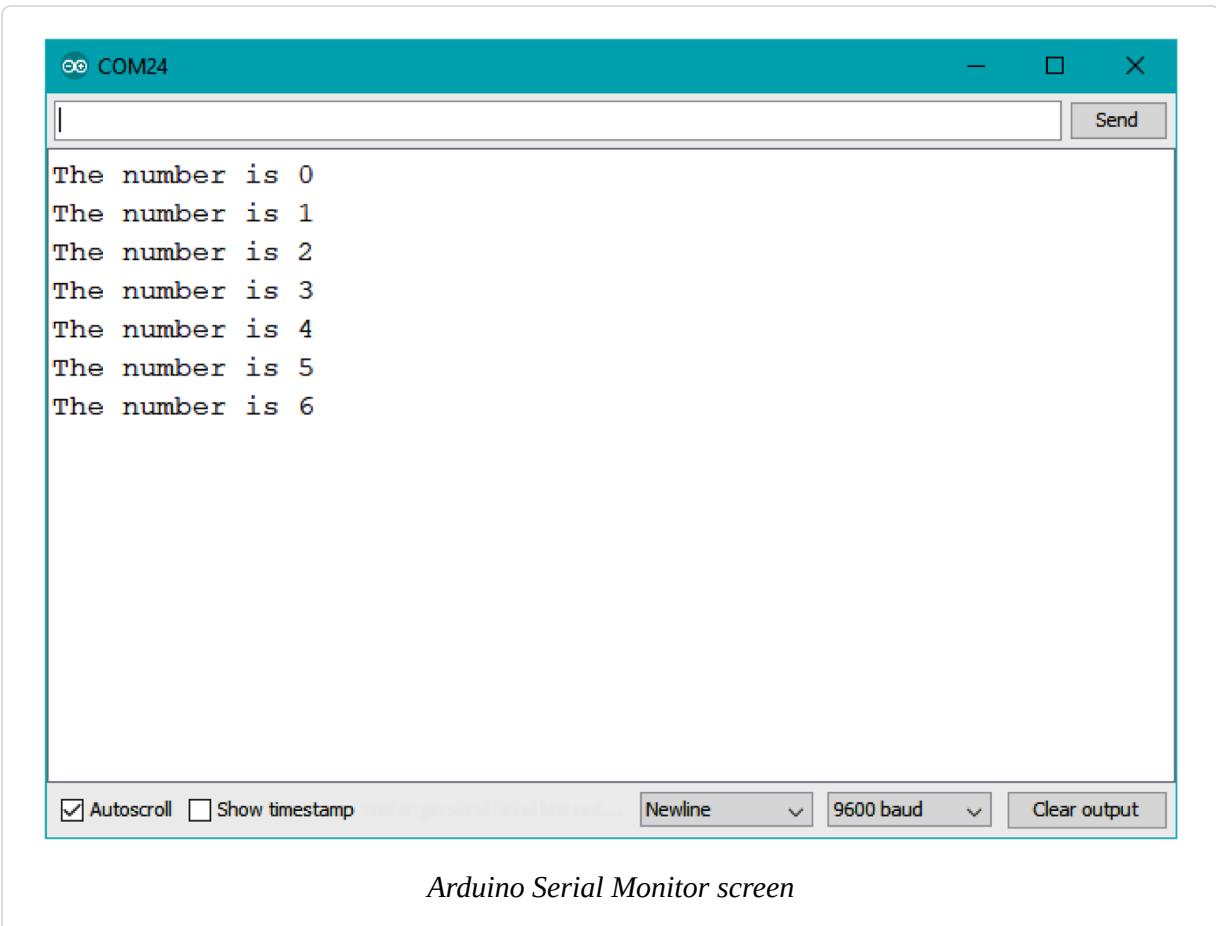
## 4.0 Introduction

Serial communications provide an easy and flexible way for your Arduino board to interact with your computer and other devices. This chapter explains how to send and receive information using this capability.

Chapter 1 described how to connect the Arduino USB serial port to your computer to upload sketches. The upload process sends data from your computer to Arduino, and Arduino sends status messages back to the computer to confirm the transfer is working. The recipes here show how you can use that same communication link to send and receive any information between Arduino and your computer or another serial device.

Serial communications are also a handy tool for debugging. You can send debug messages from Arduino to the computer and display them on your computer screen or send them to another device such as a Raspberry Pi or another Arduino. You can also use an external LCD display to show these messages, but in all likelihood, you'd use I2C or SPI to communicate with that kind of display (see Chapter 13).

The Arduino IDE (described in Recipe 1.3) provides a Serial Monitor (shown in Figure 4-1) to display serial data sent from Arduino. You can also send data from the Serial Monitor to Arduino by entering text in the text box to the left of the Send button. Arduino also includes a Serial Plotter that can graph serial data sent from Arduino (see Recipe 4.1).

*Arduino Serial Monitor screen*

You can set the speed at which data is transmitted (the baud rate, measured in bits per second) using the drop-down box on the bottom right. Make sure to set it to whatever value you use with `Serial.begin()`. The default rate of 9,600 bits per second is fine for many cases, but if you are working with a device that needs a higher speed, you can pass a number higher than 9,600 to `Serial.begin()`.

You can use the drop-down to the left of the baud rate to automatically send a newline (ASCII character 10), carriage return (ASCII character 13), a combination of newline and carriage return ("Both NL & CR"), or no terminator ("No line ending") at the end of each message.

Your Arduino sketch can use the serial port to indirectly access (usually via a proxy program written in a language like Processing or Python) all the resources (memory, screen, keyboard, mouse, network connectivity, etc.) that your computer has. Your computer can also use the serial link to

interact with certain sensors or other devices connected to Arduino. If you want to talk to multiple devices using serial communications, either you need more than one serial port or you'll need to use software serial to emulate a serial port using Arduino pins (see "Emulate Serial Hardware with Digital Pins").

> **NOTE**
>
> Many sensors and output devices that support serial communications also support SPI or I2C (see Chapter 13). While serial communications are well understood and somewhat universal, consider using SPI or I2C if either or both are supported by the sensor or output device you want to connect. Both protocols offer more flexibility when communicating with multiple devices.

Implementing serial communications involves hardware and software. The hardware provides the electrical signaling between Arduino and the device it is talking to. The software uses the hardware to send bytes or bits that the connected hardware understands. The Arduino serial libraries insulate you from most of the hardware complexity, but it is helpful for you to understand the basics, especially if you need to troubleshoot any difficulties with serial communications in your projects.

## Serial Hardware

Serial hardware sends and receives data as electrical pulses that represent sequential bits. The zeros and ones that carry the information that makes up a byte can be represented in various ways. The scheme used by Arduino is 0 volts to represent a bit value of 0, and 5 volts (or 3.3 volts) to represent a bit value of 1.

Some boards, such as the Modern Device Bare Bones Board and the (now-discontinued) Adafruit Boarduino and Arduino Pro, Mini, and Pro Mini, do not have USB support and require an adapter for connecting to your computer that converts TTL to USB. The Adafruit CP2104 Friend (Adafruit part number 3309), Modern Device USB BUB board (Modern Device part MD022X), and FTDI USB TTL Adapter all work well.

Some serial devices use the RS-232 standard for serial connection. These usually have a nine-pin connector, and an adapter is required to use them with the Arduino. RS-232 uses voltage levels that will damage Arduino digital pins, so you will need to obtain an RS-232 to TTL adapter to use it. Arduino has an Arduino RS-232 tutorial, and lots of information and links are available at the Serial Port Central website.

An Arduino Uno has a single hardware serial port, but serial communication is also possible using software libraries to emulate additional ports (communication channels) to provide connectivity to more than one device. Software serial requires a lot of help from the Arduino controller to send and receive data, so it's not as fast or efficient as hardware serial.

The Leonardo and many 32-bit boards (such as the Arduino Zero, Adafruit Metro M0, and SparkFun RedBoard Turbo) have a second hardware serial port in addition to USB serial. The Teensy 3 board from PJRC has three serial ports in addition to USB serial. The Teensy 4.0 board has seven serial ports (in addition to USB serial).

The Arduino Mega has four hardware serial ports that can communicate with up to four different serial devices. Only one of these has a USB adapter built in (you could wire a USB-TTL adapter to any of the other serial ports if you want more than one USB connection).

Table 4-1 shows the pins used for serial ports on various Arduino and Arduino-compatible boards. The pin numbers shown are for digital, rather than analog, pins.

*Serial (digital) pins for selected boards*

| Board | Serial RX/TX | Serial1 RX/TX | Serial2 RX/TX | Serial3 RX/TX |
|---|---|---|---|---|
| Arduino MKR 1010 | USB only | 13/14 | none | none |
| Arduino Uno WiFi Rev2 | USB only | 0/1 | Connected to WiFi module | none |
| Arduino Nano Every | USB only | 0/1 | none | none |
| Arduino Nano 33 BLE Sense | USB only | 0/1 | none | none |
| Arduino Uno Rev3 | 0/1 (Also USB) | none | none | none |
| Adafruit Metro Express (M0) | USB only | 0/1 | none | none |
| Arduino Zero/SparkFun RedBoard Turbo | USB only[a] | 0/1 | none | none |
| Adafruit Itsy Bitsy M4 Express | USB only | 0/1 | none | none |
| PJRC Teensy 3.2 | USB only | 0/1 | 9/10 | 7/8 |
| PJRC Teensy 4.0 | USB only | 0/1 | 7/8 | 15/14 |
| Arduino Due | 0/1 (Also USB) | 19/18 | 17/16 | 15/14 |
| Arduino Mega 2560 Rev2 | 0/1 (Also USB) | 19/18 | 17/16 | 15/14 |
| Arduino Leonardo | USB only | 0/1 | none | none |

| Board | Serial RX/TX | Serial1 RX/TX | Serial2 RX/TX | Serial3 RX/TX |
| --- | --- | --- | --- | --- |

> **NOTE**
>
> Some Teensy boards support more than three hardware serial ports, and some allow you to modify which pins are used for serial communications. See PJRC for more details.

## Serial Hardware Behavior

The number of serial ports isn't the only variable between boards. There are some fundamental differences in behavior, as well. Most boards based on the AVR ATmega chips, including the Uno, original Nano, and Mega, have a chip to convert the hardware serial port on the Arduino chip to USB for connection to the hardware serial port. On these boards, when you open a connection to the serial port (such as by opening the Serial Monitor or accessing the serial port from a program running on a computer connected to the board via USB), the board will automatically reset, causing the sketch to start from the beginning.

On some 8-bit boards (Leonardo and compatibles) and most 32-bit boards, USB serial is provided by the same processor that you run your sketches on. Because of how they are designed, opening the USB serial port does not reset these boards. As a result, your sketch will begin sending data to the USB serial port faster than you can open the serial port. This means that if you have any serial output commands (`Serial.print` or `Serial.println`) in your `setup()` function, you won't see it in the Serial Monitor because you can't open the Serial Monitor quickly enough. (You could put a `delay` in your `setup` function, but there is another way.)

Additionally, the Leonardo and compatibles have another behavior that makes working with the serial port tricky: when you first power it up, it will flash an LED for several seconds to tell you that it's in a special mode

where it allows you to load a sketch over USB. So you will not be able to open the serial port to send or receive data until it's done waiting.

On boards that do not reset automatically when you open the serial port, you can add the following code to your `setup` function (right after the call to `Serial.begin()`). This will pause execution until the serial port has been opened so you can see serial output that you send in `setup`:

```
while(!Serial) {
  ; // wait for serial port to connect
}
```

You can skip the curly brackets and consolidate it down to `while(!Serial);` but this may be confusing to novice programmers who read your code.

Because the `while(!Serial);` command will pause execution of the sketch until you open the serial port, this approach should not be used in environments where your Arduino-based solution is expected to run independently; for example when running on batteries without a USB connection. Table 4-2 shows USB serial behavior for various boards.

*USB serial behavior for various boards*

| Board | while(!Serial); needed? | Resets when serial accessed? |
| --- | --- | --- |
| Arduino MKR 1010 | Yes | No |
| Arduino Uno WiFi Rev2 | No | Yes |
| Arduino Nano Every | No; Requires a `delay(800);` after `Serial.begin()` and you must open the Serial Monitor before uploading in order to see all serial output. | No |
| Arduino Nano 33 BLE Sense | Yes | No |

| Board | while(!Serial); needed? | Resets when serial accessed? |
| --- | --- | --- |
| Arduino Uno Rev3 | No | Yes |
| Adafruit Metro Express (M0) | Yes | No |
| Adafruit Itsy Bitsy M4 Express | Yes | No |
| PJRC Teensy 3.2 | Yes | No |
| PJRC Teensy 4.0 | Yes | No |
| Arduino Mega 2560 Rev3 | No | Yes |
| Arduino Leonardo | Yes | No |

## Emulate Serial Hardware with Digital Pins

You will usually use the built-in Arduino Serial library to communicate with the hardware serial ports. Serial libraries simplify the use of the serial ports by insulating you from hardware complexities.

Sometimes you need more serial ports than the number of hardware serial ports available. If this is the case, you can use an additional *software serial* library that uses software to emulate serial hardware. Recipes 4.11 and 4.12 show how to use a software serial library to communicate with multiple devices.

## Message Protocols

The hardware or software serial libraries handle sending and receiving information. This information often consists of groups of variables that need to be sent together. For the information to be interpreted correctly, the receiving side needs to recognize where each message begins and ends. Meaningful serial communication, or any kind of machine-to-machine communication, can only be achieved if the sending and receiving sides fully agree on how information is organized in the message. The formal organization of information in a message and the range of appropriate responses to requests is called a *communications protocol*. You can establish a protocol over any underlying data transfer system, such as serial communications, but these same principles apply to other means of data transfer, such as Ethernet or WiFi networking.

Messages can contain one or more special characters that identify the start of the message—this is called the *header*. One or more characters can also be used to identify the end of a message—this is called the *footer*. The recipes in this chapter show examples of messages in which the values that make up the body of a message can be sent in either text or binary format.

Sending and receiving messages in text format involves sending commands and numeric values as human-readable letters and words. Numbers are sent as the string of digits that represent the value. For example, if the value is 1234, the characters 1, 2, 3, and 4 are sent as individual characters.

Binary messages comprise the bytes that the computer uses to represent values. Binary data is usually more efficient (requiring fewer bytes to be sent), but the data is not as human-readable as text, which makes it more difficult to debug. For example, Arduino represents 1234 as the bytes 4 and 210 (4 * 256 + 210 = 1234). If you were to look at these characters in the Serial Monitor, they wouldn't be readable because the ASCII character 4 is a control character and the ASCII character 210 is in the extended range of ASCII characters, so it will probably display an accented character or something else depending on your configuration. If the device you are connecting to sends or receives only binary data, that is what you will have to use, but if you have the choice, textual messages are easier to implement and debug.

There are many ways to approach software problems, and some of the recipes in this chapter show two or three different ways to achieve a similar result. The differences (e.g., sending text instead of raw binary data) may offer a balance between simplicity and efficiency. Where choices are offered, pick the solution that you find easiest to understand and adapt—this will probably be the first solution covered. Alternatives may be a little more efficient, or they may be more appropriate for a specific protocol that you want to connect to, but the "right way" is the one you find easiest to get working in your project.

> ### THE PROCESSING DEVELOPMENT ENVIRONMENT
>
> Some of the examples in this chapter use the Processing language to send and receive serial messages on a computer talking to Arduino.
>
> Processing is a free, open source tool that uses a similar development environment to Arduino, but instead of running your sketches on a microcontroller, your Processing sketches run on your computer. You can read more about Processing and download everything you need at the Processing website.
>
> Processing is based on the Java language, but the Processing code samples in this book should be easy to translate into other environments that support serial communications. Processing comes with some example sketches illustrating communication between Arduino and Processing. SimpleRead is a Processing example that includes Arduino code. In Processing, select File → Examples → Libraries → Serial → SimpleRead to see an example that reads data from the serial port and changes the color of a rectangle when a switch connected to Arduino is pressed and released.

## Arduino Serial Notes

Here are a few things you should be aware of when working with serial data in Arduino:

- `Serial.flush` waits for all outgoing data to be sent rather than discarding received data (which was the behavior in older versions of Arduino). You can use the following statement to discard all data in the receive buffer: `while(Serial.read() >= 0) ; // flush the receive buffer`.

- `Serial.write` and `Serial.print` do not block. Older versions of Arduino would wait until all characters were sent before returning. Instead, characters that you send using `Serial.write` or `Serial.print` (and `println`) are transmitted in the background (from an interrupt handler), allowing your sketch code to immediately resume processing. This is usually a good thing (it can make the sketch more responsive), but sometimes you want to wait until all characters are sent. You can achieve this by calling `Serial.flush()` immediately following `Serial.write()` or `Serial.print()/println()`.

- Serial print functions return the number of characters printed. This is useful when text output needs to be aligned or for applications that send data that includes the total number of characters sent.

- There is a built-in parsing capability for streams such as serial to easily extract numbers and find text. See the Discussion section of Recipe 4.5 for more on using this capability with serial.

- The SoftwareSerial library bundled with Arduino can be very helpful; see Recipes 4.11 and 4.12.

- The `Serial.peek` function lets you "peek" at the next character in the receive buffer. Unlike `Serial.read`, the character is not removed from the buffer with `Serial.peek`.

# 4.1 Sending Information from Arduino to Your Computer

**Problem**

You want to send text and data to be displayed on your PC, Mac, or other device (such as a Raspberry Pi) using the Arduino IDE or the serial terminal program of your choice.

**Solution**

This sketch prints sequential numbers on the Serial Monitor:

```
/*
 * SerialOutput sketch
 * Print numbers to the serial port
 */
void setup()
{
  Serial.begin(9600); // send and receive at 9600 baud
}

int number = 0;

void loop()
{
  Serial.print("The number is ");
  Serial.println(number);     // print the number

  delay(500); // delay half second between numbers
  number++; // to the next number
}
```

Connect Arduino to your computer just as you did in Chapter 1 and upload this sketch. Click the Serial Monitor icon in the IDE and you should see the output displayed as follows:
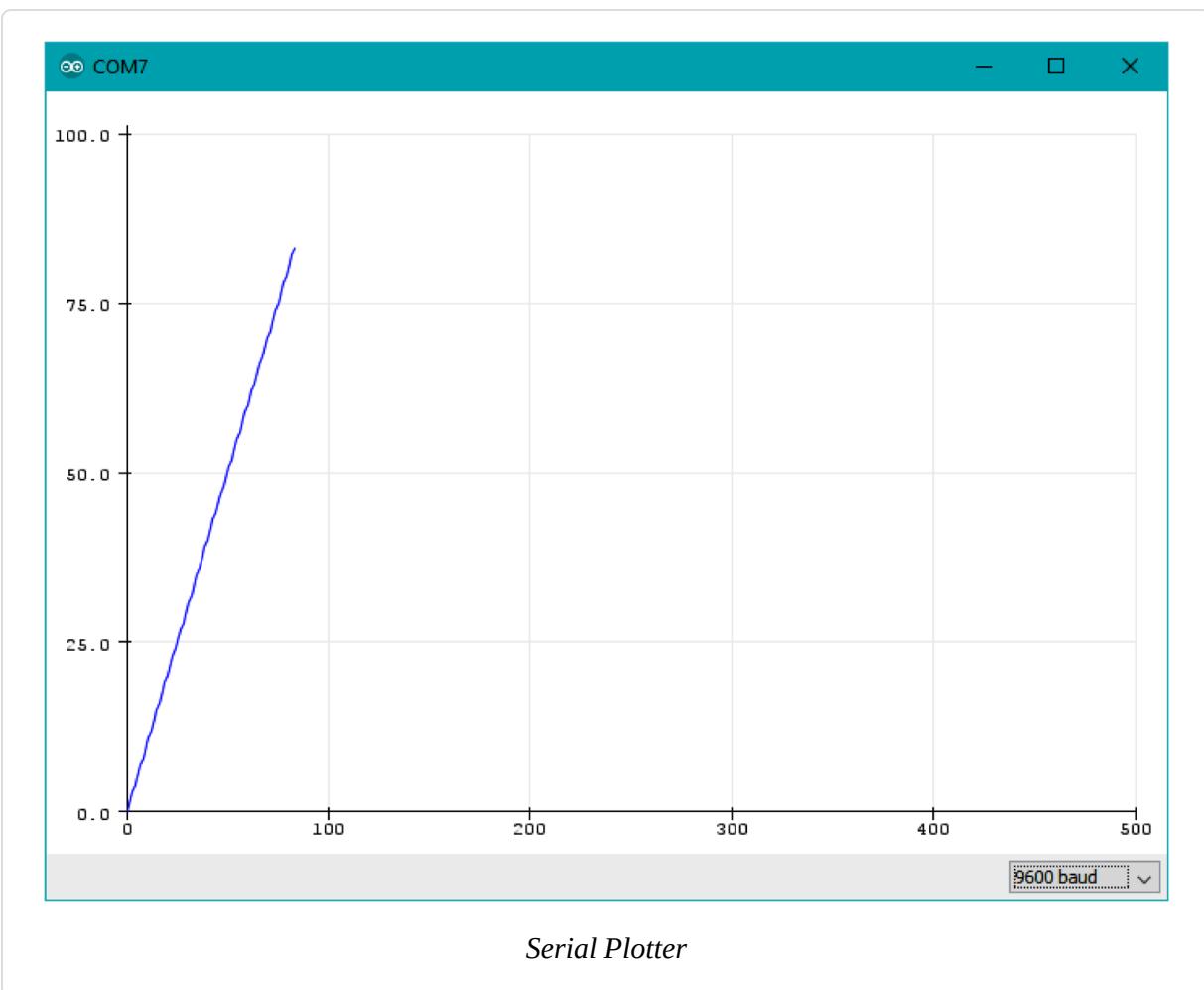
```
The number is 0
The number is 1
The number is 2
```

## Discussion

To display text and numbers from your sketch on a computer via a serial link, put the `Serial.begin(9600)` statement in `setup()`, and then use `Serial.print()` statements to print the text and values you want to see. You can then view the output in the Serial Monitor as shown in Figure 4-2.

*Arduino Serial Monitor screen*

To get a graphical display of the number being sent back, close the Serial Monitor window and Select Tools→Serial Plotter. A window will open and draw a graph of the values as they are received from the board. The plotter can isolate the numbers from the text, and identify multiple numbers separated by alpha characters and plot them separately using different color traces. Figure 4-3 shows the Serial Plotter.

*Serial Plotter*

Your sketch must call the `Serial.begin()` function before it can use serial input or output. The function takes a single parameter: the desired communication speed. You must use the same speed for the sending side and the receiving side, or you will see gobbledygook (or nothing at all) on the screen. This example and most of the others in this book use a speed of 9,600 baud (*baud* is a measure of the number of bits transmitted per second). The 9,600 baud rate is approximately 1,000 characters per second. You can send at lower or higher rates (the range is 300 to 115,200 or higher depending on your board's capabilities), but make sure both sides use the same speed. The Serial Monitor sets the speed using the baud rate drop-down (at the bottom right of the Serial Monitor window in ). If your output looks something like this:

```
`3??f<ÌxÌ    ü`³??f<
```

you should check that the selected baud rate on the Serial Monitor on your computer matches the rate set by `Serial.begin()` in your sketch.

You can transmit text using the `Serial.print()` function. Strings (text within double quotes) will be printed as is (but without the quotes). For example, the following code:

```
Serial.print("The number is ");
```

prints this:

```
The number is
```

The values (numbers) that you print depend on the type of variable; see Recipe 4.2 for more about this. For example, printing an integer will print its numeric value, so if the variable `number` is 1, the following code:

```
Serial.println(number);
```

will print whatever the current value of `number` happens to be:

```
1
```

In the example sketch, the number printed will be `0` when the loop starts and will increase by one each time through the loop. The `ln` at the end of `println` causes the next print statement to start on a new line.

> **NOTE**
>
> Keep in mind that there are two different serial port behaviors you will encounter with Arduino and Arduino-compatible boards: the Uno and most other AVR ATmega-based boards will reset when you open the serial port. This means that you will always see the count begin at zero in the Serial Monitor or Plotter. The Arduino Leonardo, as well as ARM-based boards, do not automatically reset when you open the serial port. This means that the sketch will begin counting as soon as it is powered up. As a result, the value that you see when you first open the Serial Monitor or Plotter depends on when you open the serial connection to the board. See "Serial Hardware Behavior" for more details.

That should get you started printing text and the decimal value of integers. See Recipe 4.2 for more details on print formatting options.

You may want to consider a third-party terminal program that has more features than Serial Monitor. Displaying data in text or binary format (or both), displaying control characters, and logging to a file are just a few of the additional capabilities available from the many third-party terminal programs. Here are some that have been recommended by Arduino users:

CoolTerm

An easy-to-use freeware terminal program for Windows, Mac, and Linux

CuteCom

An open source terminal program for Linux

Bray Terminal

A free executable for the PC

GNU screen

An open source virtual screen management program that supports serial communications; included with Linux and macOS

moserial

Another open source terminal program for Linux

PuTTY

An open source SSH program for Windows and Linux that supports serial communications

An open source terminal program for the PC

A shareware program for the Mac

You can use a liquid crystal display (LCD) as a serial output device, although it will be very limited in functionality. Check the documentation to see how your display handles carriage returns, as some displays may not automatically advance to a new line after `println` statements. Also, when you are using an LCD display, you will be connecting to it using the TTL serial pins (digital 0 and 1) rather than a USB connection. On most AVR ATmega boards like the Uno, these pins correspond to the `Serial` object so you can use the code shown in the Solution unchanged. However, on the Leonardo or certain ARM-based boards (SAMD-based boards, for example), pins 0 and 1 correspond to the `Serial1` object, so you'll need to change `Serial` to `Serial1` in order for it to work on those boards. See Table 4-1 for a list of Serial object pin configurations for a variety of boards.

## See Also

The Arduino LiquidCrystal library for text LCDs uses underlying print functionality similar to the Serial library, so you can use many of the suggestions covered in this chapter with that library (see Chapter 11).

# 4.2 Sending Formatted Text and Numeric Data from Arduino

**Problem**

You want to send serial data from Arduino displayed as text, decimal values, hexadecimal, or binary.

## Solution

You can print data to the serial port in many different formats; here is a sketch that demonstrates all the format options available with the serial print functions `print()` and `println`:

```
/*
  SerialFormatting
  Print values in various formats to the serial port
*/
char chrValue  = 65;   // these are the starting values to print
byte byteValue = 65;
int intValue   = 65;
float floatValue = 65.0;
char c1 = 4;
char c2 = 210;

void setup()
{
  while(!Serial); // Wait until serial port's open on Leonardo and SAMD boards
  Serial.begin(9600);
}

void loop()
{
  Serial.print("chrValue:   ");
  Serial.print(chrValue); Serial.print(" ");
  Serial.write(chrValue); Serial.print(" ");
  Serial.print(chrValue, DEC);
  Serial.println();

  Serial.print("byteValue:  ");
  Serial.print(byteValue); Serial.print(" ");
  Serial.write(byteValue); Serial.print(" ");
  Serial.print(byteValue, DEC);
  Serial.println();

  Serial.print("intValue:   ");
  Serial.print(intValue); Serial.print(" ");
  Serial.print(intValue, DEC); Serial.print(" ");
  Serial.print(intValue, HEX); Serial.print(" ");
```

```
    Serial.print(intValue, OCT); Serial.print(" ");
    Serial.print(intValue, BIN);
    Serial.println();

    Serial.print("floatValue: ");
    Serial.println(floatValue);
    Serial.println();

    delay(1000); // delay a second between numbers
    chrValue++;  // to the next value
    byteValue++;
    intValue++;
    floatValue += 1;
  }
```

The output is as follows:

```
chrValue:   A A 65
byteValue:  65 A 65
intValue:   65 65 41 101 1000001
floatValue: 65.00

chrValue:   B B 66
byteValue:  66 B 66
intValue:   66 66 42 102 1000010
floatValue: 66.00
...
```

## Discussion

Printing a text string is simple: `Serial.print("hello world");` sends the text string "hello world" to a device at the other end of the serial port. If you want your output to print a newline after the output, use `Serial.println()` instead of `Serial.print()`.

Printing numeric values can be more complicated. The way that byte and integer values are printed depends on the type of variable and an optional formatting parameter. The Arduino language is very easygoing about how you can refer to the value of different data types (see Recipe 2.2 for more on data types). But this flexibility can be confusing, because even when the numeric values are similar, the compiler considers them to be separate types

with different behaviors. For example, printing a `char`, `byte`, and `int` of the same value will not necessarily produce the same output.

Here are some specific examples; all of them create variables that have similar values:

```
char asciiValue  = 'A';   // ASCII A has a value of 65
char chrValue    = 65;    // an 8-bit signed character, this also is ASCII 'A'
byte byteValue   = 65;    // an 8-bit unsigned character, this also is ASCII
 'A'
int intValue     = 65;    // a 16-bit signed integer set to a value of 65
float floatValue = 65.0;  // float with a value of 65
```

Table 4-3 shows what you will see when you print variables using Arduino routines.

*Output formats using Serial.print*

| Data type | print (val) | print (val,DEC) | write (val) | print (val,HEX) | print (val,OCT) | print (val,BIN) |
|---|---|---|---|---|---|---|
| char | A | 65 | A | 41 | 101 | 1000001 |
| byte | 65 | 65 | A | 41 | 101 | 1000001 |
| int | 65 | 65 | A | 41 | 101 | 1000001 |
| long | Format of long is the same as int | | | | | |
| float | 65.00 | Formatting not supported for floating-point values | | | | |
| double | 65.00 | double on Uno is same as float | | | | |

The sketch in this recipe uses a separate line of source code for each print statement. This can make complex print statements bulky. For example, to print the following line:

```
At 5 seconds: speed = 17, distance = 120
```

you'd typically have to code it like this:

```
Serial.print("At ");
Serial.print(t);
Serial.print(" seconds: speed = ");
Serial.print(s);
Serial.print(", distance = ");
Serial.println(d);
```

That's a lot of code lines for a single line of output. You could combine them like this:

```
Serial.print("At "); Serial.print(t); Serial.print(" seconds, speed = ");
Serial.print(s); Serial.print(", distance = ");Serial.println(d);
```

Or you could use the *insertion-style* capability of the compiler used by Arduino to format your print statements. You can take advantage of some advanced C++ capabilities (streaming insertion syntax and templates) that you can use if you declare a streaming template in your sketch. This is most easily achieved by including the Streaming library developed by Mikal Hart. You can read more about this library at Mikal's website, and you can install it using the Arduino Library Manager (see Recipe 16.2).

If you use the Streaming library, the following gives the same output as the lines shown earlier:

```
Serial << "At " << t << " seconds, speed=" << s << ", distance=" << d << endl;
```

If you are an experienced programmer you may be wondering why Arduino does not support `printf`. In part this is due to `printf`'s use of dynamic memory and the shortage of RAM on the 8-bit boards. Recent 32-bit boards have plenty of memory, however the Arduino team has been reluctant to include the terse and easily abused syntax as part of the Arduino language's serial output capabilities.

Although Arduino does not include support for `printf`, you can use its sibling `sprintf` to store formatted text in a character buffer, and then print that buffer using `Serial.print/println`:

```
char buf[100];
sprintf(buf, "At %d seconds, speed = %d, distance = %d", t, s, d);
Serial.println(buf);
```

But `sprintf` can be dangerous. If the string you're writing is larger than your buffer, it will overflow. It's anyone's guess as to where the overflow characters will be written, but almost certainly they will cause your sketch to crash or otherwise misbehave. The `snprintf` function allows you to pass an argument specifying the maximum number of characters (including the null character that terminates all strings). You can specify the same length you use to declare the array (in this case, 100), but if you do that, you need to remember to change the length argument if you change the buffer length. Instead, you can use the `sizeof` operator to calculate the length of the buffer. Although a `char` is 1 byte in every case we can think of, it's best practice to divide the size of the array by the size of the data type it contains, so `sizeof (buf) / sizeof (buf[0])` will give you the length of the array:

```
snprintf(buf, sizeof (buf) / sizeof (buf[0]),
         "At %d seconds, speed = %d, distance = %d", t, s, d);
Serial.println(buf);
```

> **NOTE**
>
> Even though you know that `sizeof (buf[0])` is guaranteed to be 1, this is a good habit to get into. Consider the following code, which prints 400:
>
> ```
> long buf2[100];
> Serial.println(sizeof (buf2));
> ```
>
> You can get the correct result with `sizeof (buf2) / sizeof (buf2[0])`.

There is a cost associated with using `sprintf` or `snprintf`. First, you've got the overhead of the buffer, 100 bytes in this case. Additionally, there is the overhead of compiling the functionality into your sketch. On an Arduino Uno, adding in this code increases your memory usage by 1,648 bytes, which is 5% of the Uno's memory.

## See Also

Chapter 2 provides more information on data types used by Arduino. The Arduino web reference covers the serial commands as well as the streaming (insertion-style) output.

# 4.3 Receiving Serial Data in Arduino

## Problem

You want to receive data on Arduino from a computer or another serial device; for example, to have Arduino react to commands or data sent from your computer.

## Solution

This sketch receives a digit (single characters 0 through 9) and blinks the onboard LED at a rate proportional to the received digit value:

```
/*
 * SerialReceive sketch
 * Blink the LED at a rate proportional to the received digit value
*/
int blinkDelay = 0;  // blink delay stored in this variable

void setup()
{
  Serial.begin(9600); // Initialize serial port to send and receive at 9600
baud
  pinMode(LED_BUILTIN, OUTPUT); // set this pin as output
}

void loop()
{
  if (Serial.available()) // Check to see if at least one character is
available
  {
    char ch = (char) Serial.read();
    if( isDigit(ch) ) // is this an ASCII digit between 0 and 9?
    {
      blinkDelay = (ch - '0');        // ASCII value converted to numeric value
      blinkDelay = blinkDelay * 100; // actual delay is 100 ms *" received
digit
    }
  }
  blink();
}

// blink the LED with the on and off times determined by blinkDelay
void blink()
{
  digitalWrite(LED_BUILTIN, HIGH);
  delay(blinkDelay); // delay depends on blinkDelay value
  digitalWrite(LED_BUILTIN, LOW);
  delay(blinkDelay);
}
```

Upload the sketch and send messages using the Serial Monitor. Open the Serial Monitor by clicking the Monitor icon (see Recipe 4.1) and type a digit in the text box at the top of the Serial Monitor window. Clicking the Send button will send the character typed into the text box; if you type a digit, you should see the blink rate change.

## Discussion

The `Serial.read` function returns an `int` value, so you should cast it to a `char` for the comparisons that follow. Converting the received ASCII characters to numeric values may not be obvious if you are not familiar with the way ASCII represents characters. The following converts the character `ch` to its numeric value:

```
blinkDelay = (ch - '0');    // ASCII value converted to numeric value
```

The ASCII characters `'0'` through `'9'` have a value of 48 through 57 (see Appendix G). Converting `'1'` to the numeric value 1 is done by subtracting `'0'` because `'1'` has an ASCII value of 49, so 48 (ASCII `'0'`) must be subtracted to convert this to the number 1. For example, if `ch` is representing the character 1, its ASCII value is 49. The expression `49- '0'` is the same as `49-48`. This equals 1, which is the numeric value of the character 1.

In other words, the expression `(ch - '0')` is the same as `(ch - 48)`; this converts the ASCII value of the variable `ch` to a numeric value.

You can receive numbers with more than one digit using the `parseInt` and `parseFloat` methods that simplify extracting numeric values from `Serial`. (It also works with `Ethernet` and other objects derived from the `Stream` class; see the introduction to Chapter 15 for more about stream-parsing with the networking objects.)

`Serial.parseInt()` and `Serial.parseFloat()` read `Serial` characters and return their numeric representation. Nonnumeric characters before the number are ignored and the number ends with the first character that is not a numeric digit (or `"."` if using `parseFloat`). If there aren't any numeric characters in the input, the functions return 0, so you should check for zero values and handle them appropriately. If you have the Serial Monitor configured to send a newline or carriage return (or both) when you click Send, `parseInt` or `parseFloat` will try (and fail) to interpret the newline or carriage return as a number, and return a zero. This would result in

`blinkDelay` being set to zero immediately after setting it to your intended value, which would result in no blinking:

```
/*
 * SerialParsing sketch
 * Blink the LED at a rate proportional to the received digit value
 */

int blinkDelay = 0;

void setup()
{
  Serial.begin(9600); // Initialize serial port to send and receive at 9600
baud
  pinMode(LED_BUILTIN, OUTPUT); // set this pin as output
}

void loop()
{
  if ( Serial.available()) // Check to see if at least one character is
available
  {
    int i = Serial.parseInt();
    if (i != 0) {
        blinkDelay = i;
    }
  }
  blink();
}

// blink the LED with the on and off times determined by blinkDelay
void blink()
{
  digitalWrite(LED_BUILTIN, HIGH);
  delay(blinkDelay); // delay depends on blinkDelay value
  digitalWrite(LED_BUILTIN, LOW);
  delay(blinkDelay);
}
```

See the Discussion of Recipe 4.5 for another example showing `parseInt` used to find and extract numbers from Serial data.

Another approach to converting text strings representing numbers is to use the C language conversion function called `atoi` (for `int` variables) or `atol`

(for `long` variables). These obscurely named functions convert a string into integers or long integers. They provide more capability to manipulate the incoming data at the cost of greater code complexity. To use these functions, you have to receive and store the entire string in a character array before you can call the conversion function.

This code fragment terminates the incoming digits on any character that is not a digit (or if the buffer is full):

```
const int maxChars = 5;     // an int string contains up to 5 digits and
                            // is terminated by a 0 to indicate end of string
char strValue[maxChars+1]; // must be big enough for digits and terminating
null
int idx = 0;                // index into the array storing the received digits

void loop()
{
  if( Serial.available())
  {
    char ch = (char) Serial.read();
    if( idx < maxChars && isDigit(ch) ){
      strValue[idx++] = ch; // add the ASCII character to the string;
    }
    else
    {
      // here when buffer full or on the first nondigit
      strValue[idx] = 0;              // terminate the string with a 0
      blinkDelay = atoi(strValue); // use atoi to convert the string to an int
      idx = 0;
    }
  }
  blink();
}
```

`strValue` is a numeric string built up from characters received from the serial port.

---

**TIP**

See Recipe 2.6 for information about character strings.

---

`atoi` (short for ASCII to integer) is a function that converts a character string to an integer (`atol` converts to a `long` integer).

Arduino also supports the `serialEvent` function that you can use to handle incoming serial characters. If you have code within a `serialEvent` function in your sketch, this will be called once each time through the `loop` function. The following sketch performs the same function as the first sketch in this recipe but uses `serialEvent` to handle the incoming characters:

```
/*
 * SerialEvent Receive sketch
 * Blink the LED at a rate proportional to the received digit value
 */
int blinkDelay = 0;      // blink delay stored in this variable

void setup()
{
  Serial.begin(9600); // Initialize serial port to send and receive at 9600
baud
  pinMode(LED_BUILTIN, OUTPUT); // set this pin as output
}

void loop()
{
  blink();
}

void serialEvent()
{
  while(Serial.available())
  {
    char ch = (char) Serial.read();
    Serial.write(ch);
    if( isDigit(ch) ) // is this an ASCII digit between 0 and 9?
    {
      blinkDelay = (ch - '0');       // ASCII value converted to numeric value
      blinkDelay = blinkDelay * 100; // actual delay is 100 ms times digit
    }
  }
}

// blink the LED with the on and off times determined by blinkDelay
void blink()
{
```

```
    digitalWrite(LED_BUILTIN, HIGH);
    delay(blinkDelay); // delay depends on blinkDelay value
    digitalWrite(LED_BUILTIN, LOW);
    delay(blinkDelay);
  }
```

## See Also

A web search for "atoi" or "atol" provides many references to these functions (see the Wikipedia reference here).

# 4.4 Sending Multiple Text Fields from Arduino in a Single Message

## Problem

You want to send a message that contains more than one field's worth of information per message. For example, your message may contain values from two or more sensors. You want to use these values in a program such as Processing, running on a computer or a device such as a Raspberry Pi.

## Solution

An easy way to do this is to send a text string with all the fields separated by a delimiting (separating) character, such as a comma:

```
// CommaDelimitedOutput sketch

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

void loop()
{
  int value1 = 10;    // some hardcoded values to send
  int value2 = 100;
  int value3 = 1000;

  Serial.print('H'); // unique header to identify start of message
  Serial.print(",");
```

```
    Serial.print(value1,DEC);
    Serial.print(",");
    Serial.print(value2,DEC);
    Serial.print(",");
    Serial.print(value3,DEC);
    Serial.println();   // send a carriage return and line feed
    delay(100);
}
```

Here is the Processing sketch that reads this data from the serial port:

```
// Processing Sketch to read comma delimited serial
// expects format: H,1,2,3

import processing.serial.*;

Serial myPort;          // Create object from Serial class
char HEADER = 'H';      // character to identify the start of a message
short LF = 10;          // ASCII linefeed

// WARNING!
// If necessary change the definition below to the correct port
short portIndex = 0;  // select the com port, 0 is the first port

void setup() {
  size(200, 200);
  println( (Object[]) Serial.list());
  println(" Connecting to -> " + Serial.list()[portIndex]);
  myPort = new Serial(this, Serial.list()[portIndex], 9600);
}

void draw() {
  if (myPort.available() > 0) {

    String message = myPort.readStringUntil(LF); // read serial data
    if (message != null)
    {
      message = message.trim(); // Remove whitespace from start/end of string
      println(message);
      String [] data = message.split(","); // Split the comma-separated
message
      if (data[0].charAt(0) == HEADER && data.length == 4) // check validity
      {
        for (int i = 1; i < data.length; i++) // skip header (start at 1, not
0)
          {
```

```
        println("Value " + i + " = " + data[i]);  // Print the field values
      }
      println();
    }
  }
}
```

## Discussion

The Arduino code in this recipe's Solution will send the following text string to the serial port (\r indicates a carriage return and \n indicates a line feed):

```
H,10,100,1000\r\n
```

You must choose a separating character that will never occur within actual data; for instance, if your data consists only of numeric values, a comma is a good choice for a delimiter. You may also want to ensure that the receiving side can determine the start of a message to make sure it has all the data for all the fields. You do this by sending a header character to indicate the start of the message. The header character must also be unique; it should not appear within any of the data fields and it must also be different from the separator character. The example here uses an uppercase *H* to indicate the start of the message. The message consists of the header, three comma-separated numeric values as ASCII strings, and a carriage return and line feed.

The carriage return and line-feed characters are sent whenever Arduino prints using the `println()` function, and this is used to help the receiving side know that the full message string has been received. Because the Processing code `myPort.readStringUntil(LF)` will include the carriage return (`'\r'`) that appears before the line feed, this sketch uses `trim()` to remove any leading or trailing whitespace, which includes spaces, tabs, carriage returns, and line feeds.

The Processing code reads the message as a string and uses the Java `split()` method to create an array from the comma-separated fields.

Using Processing to display sensor values can save hours of debugging time by helping you to visualize the data. While CSV is a common and useful format, JSON (JavaScript Object Notation) is more expressive and also is human readable. JSON is a common data exchange format used for exchanging messages across a network. The following sketch reads the accelerometer from the Arduino WiFi Rev 2 or Arduino Nano 33 BLE Sense (uncomment the corresponding `include` line) and sends it to the serial port using JSON (for example: `{'x': 0.66, 'y': 0.59, 'z': -0.49, }`):

```
/*
   AccelerometerToJSON. Sends JSON-formatted representation of
   accelerometer readings.
*/

#include <Arduino_LSM6DS3.h>   // Arduino WiFi R2
//#include <Arduino_LSM9DS1.h> // Arduino BLE Sense

void setup() {
  Serial.begin(9600);
  while (!Serial);

  if (!IMU.begin()) {
    while (1) {
      Serial.println("Error: Failed to initialize IMU");
      delay(3000);
    }
  }
}

void loop() {
  float x, y, z;
```

```
      if (IMU.accelerationAvailable()) {
        IMU.readAcceleration(x, y, z);
        Serial.print("{");
        Serial.print("'x': "); Serial.print(x); Serial.print(", ");
        Serial.print("'y': "); Serial.print(y); Serial.print(", ");
        Serial.print("'z': "); Serial.print(z); Serial.print(", ");
        Serial.println("}");
        delay(200);
    }
}
```

The following Processing sketch adds real-time visual display of up to 12 values sent from Arduino. It displays floating-point values in a range from –5 to +5:

```
/*
 * ShowSensorData.
 *
 * Displays bar graph of JSON sensor data ranging from -127 to 127
 * expects format as: "{'label1': value, 'label2': value,}\n"
 * for example:
 * {'x': 1.0, 'y': -1.0, 'z': 2.1,}
 */

import processing.serial.*;
import java.util.Set;

Serial myPort;    // Create object from Serial class
PFont fontA;      // font to display text
int fontSize = 12;
short LF = 10;          // ASCII linefeed

int rectMargin   = 40;
int windowWidth = 600;
int maxLabelCount = 12; // Increase this if you need to support more labels
int windowHeight  = rectMargin + (maxLabelCount + 1) * (fontSize *2);
int rectWidth   = windowWidth - rectMargin*2;
int rectHeight = windowHeight - rectMargin;
int rectCenter = rectMargin + rectWidth / 2;

int origin = rectCenter;
int minValue = -5;
int maxValue = 5;

float scale = float(rectWidth) / (maxValue - minValue);
```

```
// WARNING!
// If necessary change the definition below to the correct port
short portIndex = 0;   // select the com port, 0 is the first port

void settings() {
  size(windowWidth, windowHeight);
}

void setup() {
  println( (Object[]) Serial.list());
  println(" Connecting to -> " + Serial.list()[portIndex]);
  myPort = new Serial(this, Serial.list()[portIndex], 9600);
  fontA = createFont("Arial.normal", fontSize);
  textFont(fontA);
}

void draw() {

  if (myPort.available () > 0) {
    String message = myPort.readStringUntil(LF);
    if (message != null) {

      // Load the JSON data from the message
      JSONObject json = new JSONObject();
      try {
        json = parseJSONObject(message);
      }
      catch(Exception e) {
        println("Could not parse [" + message + "]");
      }

      // Copy the JSON labels and values into separate arrays.
      ArrayList<String> labels = new ArrayList<String>();
      ArrayList<Float> values = new ArrayList<Float>();
      for (String key : (Set<String>) json.keys()) {
        labels.add(key);
        values.add(json.getFloat(key));
      }

      // Draw the grid and chart the values
      background(255);
      drawGrid(labels);
      fill(204);
      for (int i = 0; i < values.size(); i++) {
        drawBar(i, values.get(i));
      }
    }
  }
```

```
  }

  // Draw a bar to represent the current sensor reading
  void drawBar(int yIndex, float value) {
    rect(origin, yPos(yIndex)-fontSize, value * scale, fontSize);
  }

  void drawGrid(ArrayList<String> sensorLabels) {
    fill(0);

    // Draw the minimum value label and a line for it
    text(minValue, xPos(minValue), rectMargin-fontSize);
    line(xPos(minValue), rectMargin, xPos(minValue), rectHeight + fontSize);

    // Draw the center value label and a line for it
    text((minValue+maxValue)/2, rectCenter, rectMargin-fontSize);
    line(rectCenter, rectMargin, rectCenter, rectHeight + fontSize);

    // Draw the maximum value label and a line for it
    text(maxValue, xPos(maxValue), rectMargin-fontSize);
    line(
    xPos(maxValue), rectMargin, xPos(maxValue), rectHeight + fontSize);

    // Print each sensor label
    for (int i=0; i < sensorLabels.size(); i++) {
      text(sensorLabels.get(i), fontSize, yPos(i));
      text(sensorLabels.get(i), xPos(maxValue) + fontSize, yPos(i));
    }
  }

  // Calculate a y position, taking into account margins and font sizes
  int yPos(int index) {
    return rectMargin + fontSize + (index * fontSize*2);
  }

  // Calculate a y position, taking into account the scale and origin
  int xPos(int value) {
    return origin  + int(scale * value);
  }
}
```
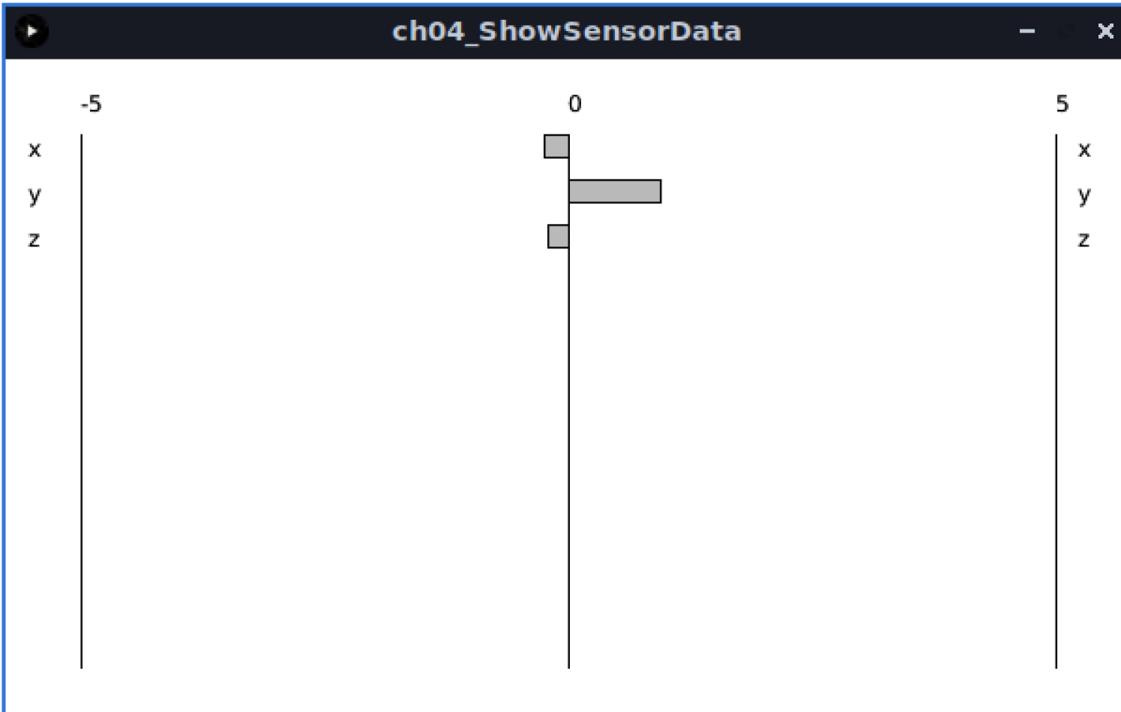
Figure 4-4 shows how accelerometer values (x, y, z) are displayed. Bars
will appear as you wave the device.

*Processing screen showing sensor data*

The range of values and the origin of the graph can be easily changed if desired. For example, to display bars originating at the lefthand axis with values from 0 to 1,024, use the following:

```
int origin = rectMargin; // rectMargin is the left edge of the graphing area
int minValue = 0;
int maxValue = 1024;
```

If you don't have an accelerometer, you can generate values with the following simple sketch that displays analog input values. If you don't have any sensors to connect, running your fingers along the bottom of the analog pins will produce levels that can be viewed in the Processing sketch. The values range from 0 to 1,023, so change the origin and min and max values in the Processing sketch, as described in the previous paragraph:

```
/*
    AnalogToJSON. Sends JSON-formatted representation of
    analog readings.
```

```
    */

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

void loop() {
  Serial.print("{");
  Serial.print("'A0': "); Serial.print(analogRead(A0)); Serial.print(", ");
  Serial.print("'A1': "); Serial.print(analogRead(A1)); Serial.print(", ");
  Serial.print("'A2': "); Serial.print(analogRead(A2)); Serial.print(", ");
  Serial.print("'A3': "); Serial.print(analogRead(A3)); Serial.print(", ");
  Serial.print("'A4': "); Serial.print(analogRead(A4)); Serial.print(", ");
  Serial.print("'A5': "); Serial.print(analogRead(A5)); Serial.print(", ");
  Serial.println("}");
}
```

## See Also

The Processing website provides more information on installing and using this programming environment.

A number of books on Processing are also available:

- *Getting Started with Processing,* Second Edition, by Casey Reas and Ben Fry (Make)

- *Processing: A Programming Handbook for Visual Designers and Artists* by Casey Reas and Ben Fry (MIT Press)

- *Visualizing Data* by Ben Fry (O'Reilly)

- *Processing: Creative Coding and Computational Art* by Ira Greenberg (Apress)

- *Making Things Talk* by Tom Igoe (Make Community) (This book covers Processing and Arduino and provides many examples of communication code.)

# 4.5 Receiving Multiple Text Fields in a Single Message in Arduino

## Problem

You want to receive a message that contains more than one field. For example, your message may contain an identifier to indicate a particular device (such as a motor or other actuator) and what value (such as speed) to set it to.

## Solution

Arduino does not have the `split()` function used in the Processing code in Recipe 4.4, but similar functionality can be implemented as shown in this recipe. The following code receives a message with a single character H as the header followed by three numeric fields separated by commas and terminated by the newline character:

```
/*
 * SerialReceiveMultipleFields sketch
 * This code expects a message in the format: H,12,345,678
 * This code requires a newline character to indicate the end of the data
 * Set the serial monitor to send newline characters
 */

const int NUMBER_OF_FIELDS = 3; // how many comma separated fields we expect
int values[NUMBER_OF_FIELDS];   // array holding values for all the fields

void setup()
{
  Serial.begin(9600); // Initialize serial port to send and receive at 9600
baud
}

void loop()
{
  if ( Serial.available())
  {
    if (Serial.read() == 'H') {

      // Read the values
      for (int i = 0; i < NUMBER_OF_FIELDS; i++)
      {
        values[i] = Serial.parseInt();
      }
```

```
    // Display the values in comma-separated format
    Serial.print(values[0]); // First value

    // Print the rest of the values with a leading comma
    for (int i = 1; i < NUMBER_OF_FIELDS; i++)
    {
      Serial.print(","); Serial.print(values[i]);
    }
    Serial.println();
  }
 }
}
```

## Discussion

This sketch uses the `parseInt` method that makes it easy to extract numeric values from serial and web streams. This is one example of how to use this capability (Chapter 15 has more examples of stream parsing). You can test this sketch by opening the Serial Monitor and sending a comma-separated message like `H1,2,3`. `parseInt` ignores anything other than a minus sign and a digit, so it doesn't have to be comma-separated. You can use another delimiter like `H1/2/3`. The sketch stores the numbers in an array, and then prints them out, separated by commas.

> **NOTE**
>
> This sketch displays a comma-separated list in a way that may seem unusual at first. It prints the first number received (for example, 1), and then prints the remaining numbers, each preceded by a comma (for example, `,2` and then `,3`). You could use fewer lines of code and print a comma after each number, but you'd end up with `1,2,3,` instead of `1,2,3`. Many other programming languages, including Processing, provide a `join` function that will combine an array into a delimited string, but Arduino does not.

The stream-parsing functions will time out waiting for a character; the default is one second. If no digits have been received and `parseInt` times out, then it will return 0. You can change the timeout by calling `Stream.setTimeout(timeoutPeriod)`. The timeout parameter is a `long`

integer indicating the number of milliseconds, so the timeout range is from 1 ms to 2,147,483,647 ms.

`Stream.setTimeout(2147483647);` will change the timeout interval to just under 25 days.

Following is a summary of the stream-parsing methods supported by Arduino (not all are used in the preceding example):

`bool find(char *target);`

> Reads from the stream until the given target is found. It returns `true` if the target string is found. A return of `false` means the data has not been found anywhere in the stream and that there is no more data available. Note that `Stream` parsing takes a single pass through the stream; there is no way to go back to try to find or get something else (see the `findUntil` method).

`bool findUntil(char *target, char *terminate);`

> Similar to the `find` method, but the search will stop if the `terminate` string is found. Returns `true` only if the target is found. This is useful to stop a search on a keyword or terminator. For example:

```
finder.findUntil("target", "\n");
```

> will try to seek to the string `"value"`, but will stop at a newline character so that your sketch can do something else if the target is not found.

`long parseInt();`

> Returns the first valid (`long`) integer value. Leading characters that are not digits or a minus sign are skipped. The integer is terminated by the first nondigit character following the number. If no digits are found, the function returns `0`.

`long parseInt(char skipChar);`

Same as `parseInt`, but the given `skipChar` within the numeric value is ignored. This can be helpful when parsing a single numeric value that uses a comma between blocks of digits in large numbers, but bear in mind that text values formatted with commas cannot be parsed as a comma-separated string (for example, 32,767 would be parsed as 32767).

`float parseFloat();`

The `float` version of `parseInt`. All characters except digits, a decimal point, or a leading minus sign are skipped.

`size_t readBytes(char *buffer, size_t length);`

Puts the incoming characters into the given buffer until timeout or length characters have been read. Returns the number of characters placed in the buffer.

`size_t readBytesUntil(char terminator, char *buf, size_t length);`

Puts the incoming characters into the given buffer until the `terminator` character is detected. Strings longer than the given `length` are truncated to fit. The function returns the number of characters placed in the buffer.

## See Also

Chapter 15 provides more examples of stream parsing used to find and extract data from a stream. The ArduinoJson library lets you parse JSON-formatted messages (see Recipe 4.4) in Arduino.

# 4.6 Sending Binary Data from Arduino

## Problem

You need to send data in binary format, because you want to pass information with the fewest number of bytes or because the application you are connecting to only handles binary data.

## Solution

This sketch sends a header followed by two integer (two-byte) values as binary data. The sketch uses `short` because it will be two bytes regardless of whether you have an 8-bit or 32-bit board (see Recipe 2.2). The values are generated using the Arduino `random` function (see Recipe 3.11). Although `random` returns a `long` value, the argument of 599 means it will never return a value over that number, which is small enough to fit in a `short`:

```
/*
 * SendBinary sketch
 * Sends a header followed by two random integer values as binary data.
 */

short intValue; // short integer value (two bytes on all boards)

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

void loop()
{
  Serial.print('H'); // send a header character

  // send a random integer
  intValue = random(599); // generate a random number between 0 and 599
  // send the two bytes that comprise that integer
  Serial.write(lowByte(intValue));  // send the low byte
  Serial.write(highByte(intValue)); // send the high byte

  // send another random integer
  intValue = random(599); // generate a random number between 0 and 599
  // send the two bytes that comprise that integer
  Serial.write(lowByte(intValue));  // send the low byte
  Serial.write(highByte(intValue)); // send the high byte

  delay(1000);
}
```

## Discussion

Sending binary data requires careful planning, because you will get gibberish unless the sending side and the receiving side understand and agree exactly how the data will be sent. Unlike text data, where the end of a message can be determined by the presence of the terminating carriage return (or another unique character you pick), it may not be possible to tell when a binary message starts or ends by looking just at the data—data that can have any value can therefore have the value of a header or terminator character.

This can be overcome by designing your messages so that the sending and receiving sides know exactly how many bytes are expected. The end of a message is determined by the number of bytes sent rather than detection of a specific character. This can be implemented by sending an initial value to say how many bytes will follow. Or you can fix the size of the message so that it's big enough to hold the data you want to send. Doing either of these is not always easy, as different platforms and languages can use different sizes for the binary data types—both the number of bytes and their order may be different from Arduino. For example, Arduino defines an `int` as two bytes (16 bits) on 8-bit platforms, four bytes (32 bits) on a 32-bit platform, and Processing (Java) defines an `int` as four bytes (`short` is the Java type for a two-byte integer). Sending an `int` value as text (as seen in earlier text recipes) simplifies this problem because each individual digit is sent as a sequential digit (just as the number is written). The receiving side recognizes when the value has been completely received by a carriage return or other nondigit delimiter. Binary transfers can only know about the composition of a message if it is defined in advance or specified in the message.

This recipe's Solution requires an understanding of the data types on the sending and receiving platforms and some careful planning. Recipe 4.7 shows example code using the Processing language to receive these messages.

Sending single bytes is easy; use `Serial.write(byteVal)`. To send an integer from Arduino you need to send the low and high bytes that make up

the integer (see Recipe 2.2 for more on data types). You do this using the `lowByte` and `highByte` functions (see Recipe 3.14):

```
Serial.write(lowByte(intValue));
Serial.write(highByte(intValue));
```

Sending a `long` integer is done by breaking down the four bytes that comprise a `long` in two steps. The `long` is first broken into two 16-bit integers; each is then sent using the method for sending integers described earlier:

```
long longValue = 2147483648;
int intValue;
```

First you send the lower 16-bit integer value:

```
intValue = longValue & 0xFFFF;  // get the value of the lower 16 bits
Serial.write(lowByte(intValue));
Serial.write(highByte(intValue));
```

Then you send the higher 16-bit integer value:

```
intValue = longValue >> 16;  // get the value of the higher 16 bits
Serial.write(lowByte(intValue));
Serial.write(highByte(intValue));
```

You may find it convenient to create functions to send the data. Here is a function that uses the code shown earlier to print a 16-bit integer to the serial port:

```
// function to send the given integer value to the serial port
void sendBinary(int value)
{
  // send the two bytes that comprise a two-byte (16-bit) integer
  Serial.write(lowByte(value));  // send the low byte
  Serial.write(highByte(value)); // send the high byte
}
```

The following function sends the value of a `long` (four-byte) integer by first sending the two low (rightmost) bytes, followed by the high (leftmost) bytes:

```
// function to send the given long integer value to the serial port
void sendBinary(long value)
{
  // first send the low 16-bit integer value
  int temp = value & 0xFFFF;  // get the value of the lower 16 bits
  sendBinary(temp);
  // then send the higher 16-bit integer value:
  temp = value >> 16;  // get the value of the higher 16 bits
  sendBinary(temp);
}
```

These functions to send binary `int` and `long` values have the same name: `sendBinary`. The compiler distinguishes them by the type of value you use for the parameter. If your code calls `printBinary` with a two-byte value, the version declared as `void sendBinary(int value)` will be called. If the parameter is a `long` value, the version declared as `void sendBinary(long value)` will be called. This behavior is called *function overloading*. Recipe 4.2 provides another illustration of this; the different functionality you saw in `Serial.print` is due to the compiler distinguishing the different variable types used.

You can also send binary data using structures. Structures are a mechanism for organizing data, and if you are not already familiar with their use you may be better off sticking with the solutions described earlier. For those who are comfortable with the concept of structure pointers, the following will send the bytes within a structure to the serial port as binary data. It includes the header character in the struct, so it sends the same messages as the Solution:

```
/*
   SendBinaryStruct sketch
   Sends a struct as binary data.
*/

typedef struct {
```

```
    char padding; // ensure same alignment on 8-bit and 32-bit
    char header;
    short intValue1;
    short intValue2;
} shortMsg;

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

void loop()
{

    shortMsg myStruct = { 0, 'H', (short) random(599), (short) random(599) };
    sendStructure((byte *)&myStruct, sizeof(myStruct));

    delay(1000);
}

void sendStructure(byte *structurePointer, int structureLength)
{
    int i;
    for (i = 0 ; i < structureLength ; i++) {
        Serial.write(structurePointer[i]);
    }
}
```

> **NOTE**
>
> If you were to declare the shortMsg struct without the padding member, you might find that the struct length is five bytes on one board, and six bytes on another board. That's because the compiler for one architecture might be perfectly happy to allow a five-byte structure, but another might insert one or more extra bytes to ensure that the size of the struct is a multiple of the board's natural data size. By putting the padding at the front, you are ensuring that the char appears at an even boundary (the second of two bytes), so the compiler is unlikely to insert padding *between* the char and short values. But this trick isn't always guaranteed to work, so you may need to experiment. One other advantage of putting the padding before the header character is that the code in Recipe 4.7 will ignore input until it sees an H character.

Sending data as binary bytes is more efficient than sending data as text, but it will only work reliably if the sending and receiving sides agree exactly on

the composition of the data. Here is a summary of the important things to check when writing your code:

Variable size

Make sure the size of the data being sent is the same on both sides. An integer is two bytes on Arduino Uno and other 8-bit boards, and four bytes on 32-bit boards and most other platforms. Always check your programming language's documentation on data type size to ensure agreement. There is no problem with receiving a two-byte Arduino integer as a four-byte integer in Processing as long as Processing expects to get only two bytes. But be sure that the sending side does not use values that will overflow the type used by the receiving side.

Byte order

Make sure the bytes within an `int` or `long` are sent in the order expected by the receiving side. The solution uses the same byte order that Arduino boards use internally, called *little endian*. This refers to the order of the bytes, in which the least significant byte appears first. Technically, ARM-based Arduino-compatible boards are bi-endian, meaning that they can be configured to use big-endian or little-endian mode, but in practice you are unlikely to encounter an Arduino board that is not little endian. When you use `lowByte` and `highByte` to pick apart an integer, you are in control of the order in which the bytes are sent. But when you send a `struct` in binary format, it will use the struct's internal representation, which is impacted by the endianness of your board. So, if you run the struct code with the Processing code from Recipe 4.7 and don't see the intended value (16,384), your struct may be flipped around.

Synchronization

Ensure that your receiving side can recognize the beginning and end of a message. If you start listening in the middle of a transmission stream, you will not get valid data. This can be achieved by sending a sequence of bytes that won't occur in the body of a message. For example, if you

are sending binary values from `analogRead`, these can only range from 0 to 1,023, so the most significant byte must be less than 4 (the `int` value of 1,023 is stored as the bytes 3 and 255); therefore, there will never be data with two consecutive bytes greater than 3. So, sending two bytes of 4 (or any value greater than 3) cannot be valid data and can be used to indicate the start or end of a message.

Flow control

Either choose a transmission speed that ensures that the receiving side can keep up with the sending side, or use some kind of *flow control*. Flow control is a handshake that tells the sending side that the receiver is ready to get more data.

## See Also

Chapter 2 provides more information on the variable types used in Arduino sketches.

See Recipe 3.15 for more on handling high and low bytes. Also, check the Arduino references for `lowByte` and `highByte`.

For more on flow control, see the Wikipedia reference.

# 4.7 Receiving Binary Data from Arduino on a Computer

## Problem

You want to respond to binary data sent from Arduino in a programming language such as Processing. For example, you want to respond to Arduino messages sent in Recipe 4.6.

## Solution

This recipe's Solution depends on the programming environment you use on your PC or Mac. If you don't already have a favorite programming tool

and want one that is easy to learn and works well with Arduino, Processing is an excellent choice.

Here are the two lines of Processing code to read a byte, taken from the Processing `SimpleRead` example (see this chapter's introduction):

```
if ( myPort.available() > 0) {  // If data is available,
    val = myPort.read();           // read it and store it in val
```

As you can see, this is very similar to the Arduino code you saw in earlier recipes.

The following is a Processing sketch that sets the size of a rectangle proportional to the integer values received from the Arduino sketch in Recipe 4.6:

```
/*
 * ReceiveBinaryData_P
 *
 * portIndex must be set to the port connected to the Arduino
 */
import processing.serial.*;

Serial myPort;        // Create object from Serial class

// WARNING!
// If necessary change the definition below to the correct port
short portIndex = 0;  // select the com port, 0 is the first port

char HEADER = 'H';
int value1, value2;          // Data received from the serial port

void setup()
{
  size(600, 600);
  // Open whatever serial port is connected to Arduino.
  String portName = Serial.list()[portIndex];
  println((Object[]) Serial.list());
  println(" Connecting to -> " + portName);
  myPort = new Serial(this, portName, 9600);
}

void draw()
{
```

```
  // read the header and two binary *(16-bit) integers:
  if ( myPort.available() >= 5)  // If at least 5 bytes are available,
  {
    if( myPort.read() == HEADER) // is this the header
    {
      value1 = myPort.read();                  // read the least significant
byte
      value1 =  myPort.read() * 256 + value1; // add the most significant byte

      value2 = myPort.read();                  // read the least significant
byte
      value2 =  myPort.read() * 256 + value2; // add the most significant byte

      println("Message received: " + value1 + "," + value2);
    }
  }
  background(255);          // Set background to white
  fill(0);                 // set fill to black

  // draw rectangle with coordinates based on the integers received from
Arduino
  rect(0, 0, value1,value2);
}
```

---

**TIP**

Make sure that you set `portIndex` to correspond to the serial port that Arduino is connected to. You may need to run the sketch once, get an error, and review the list of serial ports in the Processing console at the bottom of the screen to determine which value you should use for `portIndex`.

---

## Discussion

The Processing language influenced Arduino, and the two are intentionally similar. The `setup` function in Processing is used to handle one-time initialization, just like in Arduino. Processing has a display window, and `setup` sets its size to 600 × 600 pixels with the call to `size(600,600)`.

The line `String portName = Serial.list()[portIndex];` selects the serial port—in Processing, all available serial ports are contained in the `Serial.list` object, and this example uses the value of a variable called

`portIndex. println((Object[]) Serial.list())` prints all the available ports, and the line `myPort = new Serial(this, portName, 9600);` opens the port selected as `portName`. Ensure that you set `portIndex` to the serial port that is connected to your Arduino. Arduino is usually the first port on a Mac; on a PC, it's usually the last port if the PC has a physical RS-232 port, otherwise it may be the first port. You can also look at the list of ports in the Arduino IDE, which may display serial ports in the same order that Processing enumerates them.

The `draw` function in Processing works like `loop` in Arduino; it is called repeatedly. The code in `draw` checks if data is available on the serial port; if so, bytes are read and converted to the integer value represented by the bytes. A rectangle is drawn based on the integer values received.

## See Also

You can read more about Processing on the Processing website.

# 4.8 Sending Binary Values from Processing to Arduino

## Problem

You want to send binary bytes, integers, or long values from Processing to Arduino. For example, you want to send a message consisting of a message identifier "tag" and two 16-bit values.

## Solution

Use this code:

```
// Processing Sketch

/* SendingBinaryToArduino
 * Language: Processing
 */
import processing.serial.*;
```

```
Serial myPort;  // Create object from Serial class

// WARNING!
// If necessary change the definition below to the correct port
short portIndex = 0;  // select the com port, 0 is the first port

public static final char HEADER    = 'H';
public static final char MOUSE_TAG = 'M';

void setup()
{
  size(512, 512);
  String portName = Serial.list()[portIndex];
  println((Object[]) Serial.list());
  myPort = new Serial(this, portName, 9600);
}

void draw(){
}

void serialEvent(Serial p) {
  // handle incoming serial data
  String inString = myPort.readStringUntil('\n');
  if(inString != null) {
    print( inString );   // print text string from Arduino
  }
}

void mousePressed() {
  sendMessage(MOUSE_TAG, mouseX, mouseY);
}

void sendMessage(char tag, int x, int y){
  // send the given index and value to the serial port
  myPort.write(HEADER);
  myPort.write(tag);
  myPort.write((char)(x / 256)); // msb
  myPort.write(x & 0xff);   //lsb
  myPort.write((char)(y / 256)); // msb
  myPort.write(y & 0xff);   //lsb
}
```

When the mouse is clicked in the Processing window, `sendMessage` will be called with the 8-bit tag indicating that this is a mouse message and the two 16-bit mouse x and y coordinates. The `sendMessage` function sends the 16-bit x and y values as two bytes, with the most significant byte first.

Here is the Arduino code to receive these messages and echo the results back to Processing:

```
// BinaryDataFromProcessing
// These defines must mirror the sending program:
const char HEADER      = 'H';
const char MOUSE_TAG   = 'M';
const int  TOTAL_BYTES = 6  ; // the total bytes in a message

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

void loop(){
  if ( Serial.available() >= TOTAL_BYTES)
  {
    if( Serial.read() == HEADER)
    {
      char tag = Serial.read();
      if(tag == MOUSE_TAG)
      {
        int x = Serial.read() * 256;
        x = x + Serial.read();
        int y = Serial.read() * 256;
        y = y + Serial.read();
        Serial.println("Got mouse msg:");
        Serial.print("x=");    Serial.print(x);
        Serial.print(", y="); Serial.println(y);
      }
```

```
      else
      {
        Serial.print("Unknown tag: ");
        Serial.write(tag); Serial.println();
      }
    }
  }
}
```

## Discussion

The Processing code sends a header byte to indicate that a valid message follows. This is needed so Arduino can synchronize if it starts up in the middle of a message or if the serial connection can lose data, such as with a wireless link. The tag provides an additional check for message validity and it enables any other message types you may want to send to be handled individually. In this example, the function is called with three parameters: a tag and the 16-bit `x` and `y` mouse coordinates.

The Arduino code checks that at least `MESSAGE_BYTES` have been received, ensuring that the message is not processed until all the required data is available. After the header and tag are checked, the 16-bit values are read as two bytes, with the first multiplied by 256 to restore the most significant byte to its original value.

If you would like to send Arduino's serial output to another device, such as an LCD serial character display, you could use a SoftwareSerial port or one of your board's additional serial ports, as shown in Recipe 4.11. You would need to initialize the serial port in `setup`, and change all the `Serial.write` and `Serial.print`/`println` statements to use that serial port. For example, the following changes would send serial data to the `Serial1` TX pin 1 of the Arduino WiFi Rev 2, Leonardo, and most ARM-based Arduino compatibles. You'd first add this to `setup`:

```
  Serial1.begin(9600);
```

And change the `print`/`println`/`write` code at the end of `loop` as shown:

```
Serial1.println();
Serial1.println("Got mouse msg:");
Serial1.print("x=");   Serial1.print(x);
Serial1.print(", y="); Serial1.print(y);
```

and:

```
Serial1.println();
Serial1.print("Unknown tag: ");
Serial1.write(tag); Serial1.println();
```

> **WARNING**
>
> The sending side and receiving side must use the same message size for binary messages to be handled correctly. If you want to increase or decrease the number of bytes to send, change `TOTAL_BYTES` in the Arduino code to match.

# 4.9 Sending the Values of Multiple Arduino Pins

## Problem

You want to send groups of binary bytes, integers, or long values from Arduino. For example, you may want to send the values of the digital and analog pins to Processing.

## Solution

This recipe sends a header followed by an integer containing the bit values of digital pins 2 to 13. This is followed by six integers containing the values of analog pins 0 through 5. Chapter 5 has many recipes that set values on the analog and digital pins that you can use to test this sketch:

```
/*
 * SendBinaryFields
 * Sends digital and analog pin values as binary data
 */
```

```
const char HEADER = 'H';  // a single character header to indicate
                          // the start of a message

void setup()
{
  Serial.begin(9600);
  for(int i=2; i <= 13; i++)
  {
    pinMode(i, INPUT);      // set pins 2 through 13 to inputs
    digitalWrite(i, HIGH);   // turn on pull-ups
  }
}

void loop()
{
  Serial.write(HEADER); // send the header
  // put the bit values of the pins into an integer
  int values = 0;
  int bit = 0;

  for(int i=2; i <= 13; i++)
  {
    bitWrite(values, bit, digitalRead(i));  // set the bit to 0 or 1 depending
                                            // on value of the given pin
    bit = bit + 1;                          // increment to the next bit
  }
  sendBinary(values); // send the integer

  for(int i=0; i < 6; i++)
  {
    values = analogRead(i);
    sendBinary(values); // send the integer
  }
  delay(1000); //send every second
}

// function to send the given integer value to the serial port
void sendBinary(int value)
{
  // send the two bytes that comprise an integer
  Serial.write(lowByte(value));  // send the low byte
  Serial.write(highByte(value)); // send the high byte
}
```

## Discussion

The code sends a header (the character H), followed by an integer holding the digital pin values using the bitRead function to set a single bit in the integer to correspond to the value of the pin (see Chapter 3). It then sends six integers containing the values read from the six analog ports (see Chapter 5 for more information). All the integer values are sent using sendBinary, introduced in Recipe 4.6. The message is 15 bytes long—one byte for the header, two bytes for the digital pin values, and 12 bytes for the six analog integers. The code for the digital and analog inputs is explained in Chapter 5.

Assuming analog pins have values of 0 on pin 0, 100 on pin 1, and 200 on pin 2 through 500 on pin 5, and digital pins 2 through 7 are high and 8 through 13 are low, this is the decimal value of each byte that gets sent:

```
72    // the character 'H' - this is the header
      // two bytes in low high order containing bits representing pins 2-13
63    // binary 00111111 :  this indicates that pins 2-7 are high
0     // this indicates that  8-13 are low

      // two bytes for each pin representing the analog value
0     // pin 0's analog value is 0 and this is sent as two bytes
0

100  // pin 1 has a value of 100, sent as a byte of 100 and a byte of 0
0
...
      // pin 5 has a value of 500
244  // the remainder when dividing 500 by 256
1    //  the number of times 500 can be divided by 256
```

This Processing code reads the message and prints the values to the Processing console:

```
// Processing Sketch

/*
 * ReceiveMultipleFieldsBinary_P
 *
 * portIndex must be set to the port connected to the Arduino
 */
```

```processing
import processing.serial.*;

Serial myPort;          // Create object from Serial class
short portIndex = 0;    // select the com port, 0 is the first port

char HEADER = 'H';

void setup()
{
  size(200, 200);
  // Open whatever serial port is connected to Arduino.
  String portName = Serial.list()[portIndex];
  println((Object[]) Serial.list());
  println(" Connecting to -> " + portName);
  myPort = new Serial(this, portName, 9600);
}

void draw()
{
int val;

  if ( myPort.available() >= 15)  // wait for the entire message to arrive
  {
    if( myPort.read() == HEADER) // is this the header
    {
      println("Message received:");
      // header found
      // get the integer containing the bit values
      val = readArduinoInt();
      // print the value of each bit
      for(int pin=2, bit=1; pin <= 13; pin++){
        print("digital pin " + pin + " = " );
        int isSet = (val & bit);
        if( isSet == 0) {
          println("0");
        }
        else{
          println("1");
        }
        bit = bit * 2; //shift the bit to the next higher binary place
      }
      println();
      // print the six analog values
      for(int i=0; i < 6; i ++){
        val = readArduinoInt();
        println("analog port " + i + "=" + val);
      }
      println("----");
```

```
      }
    }
  }

  // return integer value from bytes received from serial port (in low,high
  order)
  int readArduinoInt()
  {
    int val;        // Data received from the serial port

    val = myPort.read();            // read the least significant byte
    val = myPort.read() * 256 + val;  // add the most significant byte
    return val;
  }
```

> ### TIP
>
> Make sure that you set `portIndex` to correspond to the serial port that Arduino is connected to. You may need to run the sketch once, get an error, and review the list of serial ports in the Processing console at the bottom of the screen to determine which value you should use for `portIndex`.

The Processing code waits for 15 characters to arrive. If the first character is the header, it then calls the function named `readArduinoInt` to read two bytes and transform them back into an integer by doing the complementary mathematical operation that was performed by Arduino to get the individual bits representing the digital pins. The six integers are then representing the analog values. Note that the digital pins will all default to 1 (`HIGH`). This is because the pull-ups have been enabled on them using `INPUT_PULLUP`. This means that if you had a button connected to them, a value of 1 indicates the button is not pressed, while 0 indicates it is pressed. Recipe 2.4 has a discussion of this mode.

## See Also

To send Arduino values back to the computer or drive the pins from the computer (without making decisions on the board), consider using Firmata. The Firmata library and example sketches (File → Examples → Firmata) are

included in the Arduino software distribution, and a library is available to use in Processing. You load the Firmata code onto Arduino, control whether pins are inputs or outputs from the computer, and then set or read those pins.

# 4.10 Logging Arduino Data to a File on Your Computer

## Problem

You want to create a file containing information received over the serial port from Arduino. For example, you want to save the values of the digital and analog pins at regular intervals to a logfile.

## Solution

We covered sending information from Arduino to your computer in previous recipes. This solution uses the same Arduino code explained in Recipe 4.9. The Processing sketch that handles file logging is based on the Processing sketch also described in that recipe.

This Processing sketch creates a file (using the current date and time as the filename) in the same directory as the Processing sketch. Messages received from Arduino are added to the file. Pressing any key saves the file and exits the program:

```
/*
 * ReceiveMultipleFieldsBinaryToFile_P
 *
 * portIndex must be set to the port connected to the Arduino
 * based on ReceiveMultipleFieldsBinary, this version saves data to file
 * Press any key to stop logging and save file
 */

import processing.serial.*;
import java.util.*;
import java.text.*;

PrintWriter output;
```

```
DateFormat fnameFormat = new SimpleDateFormat("yyMMdd_HHmm");
DateFormat  timeFormat = new SimpleDateFormat("hh:mm:ss");
String fileName;

Serial myPort;          // Create object from Serial class
short portIndex = 0;    // select the com port, 0 is the first port
char HEADER = 'H';

void setup()
{
  size(200, 200);
  // Open whatever serial port is connected to Arduino.
  String portName = Serial.list()[portIndex];
  println((Object[]) Serial.list());
  println(" Connecting to -> " + portName);
  myPort = new Serial(this, portName, 9600);
  Date now = new Date();
  fileName = fnameFormat.format(now);
  output = createWriter(fileName + ".txt"); // save the file in the sketch
folder
}

void draw()
{
  int val;

  if ( myPort.available() >= 15)  // wait for the entire message to arrive
  {
    if( myPort.read() == HEADER) // is this the header
    {
      String timeString = timeFormat.format(new Date());
      println("Message received at " + timeString);
      output.println(timeString);

      // get the integer containing the bit values
      val = readArduinoInt();
      // print the value of each bit
      for (int pin=2, bit=1; pin <= 13; pin++){
        print("digital pin " + pin + " = " );
        output.print("digital pin " + pin + " = " );
        int isSet = (val & bit);
        if (isSet == 0){
          println("0");
          output.println("0");
        }
        else
        {
          println("1");
```

```
        output.println("1");
      }
      bit = bit * 2; // shift the bit
    }
    // print the six analog values
    for (int i=0; i < 6; i ++){
      val = readArduinoInt();
      println("analog port " + i + "=" + val);
      output.println("analog port " + i + "=" + val);
    }
    println("----");
    output.println("----");
   }
  }
}

void keyPressed() {
  output.flush(); // Writes the remaining data to the file
  output.close(); // Finishes the file
  exit(); // Stops the program
}

// return the integer value from bytes received on the serial port
// (in low,high order)
int readArduinoInt()
{
  int val;        // Data received from the serial port

  val = myPort.read();               // read the least significant byte
  val = myPort.read() * 256 + val; // add the most significant byte
  return val;
}
```

Don't forget that you need to set `portIndex` to the serial port connected to Arduino. If you choose the wrong value for `portIndex`, review the initial output of the Processing sketch where it prints the list of available serial ports and choose the correct one.

## Discussion

The base name for the logfile is formed using the `DateFormat` function in Processing:

```
DateFormat fnameFormat= new SimpleDateFormat("yyMMdd_HHmm");
```

The full filename is created with code that adds a directory and file
extension:

```
output = createWriter(fileName + ".txt");
```

To create the file and exit the sketch, you can press any key while the
Processing sketch main window is active. Don't press the Escape key
because it will terminate the sketch without saving the file. The file will be
created in the same directory as the Processing sketch (the sketch needs to
be saved at least once to ensure that the directory exists). To find this
directory, choose Sketch → Show Sketch Folder in Processing.

createWriter is the Processing function that opens the file; this creates an
object (a unit of runtime functionality) called output that handles the actual
file output. The text written to the file is the same as what is printed to the
console in Recipe 4.9, but you can format the file contents as required by
using the standard string-handling capabilities of Processing. For example,
the following variation on the draw routine produces a comma-separated
file that can be read by a spreadsheet or database. The rest of the Processing
sketch can be the same, although you may want to change the extension
from *.txt* to *.csv*:

```
void draw()
{
  int val;

  if (myPort.available() >= 15)  // wait for the entire message to arrive
  {
    if (myPort.read() == HEADER) // is this the header
    {
      String timeString = timeFormat.format(new Date());
      output.print(timeString);

      val = readArduinoInt();
      // print the value of each bit
      for (int pin=2, bit=1; pin <= 13; pin++){
        int isSet = (val & bit);
        if (isSet == 0){
```

```
        output.print(",0");
      }
      else
      {
        output.print(",1");
      }
      bit = bit * 2; // shift the bit
    }

    // output the six analog values delimited by a comma
    for (int i=0; i < 6; i ++){
      val = readArduinoInt();
      output.print("," + val);
    }
    output.println();
      }
    }
  }
```

## See Also

For more on `createWriter`, see the Processing page. Processing also includes the `Table` object for creating, manipulating, and saving CSV files.

# 4.11 Sending Data to More than One Serial Device

## Problem

You want to send data to a serial device such as a serial LCD, but you are already using the built-in serial-to-USB port to communicate with your computer.
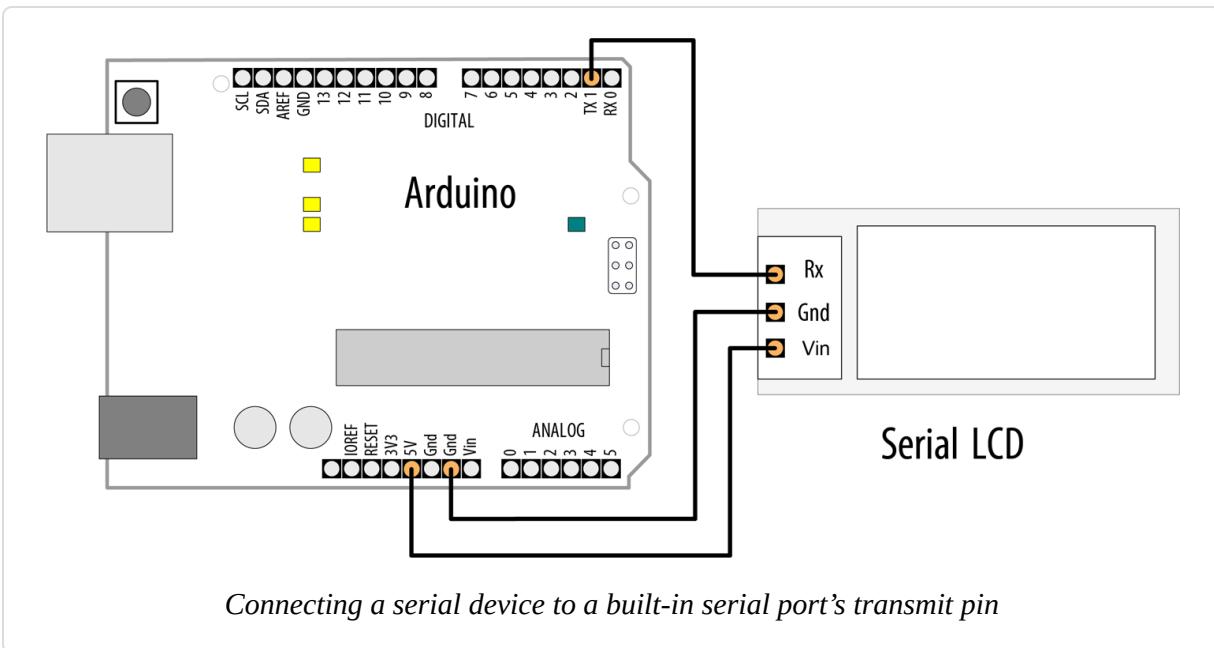
## Solution

On a board with more than one serial port (see the introduction for some suggestions) this is not a problem; first, you will need to wire the board to the serial device as shown in Figure 4-5. Next, you can initialize two serial ports and use `Serial` for the connection to your computer, and the other (usually `Serial1`) for the device:

```
void setup() {
  // initialize two serial ports on a board that supports this
  Serial.begin(9600);  // primary serial port
  Serial1.begin(9600); // Some boards have even more serial ports
}
```



*Connecting a serial device to a built-in serial port's transmit pin*
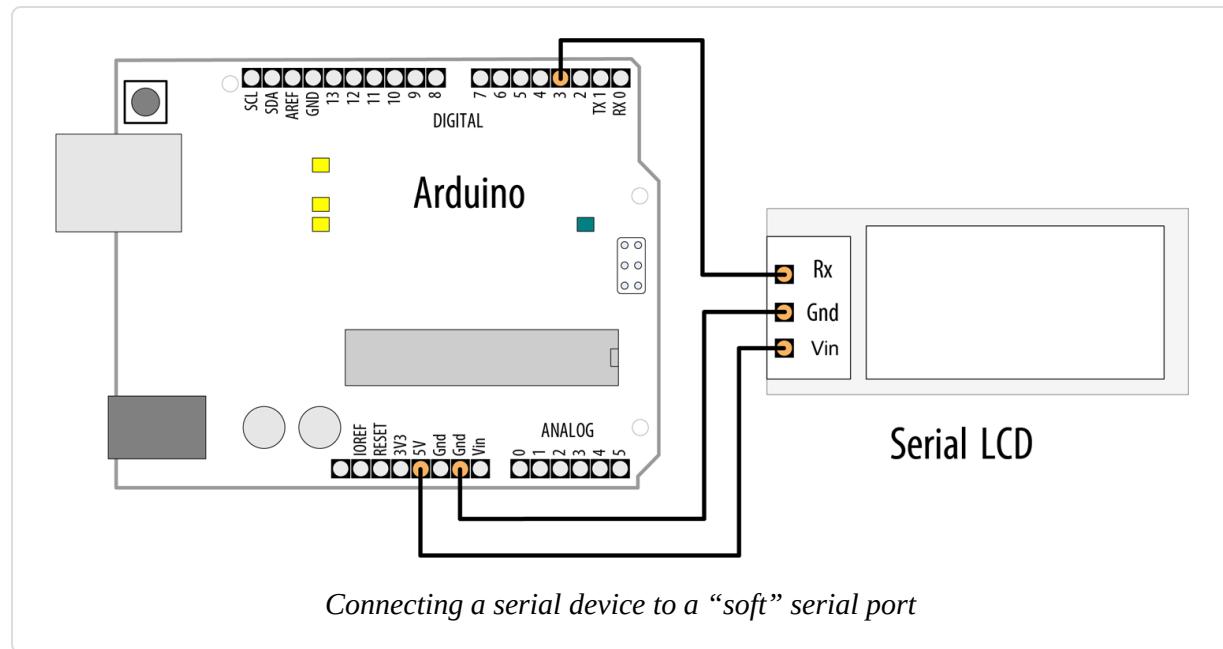
### WARNING

If you are using an Arduino or Arduino-compatible board that operates at 3.3V (such as a SAMD-based board), you can safely transmit to a device that uses 5V. But if you are using a board that operates at 5V (such as an Uno) with a device that uses 3.3V, you will eventually damage the device unless you incorporate a voltage divider into the circuit to bring the voltage down. See Recipes 4.13 and 5.11 for examples of voltage dividers.

On an ATmega328-based Arduino board (or similar), such as the Uno, which has only one hardware serial port, you will need to create an emulated or "soft" serial port using the SoftwareSerial library.

Select two available digital pins, one each for transmit and receive, and connect your serial device to them. Connect the device's transmit line to the receive pin and the receive line to the transmit pin. For scenarios where you are only sending data, such as when displaying characters on a Serial LCD display, you only need to wire up the transmit (TX) pin to the device's

receive (RX) pin, as shown in Figure 4-6, where we have selected pin 3 as the transmit pin.



*Connecting a serial device to a "soft" serial port*

In your sketch, create a `SoftwareSerial` object and tell it which pins you chose as your emulated serial port. In this example, we're creating an object named `serial_lcd`, which we instruct to use pins 2 and 3. Even though we're not going to be receiving any data from this serial connection, we need to specify a receive pin, so you shouldn't use pin 2 for anything else when you are using the SoftwareSerial port:

```
/*
 * SoftwareSerialOutput sketch
 * Output data to a software serial port
 */

#include <SoftwareSerial.h>

const int rxpin = 2;           // pin used to receive (not used in this
version)
const int txpin = 3;           // pin used to send to LCD
SoftwareSerial serial_lcd(rxpin, txpin); // new serial port on pins 2 and 3

void setup()
{
   Serial.begin(9600); // 9600 baud for the built-in serial port
```

```
   serial_lcd.begin(9600); //initialize the software serial port also for 9600
 }

 int number = 0;

 void loop()
 {
   serial_lcd.print("Number: ");   // send text to the LCD
   serial_lcd.println(number);     // print the number on the LCD
   Serial.print("Number: ");
   Serial.println(number);          // print the number on the PC console

   delay(500); // delay half second between numbers
   number++;   // to the next number
 }
```

To use the sketch with a built-in hardware serial port, connect the pins as shown in Figure 4-5, then remove these lines:

```
#include <SoftwareSerial.h>
const int rxpin = 2;
const int txpin = 3;
SoftwareSerial serial_lcd(rxpin, txpin);
```

Finally, add this line in their place: `#define serial_gps Serial1` (change `Serial1` as needed if you are using a different port).

> **NOTE**
>
> Some of the boards that support multiple hardware serial ports, such as the Leonardo, Mega, and Mega 2560, have restrictions on which pins you can use for SoftwareSerial receive (RX). Even though we aren't using the receive capability here, and even though you'd most likely use the hardware serial pins for `Serial1` on those boards (see Table 4-1), you should be aware that those boards do not support the RX capability on pin 2, so if you were to try to read from a software serial connection on one of those boards, you'd need to use a supported pin. Recipe 4.12 uses RX pins that will work on a wide variety of boards.

This sketch assumes that a serial LCD has been connected to pin 3 as shown in Figure 4-6, and that a serial console is connected to the built-in port. The loop will repeatedly display the same message on each:

```
Number: 0
Number: 1
...
```

## Discussion

The Arduino microcontroller contains at least one built-in hardware serial port. On the Arduino Uno, this port is connected to the USB serial connection, and is also wired to pins 0 (receive) and 1 (transmit), allowing you to connect a device such as an LCD serial display to the Arduino. The characters you transmit over the `Serial` object are displayed on the LCD.

> **NOTE**
>
> Although you can use a separate power supply for the serial device, you must connect the Arduino's ground pin to the device's pin, thus giving the Arduino and the serial device a *common ground*. In the Solution, we did this, but we also used the Arduino's 5V output to power the device.

In addition to the onboard USB serial connection, some boards support one or more direct serial connections. On these boards, pins 0 and 1 are typically tied to the `Serial1` object, which allows you to maintain a USB serial connection to your computer while you exchange data with the device on pins 0 and 1. Some boards support additional serial ports on a different set of pins (see Table 4-1 for a table of serial ports available on several boards). All the pins that support serial input and output, in addition to being general-purpose digital pins, are backed by universal asynchronous receiver-transmitter (UART) hardware that's built into the chip. This special piece of hardware is responsible for generating the series of precisely timed pulses its partner device sees as data and for interpreting the similar stream that it receives in return.

Although ARM SAMD-based boards (M0 and M4 boards) have two hardware-supported serial ports and the Mega has four such ports, the Arduino Uno and most similar boards based on the ATmega328 have only one. On the Uno and similar boards, you'll need a software library that

emulates the additional ports for projects that require connections to two or more serial devices. A "software serial" library effectively turns an arbitrary pair of digital I/O pins into a new serial port.

To build your software serial port, you select a pair of pins that will act as the port's transmit and receive lines in much the same way that a hardware serial port uses its assigned pins. In Figure 4-6, pins 3 and 2 are shown, but any available digital pins can be used, with some exceptions for certain boards. It's wise to avoid using 0 and 1, because these are already being driven by the built-in port.

The syntax for writing to the soft port is identical to that for the hardware port. In the example sketch, data is sent to both the "real" and emulated ports using `print()` and `println()`:

```
serial_lcd.print("Number: ");  // send text to the LCD
serial_lcd.println(number);    // print the number on the LCD
Serial.print("Number: ");
Serial.println(number);        // print the number on the PC console
```

> **TIP**
>
> If the combined text (`"Number:  "`) and the number itself are longer than the width of your LCD serial display, the output may be truncated or may scroll off the display. Many LCD character displays have two rows of 20 characters each.

## See Also

Nick Gammon maintains a transmit-only version of SoftwareSerial that lets you avoid the need to allocate a pin for receiving data when you don't need it.

# 4.12 Receiving Serial Data from More than One Serial Device
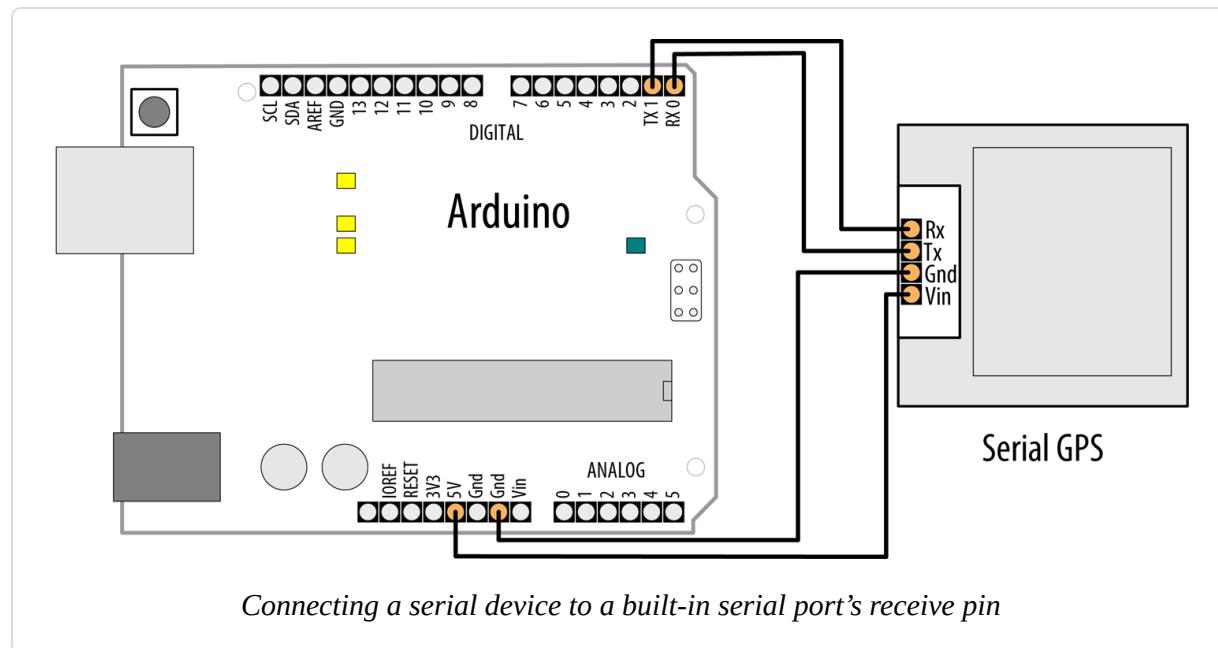
**Problem**

You want to receive data from a serial device such as a serial GPS, but you are already using the built-in serial-to-USB port to communicate with your computer.

## Solution

On a board with more than one serial port (see the introduction for some suggestions) this is not a problem; first, you will need to wire the board to the serial device as shown in Figure 4-7. Next, you can initialize two serial ports and use `Serial` for the connection to your computer, and the other (usually `Serial1`) for the device:

```
void setup() {
  // initialize two serial ports on a board that supports this
  Serial.begin(9600);  // primary serial port
  Serial1.begin(9600); // Some boards have even more serial ports
}
```



*Connecting a serial device to a built-in serial port's receive pin*

On the Arduino Uno and other boards based on the ATmega328, which has only one hardware serial port, you will need to create an emulated or "soft" serial port using the SoftwareSerial library. You will be limited to slower transfer speeds than with a built-in hardware serial port.

This problem is similar to the one in Recipe 4.11, and indeed the solution is much the same. If your Arduino's serial port is connected to the console and you want to attach a second serial device, you must create an emulated port using a software serial library. In this case, we will be receiving data from the emulated port instead of writing to it, but the basic solution is very similar.

Select two pins to use as your transmit and receive lines. This Solution uses pins 8 and 9 because some boards (such as the Arduino Leonardo) can only support SoftwareSerial receive on pins 8, 9, 10, 11, and 14. These also happen to be among the pins that the Arduino Mega and Mega 2560 support for SoftwareSerial receive. In practice, you would probably use the hardware serial available on pins 0 and 1 (`Serial1`) on these boards (the Mega boards have pins that support hardware serial as `Serial2` and `Serial3`). But we chose SoftwareSerial pins that would work on the widest possible range of boards in case you decide to test this code on one of them.

Connect your GPS as shown in Figure 4-8.

*Connecting a serial GPS device to a "soft" serial port*

As you did in Recipe 4.11, create a `SoftwareSerial` object in your sketch and tell it which pins to control. In the following example, we define a soft serial port called `serial_gps`, using pins 8 and 9 for receive and transmit, respectively. Even though we're not going to be sending any data to this serial device, we need to specify a transmit pin, so you shouldn't use pin 9 for anything else when you are using the software serial port:

---

### TIP

To use the following code with a built-in hardware serial port, connect the pins as shown in Figure 4-7, then remove these lines:

```
#include <SoftwareSerial.h>
const int rxpin = 8;
const int txpin = 9;
SoftwareSerial serial_gps(rxpin, txpin);
```

Finally, add this line in their place (change `Serial1` if you are using a different port):

```
#define serial_gps Serial1
```

---

```
/*
 * SoftwareSerialInput sketch
 * Read data from a software serial port
 */

#include <SoftwareSerial.h>
const int rxpin = 8;                    // pin used to receive from GPS
const int txpin = 9;                    // pin used to send to GPS
SoftwareSerial serial_gps(rxpin, txpin); // new serial port on these pins

void setup()
{
  Serial.begin(9600); // 9600 baud for the built-in serial port
  serial_gps.begin(9600); // initialize the port, most GPS devices
                          // use 9600 bits per second
}

void loop()
{
  if (serial_gps.available() > 0) // any character arrived yet?
  {
    char c = serial_gps.read();   // if so, read it from the GPS
    Serial.write(c);              // and echo it to the serial console
  }
}
```

This short sketch simply forwards all incoming data from the GPS to the Arduino Serial Monitor. If the GPS is functioning and your wiring is correct, you should see GPS data displayed on the Serial Monitor.

## Discussion

You initialize an emulated SoftwareSerial port by providing pin numbers for transmit and receive. The following code will set up the port to receive on pin 8 and send on pin 9:

```
const int rxpin = 8;                    // pin used to receive from GPS
const int txpin = 9;                    // pin used to send to GPS
SoftwareSerial serial_gps(rxpin, txpin); // new serial port on these pins
```

The syntax for reading an emulated port is very similar to that for reading from a built-in port. First check to make sure a character has arrived from

the GPS with `available()`, and then read it with `read()`.

It's important to remember that software serial ports consume time and resources. An emulated serial port must do everything that a hardware port does, using the same processor your sketch is trying to do "real work" with. Whenever a new character arrives, the processor must interrupt whatever it is doing to handle it. This can be time-consuming. At 4,800 baud, for example, it takes the Arduino about 2 ms to process a single character. While 2 ms may not sound like much, consider that if your connected device transmits 200 to 250 characters per second, your sketch is spending 40 to 50% of its time trying to keep up with the serial input. This leaves very little time to actually *process* all that data. The lesson is that if you have two serial devices, when possible connect the one with the higher bandwidth consumption to the built-in (hardware) port. If you must connect a high-bandwidth device to a software serial port, make sure the rest of your sketch's loop is very efficient.

## Receiving data from multiple SoftwareSerial ports

With the SoftwareSerial library included with Arduino, it is possible to create multiple "soft" serial ports in the same sketch. This is a useful way to control, say, several XBee radios (see Recipe 14.2) or serial displays in the same project. The caveat is that at any given time, only one of these ports can actively receive data. Reliable communication on a software port requires the processor's undivided attention. That's why SoftwareSerial can only actively communicate with one port at a given time.

It *is* possible to receive on two different SoftwareSerial ports in the same sketch. You just have to take some care that you aren't trying to receive from both simultaneously. There are many successful designs which, say, monitor a serial GPS device for a while, then later accept input from an XBee. The key is to alternate slowly between them, switching to a second device only when a transmission from the first is complete.

For example, in the sketch that follows, an XBee module is connected to the Arduino. The module is receiving commands from a remote device connected to another XBee module. The sketch listens to the command

stream through the "xbee" port until it receives the signal to begin gathering data from a GPS module attached to a second SoftwareSerial port. The sketch then monitors the GPS for 10 seconds—hopefully long enough to establish a "fix"—before returning to the XBee.

In a system with multiple "soft" ports, only one is actively receiving data. By default, the "active" port is the one for which `begin()` has been called most recently. However, you can change which port is active by calling its `listen()` method. `listen()` instructs the SoftwareSerial system to stop receiving data on one port and begin listening for data on another.

> **NOTE**
>
> Because the following example is only receiving data, you can choose any pins for `txpin1` and `txpin2`. If you need to use pins 9 and 11 for something else, you can change `txpin1/2` to another pin. We recommend against changing it to a nonexistent pin number, because that could lead to some unusual behavior.

The following code fragment illustrates how you might design a sketch to read first from one port and then another:

```
/*
 * MultiRX sketch
 * Receive data from two software serial ports
 */
#include <SoftwareSerial.h>
const int rxpin1 = 8;
const int txpin1 = 9;
const int rxpin2 = 10;
const int txpin2 = 11;

SoftwareSerial gps(rxpin1, txpin1);   // gps TX pin connected to Arduino pin 9
SoftwareSerial xbee(rxpin2, txpin2);  // xbee TX pin connected to Arduino pin 10

void setup()
{
  Serial.begin(9600);
  xbee.begin(9600);
  gps.begin(9600);
```

```
    xbee.listen(); // Set "xbee" to be the active device
  }

  void loop()
  {
    if (xbee.available() > 0) // xbee is active. Any characters available?
    {
      if (xbee.read() == 'y') // if xbee received a 'y' character?
      {
        gps.listen(); // now start listening to the gps device

        unsigned long start = millis(); // begin listening to the GPS
        while (start + 100000 > millis()) // listen for 10 seconds
        {
          if (gps.available() > 0) // now gps device is active
          {
            char c = gps.read();
            Serial.write(c);              // echo it to the serial console
          }
        }
        xbee.listen(); // After 10 seconds, go back to listening to the xbee
      }
    }
  }
```

This sketch is designed to treat the XBee radio as the active port until it receives a y character, at which point the GPS becomes the active listening device. After processing GPS data for 10 seconds, the sketch resumes listening on the XBee port. Data that arrives on an inactive port is simply discarded.

Note that the "active port" restriction only applies to multiple *soft* ports. If your design really must receive data from more than one serial device simultaneously, consider using a board, such as the Teensy, that supports several serial ports (the Teensy 4.0 supports seven in total). Table 4-1 shows the pins used for serial ports on various Arduino and Arduino-compatible boards.

## See Also

If you would like to go further with a GPS and parse the messages that you receive, see Recipe 6.14. For an alternative to SoftwareSerial that can

support multiple devices more robustly, see the AltSoftSerial library.

# 4.13 Using Arduino with the Raspberry Pi

## Problem

You want to use Arduino capabilities together with the processing power of a single-board Linux computer such as Raspberry Pi. For example, you want to send commands to Arduino from a script running on the Pi.
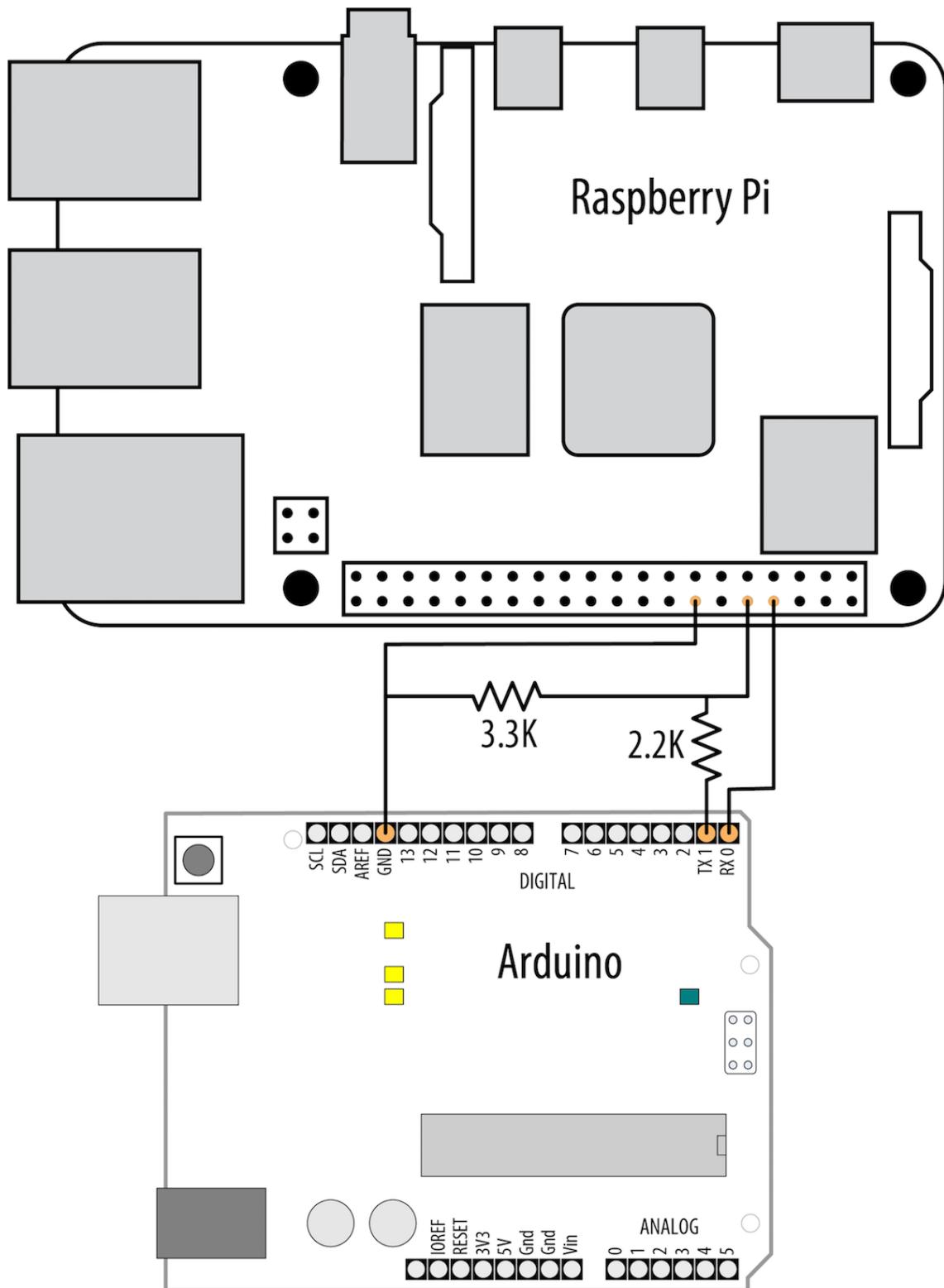
## Solution

Arduino can monitor and respond to serial commands from the Raspberry Pi. The code here controls Arduino LEDs from Python scripts running on Pi.

> **NOTE**
>
> It is also possible to connect Arduino to a Raspberry Pi using one of the Raspberry Pi's USB ports. In fact, you can even run the Arduino IDE on the Raspberry Pi. Download one of the ARM versions. At the time of this writing, the Raspberry Pi operating system, Raspbian, operated in 32-bit mode, so you'd choose the 32-bit version unless you're running a 64-bit operating system.

Connect the Arduino serial receive pin (pin 0 marked RX on the board) to pin 8 on the Pi's header pins. Connect Arduino TX pin 1 to the Pi GPIO pin 10. The Arduino ground (GND) pin connects to any of the Pi ground pins (pin 14 is used in Figure 4-9).

*Arduino board connected to Raspberry Pi*

Here is an Arduino sketch that monitors serial messages from the Pi. Upload this to the Arduino board:

```
/*
 * ArduinoPi sketch
 * Pi control Arduino pins using serial messages
 * format is: Pn=state
 * where 'P' is header character, n is pin number, state is 0 or 1
 * example: P13=1  turns on pin 13
 */

// Replace Serial with Serial1 on boards with an additional serial port
#define mySerial Serial

void setup()
{
  mySerial.begin(9600); // Initialize serial port to send/receive at 9600 baud
}

void loop()
{
  if (mySerial.available()) // Check whether at least one character is
available
  {
    char ch = mySerial.read();
    int pin = -1;
    if( ch == 'P') // is this the beginning of a message to set a pin?
    {
      pin = mySerial.parseInt();   // get the pin number
    }
    else if (ch == 'B') // Message to set LED_BUILTIN
    {
      pin = LED_BUILTIN;
    }

    if( pin > 1) { // 0 and 1 are usually serial pins (leave them alone)
```

```
        int state = mySerial.parseInt(); // 0 is off, 1 is on
        pinMode(pin, OUTPUT);
        digitalWrite(pin, state);
      }

    }
  }
```

Save the following Python script as *blinkArduino.py* on the Pi, and run it
with `python blinkArduino.py`. The script will blink the onboard LED on
the Arduino board. You must have the python-serial library installed before
you run it. You can install it on the Raspberry Pi with `sudo apt-get
install python-serial`:

```python
#!/usr/bin/env python

import serial
from time import import sleep

ser = serial.Serial('/dev/serial0', 9600)
ser.write('P13=1')
sleep(1)
ser.write('P13=0')
```

When the script is run, an LED on pin 13 should turn on for one second and
then go off.

## Discussion

The Arduino sketch detects the start of a message when the character *P* is
received. The Arduino `parseInt` function is used to extract the desired pin
number and pin state. Sending *P13=1* will turn on the LED on pin 13.
*P13=0* will turn the LED off. More information on serial messages and
`parseInt` can be found in Chapter 4. Many Arduino and Arduino-
compatible boards use some pin other than 13, so to save you the trouble of
having to look that up, the Arduino sketch will use the `LED_BUILTIN`
constant as the pin number when you send it a message like B=1 (no pin
number required).

The Python script sends the appropriate messages to turn the LED on and then off.

If your board's built-in LED is not on pin 13, use this version, which uses the B command to toggle whichever LED is assigned to LED_BUILTIN:

```python
#!/usr/bin/env python

import serial
from time import sleep

ser = serial.Serial('/dev/serial0', 9600)
ser.write('B=1')
sleep(1)
ser.write('B=0')
```

Raspberry Pi pins are not 5-volt tolerant, so you must use the voltage divider shown in the diagram if you are connecting a 5V Arduino-compatible board to the Pi. If you are using a 3.3V Arduino, it's generally safe to omit the voltage divider, but the voltage divider won't hurt it. See Recipe 5.11 for more details on voltage dividers.

The messages sent in this recipe are very simple but can be expanded to enable the Pi to control almost any Arduino function and for Arduino to send information back to the Pi. See Recipe 4.0 for more on getting Arduino working with a computer over the serial link.

Details on Python and the Pi can be found online and in books such as *Raspberry Pi Cookbook,* Third Edition, by Simon Monk.

### WHY ARDUINO CAN SEEM MUCH FASTER THAN A RASPBERRY PI

The Raspberry Pi is a remarkable piece of technology. It has the ability to run a complex operating system such as Linux or Windows 10, and that is something that a standard Arduino board cannot do. This capability can be essential if you want to use complex drivers such as those for speech recognition, visual pattern matching, and countless other capabilities supported on Linux and Windows. However, if your application requires precise high-speed software control of input or output pins, then the Arduino can, in some situations, respond faster than the Pi.

This is because the pin control on the Pi is handled through layers of software designed to isolate the hardware from the operating system, and one of those layers is the programming language you are using. This overhead slows down the rate at which pins can be controlled. And because the operating system is constantly interrupting each task to support other tasks, there can be a small but inconsistent delay in the intervals given to the task-controlling pins.

A basic Arduino Uno board can achieve a consistent rate of 8 MHz (see Recipe 18.11). Joonas Pihlajamaa performed a set of benchmarks on Raspberry PI GPIO speed and reached only 70 KHz on the Raspberry Pi using Python and RPi.GPIO. However, using native libraries and the C programming language, it's possible to beat even the Uno. Pihlajamaa reached 22 MHz with that approach.

More recent Arduino and Arduino-compatible hardware is even faster. For example, the Teensy 3 can achieve a toggle rate of 48 MHz.

# Simple Digital and Analog Input

## 5.0 Introduction
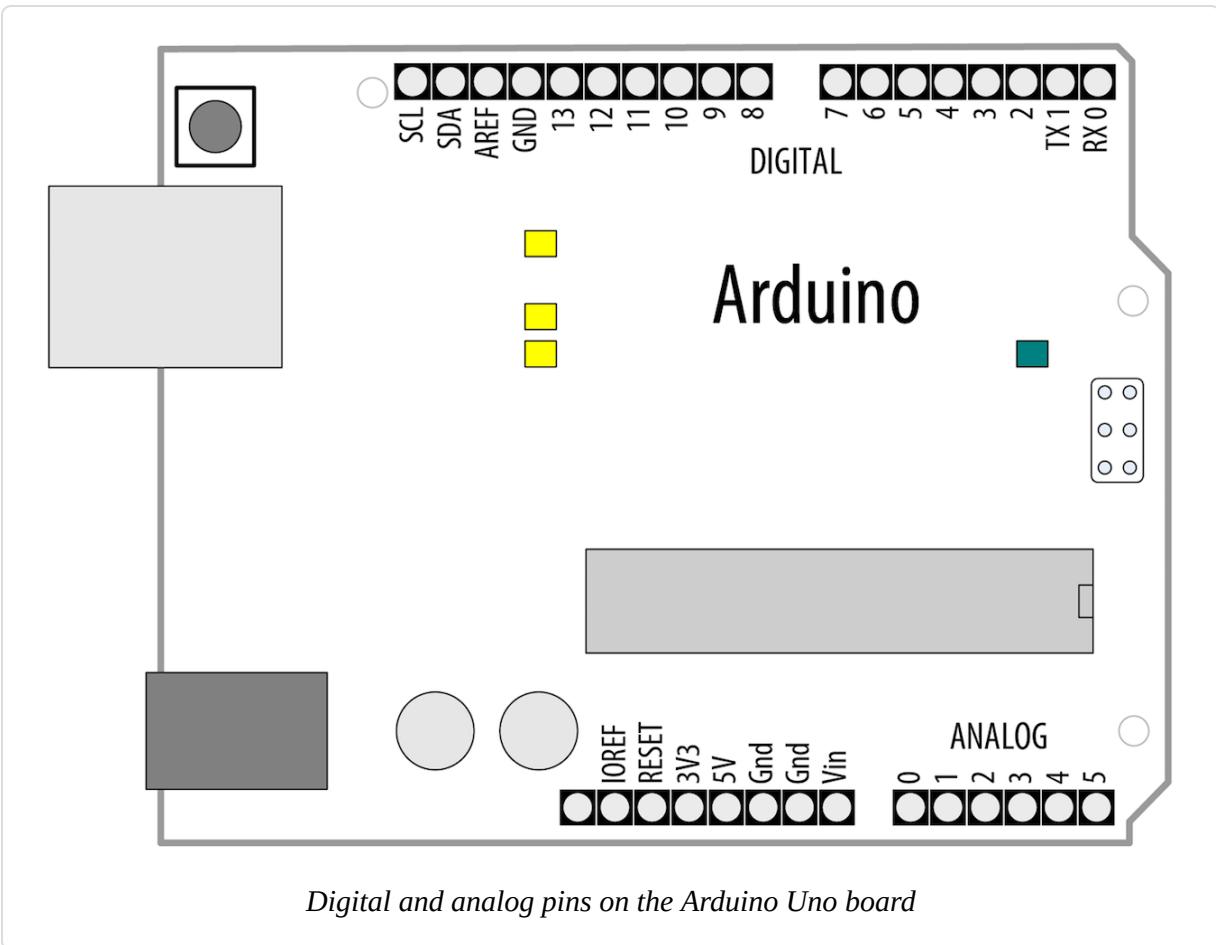
The Arduino's ability to sense digital and analog inputs allows it to respond to you and to the world around you. This chapter introduces techniques you can use to monitor and respond to these inputs: detect when a switch is pressed, read input from a numeric keypad, and read a range of voltage values.

This chapter covers the Arduino pins that can sense *digital* and *analog* inputs. Digital input pins sense the presence and absence of voltage on a pin. Analog input pins measure a range of voltages on a pin.

Figure 5-1 shows the arrangement of pins on the Arduino Uno. This pin arrangement is used by many Arduino-compatible boards, including the Adafruit Metro line and SparkFun. See this list of the official boards, which links to connection information for each. If your board is not on that list, check your board supplier's website for connection information.

The Arduino function to detect digital input is `digitalRead`, and it tells your sketch if a voltage on a pin is `HIGH` or `LOW`. `HIGH` is between 3 and 5 volts for boards such as the Uno (between 2 and 3.3 volts on ARM-based boards and any other 3.3V boards), `LOW` is 0 volts. The Arduino function to configure a pin for reading input is `pinMode(pin, INPUT)`.

On a board with the Uno-style pin layout (including the Arduino Leonardo, several of the Adafruit Metro boards, and SparkFun RedBoard), there are 14 digital pins (numbered 0 to 13) as shown at the top of Figure 5-1. On the Uno and 100% compatible boards (typically boards based on the ATmega328), pins 0 and 1 (marked RX and TX) are used for the USB serial connection and should be avoided for other uses. See Chapter 4 for more details on serial connections.

*Digital and analog pins on the Arduino Uno board*

Arduino has logical names that can be used to reference many of the pins. The constants in Table 5-1 can be used in all functions that expect a pin number. It is very likely that you will encounter example code that uses the actual pin numbers. But given the wide diversity of Arduino and Arduino-compatible boards, you should avoid using the numeric pin number and use these constants instead. For example, on the Arduino Uno, A0 is pin 14, but it's 15 on the MKR WiFi 1010, and 54 on the Arduino Mega.

*Pin constants for Uno-style layout*
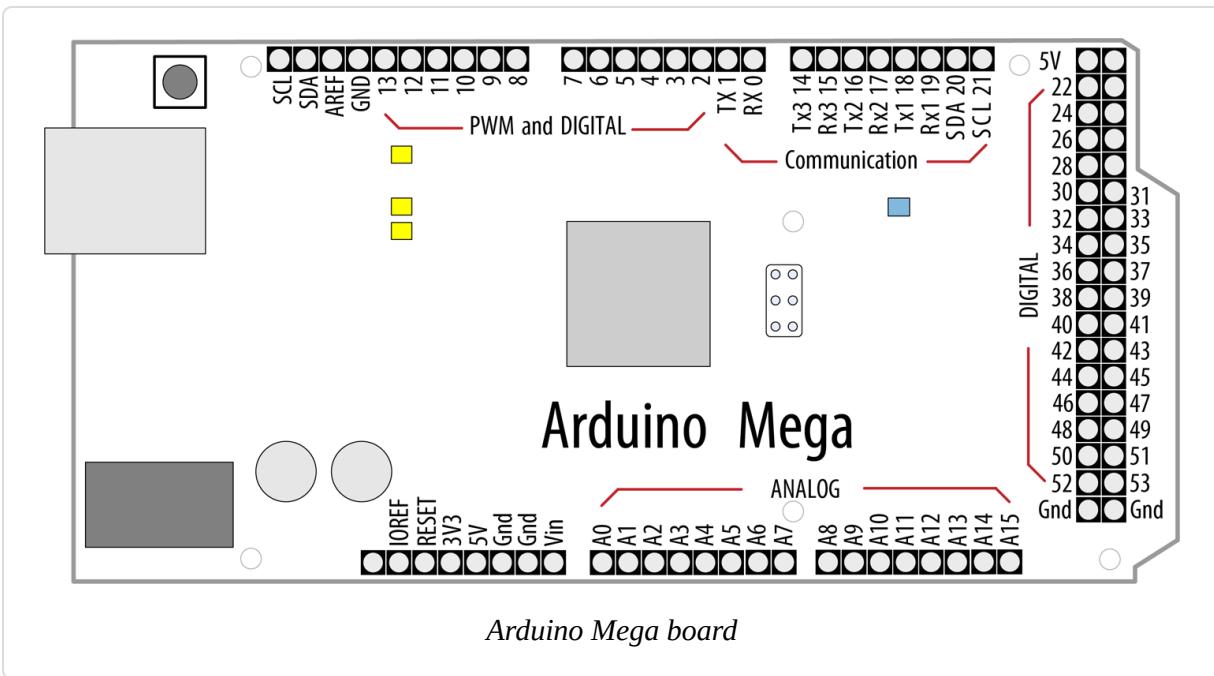
| Constant | Pin | Constant | Pin |
|---|---|---|---|
| A0 | Analog input 0 | LED_BUILTIN | Onboard LED |
| A1 | Analog input 1 | SDA | I2C Data |
| A2 | Analog input | SCL | I2C Clock |

| A3 | Analog input | SS | SPI Select |
|----|--------------|------|------------|
| A4 | Analog input | MOSI | SPI Input |
| A5 | Analog input | MISO | SPI Output |
|    |              | SCK | SPI Clock |

> **TIP**
>
> If you need more digital pins, you can use the analog pins as digital pins (when you do this, you can refer to them through their symbolic names, for example with `pinMode(A0, INPUT);`).

Boards such as the Mega and Due have many more digital and analog pins. Digital pins 0 through 13 and analog pins 0 through 5 are located in the same place as on the standard board, so that hardware shields designed for the standard board can fit. As with the standard board, you can use analog pins as digital pins, but with the Mega, analog numbering goes from A0 through A15. Figure 5-2 shows the Mega pin layout.



*Arduino Mega board*

The Uno, Leonardo, and many other boards have an LED connected to pin 13, but the pin number will be different on other boards, so you should

always use the `LED_BUILTIN` constant to refer to the built-in LED. If your board does not have a built-in LED, skip ahead to Recipe 7.1 if you need help connecting an LED to a digital pin. You'll also need to change the output pin from `LED_BUILTIN` to the pin number you are using.

Recipes covering digital input sometimes use internal or external resistors to force the input pin to stay in a known state when the input is not engaged. Without such a resistor, the pin's value would be in a state known as *floating*, and `digitalRead` might return `HIGH` at first, but then would return `LOW` milliseconds later, regardless of whether the input is engaged (such as when a button is pressed). A *pull-up* resistor is so named because the voltage is "pulled up" to the logic level (5V or 3.3V) of the board. When you press the button in a pull-up configuration, `digitalRead` will return `LOW`. At all other times, it returns `HIGH` because the pull-up resistor is keeping it high. A *pull-down* resistor pulls the pin down to 0 volts. In this configuration, `digitalRead` will return `HIGH` when the button is pressed. Although 10K ohms is a commonly used value for a pull-up or pull-down resistor, anything between 4.7K and 20K or more will work; see Appendix A for more information about the components used in this chapter.

**WORKING WITH ELECTRONIC COMPONENTS**

This is the first of many chapters to come that cover electrical connections to Arduino. If you don't have an electronics background, you may want to look through Appendix A on electronic components, Appendix B on schematic diagrams and datasheets, Appendix C on building and connecting circuits, and Appendix E on hardware troubleshooting. In addition, many good introductory tutorials are available. Two that are particularly relevant to Arduino are *Getting Started with Arduino* by Massimo Banzi and Michael Shiloh (Make Community) and *Making Things Talk* by Tom Igoe (Make Community). Other books offering a background on electronics topics covered in this and the following chapters include *Getting Started in Electronics* by Forrest M. Mims, III (Master Publishing), *Make: Electronics* by Charles Platt (Make Community), and *Physical Computing* by Tom Igoe (Cengage).

If wiring components to your Arduino is new to you, be careful about how you connect and power the things you attach. The Arduino Uno uses a robust controller chip that can take a fair amount of abuse, but you can damage the chip if you connect the wrong voltages or short-circuit an output pin. 32-bit Arduino and compatible boards are generally a bit more fragile. Arduino controller chips on boards such as the Uno are powered by 5 volts, and you must not connect external power to Arduino pins with a higher voltage than this. But most of the newer Arduino boards and compatibles can tolerate a maximum of 3.3 volts. See the online documentation for your board to find the maximum pin voltage.

Some Arduino boards have the main chip in a socket that can be removed and replaced, so you don't need to replace the whole board if you damage the chip. If you are new to electronics and want to experiment, boards with replaceable microcontrollers such as the Uno are a good choice. The Arduino Uno Rev3 SMD (Surface Mount Device) has a soldered-on microcontroller that is not replaceable.

Arduino boards have internal pull-up resistors that you can activate when you use the `INPUT_PULLUP` mode with `pinMode`, as shown in Recipe 5.2. This eliminates the need for external pull-up resistors.

Unlike a digital value, which is only on or off, analog values are continuously variable. The volume setting of a device is a good example; it is not just on or off, but it can have a range of values in between. Many sensors provide information by varying the voltage to correspond to the sensor measurement. Arduino code uses a function called `analogRead` to get a value proportional to the voltage it sees on one of its analog pins. The value will be 0 if there are 0 volts on the pin and 1,023 for 5 volts (or 3.3 volts on a 3.3-volt board). The value in between will be proportional to the voltage on the pin, so 2.5 volts (half of 5 volts) will result in a value of roughly 511 (half of 1,023). You can see the six analog input pins (marked 0 to 5) at the bottom of Figure 5-1 (these pins can also be used as digital pins

if they are not needed for analog). Some of the analog recipes use a *potentiometer* (*pot* for short, also called a *variable resistor*) to vary the voltage on a pin. When choosing a potentiometer, a value of 10K is the best option for connecting to analog pins.

Although most of the circuits in this chapter are relatively easy to connect, you will want to consider getting a solderless breadboard to simplify your wiring to external components. A full-length breadboard has 830 *tie points* (the holes you insert the wire into) and two power bus rows per side. These include: Jameco part number 20723, Adafruit Industries part number 239, Digi-Key 438-1045-ND, and SparkFun PRT-12615. Half-length breadboards with 400 tie points are popular in part because they are about the same size as the Arduino Uno.

Another handy item is an inexpensive multimeter. Almost any will do, as long as it can measure voltage and resistance. Continuity checking and current measurement are nice additional features to have. (The Jameco 220759, Adafruit 2034, Digi-Key 1742-1135-ND, and SparkFun TOL-12966 offer these features.)

# 5.1 Using a Switch

## Problem

You want your sketch to respond to the closing of an electrical contact; for example, a pushbutton or other switch or an external device that makes an electrical connection.

## Solution

Use `digitalRead` to determine the state of a switch connected to an Arduino digital pin set as input. The following code lights an LED when a switch is pressed (Figure 5-3 shows how it should be wired up):

```
/*
  Pushbutton sketch
  a switch connected to pin 2 lights the built-in LED
```
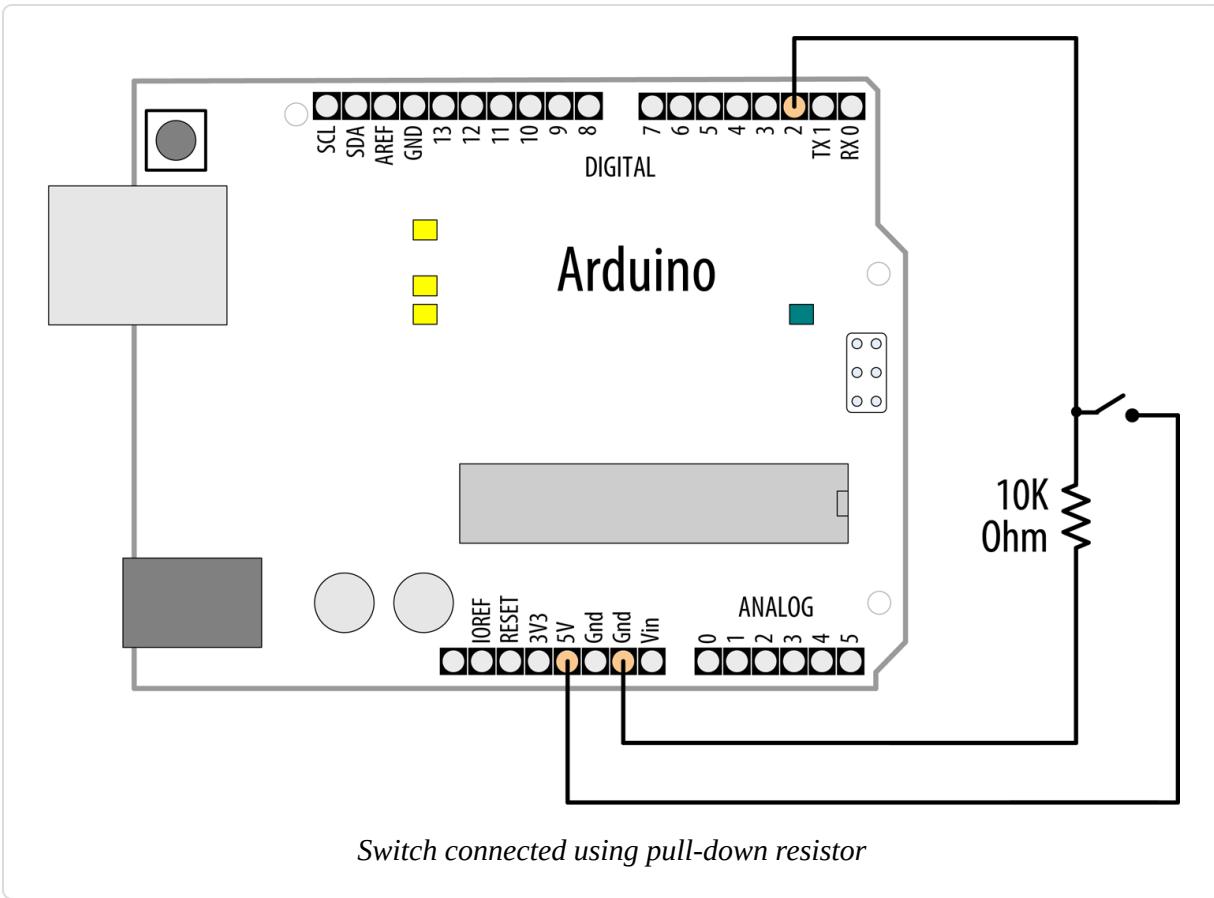
```
*/

const int inputPin = 2;            // choose the input pin (for a pushbutton)

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);    // declare LED as output
  pinMode(inputPin, INPUT);        // declare pushbutton as input
}

void loop(){
  int val = digitalRead(inputPin);  // read input value
  if (val == HIGH)                  // check if the input is HIGH
  {
    digitalWrite(LED_BUILTIN, HIGH); // turn LED on if switch is pressed
  }
  else
  {
    digitalWrite(LED_BUILTIN, LOW);  // turn LED off
  }
}
```



*Switch connected using pull-down resistor*

## Discussion

The `setup` function configures the LED pin as `OUTPUT` and the switch pin as `INPUT`.

The `digitalRead` function reads the voltage on the input pin (`inputPin`), and it returns a value of `HIGH` if the voltage is high (5 volts on most 8-bit boards, 3.3 volts on most 32-bit boards) and `LOW` if the voltage is low (0 volts). Any voltage between 3 and 5 volts (or between 2 and 3.3 volts on 3.3V boards) is considered `HIGH`, and less than this is treated as `LOW`. If the pin is left unconnected (known as *floating*), the value returned from `digitalRead` is indeterminate (it may be `HIGH` or `LOW`, and it cannot be reliably used). The resistor shown in Figure 5-3 ensures that the voltage on the pin will be low when the switch is not pressed, because the resistor "pulls down" the voltage to ground (labeled GND on most boards), which is 0 volts. When the switch is pushed, a connection is made between the pin and +5 volts, so the value on the pin returned by `digitalRead` changes from `LOW` to `HIGH`.

In this example, the value from `digitalRead` is stored in the variable `val`. This will be `HIGH` if the button is pressed, `LOW` otherwise.

**NOTE**

The switch used in this example (and almost everywhere else in this book) makes electrical contact when pressed and breaks contact when not pressed. These switches are called Normally Open (NO). The other kind of momentary switch is called Normally Closed (NC).

The output pin connected to the LED is turned on when you set `val` to `HIGH`, illuminating the LED.

Although Arduino sets all digital pins as inputs by default, it is a good practice to set this explicitly in your sketch to remind yourself about the pins you are using.

You may see similar code that uses `true` instead of `HIGH`; these can be used interchangeably (they are also sometimes represented as `1`). Likewise, `false` is the same as `LOW` and `0`. Use the form that best expresses the meaning of the logic in your application.

Almost any switch can be used, although the ones called *momentary tactile switches* are popular because they are inexpensive and can plug directly into a breadboard.

Here is another way to implement the logic in the preceding sketch:
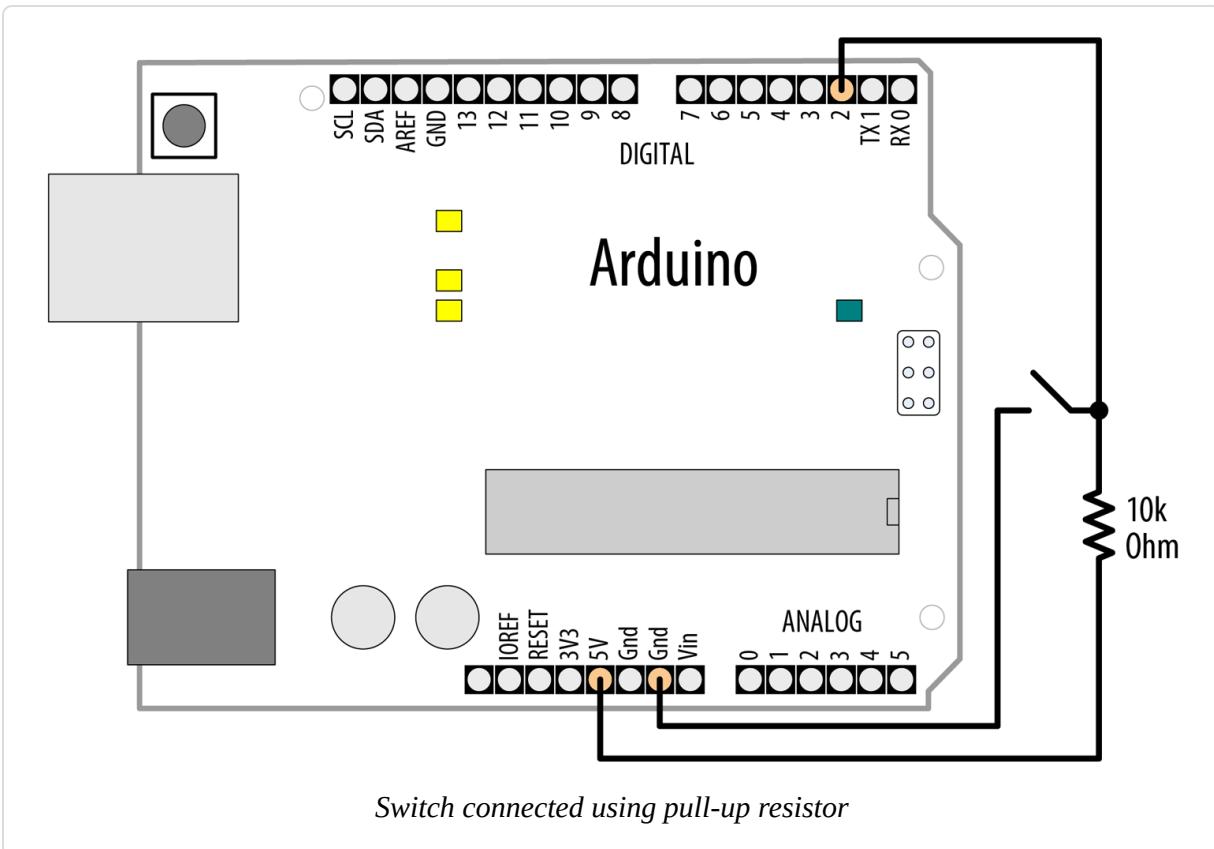
```
void loop()
{
```

```
    // turn LED ON if input pin is HIGH, else turn OFF
    digitalWrite(LED_BUILTIN, digitalRead(inputPin));
  }
```

This doesn't store the button state into a variable. Instead, it sets the LED
on or off directly from the value obtained from `digitalRead`. It is a handy
shortcut, but if you find it overly terse, there is no practical difference in
performance, so pick whichever form you find easier to understand.

The pull-up code is similar to the pull-down version, but the logic is
reversed: the value on the pin goes LOW when the button is pressed (see
Figure 5-4 for a schematic diagram of this). It may help to think of this as
pressing the switch DOWN, causing the pin to go LOW:

```
void loop()
{
  int val = digitalRead(inputPin);  // read input value
  if (val == HIGH)                   // check if the input is HIGH
  {
    digitalWrite(LED_BUILTIN, LOW);  // turn LED OFF
  }
  else
  {
    digitalWrite(LED_BUILTIN, HIGH); // turn LED ON
  }
}
```

*Switch connected using pull-up resistor*

## See Also

The Arduino references for:

- `digitalRead`
- `digitalWrite`
- `pinMode`
- constants (HIGH, LOW, etc.)

This Arduino tutorial on digital pins

# 5.2 Using a Switch Without External Resistors
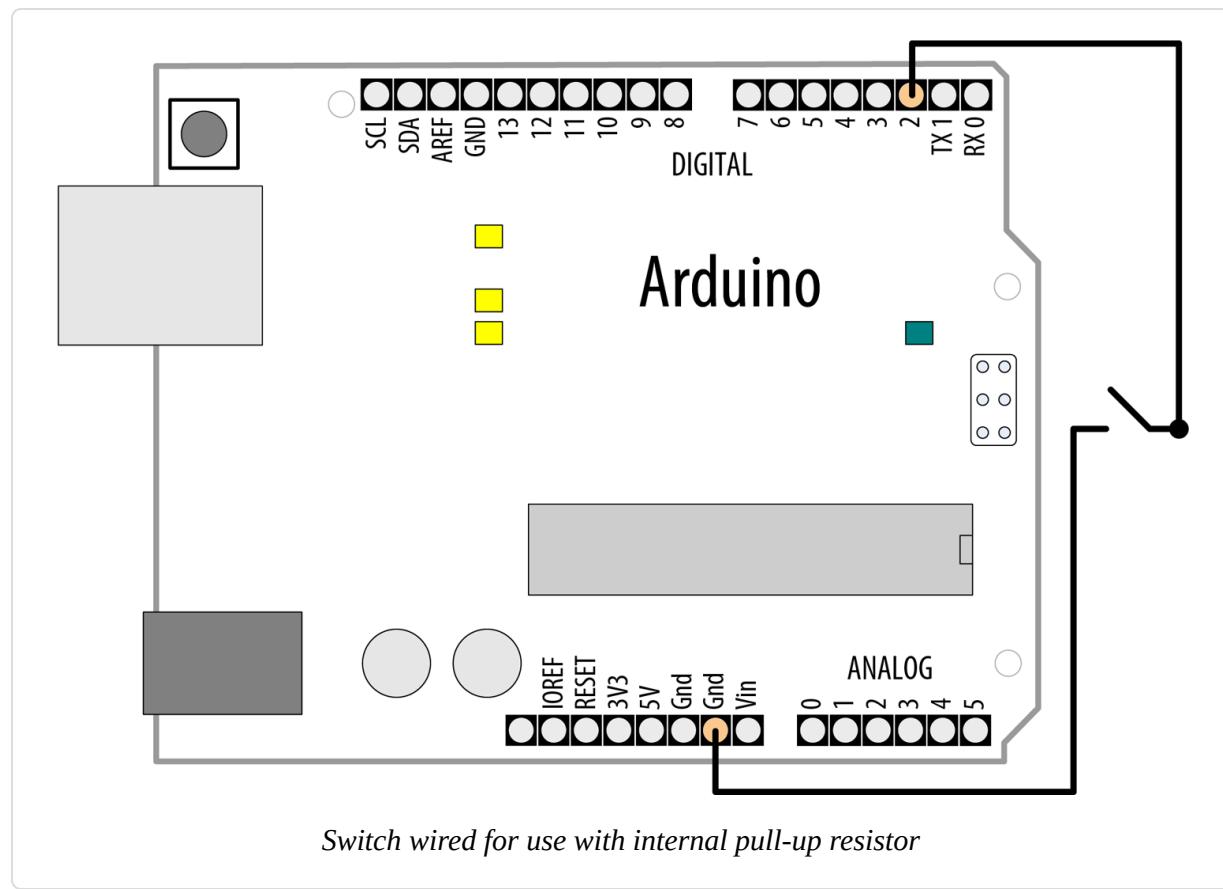
**Problem**

You want to simplify your wiring by eliminating external pull-up resistors when connecting switches.

## Solution

As explained in Recipe 5.1, digital inputs must have a resistor to hold the pin to a known value when the switch is not pressed. Arduino has internal pull-up resistors that can be enabled by using the `INPUT_PULLUP` mode with `pinMode`.

For this example, the switch is wired as shown in Figure 5-5. This is almost exactly the same as Figure 5-4, but without an external resistor.



*Switch wired for use with internal pull-up resistor*

The switch is only connected between pin 2 and ground (labeled GND on most boards). Ground is at 0 volts by definition:

```
/*
  Input pullup sketch
```

```
    a switch connected to pin 2 lights the built-in LED
 */

const int inputPin = 2; // input pin for the switch

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(inputPin, INPUT_PULLUP); // use internal pull-up on inputPin
}

void loop(){
  int val = digitalRead(inputPin);   // read input value
  if (val == HIGH)                   // check if the input is HIGH
  {
    digitalWrite(LED_BUILTIN, LOW);  // turn LED off
  }
  else
  {
    digitalWrite(LED_BUILTIN, HIGH); // turn LED on
  }
}
```

---

**TIP**

There is more than one ground pin on an Arduino board; they are all connected together, so pick whichever is convenient.

---

## Discussion

When you use pull-up resistors, the logic is reversed: the value of `digitalRead` will be `LOW` when the button is pushed, and `HIGH` when it is not. The internal pull-up resistors are 20K ohms or more (between 20K and 50K). This is suitable for most applications, but some devices may require lower-value resistors—see the datasheet for external devices you want to connect to Arduino to see whether the internal pull-ups are suitable or not.

# 5.3 Reliably Detect (Debounce) When a Switch Is Pressed

## Problem

You want to avoid false readings due to *contact bounce* (contact bounce produces spurious signals at the moment the switch contacts close or open). The process of eliminating spurious readings is called *debouncing*.

## Solution

There are many ways to solve this problem; here is one using the wiring shown in Figure 5-3 from Recipe 5.1:

```
/*
 * Debounce sketch
 * a switch connected to pin 2 lights the built-in LED
 * debounce logic prevents misreading of the switch state
 */

const int inputPin = 2;         // the number of the input pin
const int debounceDelay = 10;   // iterations to wait until pin is stable
bool last_button_state = LOW;   // Last state of the button
int ledState = LOW;             // On or off (HIGH or LOW)

// debounce returns the state when the switch is stable
bool debounce(int pin)
{
  bool state;
  bool previousState;
```

```
    previousState = digitalRead(pin);         // store switch state
    for(int counter=0; counter < debounceDelay; counter++)
    {
        delay(1);                     // wait for 1 ms
        state = digitalRead(pin);   // read the pin
        if( state != previousState)
        {
            counter = 0; // reset the counter if the state changes
            previousState = state;   // and save the current state
        }
    }
    // here when the switch state has been stable longer than the debounce
period
    return state;
}

void setup()
{
  pinMode(inputPin, INPUT);
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop()
{
  bool button_state = debounce(inputPin);

  // If the button state changed and the button was pressed
  if (last_button_state != button_state && button_state == HIGH) {
    // Toggle the LED
    ledState = !ledState;
    digitalWrite(LED_BUILTIN, ledState);
  }
  last_button_state = button_state;
}
```

When you want to reliably check for a button press, you should call the
debounce function with the pin number of the switch you want to
debounce; the function returns HIGH if the switch is pressed and stable. It
returns LOW if it is not pressed or not yet stable.

## Discussion

The `debounce` method checks to see if it gets the same reading from the switch after a delay that needs to be long enough for the switch contacts to stop bouncing. You may require more iterations for "bouncier" switches (some switches can require as much as 50 ms or more). The function works by repeatedly checking the state of the switch as many times as defined in the `debounceDelay` time. If the switch remains stable throughout that time period, the state of the switch will be returned (`HIGH` if pressed and `LOW` if not). In a boolean context such as an `if` statement, `HIGH` evaluates to true and `LOW` to false. If the switch state changes within the debounce period, the counter is reset so that the checks start over until the switch state does not change within the debounce time.

> **NOTE**
>
> Although `debounceDelay` is defined as 10, and there is a delay of 1 ms per iteration, the actual length of the delay may be higher than 10 ms for two reasons. First (depending on the speed of your board), all the other operations in the loop take measurable time to complete, which adds to the delay. Second, if the state of the switch changes within the loop, the counter is reset to 0.

In the `loop` function, this sketch repeatedly checks the state of the button. If the button state changes (from `HIGH` to `LOW` or vice versa), and if the button state is `HIGH` (pressed), the sketch toggles the state of the LED. So if you press the button once, the LED is turned on. Press it a second time, and the LED will turn off.

If your wiring uses pull-up resistors instead of pull-down resistors (see Recipe 5.2) you need to invert the value returned from the `debounce` function, because the state goes `LOW` when the switch is pressed using pull-ups, but the function should return `true` (`true` is equivalent to `HIGH`) when the switch is pressed. The debounce code using pull-ups is as follows; only the last four lines (highlighted) are changed from the previous version:

```
bool debounce(int pin)
{
  bool state;
```

```
bool previousState;

previousState = digitalRead(pin);            // store switch state
for(int counter=0; counter < debounceDelay; counter++)
{
    delay(1);                        // wait for 1 ms
    state = digitalRead(pin);    // read the pin
    if( state != previousState)
    {
        counter = 0; // reset the counter if the state changes
        previousState = state;   // and save the current state
    }
}
// here when the switch state has been stable longer than the debounce
period
if(state == LOW)   // LOW means pressed (because pull-ups are used)
    return true;
else
    return false;
}
```

For testing, you can add a `count` variable to display the number of presses.
If you view this on the Serial Monitor (see Chapter 4), you can see whether
it increments once per press. Increase the value of `debounceDelay` until the
count keeps step with the presses. The following fragment prints the value
of `count` when used with the `debounce` function shown earlier:

```
int count;    // add this variable to store the number of presses

void setup()
{
  pinMode(inPin, INPUT);
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600); // add this to the setup function
}

void loop()
{
  bool button_state = debounce(inputPin);
  if (button_state)
  {
      count++; // increment count
      Serial.println(count);  // display the count on the Serial Monitor
  }
```

```
    // If the button state changed and the button was pressed
    if (last_button_state != button_state && button_state == HIGH) {
      // Toggle the LED
      ledState = !ledState;
      digitalWrite(LED_BUILTIN, ledState);
    }
    last_button_state = button_state;
  }
```

This `debounce()` function will work for any number of switches, but you must ensure that the pins used are in input mode.

A potential disadvantage of this method for some applications is that from the time the `debounce` function is called, everything waits until the switch is stable. In most cases this doesn't matter, but your sketch may need to be attending to other things while waiting for your switch to stabilize. You can use the code shown in Recipe 5.4 to overcome this problem.

## See Also

See the Debounce example sketch distributed with Arduino. From the File menu, select Examples → Digital → Debounce.

# 5.4 Determining How Long a Switch Is Pressed

## Problem

Your application wants to detect the length of time a switch has been in its current state. Or you want to increment a value while a switch is pushed and you want the rate to increase the longer the switch is held (the way many electronic clocks are set). Or you want to know if a switch has been pressed long enough for the reading to be stable (see Recipe 5.3).

## Solution

The following sketch demonstrates the setting of a countdown timer. The wiring is the same as in Figure 5-5 from Recipe 5.2. Pressing a switch sets the timer by incrementing the timer count; releasing the switch starts the countdown. The code debounces the switch and accelerates the rate at which the counter increases when the switch is held for longer periods. The timer count is incremented by one when the switch is initially pressed (after debouncing). Holding the switch for more than one second increases the increment rate by four; holding the switch for four seconds increases the rate by 10. Releasing the switch starts the countdown, and when the count reaches zero, a pin is set HIGH (in this example, lighting an LED):

```
/*
 SwitchTime sketch
 Countdown timer that decrements every tenth of a second
 lights an LED when 0
 Pressing button increments count, holding button down increases
 rate of increment

 */
const int ledPin = LED_BUILTIN;        // the number of the output pin
const int inPin = 2;                   // the number of the input pin

const int debounceTime = 20;           // the time in milliseconds required
                                       // for the switch to be stable
const int fastIncrement = 1000;        // increment faster after this many
                                       // milliseconds
const int veryFastIncrement = 4000;    // and increment even faster after
                                       // this many milliseconds
int count = 0;                         // count decrements every tenth of a
                                       // second until reaches 0

void setup()
{
  pinMode(inPin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  int duration = switchTime();
  if(duration > veryFastIncrement)
  {
    count = count + 10;
```

```arduino
  }
  else if (duration > fastIncrement)
  {
    count = count + 4;
  }
  else if (duration > debounceTime)
  {
    count = count + 1;
  }
  else
  {
    // switch not pressed so service the timer
    if( count == 0)
    {
      digitalWrite(ledPin, HIGH);   // turn the LED on if the count is 0
    }
    else
    {
      digitalWrite(ledPin, LOW);    // turn the LED off if the count is not 0
      count = count - 1;            // and decrement the count
    }
  }
  Serial.println(count);
  delay(100);
}

// return the time in milliseconds that the switch has been pressed (LOW)
long switchTime()
{
  // these variables are static - see Discussion for an explanation
  static unsigned long startTime = 0; // when switch state change was detected
  static bool state;                  // the current state of the switch

  if(digitalRead(inPin) != state) // check to see if switch has changed state
  {
    state = ! state;       // yes, invert the state
    startTime = millis();  // store the time
  }
  if(state == LOW)
  {
    return millis() - startTime;  // switch pushed, return time in
milliseconds
  }
  else
  {
    return 0; // return 0 if the switch is not pushed (in the HIGH state);
  }
}
```

## Discussion

The heart of this recipe is the `switchTime` function. This returns the number of milliseconds that the switch has been pressed. Because this recipe uses internal pull-up resistors (see Recipe 5.2), the `digitalRead` of the switch pin will return `LOW` when the switch is pressed.

The `loop` checks the value returned from `switchTime` to see what should happen. If the time the switch has been held down is long enough for the fastest increment, the counter is incremented by that amount; if not, it checks the `fast` value to see if that should be used; if not, it checks if the switch has been held down long enough to stop bouncing and if so, it increments a small amount. At most, one of those will happen. If none of them are `true`, the switch is not being pressed, or it has not been pressed long enough to have stopped bouncing. The counter value is checked and an LED is turned on if it is zero; if it's not zero, the counter is decremented and the LED is turned off.

You can use the `switchTime` function just for debouncing a switch. The following code handles debounce logic by calling the `switchTime` function:

```
// the time in milliseconds that the switch needs to be stable
const int  debounceTime = 20;

if( switchTime() > debounceTime)
{
  Serial.print("switch is debounced");
}
```

This approach to debouncing can be handy if you have more than one switch, because you can peek in and look at the amount of time a switch has been pressed and process other tasks while waiting for a switch to become stable. To implement this, you need to store the current state of the switch (pressed or not) and the time the state last changed. There are many ways to do this—in this example, you will use a separate function for each switch. You could store the variables associated with all the switches at the top of your sketch as *global variables* (called "global" because they are accessible

everywhere). But it is more convenient to have the variables for each switch contained with the function.

To retain the values of variables defined in the function, this sketch uses *static variables*. Static variables within a function provide permanent storage for values that must be maintained between function calls. A value assigned to a static variable is retained even after the function returns. The last value set will be available the next time the function is called. In that sense, static variables are similar to the global variables (variables declared outside a function, usually at the beginning of a sketch) that you saw in the other recipes. But unlike global variables, static variables declared in a function are only accessible within that function. The benefit of static variables is that they cannot be accidentally modified by some other function.

This sketch shows an example of how you can add separate functions for different switches. The wiring for this is similar to Recipe 5.2, with the second switch wired similarly to the first (as shown in Figure 5-5) but connected between pin 3 and GND:

```
/*
 SwitchTimeMultiple sketch
 Prints how long more than one switch has been pressed
*/

const int switchAPin = 2;  // the pin for switch A
const int switchBPin = 3;  // the pin for switch B

void setup()
{
  pinMode(switchAPin, INPUT_PULLUP);
  pinMode(switchBPin, INPUT_PULLUP);
  Serial.begin(9600);
}

void loop()
{
  unsigned long timeA;
  unsigned long timeB;

  timeA = switchATime();
  timeB = switchBTime();
```

```
    if (timeA > 0 || timeB > 0)
    {
      Serial.print("switch A time=");
      Serial.print(timeA);

      Serial.print(", switch B time=");
      Serial.println(timeB);
    }
}

unsigned long switchTime(int pin, bool &state, unsigned long  &startTime)
{
  if(digitalRead(pin) != state) // check to see if the switch has changed
state
  {
    state = ! state;        // yes,  invert the state
    startTime = millis();  // store the time
  }
  if(state == LOW)
  {
    return millis() - startTime;   // return the time in milliseconds
  }
  else
  {
    return 0; // return 0 if the switch is not pushed (in the HIGH state);
  }
}

long switchATime()
{
  // these variables are static - see text for an explanation
  // the time the switch state change was first detected
  static unsigned long startTime = 0;
  static bool state;  // the current state of the switch
  return switchTime(switchAPin, state, startTime);
}

long switchBTime()
{
  // these variables are static - see text for an explanation
  // the time the switch state change was first detected
  static unsigned long startTime = 0;
  static bool state;  // the current state of the switch
  return switchTime(switchBPin, state, startTime);
}
```

The sketch performs its time calculation in a function called `switchTime()`. This function examines and updates the switch state and duration. The function uses references to handle the parameters—references were covered in Recipe 2.11. A function for each switch (`switchATime()` and `switchBTime()`) is used to retain the start time and state for each switch. Because the variables holding the values are declared as static, the values will be retained when the functions exit. Holding the variables within the function ensures that the wrong variable will not be used. The pins used by the switches are declared as global variables because the values are needed by `setup` to configure the pins. But because these variables are declared with the `const` keyword, the compiler will not allow the values to be modified, so there is no chance that these will be accidentally changed by the sketch code.

> **TIP**
>
> Limiting the exposure of a variable becomes more important as projects become more complex. The Arduino environment provides a more elegant way to handle this; see Recipe 16.4 for a discussion on how to implement this using classes.

Within `loop`, the sketch checks to see how long the button was held down. If either button was held down for more than zero milliseconds, the sketch prints the time that each switch was held. If you open the Serial Monitor and hold down either or both buttons, you'll see the time values increase and scroll by. If you release both buttons, the `switchTime` function returns zero, so the sketch stops printing output until you press one or both again.
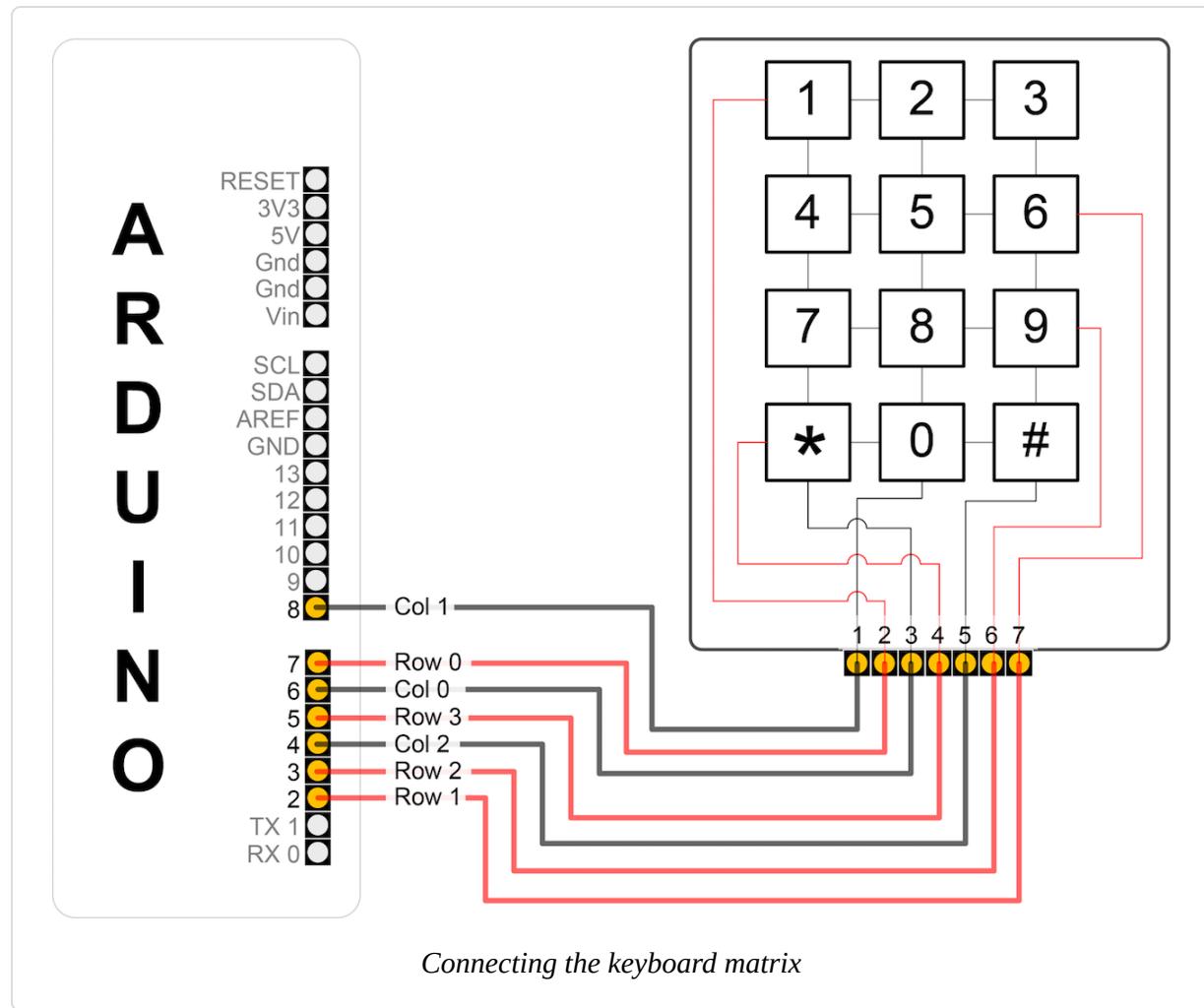
# 5.5 Reading a Keypad

## Problem

You have a matrix keypad and want to read the key presses in your sketch. For example, you have a telephone-style keypad similar to the Adafruit 12-button keypad (Adafruit ID: 419)

# Solution

Wire the rows and columns from the keypad connector to the Arduino, as shown in Figure 5-6.



*Connecting the keyboard matrix*

If you've wired your Arduino and keypad as shown in Figure 5-6, the following sketch will print key presses to the Serial Monitor:

```
/*
  Keypad sketch
  prints the key pressed on a keypad to the serial port
*/

const int numRows = 4;        // number of rows in the keypad
const int numCols = 3;        // number of columns
const int debounceTime = 20;  // number of milliseconds for switch to be stable
```

```cpp
// keymap defines the character returned when the corresponding key is pressed
const char keymap[numRows][numCols] = {
  { '1', '2', '3'  } ,
  { '4', '5', '6'  } ,
  { '7', '8', '9'  } ,
  { '*', '0', '#'  }
};

// this array determines the pins used for rows and columns
const int rowPins[numRows] = {8, 7, 6, 5}; // Rows 0 through 3
const int colPins[numCols] = {4, 3, 2};    // Columns 0 through 2

void setup()
{
  Serial.begin(9600);
  for (int row = 0; row < numRows; row++)
  {
    pinMode(rowPins[row],INPUT_PULLUP); // Set row pins as input with pullups
  }
  for (int column = 0; column < numCols; column++)
  {
    pinMode(colPins[column],OUTPUT);    // Set column pins as outputs
    digitalWrite(colPins[column],HIGH);  // Make all columns inactive
  }
}

void loop()
{
  char key = getKey();
  if( key != 0) { // if the character is not 0 then it's a valid key press
    Serial.print("Got key ");
    Serial.println(key);
  }
}

// returns the key pressed, or zero if no key is pressed
char getKey()
{
  char key = 0;                              // 0 indicates no key pressed

  for(int column = 0; column < numCols; column++)
  {
    digitalWrite(colPins[column],LOW);        // Activate the current column.
    for(int row = 0; row < numRows; row++)    // Scan all rows for key press
    {
      if(digitalRead(rowPins[row]) == LOW)    // Is a key pressed?
      {
        delay(debounceTime);                  // debounce
```

```
            while(digitalRead(rowPins[row]) == LOW)
                ;                             // wait for key to be released
            key = keymap[row][column];        // Remember which key
                                              // was pressed.
        }
    }
    digitalWrite(colPins[column],HIGH);     // De-activate the current column.
  }
  return key;  // returns the key pressed or 0 if none
}
```

This sketch will only work correctly if the wiring agrees with the code.
Table 5-2 shows how the rows and columns should be connected to
Arduino pins. If you are using a different keypad, check your datasheet to
determine the row and column connections. Check carefully, as incorrect
wiring can short out the pins, and that could damage your controller chip.

*Mapping of Arduino pins to keypad rows and columns*

| Arduino pin | Keypad connector | Keypad row/column |
| --- | --- | --- |
| 2 | 7 | Column 2 |
| 3 | 6 | Column 1 |
| 4 | 5 | Column 0 |
| 5 | 4 | Row 3 |
| 6 | 3 | Row 2 |
| 7 | 2 | Row 1 |
| 8 | 1 | Row 0 |

## Discussion

Matrix keypads typically consist of Normally Open switches that connect a
row with a column when pressed. (A Normally Open switch only makes an
electrical connection when pushed.) Figure 5-6 shows how the internal
conductors connect the button rows and columns to the keyboard connector.

Each of the four rows is connected to an input pin and each column is connected to an output pin. The `setup` function sets the pin modes to enable pull-up resistors on the input pins (see the pull-up recipes in the beginning of this chapter).

The `getkey` function sequentially sets the pin for each column `LOW` and then checks to see if any of the row pins are `LOW`. Because pull-up resistors are used, the rows will be `HIGH` (pulled up) unless a switch is closed (closing a switch produces a `LOW` signal on the input pin). If they are `LOW`, this indicates that the switch for that row and column is closed. A delay is used to ensure that the switch is not bouncing (see Recipe 5.3); the code waits for the switch to be released, and the character associated with the switch is found in the `keymap` array and returned from the function. A `0` is returned if no switch is pressed.

The Keypad library for Arduino makes it easier to handle a different number of keys and can be made to work while sharing some of the pins with an LCD character display. It is available in the Arduino Library Manager (see Recipe 16.2 for instructions on installing libraries).

### See Also

More information on the Adafruit 12-button keypad

# 5.6 Reading Analog Values

### Problem

You want to read the voltage on an analog pin. Perhaps you want a reading from a potentiometer (pot), a variable resistor, or a sensor that provides a varying voltage.

### Solution

This sketch reads the voltage on an analog pin (A0) and flashes an LED at a proportional rate to the value returned from the `analogRead` function. The

voltage is adjusted by a potentiometer connected as shown in Figure 5-7:

```
/*
 Pot sketch
 blink an LED at a rate set by the position of a potentiometer
*/

const int potPin = A0;           // select the input pin for the potentiometer
const int ledPin = LED_BUILTIN; // select the pin for the LED
int val = 0;  // variable to store the value coming from the sensor

void setup()
{
  pinMode(ledPin, OUTPUT);  // declare the ledPin as an OUTPUT
}

void loop() {
  val = analogRead(potPin);    // read the voltage on the pot
  digitalWrite(ledPin, HIGH); // turn the ledPin on
  delay(val);                 // blink rate set by pot value (in milliseconds)
  digitalWrite(ledPin, LOW);  // turn the ledPin off
  delay(val);                 // turn led off for same period as it was turned
on
}
```
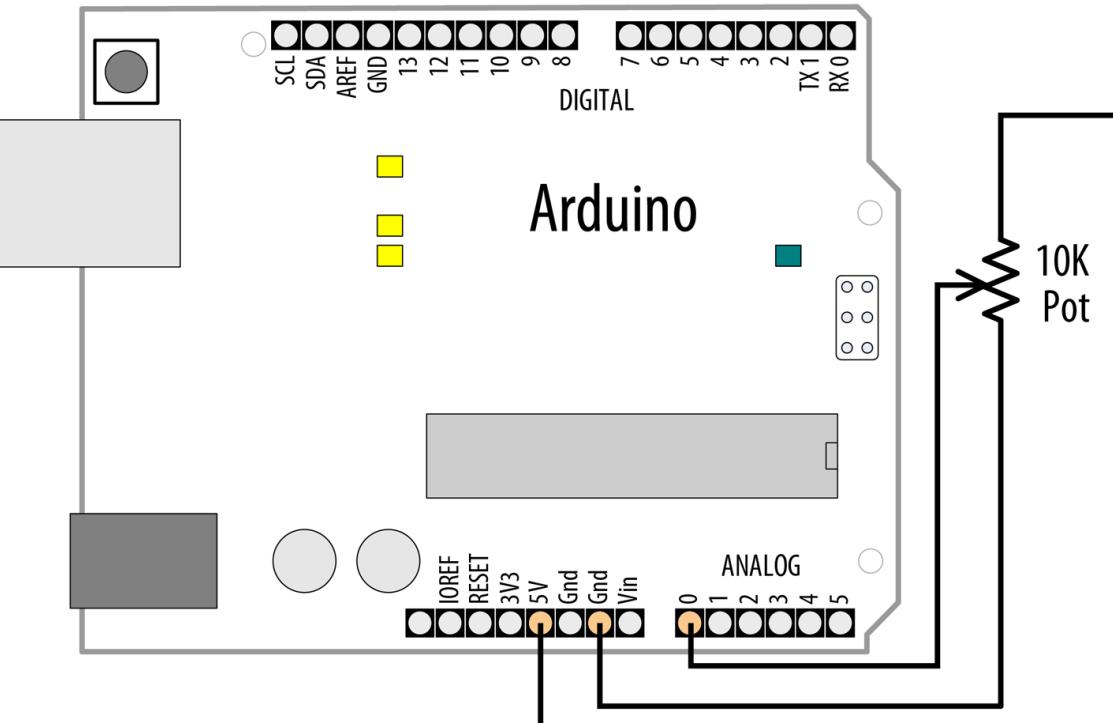
## WARNING

If your board is not 5V tolerant, do not connect the potentiometer to 5V even if your board has a 5V power pin. Many boards that are not 5V tolerant have a 5V power pin that draws power directly from the USB power. This pin can be used to power devices that require 5V to run, but you must be careful to never connect a 5V output to a pin that can tolerate no more than 3.3V. You should connect the potentiometer to the 3.3V pin on the board instead.

*Connecting a potentiometer to Arduino*

## Discussion

This sketch uses the `analogRead` function to read the voltage on the potentiometer's *wiper* (the center pin). A pot has three pins; two are connected to a resistive material and the third pin (usually in the middle) is connected to a wiper that can be rotated to make contact anywhere on the resistive material. As the potentiometer rotates, the resistance between the wiper and one of the pins increases, while the other decreases. The schematic diagram for this recipe (Figure 5-7) may help you visualize how a potentiometer works; as the wiper moves toward the bottom end, the wiper (the line with the arrow) will have lower resistance connecting to GND and higher resistance connecting to 5 volts (or 3.3 volts, depending on your board). As the wiper moves down, the voltage on the analog pin will decrease (to a minimum of 0 volts). Moving the wiper upward will have the opposite effect, and the voltage on the pin will increase (up to a maximum of 5 or 3.3 volts).

> **TIP**
>
> If the voltage on the pin decreases, rather than increases, as you increase the rotation of the potentiometer, you need to reverse the connections to the 5V and GND pins.

The voltage is measured using `analogRead`, which provides a value proportional to the actual voltage on the analog pin. The value will be 0 when the voltage on the pin is 0 and 1,023 when the voltage is at 5V (or 3.3V for a 3.3V board such as most 32-bit boards). A value in between will be proportional to the ratio of the voltage on the pin to 5 (or 3.3, depending on your board) volts.

Potentiometers with a value of 10K ohms are the best choice for connecting to analog pins.

> **NOTE**
>
> `potPin` does not need to be set as input. (This is done for you automatically each time you call `analogRead`.)

## See Also

Appendix B for tips on reading schematic diagrams

This Arduino reference for `analogRead`

*Getting Started with Arduino* by Massimo Banzi and Michael Shiloh (Make Community)

# 5.7 Changing the Range of Values

## Problem

You want to change the range of a value, such as the value from `analogRead` obtained by connecting a potentiometer (pot) or other device

that provides a variable voltage. For example, suppose you want to display the position of a potentiometer knob as a percentage from 0% to 100%.

## Solution

Use the Arduino `map` function to scale values to the range you want. This sketch reads the voltage on a pot into the variable `val` and scales this from 0 to 100 as the pot is rotated from one end to the other. It blinks an LED with a rate proportional to the voltage on the pin and prints the scaled range to the serial port (see Recipe 4.2 for instructions on monitoring the serial port). Recipe 5.6 shows how the pot is connected (see Figure 5-7):

```
/*
 * Map sketch
 * map the range of analog values from a pot to scale from 0 to 100
 * resulting in an LED blink rate ranging from 0 to 100 ms.
 * and Pot rotation percent is written to the serial port
 */

const int potPin = A0;       // select the input pin for the potentiometer
int ledPin = LED_BUILTIN;  // select the pin for the LED

void setup()
{
  pinMode(ledPin, OUTPUT);     // declare the ledPin as an OUTPUT
  Serial.begin(9600);
}

void loop() {
  int val;      // The value coming from the sensor
  int percent; // The mapped value

  val = analogRead(potPin);        // read the voltage on the pot (val ranges
                                   // from 0 to 1023)
  percent = map(val,0,1023,0,100); // percent will range from 0 to 100.
  digitalWrite(ledPin, HIGH);      // turn the ledPin on
  delay(percent);                  // On time given by percent value
  digitalWrite(ledPin, LOW);       // turn the ledPin off
  delay(100 - percent);            // Off time is 100 minus On time
  Serial.println(percent);         // show % of pot rotation on Serial Monitor
}
```

## Discussion

Recipe 5.6 describes how the position of a pot is converted to a value. Here you use this value with the `map` function to scale the value to your desired range. In this example, the value provided by `analogRead` (0 to 1023) is mapped to a percentage (0 to 100). This percentage is used to set the LED's *duty cycle*. Duty cycle is the percentage of time that the LED is active, measured over a duration called the *period*, which is 100 ms. The time that the LED is off is calculated by subtracting the duty cycle from 100. So, if the analog reading is 620, it will be scaled to 60 by `map`. The LED will then be turned on for 60 ms, and turned off for 40 ms (100–60).

The values from `analogRead` range from `0` to `1023` if the voltage ranges from 0 to 5 volts (3.3 volts on 3.3-volt boards), but you can use any appropriate values for the source and target ranges. For example, a typical pot only rotates 270 degrees from end to end, and if you wanted to display the angle of the knob on your pot, you could use this code:

```
int angle = map(val,0,1023,0,270); // angle of pot derived from analogRead val
```

Range values can also be negative. If you want to display `0` when the pot is centered and negative values when the pot is rotated left and positive values when it is rotated right, you can do this:

```
>
// show angle of 270 degree pot with center as 0
angle = map(val,0,1023,-135,135);
```

The `map` function can be handy where the input range you are concerned with does not start at zero. For example, if you have a battery where the available capacity is proportional to a voltage that ranges from 1.1 volts to 1.5 volts, you can do the following:

```
const int board_voltage = 5.0; // Set to 3.3 on boards that use 3.3 volt logic

const int empty = 1.1/(5.0/1023.0); // voltage is 1.1V (1100mv) when empty
const int full  = 1.5/(5.0/1023.0); // voltage is 1.5V (1500mv) when full
```

```
int val = analogRead(potPin);              // read the analog voltage
int percent = map(val, empty, full, 0,100); // map the voltage to a percent
Serial.println(percent);
```

If you are using sensor readings with `map`, you will need to determine the minimum and maximum values from your sensor. You can monitor the reading on the serial port to determine the lowest and highest values. Enter these as the lower and upper bound into the `map` function.

If the range can't be determined in advance, you can determine the values by calibrating the sensor. Recipe 8.11 shows one technique for calibration; another can be found in the Calibration examples sketch distributed with Arduino (Examples → Analog → Calibration).

Bear in mind that if you feed values into `map` that are outside the upper and lower limits, the output will also be outside the specified output range. You can prevent this from happening by using the `constrain` function; see Recipe 3.5.

---

**NOTE**

`map` uses integer math, so it will only return whole numbers in the range specified. Any fractional element is truncated, not rounded.

---

(See Recipe 5.9 for more details on how `analogRead` values relate to actual voltage.)

## See Also

The Arduino reference for `map`

# 5.8 Reading More than Six Analog Inputs

## Problem

You have more analog inputs to monitor than you have available analog pins. A standard Arduino board has six analog inputs (the Mega has 16) and there may not be enough analog inputs available for your application. Perhaps you want to adjust eight parameters in your application by turning knobs on eight potentiometers.

## Solution

Use a multiplexer chip to select and connect multiple voltage sources to one analog input. By sequentially selecting from multiple sources, you can read each source in turn. This recipe uses the popular 4051 chip connected to Arduino as shown in Figure 5-8. Connect your analog inputs (such as a pot or resistive sensor) to the 4051 pins marked Ch 0 to Ch 7. Make sure the voltage on the channel input pins is never higher than 5 volts. If you are not using an input pin, you must connect it to ground with a 10K resistor:

```
/*
 * multiplexer sketch
 * read 1 of 8 analog values into single analog input pin with 4051
multiplexer
 */

// array of pins used to select 1 of 8 inputs on multiplexer
const int select[] = {2,3,4}; // pins connected to the 4051 input select lines
const int analogPin = A0;      // the analog pin connected to multiplexer
output

// this function returns the analog value for the given channel
int getValue(int channel)
{
    // set the selector pins HIGH and LOW to match the binary value of channel
    for(int bit = 0; bit < 3; bit++)
    {
        int pin = select[bit]; // the pin wired to the multiplexer select bit
        int isBitSet =  bitRead(channel, bit); // true if given bit set in
channel
        digitalWrite(pin, isBitSet);
    }
    return analogRead(analogPin);
}

void setup()
```
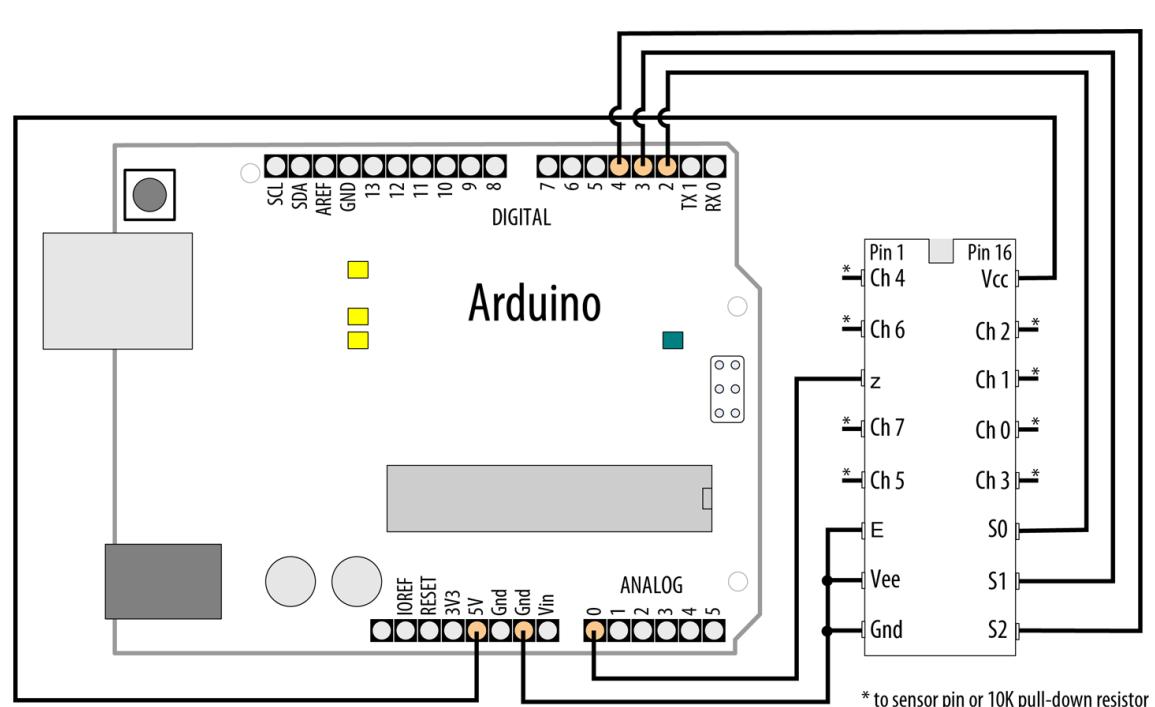
```
{
  for(int bit = 0; bit < 3; bit++)
  {
    pinMode(select[bit], OUTPUT);  // set the three select pins to output
  }
  Serial.begin(9600);
}
void loop () {
  // print the values for each channel once per second
  for(int channel = 0; channel < 8; channel++)
  {
    int value = getValue(channel);
    Serial.print("Channel ");
    Serial.print(channel);
    Serial.print(" = ");
    Serial.println(value);
  }
  delay (1000);
}
```



*The 4051 multiplexer connected to Arduino*

# Discussion

Analog multiplexers are digitally controlled analog switches. The 4051 selects one of eight inputs through three selector pins (S0, S1, and S2). There are eight different combinations of values for the three selector pins, and the sketch sequentially selects each of the possible bit patterns; see .

You must connect the ground from the devices you are measuring to the ground on the 4051 and Arduino. In order to read values accurately, they must have a common ground. If you are planning on powering all the devices from a 5V or 3.3V pin on your board, be sure that your power draw does not exceed either the maximum power from your power supply or the maximum power that the pin is capable of delivering (whichever is lower). For example, the Arduino Uno's 5V pin can safely deliver 900 mA when it is being powered by an external power supply (400 mA max on USB power). But if you are using a 500 mA power supply, then the maximum you can draw is less than 500 mA because the microcontroller, LEDs, and other components draw power as well. This is a best-case maximum, so you should stay below that. You may need to dig into the documentation for your board, and possibly the datasheet for its microcontroller and voltage regulator to confirm the limits. If you find that your total current draw is taking you anywhere near the limit, use a separate power supply for the devices you are connecting.

*Truth table for 4051 multiplexer*

| Selector pins | | | Input channel |
|---|---|---|---|
| S2 | S1 | S0 | |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 2 |
| 0 | 1 | 1 | 3 |
| 1 | 0 | 0 | 4 |

| Selector pins | | | Input channel |
|---|---|---|---|
| 1 | 0 | 1 | 5 |
| 1 | 1 | 0 | 6 |
| 1 | 1 | 1 | 7 |

You may recognize the pattern in Table 5-3 as the binary representation of the decimal values from 0 to 7.

In the Solution sketch, `getValue()` is the function that sets the correct selector bits for the given channel using `digitalWrite(pin, isBitSet)` and reads the analog value from the selected 4051 input with `analogRead(analogPin)`. The code to produce the bit patterns uses the built-in `bitRead` function (see Recipe 3.12).

Bear in mind that this technique selects and monitors the eight inputs sequentially, so it requires more time between the readings on a given input compared to using `analogRead` directly. If you are reading eight inputs, it will take eight times longer for each input to be read. This may make this method unsuitable for inputs that change value quickly.

## See Also

The Arduino Playground tutorial for the 4051

The 74HC4051 datasheet
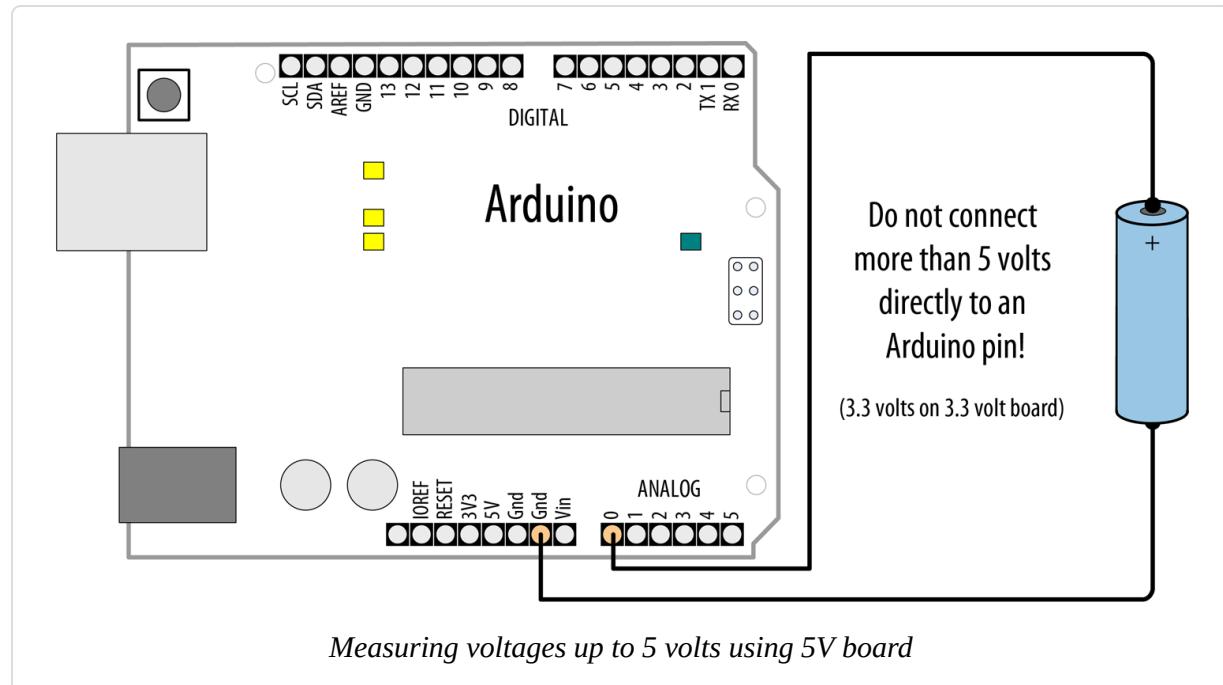
# 5.9 Measuring Voltages Up to 5V

## Problem

You want to monitor and display the value of a voltage between 0 and 5 volts. For example, suppose you want to display the voltage of a single 1.5V cell on the Serial Monitor.

## Solution

Use `AnalogRead` to measure the voltage on an analog pin. Convert the reading to a voltage by using the ratio of the reading to the reference voltage (5 volts), as shown in Figure 5-9.



*Measuring voltages up to 5 volts using 5V board*

## Measuring Analog Voltages on ESP8266 Boards

If you are using an ESP8266-based board, you may be limited to voltages in the range of 0 to 1 volt. Some ESP8266-based boards have built-in voltage dividers that allow you to read up to 3.3V (the ESP8266 itself runs at 3.3 volts), so be sure to check the documentation for your board. Without a voltage divider, the ESP8266 analog input pins max out at 1V. (See Recipe 5.11.)

The simplest solution uses a floating-point calculation to print the voltage; this example sketch calculates and prints the ratio as a voltage:

```
/*
 * Display5vOrless sketch
 * prints the voltage on analog pin to the serial port
 * Warning - do not connect more than 5 volts directly to an Arduino pin.
 */
```

```
const float referenceVolts = 5.0;  // the default reference on a 5-volt board
const int batteryPin = A0;          // battery is connected to analog pin 0

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

void loop()
{
   int val = analogRead(batteryPin);  // read the value from the sensor
   float volts = (val / 1023.0) * referenceVolts;  // calculate the ratio
   Serial.println(volts);  // print the value in volts
}
```

The formula is:

*volts = (analog reading / analog steps) × reference voltage*

Printing a floating-point value to the serial port with `println` will format the value to two decimal places.

---

**NOTE**

Make the following change if you are using a board that uses 3.3V logic:

```
const float referenceVolts = 3.3;
```

---

Floating-point numbers consume lots of memory, so unless you are already using floating point elsewhere in your sketch, it is more efficient to use integer values. The following code looks a little strange at first, but because `analogRead` returns a value of `1023` for 5 volts, each step in value will be 5 divided by 1,023. In units of millivolts, this is 5,000 divided by 1,023.

This code prints the value in millivolts:

```
const int batteryPin = A0;

void setup()
```

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

void loop()
{
  long val = analogRead(batteryPin);    // read the value from the sensor -
                                        // note val is a long int
  Serial.println( (val * (500000/1023L)) / 100);   // the value in millivolts
}
```

> ## TIP
>
> If you are using a 3.3V board, change (500000/1023L) to (330000/1023L).

The following code prints the value using decimal points. It prints 1.5 if the voltage is 1.5 volts:

```
const int batteryPin = A0;

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

void loop()
{
  int val = analogRead(batteryPin); // read the value from the sensor

  long mv =  (val * (500000/1023L)) / 100; // calculate the value in
millivolts
  Serial.print(mv/1000); // print the integer portion of the voltage
  Serial.print('.');
  int fraction = (mv % 1000); // calculate the fraction
  if (fraction == 0)
  {
     Serial.print("000");        // add three zeros
  }
  else if (fraction < 10)  // if fractional < 10 the 0 is ignored giving a
wrong
                           // time, so add the zeros
  {
     Serial.print("00");        // add two zeros
```

```
    }
    else if (fraction < 100)
    {
        Serial.print("0");
    }
    Serial.println(fraction); // print the fraction
}
```

## Discussion

The `analogRead()` function returns a value that is proportional to the ratio of the measured voltage to the reference voltage (5 volts on an Uno). To avoid the use of floating point yet maintain precision, the code operates on values as millivolts instead of volts (there are 1,000 millivolts in 1 volt). Because a value of `1023` indicates 5,000 millivolts, each unit represents 5,000 divided by 1,023 millivolts (that is, 4.89 millivolts).

> **NOTE**
>
> You will see both 1,023 and 1,024 used for converting `analogRead` values to millivolts. 1,024 is commonly used by engineers because there are 1,024 possible values between 0 and 1,023. However, 1,023 is more intuitive for some because the highest possible value is 1,023. In practice, the hardware inaccuracy is greater than the difference between the calculations, so choose whichever value you feel more comfortable with.

To eliminate the decimal point, the values are multiplied by 100. In other words, 5,000 millivolts times 100 divided by 1,023 gives the number of millivolts times 100. Dividing this by 100 yields the value in millivolts. If multiplying fractional numbers by 100 to enable the compiler to perform the calculation using fixed-point arithmetic seems convoluted, you can stick to the slower and more memory-hungry floating-point method.

This solution assumes you are using an Arduino Uno or similar 8-bit board that uses 5-volt logic. If you are using a 3.3V board, the maximum voltage you can measure is 3.3 volts without using a voltage divider—see Recipe 5.11.

# 5.10 Responding to Changes in Voltage

## Problem

You want to monitor one or more voltages and take some action when the voltage rises or falls below a threshold. For example, you want to flash an LED to indicate a low battery level—perhaps to start flashing when the voltage drops below a warning threshold and increasing in urgency as the voltage drops further.

## Solution

You can use the connections shown in Figure 5-7 in Recipe 5.9, but here we'll compare the value from `analogRead` to see if it drops below a threshold. This example starts flashing an LED at 1.2 volts and increases the on-to-off time as the voltage decreases below the threshold. If the voltage drops below a second threshold, the LED stays lit:

```
/*
 * RespondingToChanges sketch
 * flash an LED to indicate low voltage levels
 */

long batteryFull       = 1500;  // millivolts for a full battery
long warningThreshold  = 1200;  // Warning level in millivolts - LED flashes
long criticalThreshold = 1000;  // Critical voltage level - LED stays on

const int batteryPin = A0;
const int ledPin = LED_BUILTIN;

void setup()
{
    pinMode(ledPin, OUTPUT);
}

void loop()
{
  int val = analogRead(batteryPin);    // read the value from the sensor
  int mv = map(val, 0, 1023, 0, 5000);
  if(mv < criticalThreshold) {
    digitalWrite(ledPin, HIGH);
  }
  else if (mv < warningThreshold) {
```

```
      int blinkDelay = map(mv, criticalThreshold, batteryFull, 0, 250);
      flash(blinkDelay);
  }
  else
  {
    digitalWrite(ledPin, LOW);
  }
  delay(1);
}

// function to flash an led with specified on/off time
void flash(int blinkDelay)
{
  digitalWrite(ledPin, HIGH);
  delay(blinkDelay);
  digitalWrite(ledPin, LOW);
  delay(blinkDelay);
}
```

## Discussion

This sketch maps the value read from the analog port to the range of the threshold voltage (0 to 5,000 millivolts). For example, with a warning threshold of 1 volt and a reference voltage of 5 volts, you want to know when the analog reading is one-fifth of the reference voltage. When the value from `analogRead` returns 205, the `map` function will return 1,000 (1,000 millivolts = 1 volt).

When the voltage (`mv`) is below `criticalThreshold`, the LED stays on. If it's not, the sketch checks to see if the voltage is below `warningThreshold`. If it is, the sketch then calculates the blink delay by mapping the voltage (`mv`) to a value between 0 and 250. The closer the value is to `criticalThreshold`, the lower the blink delay, so the LED blinks faster as it approaches that threshold. If the voltage is above the `warningThreshold`, the LED stays off.

# 5.11 Measuring Voltages More than 5V (Voltage Dividers)

## Problem

You want to measure voltages greater than 5 volts. For example, you want to display the voltage of a 9V battery and trigger an alarm LED when the voltage falls below a certain level.

## Solution
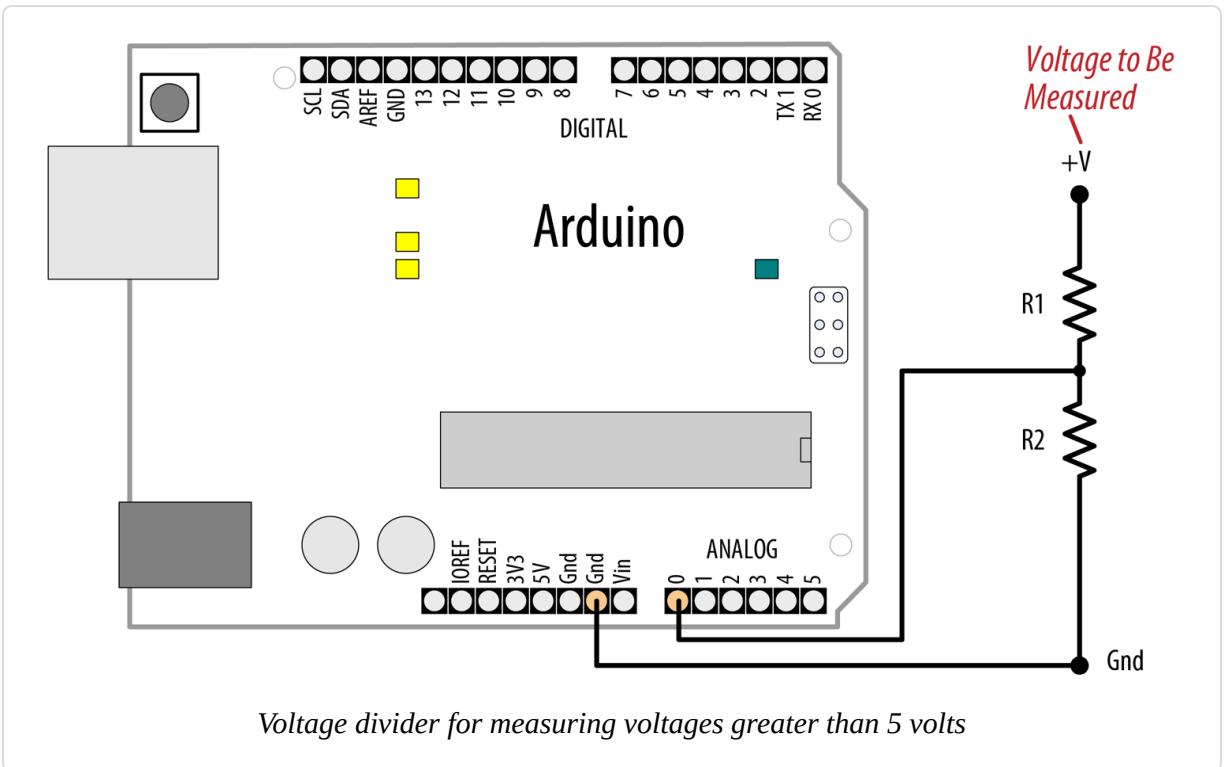
Use a solution similar to Recipe 5.9, but connect the voltage through a voltage divider (see Figure 5-10). For voltages up to 10 volts, you can use two 4.7K ohm resistors. For higher voltages, you can determine the required resistors using Table 5-4.

*Resistor values*

| Max voltage | R1 | R2 | Calculation R2/(R1 + R2) | Value of resistorFactor |
|---|---|---|---|---|
| 5 | Short[a] | None[b] | None | 1023 |
| 10 | 1K | 1K | 1(1 + 1) | 511 |
| 15 | 2K | 1K | 1(2 + 1) | 341 |
| 20 | 3K | 1K | 1(3 + 1) | 255 |
| 30 | 5K (5.1K) | 1K | 1(5 + 1) | 170 |

[a] +V connected to analog pin

[b] No connection

*Voltage divider for measuring voltages greater than 5 volts*

Select the row with the highest voltage you need to measure to find the values for the two resistors:

```
/*
  DisplayMoreThan5V sketch
  prints the voltage on analog pin to the serial port
  Do not connect more than 5 volts directly to an Arduino pin.
*/

const float referenceVolts = 5;      // the default reference on a 5-volt
board
//const float referenceVolts = 3.3;  // use this for a 3.3-volt board

const float R1 = 1000; // value for a maximum voltage of 10 volts
const float R2 = 1000;
// determine by voltage divider resistors, see text
const float resistorFactor = 1023.0 * (R2/(R1 + R2));
const int batteryPin = 0; // +V from battery is connected to analog pin 0

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

```
void loop()
{
    int val = analogRead(batteryPin);  // read the value from the sensor
    float volts = (val / resistorFactor) * referenceVolts; // calculate the
ratio
    Serial.println(volts);  // print the value in volts
}
```

## Discussion

Like the previous analog recipes, this recipe relies on the fact that the
`analogRead` value is a ratio of the measured voltage to the reference. But
because the measured voltage is divided by the two *dropping resistors*, the
`analogRead` value needs to be multiplied to get the actual voltage. This
code is similar to that in Recipe 5.7, but the value read from the analog pin
is divided not by 1,023, but by the `resistorFactor`:

```
float volts = (val / resistorFactor) * referenceVolts ;  // calculate the
ratio
```

The calculation used to produce the table is based on the following formula:
the output voltage is equal to the input voltage times R2 divided by the sum
of R1 and R2. In the example where two equal-value resistors are used to
drop the voltage from a 9V battery by half, `resistorFactor` is 511 (half of
1,023), so the value of the `volts` variable will be twice the voltage that
appears on the input pin. With resistors selected for 10 volts, the analog
reading from a 9V battery will be approximately 920.

> **WARNING**
>
> More than 5 volts on the pin (3.3V on 3.3V boards) can damage the pin and possibly destroy the
> chip; double-check that you have chosen the right value resistors and wired them correctly before
> connecting them to an Arduino input pin. If you have a multimeter, measure the voltage before
> connecting anything that could possibly carry voltages higher than 5 volts.

# Getting Input from Sensors

## 6.0 Introduction

Getting and using input from sensors enables Arduino to respond to or report on the world around it. This is one of the most common tasks you will encounter. This chapter provides simple and practical recipes for how to use the most popular input devices and sensors. Wiring diagrams show how to connect and power the devices, and code examples demonstrate how to use data derived from the sensors.

Sensors respond to input from the physical world and convert this into an electrical signal that Arduino can read on an input pin. The nature of the electrical signal provided by a sensor depends on the kind of sensor and how much information it needs to transmit. Some sensors (such as photoresistors and Piezo knock sensors) are constructed from a substance that alters its electrical properties in response to physical change. Others are sophisticated electronic modules that use their own microcontroller to process information before passing a signal on for the Arduino.

Sensors use the following methods to provide information:

Digital on/off

> Some devices, such as the tilt sensor in Recipe 6.2 and the motion sensor in Recipe 6.4, simply switch a voltage on and off. These can be treated like the switch recipes shown in Chapter 5.

Analog

> Other sensors provide an analog signal (a voltage that is proportional to what is being sensed, such as temperature or light level). The recipes for detecting light (Recipe 6.3), temperature (Recipe 6.9), and sound (Recipe 6.8) demonstrate how analog sensors can be used. All of them use the `analogRead` command that is discussed in Chapter 5.

Pulse width

Distance sensors, such as the PING))) in Recipe 6.5, provide data using pulse duration proportional to the distance value. Applications using these sensors measure the duration of a pulse using the `pulseIn` command.

Serial

Some sensors provide values using a serial protocol. For example, the GPS in Recipe 6.14 communicates through the Arduino serial port (see Chapter 4 for more on serial). Most Arduino boards only have one hardware serial port, so read Recipe 6.14 for an example of how you can add additional software serial ports if you have multiple serial sensors or the hardware serial port is occupied for some other task.

Synchronous protocols: I2C and SPI

The I2C and SPI digital serial communications interfaces were created for processors and microcontrollers like Arduino to talk to external sensors and modules. For example, Recipe 6.15 shows how a gyroscope module is connected using I2C. These protocols are used extensively for sensors, actuators, and peripherals, and they are covered in detail in Chapter 13.

There is another generic class of sensing devices that you may make use of. These are consumer devices that contain sensors but are sold as devices in their own right, rather than as sensors. An example of this in this chapter is a PS/2 mouse. These devices can be very useful; they provide sensors already incorporated into robust and ergonomic devices. They are also inexpensive (often less expensive than buying the raw sensors that they contain), as they are mass-produced. You may have some of these lying around.

If you are using a device that is not specifically covered in a recipe, check the Arduino Library Manager to see if there is a library available for it (see Recipe 16.2). If not, you may be able to adapt a recipe for a device that produces a similar type of output. Information about a sensor's output signal is usually available from the company from which you bought the

device or from a datasheet for your device (which you can find through a Google search of the device part number or description).

Datasheets are aimed at engineers designing products to be manufactured, and they usually provide more detail than you need to just get the product up and running. If you can't find a datasheet at the component vendor's website, you can usually find it with a search engine by specifying the name of the component and the word "datasheet." The information on output signal will usually be in a section referring to data format, interface, output signal, or something similar. Don't forget to check the maximum voltage (usually in a section labeled Absolute Maximum Ratings) to ensure that you don't damage the component.

---

### WARNING

Sensors designed for a maximum of 3.3 volts can be destroyed by connecting them to a voltage above that, such as an output pin on an Arduino board that operates at a 5-volt logic level. Check the absolute maximum rating for your device before connecting. If you need to connect a 5V output to a 3.3V-tolerant input, you can use a voltage divider in most cases. See Recipe 5.11 for more details on working with a voltage divider.

---

Reading sensors from the messy analog world is a mixture of science, art, and perseverance. You may need to use ingenuity and trial and error to get a successful result. A common problem is that the sensor just tells you a physical condition has occurred, not what caused it. Putting the sensor in the right context (location, range, orientation) and limiting its exposure to things that you don't want to activate it are skills you will acquire with experience.

Another issue concerns separating the desired signal from background noise; Recipe 6.7 shows how you can use a threshold to detect when a signal is above a certain level, and Recipe 6.8 shows how you can take the average of a number of readings to smooth out noise spikes.

## See Also

For information on working with and connecting electronic components, see *Make: Electronics* by Charles Platt (Make Community).

*Making Things Talk* by Tom Igoe (Make Community) addresses the intersection of science, art, and perseverance in designing and implementing sensor-based systems with Arduino.

See the introduction to Chapter 5 and Recipe 5.6 for more on reading analog values from sensors.

# 6.1 You Want an Arduino with Many Built-in Sensors

## Problem

You want to use an Arduino with multiple sensors built in.

## Solution

The Arduino Nano 33 BLE Sense is designed exactly for this type of situation. It is very small, inexpensive, fast, and includes eight sensor capabilities that are provided by a group of components built right into the board. Table 6-1 lists the components, their capabilities, and the name of the supporting library. Before you can use the Nano 33 BLE Sense, first open the Arduino Boards Manager and install the Arduino nRF528x Boards (Mbed OS) package (see Recipe 1.7). Next, install each of the libraries listed in the Library name column using the Library Manager (see Recipe 16.2).

*Nano 33 BLE Sense built-in sensors*

| Component | Features | Library name |
| --- | --- | --- |
| Broadcom APDS-9960 | Gesture, Proximity, RGB Color | Arduino_APDS9960 |
| ST HTS221 | Temperature, Relative Humidity | Arduino_HTS221 |
| ST LPS22HB | Barometric Pressure | Arduino_LPS22HB |

| Component | Features | Library name |
|---|---|---|
| ST LSM9DS1 | 9DOF Inertial Measurement Unit (IMU): accelerometer, gyroscope, magnetometer | Arduino_LSM9DS1 |
| ST MP34DT05 | Digital microphone | (Installed by default with Nano 33 BLE board package) |

After you've installed support for the Nano 33 BLE Sense board and the supporting libraries, use the Tools menu to configure the Arduino IDE to use the Nano 33 BLE board and set the correct port. As of this writing, both the Nano 33 BLE and Nano 33 BLE Sense use the same board setting in the IDE (the Nano 33 BLE is the same as the Nano 33 BLE Sense, just without all the cool sensors). Next, load the following sketch onto the board and open the Serial Monitor:

```
/*
 * Arduino Nano BLE Sense sensor demo
 */

#include <Arduino_APDS9960.h>
#include <Arduino_HTS221.h>
#include <Arduino_LPS22HB.h>
#include <Arduino_LSM9DS1.h>

void setup() {

  Serial.begin(9600);
  while (!Serial);

  if (!APDS.begin()) { // Initialize gesture/color/proximity sensor
    Serial.println("Could not initialize APDS9960.");
    while (1);
  }
  if (!HTS.begin()) {  // Initialize temperature/humidity sensor
    Serial.println("Could not initialize HTS221.");
    while (1);
  }
  if (!BARO.begin()) { // Initialize barometer
    Serial.println("Could not initialize LPS22HB.");
    while (1);
  }
  if (!IMU.begin()) {  // Initialize inertial measurement unit
```

```
      Serial.println("Could not initialize LSM9DS1.");
      while (1);
    }

    prompt(); // Tell users what they can do.
}

void loop() {

    // If there's a gesture, run the appropriate function.
    if (APDS.gestureAvailable()) {
      int gesture = APDS.readGesture();
      switch (gesture) {
        case GESTURE_UP:
          readTemperature();
          break;

        case GESTURE_DOWN:
          readHumidity();
          break;

        case GESTURE_LEFT:
          readPressure();
          break;

        case GESTURE_RIGHT:
          Serial.println("Spin the gyro!\nx, y, z");
          for (int i = 0; i < 10; i++)
          {
            readGyro();
            delay(250);
          }
          break;

        default:
          break;
      }
      prompt(); // Show the prompt again
    }
}
void prompt() {
    Serial.println("\nSwipe!");
    Serial.println("Up for temperature, down for humidity");
    Serial.println("Left for pressure, right for gyro fun.\n");
}

void readTemperature()
{
```

```
  float temperature = HTS.readTemperature(FAHRENHEIT);
  Serial.print("Temperature: "); Serial.print(temperature);
  Serial.println(" °F");
}

void readHumidity()
{
  float humidity = HTS.readHumidity();
  Serial.print("Humidity: "); Serial.print(humidity);
  Serial.println(" %");
}

void readPressure()
{
  float pressure = BARO.readPressure(PSI);
  Serial.print("Pressure: "); Serial.print(pressure);
  Serial.println(" psi");
}

void readGyro()
{
  float x, y, z;
  if (IMU.gyroscopeAvailable()) {
    IMU.readGyroscope(x, y, z);
    Serial.print(x); Serial.print(", ");
    Serial.print(y); Serial.print(", ");
    Serial.println(z);
  }
}
```

The Serial Monitor will display a prompt that tells you how you can interact with the Arduino Nano 33 BLE Sense. To swipe in a given direction, hold your hand over the top of the board and make a wiping motion. To swipe up, wave your hand in a motion that moves from the board's USB port up to the u-blox module on the opposite end.

## Discussion

The code in this recipe uses several of the sensors built into the Nano 33 BLE Sense: the gesture sensor (APDS-9960), the temperature/humidity sensor (HTS221), the barometer (LPS22HB), and the gyroscope (LSM9DS1). The `setup` function waits until the serial port is open, then it initializes each of these devices, and if it encounters an error, it will display

an error message and hang by entering an endless loop with `while(1);`. At the end of `setup`, the sketch calls the `prompt` routine, which displays instructions on the Serial Monitor.

Within the `loop`, the sketch checks to see if the APDS-9960 has detected a gesture. If so, it dispatches execution to a function that corresponds to the desired sensor. Each of these functions reads the state of the sensor and displays it on the Serial Monitor. For the gyroscope, the sketch prompts you to spin the board around, and then enters a loop where it reads the gyro 10 times with a slight delay so you can see how the values change with your motion.

## See Also

Arduino has a forum dedicated to the Nano 33 BLE Sense. You may also want to visit the forum for the Nano 33 BLE, which is a variant of the board without all the built-in sensors.

Recipe 6.15 has more on using a gyroscope with Arduino.

Recipe 6.17 has more on accelerometers.

# 6.2 Detecting Movement

## Problem

You want to detect when something is moved, tilted, or shaken.

## Solution

This sketch uses a switch that closes a circuit when tilted, called a *tilt sensor*. The switch recipes in Chapter 5 (Recipes 5.1 and 5.2) will work with a tilt sensor substituted for the switch.

The following sketch (circuit shown in Figure 6-1) will switch on the LED attached to pin 11 when the tilt sensor is tilted one way, and the LED connected to pin 12 when it is tilted the other way:

```
/*
 * tilt sketch
 *
 * a tilt sensor attached to pin 2 lights one of
 * the LEDs connected to pins 11 and 12 depending
 * on which way the sensor is tilted
 */

const int tiltSensorPin = 2;   // pin the tilt sensor is connected to
const int firstLEDPin  = 11;   // pin for one LED
const int secondLEDPin = 12;   // pin for the other

void setup()
{
  pinMode (tiltSensorPin, INPUT_PULLUP); // Tilt sensor connected to this pin

  pinMode (firstLEDPin, OUTPUT);    // first output LED
  pinMode (secondLEDPin, OUTPUT);   // and the second
}

void loop()
{
  if (digitalRead(tiltSensorPin) == LOW){ // The switch is on (upright)
    digitalWrite(firstLEDPin, HIGH);      // Turn on the first LED
    digitalWrite(secondLEDPin, LOW);      // and turn off the second.
  }
  else{                                   // The switch is off (tilted)
    digitalWrite(firstLEDPin, LOW);       // Turn the first LED off
    digitalWrite(secondLEDPin, HIGH);     // and turn on the second.
  }
}
```
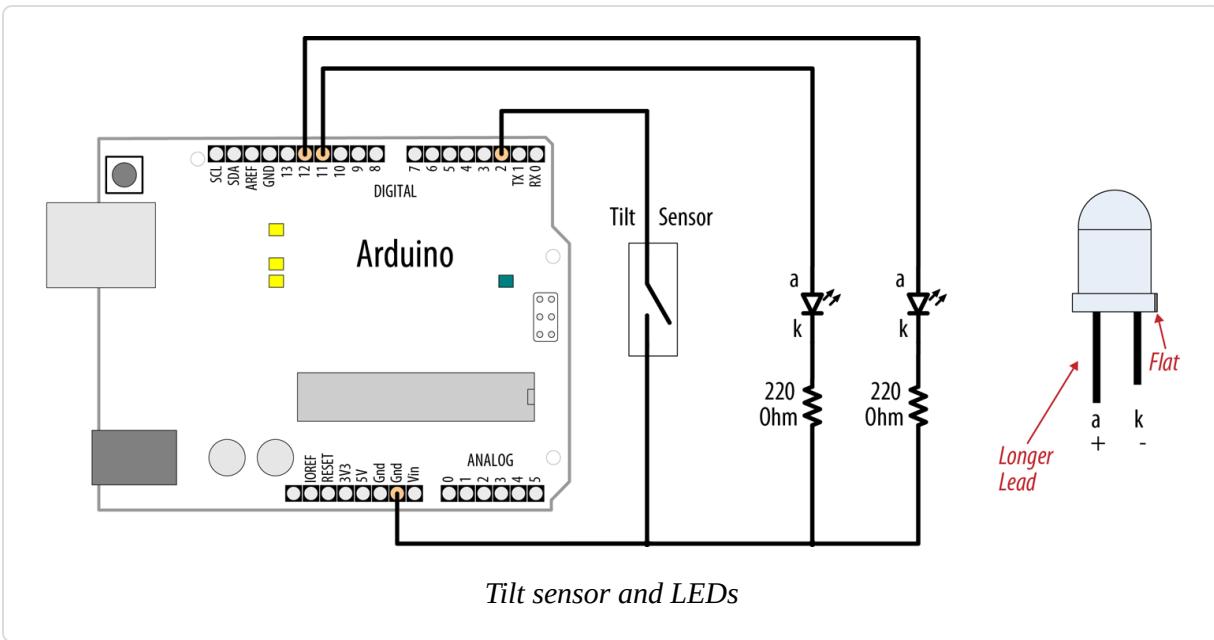
*Tilt sensor and LEDs*

## Discussion

The most common tilt sensor is a ball bearing in a tube with contacts at one end. When the tube is tilted the ball rolls away from the contacts and the connection is broken. When the tube is tilted to roll the other way, the ball touches the contacts and completes a circuit. Markings, or pin configurations, may show which way the sensor should be oriented. Tilt sensors are sensitive to small movements of around 5 to 10 degrees when oriented with the ball just touching the contacts. If you position the sensor so that the ball bearing is directly above the contacts, the LED state will only change if it is just turned over. This can be used to tell if something is upright or upside down.

To determine if something is being shaken, you need to check how long it's been since the state of the tilt sensor changed (this recipe's Solution just checks if the switch was open or closed). If it hasn't changed for a time you consider significant, the object is not shaking. Changing the orientation of the tilt sensor will change how vigorous the shaking needs to be to trigger it. The following code lights the built-in LED when the sensor is shaken:

```
/*
 * shaken sketch
```

```
 * tilt sensor connected to pin 2
 * using the built-in LED
 */

const int tiltSensorPin = 2;
const int ledPin = LED_BUILTIN;
int tiltSensorPreviousValue = 0;
int tiltSensorCurrentValue  = 0;
long lastTimeMoved = 0;
int shakeTime = 50;

void setup()
{
  pinMode (tiltSensorPin, INPUT_PULLUP);
  pinMode (ledPin, OUTPUT);
}

void loop()
{
  tiltSensorCurrentValue = digitalRead(tiltSensorPin);
  if (tiltSensorPreviousValue != tiltSensorCurrentValue)
  {
    lastTimeMoved = millis();
    tiltSensorPreviousValue = tiltSensorCurrentValue;
  }

  if (millis() - lastTimeMoved < shakeTime){
    digitalWrite(ledPin, HIGH);
  }
  else {
    digitalWrite(ledPin, LOW);
  }
}
```

Many mechanical switch sensors can be used in similar ways. A float switch can turn on when the water level in a container rises to a certain level (similar to the way a float valve works in a toilet cistern). A pressure pad such as the one used in shop entrances can be used to detect when someone stands on it. If your sensor turns a digital signal on and off, something similar to this recipe's sketch will be suitable.

## See Also

contains background information on using switches with Arduino.

has more on using the `millis` function to determine delay.

# 6.3 Detecting Light

## Problem

You want to detect changes in light levels. You may want to detect a change when something passes in front of a light detector or to measure the light level—for example, detecting when a room is getting too dark.

## Solution

The easiest way to detect light levels is to use a photoresistor, also known as a light-dependent resistor (LDR). This changes resistance with changing light levels, and when connected in the circuit shown in Figure 6-2 it produces a change in voltage that the Arduino analog input pins can sense.

The sketch for this recipe is simple:

```
/*
 * Light sensor sketch
 *
 * Varies the blink rate based on the measured brightness
 */
const int ledPin = LED_BUILTIN; // Built-in LED
const int sensorPin = A0;       // connect sensor to analog input 0

void setup()
{
  pinMode(ledPin, OUTPUT);  // enable output on the led pin
}

void loop()
{
  int rate = analogRead(sensorPin);    // read the analog input
  digitalWrite(ledPin, HIGH);   // set the LED on
  delay(rate);                         // wait duration dependent on light level
  digitalWrite(ledPin, LOW);    // set the LED off
```

```
    delay(rate);
  }
```

*Connecting a light-dependent resistor*

## Discussion

The circuit for this recipe is the standard way to use any sensor that changes its resistance based on some physical phenomenon (see Chapter 5 for background information on responding to analog signals). With the circuit in Figure 6-2, the voltage on analog pin 0 changes as the resistance of the photoresistor (or phototransistor) changes with varying light levels.

A circuit such as this will not give the full range of possible values from the analog input—0 to 1,023—as the voltage will not be swinging from 0 volts to 5 volts. This is because there will always be a voltage drop across each resistance, so the voltage where they meet will never reach the limits of the power supply. When using sensors such as these, it is important to check the actual values the device returns in the situation in which you will be using it. Then you have to determine how to convert them to the values you need to control whatever you are going to control. See Recipe 5.7 for more details on changing the range of values.

The photoresistor is a simple kind of sensor called a *resistive sensor*. A range of resistive sensors respond to changes in different physical characteristics.

Arduino cannot measure resistance directly, so the Solution uses a fixed-value resistor in combination with a resistive sensor to form a voltage divider like you saw back in Recipe 5.11. The analog pins read voltage, not resistance, so the only way for Arduino to measure resistance is if that resistance is somehow changing a voltage. A voltage divider uses a pair of resistors to produce an output voltage that is dependent on the relationship between the input voltage and two resistors. So, you can combine a fixed-value resistor with a component of variable resistance, such as a photoresistor, and Arduino's analog pin will see a voltage that changes based on what the photoresistor is sensing.

Similar circuits will work for other kinds of simple resistive sensors, although you may need to adjust the resistor to suit the sensor. Choosing the best resistor value depends on the photoresistor you are using and the range of light levels you want to monitor. Engineers would use a light meter and consult the datasheet for the photoresistor, but if you have a multimeter, you can measure the resistance of the photoresistor at a light level that is approximately midway in the range of illumination you want to monitor. Note the reading and choose the nearest convenient resistor to this value. You can also read the values from Arduino, print it to the serial port, and use the Serial Plotter to show the highs and lows (see Recipe 4.1).

Be aware of any artificial light sources in your environment that flicker on and off at an unusual rate, such as neon or some LED lights. Even though they turn off and on too quickly for a human to discern, these may register as low-light conditions to an Arduino. You can adjust for this by taking a moving average of the readings (you can see an example of this calculation in Recipe 6.8).

## See Also

This sketch was introduced in Recipe 1.6; see that recipe for more on this and variations on this sketch.

# 6.4 Detecting Motion of Living Things
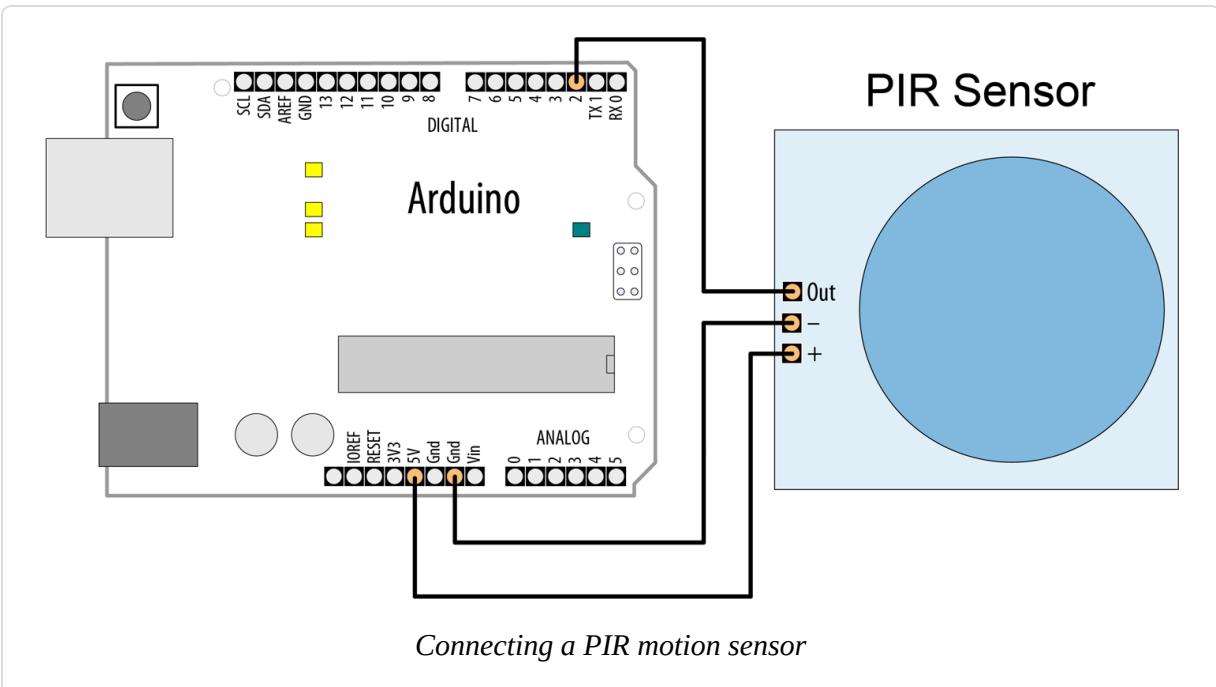
## Problem

You want to detect when people or animals are moving near a sensor.

## Solution

Use a motion sensor such as a Passive Infrared (PIR) sensor to change values on a digital pin when a living creature (or an object that radiates warmth) moves nearby.

Sensors such as the Adafruit PIR (motion) Sensor (part number 189) and the Parallax PIR Sensor (555-28027) can be easily connected to Arduino pins, as shown in Figure 6-3. Some PIR sensors, such as the SparkFun PIR Motion Sensor (SEN-13285) require a pull-up resistor on the sensor's output. If you use the pull-up resistor, you will need to use the `INPUT_PULLUP` mode and invert the logic in the sketch as described in the Discussion.

*Connecting a PIR motion sensor*

Check the datasheet for your sensor to identify the correct pins. For example, the Adafruit sensor has pins marked "OUT," "-," and "+" (for Output, GND, and +5V) and the Parallax sensor is labeled GND, VCC, and OUT.

The following sketch will light your board's built-in LED when the sensor detects motion:

```
/*
   PIR sketch
   a Passive Infrared motion sensor connected to pin 2
   lights the LED on the built-in LED
*/

const int ledPin = LED_BUILTIN;   // choose the pin for the LED
const int inputPin = 2;           // choose the input pin (for the PIR sensor)

void setup() {
  pinMode(ledPin, OUTPUT);        // declare LED as output
  pinMode(inputPin, INPUT);       // declare pin as input
}

void loop(){
  int val = digitalRead(inputPin);  // read input value
  if (val == HIGH)                  // check if the input is HIGH
```

```
  {
    digitalWrite(ledPin, HIGH);     // turn LED on if motion detected
    delay(500);
    digitalWrite(ledPin, LOW);      // turn LED off
  }
}
```

## Discussion

This code is similar to the pushbutton examples shown in . That's because the sensor acts like a switch when motion is detected. Different kinds of PIR sensors are available, and you should check the information for the one you have connected.

Some sensors, such as the Parallax and Adafruit PIR sensors, have a jumper that determines how the output behaves when motion is detected. In one mode, the output remains `HIGH` while motion is detected, or it can be set so that the output goes `HIGH` briefly and then `LOW` when triggered. The example sketch in this recipe's Solution will work in either mode.

Other sensors may go `LOW` on detecting motion. If your sensor's output pin goes `LOW` when motion is detected, change the line that checks the input value so that the LED is turned on when `LOW`:

```
if (val == LOW) // motion detected when the input is LOW
```

If your sensor's documentation indicates that it needs a pull-up resistor, you should change the code in `setup` that initializes `inputPin`:

```
pinMode(inputPin, INPUT_PULLUP); // declare pin as input with pull-up resistor
```

PIR sensors come in a variety of styles and are sensitive over different distances and angles. Careful choice and positioning can make them respond to movement in part of a room, rather than all of it. Some PIR sensors have a potentiometer that you can adjust with a screwdriver to change the PIR's sensitivity.

# 6.5 Measuring Distance

**Problem**

You want to measure the distance to something, such as a wall or someone walking toward the Arduino.

## Solution

This recipe uses the Parallax PING))) ultrasonic distance sensor to measure the distance of an object ranging from 2 centimeters to around 3 meters. It displays the distance on the Serial Monitor and flashes an LED faster as objects get closer (Figure 6-4 shows the connections):

```
/* Ping))) Sensor
 * prints distance and changes LED flash rate
 * depending on distance from the Ping))) sensor
 */

const int pingPin = 5;
const int ledPin = LED_BUILTIN; // LED pin

void setup()
{
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  int cm = ping(pingPin);
  Serial.println(cm);

  digitalWrite(ledPin, HIGH);
  delay(cm * 10); // each centimeter adds 10 ms delay
  digitalWrite(ledPin, LOW);
  delay(cm * 10);
}

// Measure distance and return the result in centimeters
int ping(int pingPin)
{
  long duration; // This will store the measured duration of the pulse
```
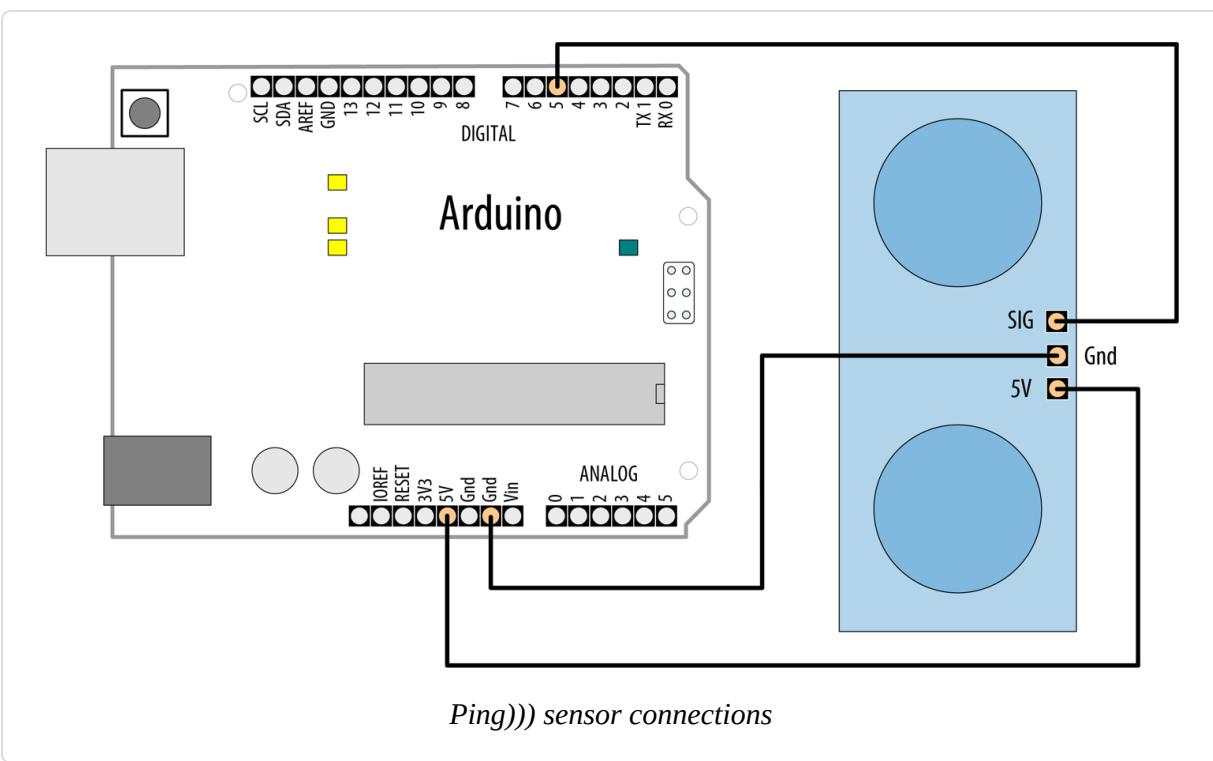
```
  // Set the pingPin to output.
  pinMode(pingPin, OUTPUT);
  digitalWrite(pingPin, LOW); // Stay low for 2μs to ensure a clean pulse
  delayMicroseconds(2);

  // Send a pulse of 5μs
  digitalWrite(pingPin, HIGH);
  delayMicroseconds(5);
  digitalWrite(pingPin, LOW);

  // Set the pingPin to input and read the duration of the pulse.
  pinMode(pingPin, INPUT);
  duration = pulseIn(pingPin, HIGH);

  // convert the time into a distance
  return duration / 29 / 2;
}
```



*Ping))) sensor connections*

## Discussion

Ultrasonic sensors provide a measurement of the time it takes for sound to bounce off an object and return to the sensor.

The "ping" sound pulse is generated when the `pingPin` level goes `HIGH` for two microseconds. The sensor will then generate a pulse that terminates when the sound returns. The width of the pulse is proportional to the distance the sound traveled, and the sketch then uses the `pulseIn` function to measure that duration. The speed of sound is about 340 meters per second, which is 29 microseconds per centimeter. The formula for the distance of the round trip is: `duration in microseconds / 29`.

So, the formula for the one-way distance in centimeters is: `duration in microseconds / 29 / 2`. The 340 meters per second figure is the approximate speed of sound at 20°C/68°F. If your ambient temperature is significantly different, you can use a speed of sound calculator such as that hosted by the United States National Weather Service.

A lower-cost alternative to the Parallax PING))) sensor is the HC-SR04, which is available from many suppliers and also on eBay. Although this has less accuracy and range, it can be suitable where the price is more important than performance. The HC-SR04 has separate pins to trigger the sound pulse and detect the echo. This variation on the previous sketch shows its use:

```
/* HC-SR04 Sensor
 * prints distance and changes LED flash rate
 * depending on distance from the HC-SR04 sensor
 */

const int trigPin = 5; // Pin to send the ping from
const int echoPin = 6; // Pin to read the response from
const int ledPin  = LED_BUILTIN; // LED pin

void setup()
{
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}

void loop()
{
  int cm = calculateDistance(trigPin);
```

```
    Serial.println(cm);

    digitalWrite(ledPin, HIGH);
    delay(cm * 10); // each centimeter adds 10 ms delay
    digitalWrite(ledPin, LOW);
    delay(cm * 10);

    delay(60); // datasheet recommends waiting at least 60ms between
  measurements
}

int calculateDistance(int trigPin)
{
  long duration; // This will store the measured duration of the pulse

  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);   // Stay low for 2µs to ensure a clean pulse
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10); // Send a pulse of 10µs to ensure a clean pulse
  digitalWrite(trigPin, LOW);

  // Read the duration of the response pulse
  duration = pulseIn(echoPin, HIGH);

  // convert time into distance
  return duration / 29 / 2;
}
```
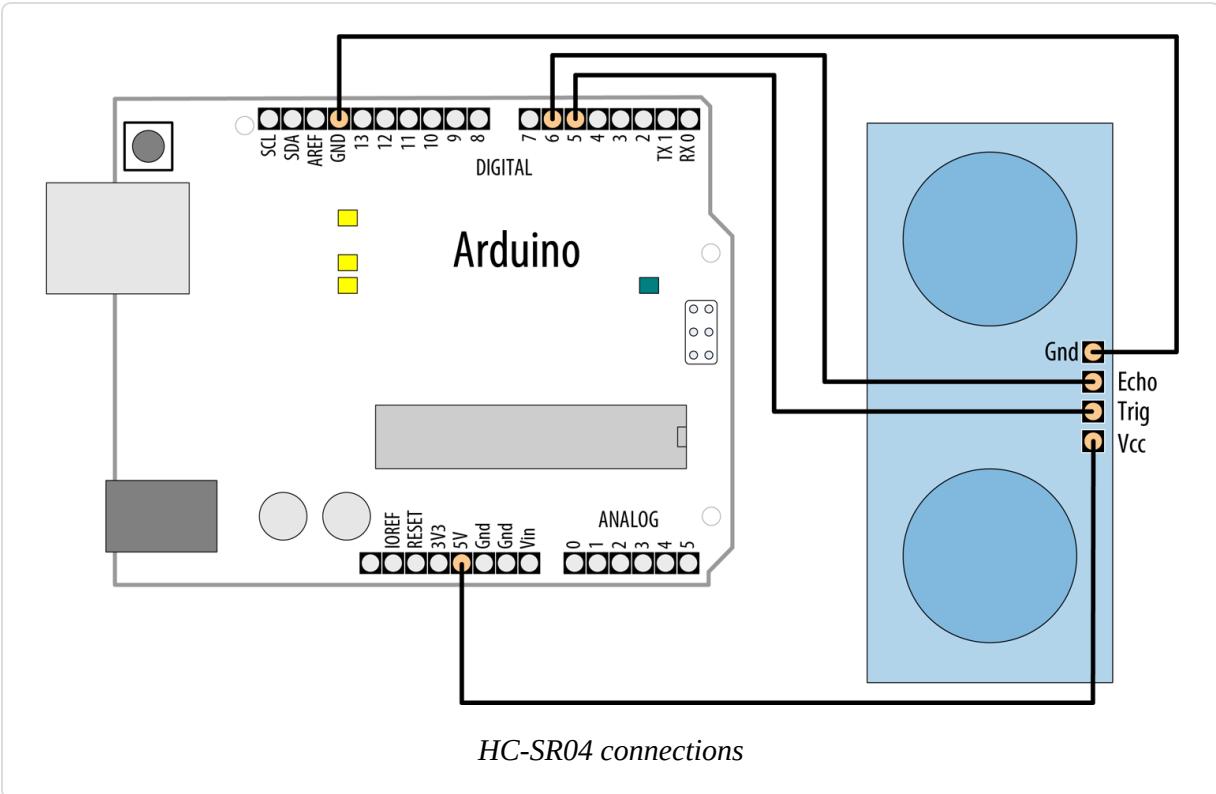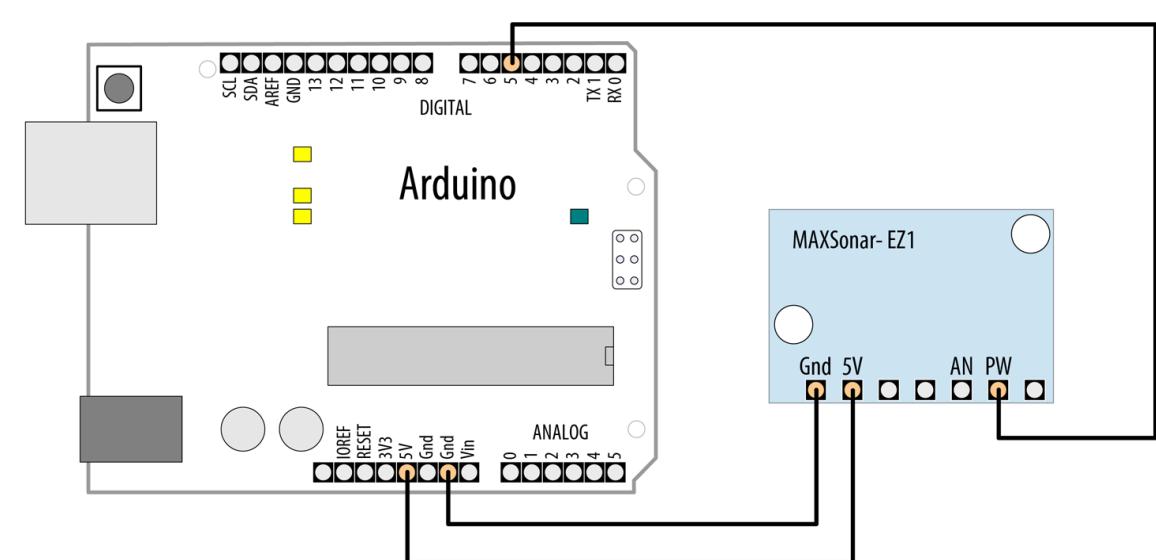
The HC-SR04 datasheet recommends at least 60 ms between measurements, but blinking the LED takes up some time, so the `delay(60);` adds more of a delay than is needed. But if you are writing code that does not add its own delay, you'll want to keep that 60 ms delay in there.

The HC-SR04 works best with 5 volts but can be used with 3.3V boards that are 5-volt tolerant, such as the Teensy 3. Figure 6-5 shows the wiring for a 5V board.

*HC-SR04 connections*

The MaxBotix EZ1 is another ultrasonic sensor that can be used to measure distance. It is easier to integrate than the Ping))) or the HC-SR04 because it does not need to be "pinged" and it can operate on 3.3 or 5 volts. It provides continuous distance information, either as an analog voltage or proportional to pulse width. Figure 6-6 shows the connections.

*Connecting EZ1 PW output to a digital input pin*

The sketch that follows uses the EZ1 pulse width (PW) output to produce output similar to that of the previous sketch:

```
/*
 * EZ1Rangefinder Distance Sensor
 * prints distance and changes LED flash rate
 * depending on distance from the sensor
 */

const int sensorPin = 5;
const int ledPin    = LED_BUILTIN;

void setup()
{
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  long value = pulseIn(sensorPin, HIGH) ;
  int cm = value / 58;        // pulse width is 58 microseconds per cm
  Serial.println(cm);

  digitalWrite(ledPin, HIGH);
  delay(cm * 10 ); // each centimeter adds 10 ms delay
  digitalWrite(ledPin, LOW);
```

```
    delay( cm * 10);

    delay(20);
}
```

The EZ1 is powered through +5V and ground pins and these are connected to the respective Arduino pins. Connect the EZ1 PW pin to Arduino digital pin 5. The sketch measures the width of the pulse with the `pulseIn` command. The width of the pulse is 58 microseconds per centimeter, or 147 microseconds per inch.

> **NOTE**
>
> You may need to add a capacitor across the +5V and GND lines to stabilize the power supply to the sensor if you are using long connecting leads. If you get erratic readings, connect a 10 uF capacitor at the sensor (see Appendix C for more on using decoupling capacitors).

You can also obtain a distance reading from the EZ1 through its analog output—connect the AN pin to an analog input and read the value with `analogRead`. The following code prints the analog input converted to cm:

```
int value = analogRead(A0);
float mv = (value / 1024.0) * 5000 ;
float inches  =  mv / 9.8; // 9.8mv per inch per datasheet
float cm = inches * 2.54;
Serial.print("in: "); Serial.println(inches);
Serial.print("cm: "); Serial.println(cm);
```

The value from `analogRead` is around 4.8mV per unit (see Recipe 5.6 for more on `analogRead`), and according to the datasheet, the EZ1 output is 9.8mV/inch when powered at 5V, and 6.4mV/inch at 3.3V. Multiply the result in inches by 2.54 to get the distance in centimeters.

## See Also

Recipe 5.6 explains how to convert readings from `analogInput` into voltage values.

The Arduino reference for `pulseIn`

# 6.6 Measuring Distance Precisely

**Problem**

You want to measure how far objects are from the Arduino with more precision than in Recipe 6.5.

## Solution

Time of flight distance sensors use a tiny laser and sensor to measure how long it takes for a laser light signal to return to it. While they have a much more narrow field of view than the ultrasonic sensors you saw in Recipe 6.5, laser-based time of flight sensors can be more precise. However, time of flight sensors typically have a smaller range. For example, while the HC-SR04 has a range of 2 cm to 4 meters, the VL6180X time of flight sensor can measure 5 cm to 10 cm. This sketch provides similar functionality to Recipe 6.5, but it uses the VL6180X Time of Flight Distance Ranging Sensor from Adafruit (product ID 3316). Figure 6-7 shows the connections. To use this sketch, you'll need to install the Adafruit_VL6180X library (see Recipe 16.2):

```
/* tof-distance sketch
 * prints distance and changes LED flash rate based on distance from sensor
 */

#include <Wire.h>
#include "Adafruit_VL6180X.h"

Adafruit_VL6180X sensor = Adafruit_VL6180X();

const int ledPin  = LED_BUILTIN; // LED pin

void setup() {
  Serial.begin(9600);
  while (!Serial);

  if (! sensor.begin()) {
    Serial.println("Could not initialize VL6180X");
```
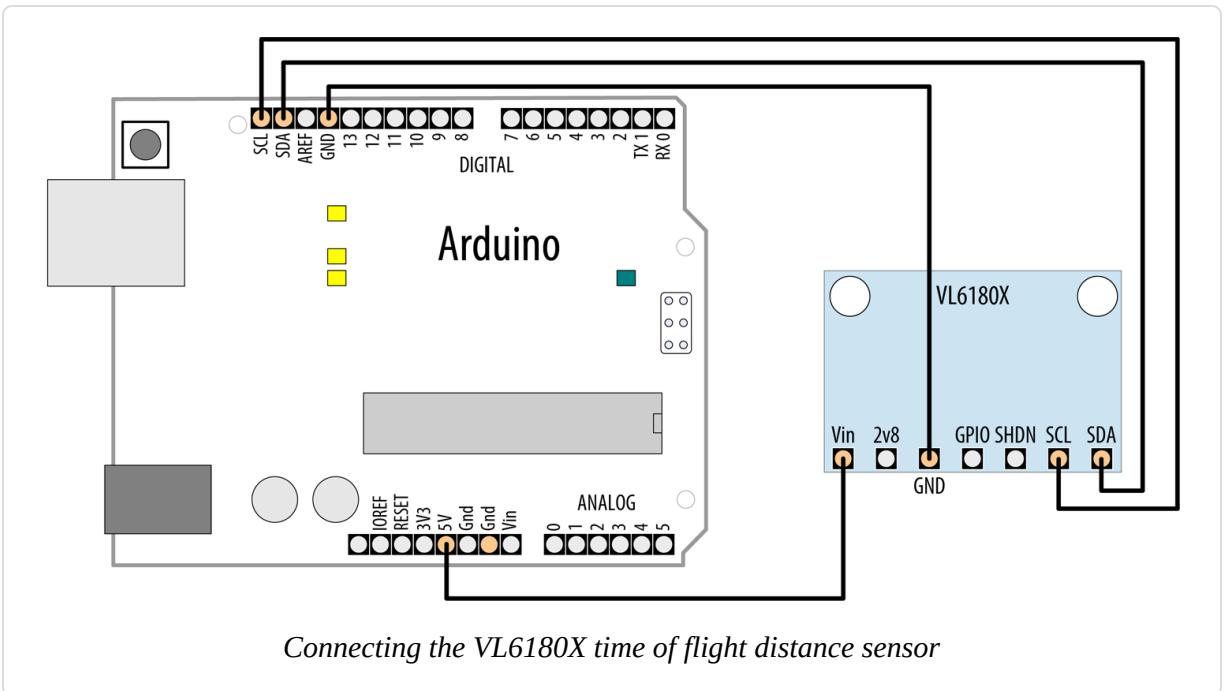
```
    while (1);
  }
}

void loop() {

  // Read the range and check the status for any errors
  byte cm     = sensor.readRange();
  byte status = sensor.readRangeStatus();

  if (status == VL6180X_ERROR_NONE)
  {
    Serial.println(cm);

    digitalWrite(ledPin, HIGH);
    delay(cm * 10); // each centimeter adds 10 ms delay
    digitalWrite(ledPin, LOW);
    delay(cm * 10);

  }
  else
  {
    // Major errors are worth mentioning
    if  ((status >= VL6180X_ERROR_SYSERR_1) &&
         (status <= VL6180X_ERROR_SYSERR_5))
    {
      Serial.println("System error");
    }
  }
  delay(50);
}
```

*Connecting the VL6180X time of flight distance sensor*

## Discussion

The VL6180X sensor uses the I2C protocol (see Chapter 13) to communicate, which requires a connection between the Arduino and the sensor's SCL and SDA pins. The sketch includes the Wire library, which provides support for I2C, and also includes the Adafruit_VL6180X library to provide functions for working with the sensor. Before the `setup` function, the sketch defines an object (`sensor`) to represent the sensor, and later initializes it in `setup`.

The `setup` function initializes the serial port and attempts to initialize the sensor. If that fails, it prints an error message to the serial port, and stops running the sketch by entering an infinite `while` loop.

On each run through the `loop`, the sketch reads the range and also checks the sensor status to make sure it's not in an error state. If it gets a good reading, it displays the distance to the serial port and blinks the LED at a rate determined by the distance it measured. The example included with the Adafruit_VL6180X library has a more exhaustive check of all the possible

error states. With the exception of the system errors that this sketch checks for, most errors are transient and will be corrected on a subsequent reading.

## See Also

Detailed comparisons of ultrasonic, LED, and laser-based distance sensors are available from DIY Projects and SparkFun.

# 6.7 Detecting Vibration

## Problem

You want to respond to vibration; for example, when a door is knocked on.

## Solution

A Piezo sensor responds to vibration. It works best when connected to a larger surface that vibrates. Figure 6-8 shows the connections:

```
/* piezo sketch
 * lights an LED when the Piezo is tapped
 */

const int sensorPin = 0;  // the analog pin connected to the sensor
const int ledPin    = LED_BUILTIN;    // pin connected to LED
const int THRESHOLD = 100;

void setup()
{
    pinMode(ledPin, OUTPUT);
}

void loop()
{
  int val = analogRead(sensorPin);
  if (val >= THRESHOLD)
  {
    digitalWrite(ledPin, HIGH);
    delay(100);  // to make the LED visible
  }
  else
```
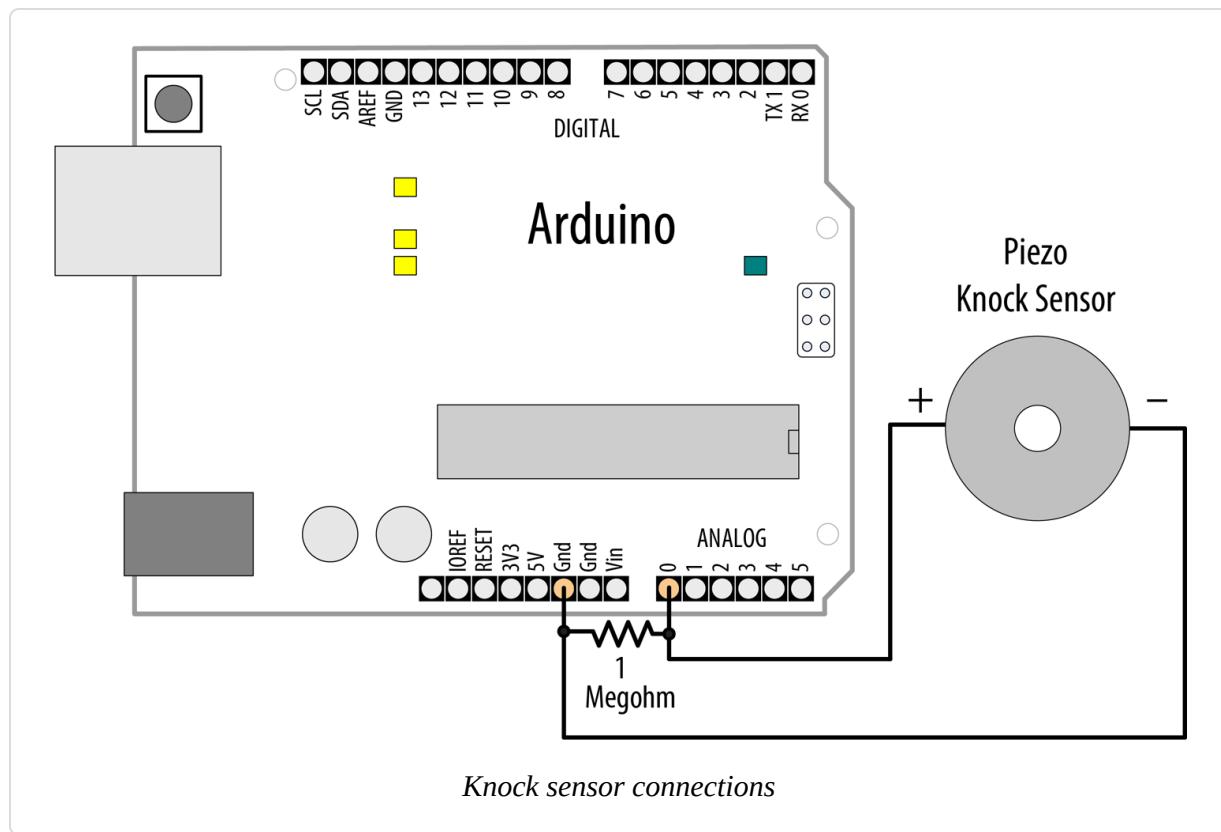
```
        digitalWrite(ledPin, LOW);
}
```

## Discussion

A Piezo sensor, also known as a knock sensor, produces a voltage in response to physical stress. The more it is stressed, the higher the voltage. The Piezo is polarized and the positive side (usually a red wire or a wire marked with a "+") is connected to the analog input; the negative wire (usually black or marked with a "–") is connected to ground. A high-value resistor (1 megohm) is connected across the sensor. The resistor is included to protect the Arduino pins against excessive current or voltage.



*Knock sensor connections*

The voltage is detected by Arduino `analogRead` to turn on an LED (see Chapter 5 for more about the `analogRead` function). The `THRESHOLD` value determines the level from the sensor that will turn on the LED, and you can decrease or increase this value to make the sketch more or less sensitive.

Piezo sensors can be bought in plastic cases or as bare metal disks with two wires attached. The components are the same; use whichever fits your project best.

Some sensors, such as the Piezo, can be driven by the Arduino to produce the thing that they can sense. Chapter 9 has more about using a Piezo to generate sound.
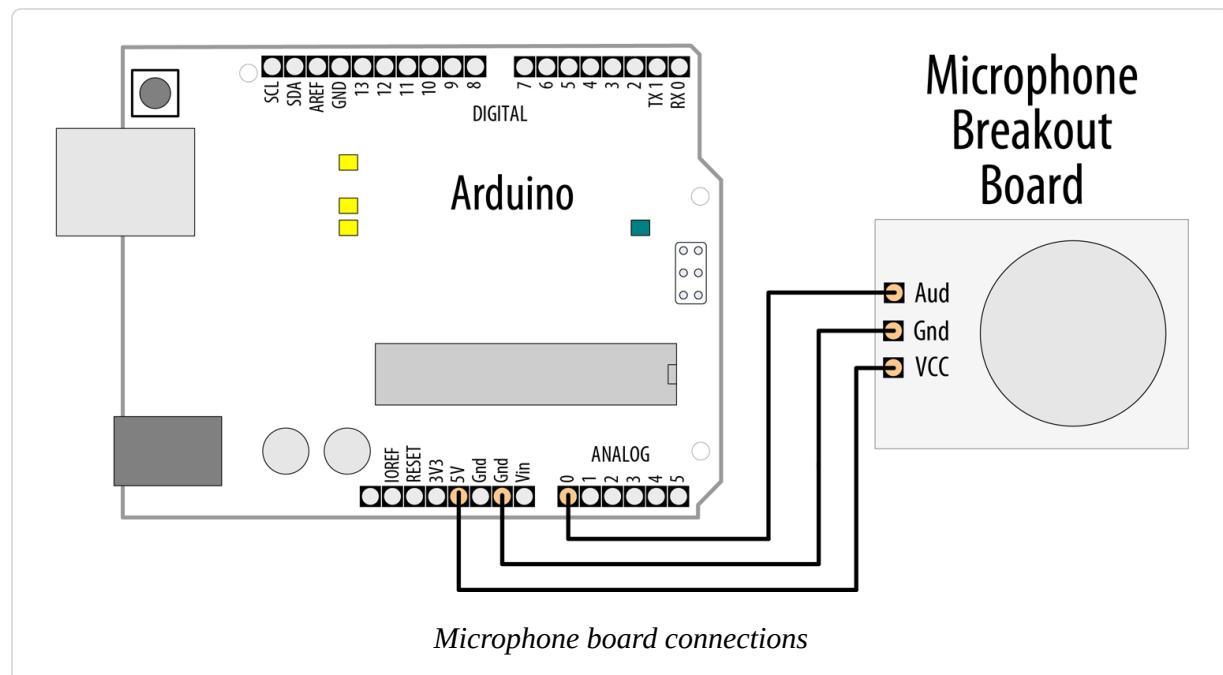
# 6.8 Detecting Sound

## Problem

You want to detect sounds such as clapping, talking, or shouting.

## Solution

This recipe uses the BOB-12758 breakout board for the Electret Microphone (SparkFun). Connect the board as shown in Figure 6-9 and load the code to the board. If you are using a 3.3V board, you should connect the microphone's VCC pin to 3.3V instead of 5V.



*Microphone board connections*

The built-in LED will turn on when you clap, shout, or play loud music near the microphone. You may need to adjust the threshold—use the Serial Monitor to view the high and low values, and change the threshold value so that it is between the high values you get when noise is present and the low values when there is little or no noise. Upload the changed code to the board and try again:

```
/* microphone sketch
 * SparkFun breakout board for Electret Microphone is connected to analog pin
0
 */

const int micPin = A0;              // Microphone connected to analog 0
const int ledPin = LED_BUILTIN;     // the code will flash the built-in LED
const int middleValue = 512;        // the middle of the range of analog values
const int numberOfSamples = 128;    // how many readings will be taken each time

int sample;                         // the value read from microphone each time
long signal;                        // the reading once you have removed DC
offset
long newReading;                    // the average of that loop of readings

long runningAverage = 0;            // the running average of calculated values
const int averagedOver = 16;        // how quickly new values affect running avg
                                    // bigger numbers mean slower

const int threshold = 400;          // at what level the light turns on

void setup()
{
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  long sumOfSquares = 0;
  for (int i=0; i<numberOfSamples; i++) { // take many readings and average
them
    sample = analogRead(micPin);          // take a reading
    signal = (sample - middleValue);      // work out its offset from the
center
    signal *= signal;                     // square it
    sumOfSquares += signal;               // add to the total
  }
```

```
    newReading = sumOfSquares/numberOfSamples;

    // calculate running average
    runningAverage=(((averagedOver-1)*runningAverage)+newReading)/averagedOver;

    Serial.print("new:"); Serial.print(newReading);
    Serial.print(",");
    Serial.print("running:"); Serial.println(runningAverage);

    if (runningAverage > threshold){        // is average more than the
threshold?
        digitalWrite(ledPin, HIGH);         // if it is turn on the LED
    } else {
        digitalWrite(ledPin, LOW);          // if it isn't turn the LED off
    }

}
```
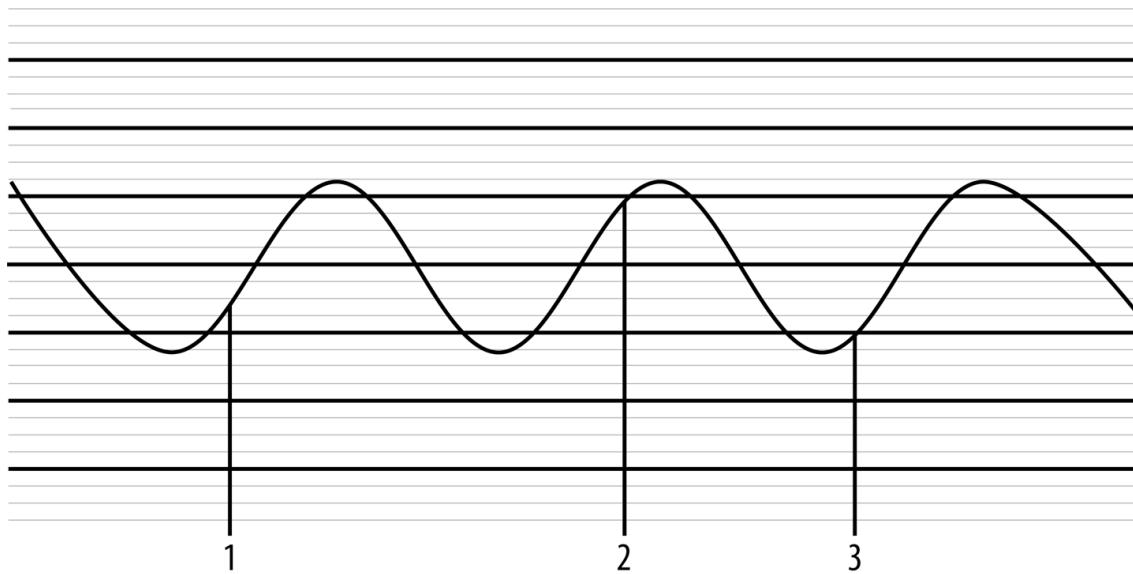
## Discussion

A microphone produces very small electrical signals. If you connected it straight to the pin of an Arduino, you would not get any detectable change. The signal needs to be amplified first to make it usable by Arduino. The SparkFun board has the microphone with an amplifier circuit built in to amplify the signal to a level readable by Arduino.

Because you are reading an audio signal in this recipe, you will need to do some additional calculations to get useful information. An audio signal changes fairly quickly, and the value returned by `analogRead` will depend on what point in the undulating signal you take a reading. If you are unfamiliar with using `analogRead`, see Chapter 5 and Recipe 6.3. An example waveform for an audio tone is shown in Figure 6-10. As time changes from left to right, the voltage goes up and down in a regular pattern. If you take readings at the three different times marked on it, you will get three different values. If you used this to make decisions, you might incorrectly conclude that the signal got louder in the middle.
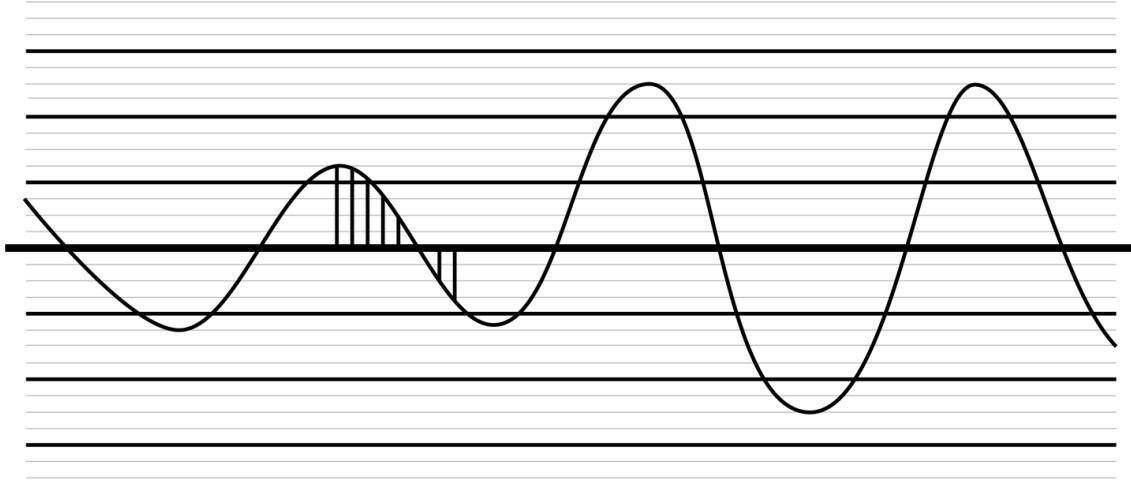
An accurate measurement requires multiple readings taken close together. The peaks and troughs increase as the signal gets bigger. The difference

between the bottom of a trough and the top of a peak is called the *amplitude* of the signal, and this increases as the signal gets louder.



*Audio signal measured in three places*

To measure the size of the peaks and troughs, you measure the difference between the midpoint voltage and the levels of the peaks and troughs. You can visualize this midpoint value as a line running midway between the highest peak and the lowest trough, as shown in Figure 6-11. The line represents the DC offset of the signal (it's the DC value when there are no peaks or troughs). If you subtract the DC offset value from your `analogRead` values, you get the correct reading for the signal amplitude.
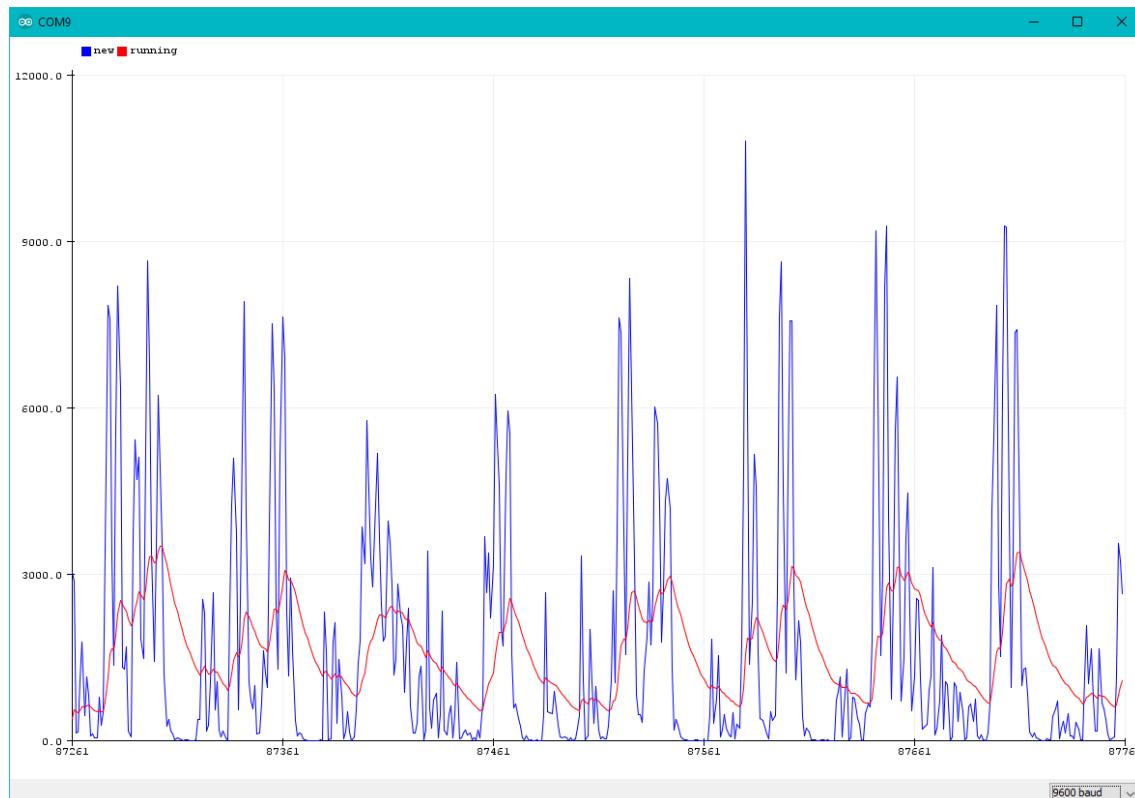
*Audio signal showing DC offset (signal midpoint)*

As the signal gets louder, the average size of these values will increase, but as some of them are negative (where the signal has dropped below the DC offset), they will cancel each other out, and the average will tend to be zero. To fix that, we square each value (multiply it by itself). This will make all the values positive, and it will increase the difference between small changes, which helps you evaluate changes as well. The average value will now go up and down as the signal amplitude does.

To do the calculation, we need to know what value to use for the DC offset. To get a clean signal, the amplifier circuit for the microphone will have been designed to have a DC offset as close as possible to the middle of the possible range of voltage so that the signal can get as big as possible without distorting. The code assumes this and uses the value 512 (right in the middle of the analog input range of 0 to 1,023). Each time the sketch takes the average of the squared values to calculate a new reading, the sketch updates the running average. The running average is calculated by multiplying the current running average by `averagedOver - 1`. With `averagedOver` set to 16, this weights the current running average by 15. Next, the sketch adds the new reading in (a weighting of 1), and divides by `averagedOver` to get the weighted average, which yields the new running average: `(currentAverage * 15 + newReading)/16.`

The sketch prints the values of the new reading and the running average in such a way that you can view them with the Serial Plotter (Tools → Serial Plotter). You can see the relationship between the new reading and the running average in Figure 6-12. The running average is less spiky which means the LED will stay on long enough for someone to notice it, rather than just flickering briefly during a spike.



*Readings and moving average displayed in the Serial Plotter*

The values of variables at the top of the sketch can be varied if the sketch does not trigger well for the level of sound you want.

The `numberOfSamples` is set at 128—if it is set too small, the average may not adequately cover complete cycles of the waveform and you will get erratic readings. If the value is set too high, you will be averaging over too long a time, and a very short sound might be missed as it does not produce enough change once a large number of readings are averaged. It could also start to introduce a noticeable delay between a sound and the light going on.

Constants used in calculations, such as `numberOfSamples` and `averagedOver`, are set to powers of 2 (128 and 16, respectively). Try to use values evenly divisible by two for these to give you the fastest performance (see Chapter 3 for more on math functions).

While the values as calculated work well for detecting sound levels, you can change the sketch so it lines up with standard methods for measuring sound levels (decibels). First, you'll need to change the way `newReading` is calculated to take the square root of the average (this is called a Root Mean Square, or RMS). Next, you'll want to take the common logarithm of both values and multiply it by 20 to get decibels. This is unlikely to yield an accurate measurement without calibration, but it is a starting point:

```
newReading = sqrt(sumOfSquares/numberOfSamples);

// calculate running average
runningAverage=(((averagedOver-1)*runningAverage)+newReading)/averagedOver;

Serial.print("new:"); Serial.print(20*log10(newReading));
Serial.print(",");
Serial.print("running:"); Serial.println(20*log10(runningAverage));
```

You will also need to modify the threshold to something much lower:

```
const int threshold = 30;        // at what level the light turns on
```

# 6.9 Measuring Temperature

## Problem
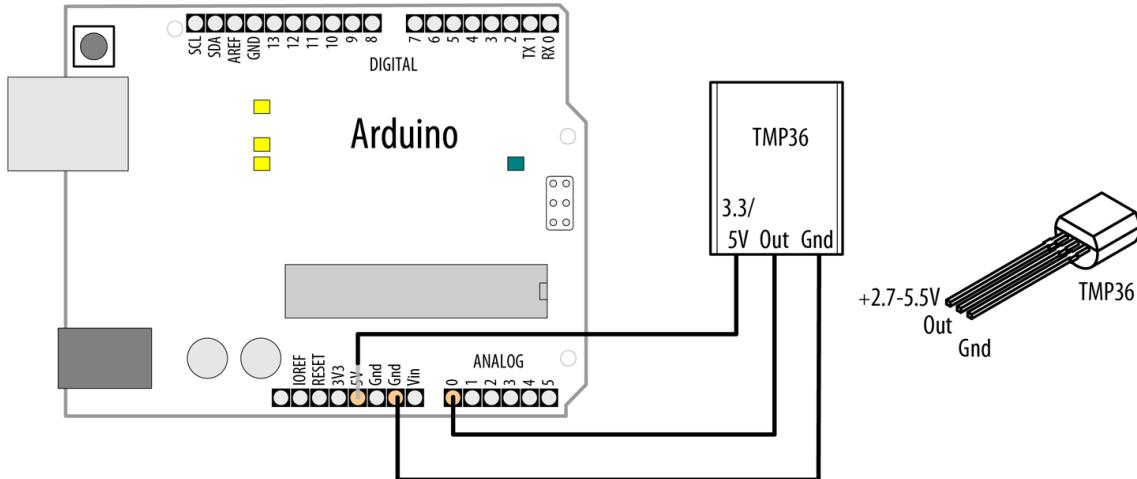
You want to display the temperature or use the value to control a device; for example, to switch something on when the temperature reaches a threshold.

## Solution

This recipe displays the temperature in Fahrenheit and Celsius (Centigrade) using the popular TMP36 heat detection sensor. The sensor looks similar to a transistor and is connected as shown in Figure 6-13.

*Connecting the TMP36 temperature sensor*

```
/*
 * tmp36 sketch
 * prints the temperature to the Serial Monitor
 * and turns on the LED when a threshold is reached
 */

const int inPin  = A0; // analog pin
const int ledPin = LED_BUILTIN;
const int threshold = 80; // Turn on the LED over 80F

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

void loop()
{
  int value = analogRead(inPin);

  // Use 3300 instead of 5000 for 3.3V boards
  float millivolts = (value / 1024.0) * 5000;
```

```
    // 10mV per degree Celsius with a 500mv offset
    float celsius = (millivolts - 500) / 10;
    float fahrenheit = (celsius * 9)/ 5 + 32;

    Serial.print("C:");
    Serial.print(celsius);
    Serial.print(",");
    Serial.print("F:");
    Serial.println( fahrenheit );  //  converts to fahrenheit

    if (fahrenheit > threshold){    // is the temperature over the threshold?
      digitalWrite(ledPin, HIGH);  // if it is turn on the LED
    } else {
      digitalWrite(ledPin, LOW);   // if it isn't turn the LED off
    }
    delay(1000); // wait for one second

  }
```

## Discussion
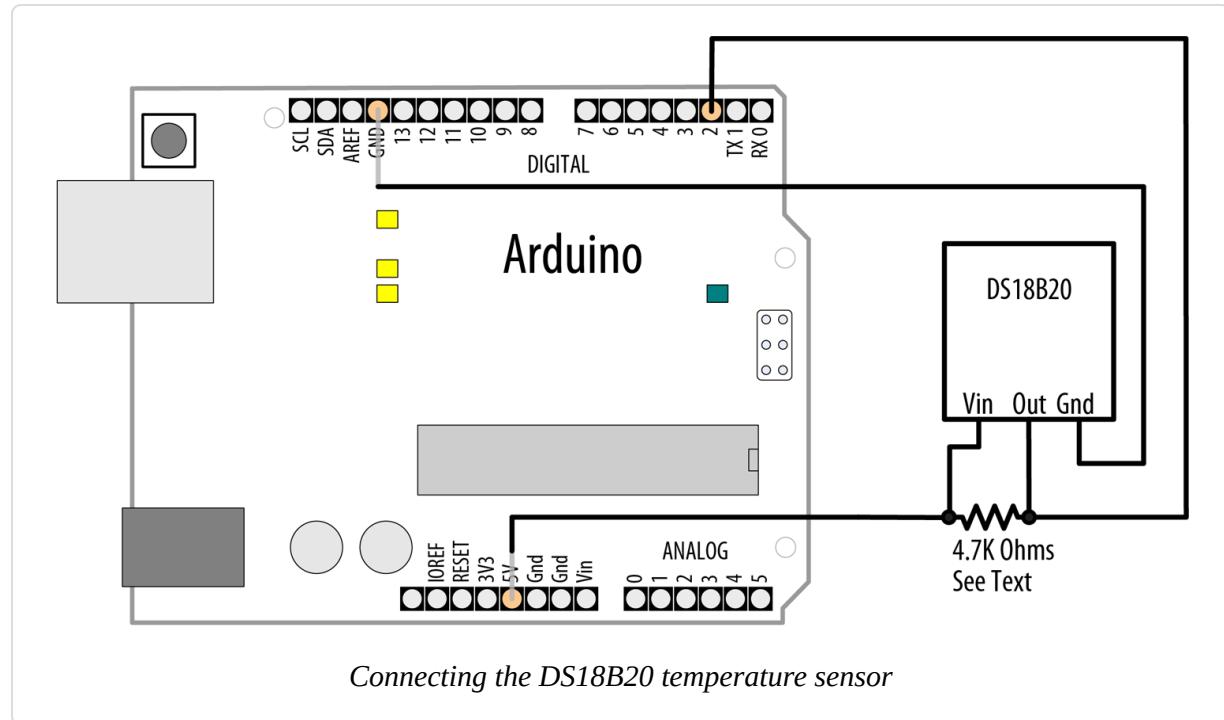
The TMP36 temperature sensor produces an analog voltage directly proportional to temperature with an output of 1 millivolt (mV) per 0.1°C (10mV per degree), but with a 500mV offset.

The sketch converts the `analogRead` values into millivolts (see Chapter 5). It then subtracts 0.5V (500mV), the offset voltage specified in the TMP36 datasheet, and then divides the result by 10 to get degrees C. If the temperature exceeds the threshold value, the sketch lights the onboard LED. You can easily get the sensor to go over F80 by holding the sensor between two fingers, but avoid touching your fingers to the sensor's leads so as to not interfere with the electrical signaling.

There are many temperature sensors available, but an interesting alternative is the waterproof DS18B20 digital temperature sensor (Adafruit part 381, SparkFun part SEN-11050, available from other suppliers as well). It is wired and used differently than the TMP36.

The DS18B20 is based on the 1-Wire protocol pioneered by Dallas Semiconductor (now Maxim), and requires two libraries. The first is the OneWire library. There are several libraries available with OneWire in their

name, so be sure to choose the OneWire library by Jim Studt, Tom Pollard, et al. You will also need the DallasTemperature library. You can install both using the Library Manager (see Recipe 16.2). To wire the DS18B20, connect the red wire to 5V (or 3.3V if on a 3.3V board), black to ground, and the signal wire (yellow, white, or some other color) to digital pin 2 with a 4.7K resistor between the signal and power (5V or 3.3V) pin, as shown in Figure 6-14.



*Connecting the DS18B20 temperature sensor*

Here's the sketch for reading the temperature:

```
/* DS18B20 temperature
 * Reads temperature from waterproof sensor probe
 */
#include <OneWire.h>
#include <DallasTemperature.h>

#define ONE_WIRE_BUS 2  // The pin that the sensor wire is connected to

const int ledPin = LED_BUILTIN;
const int threshold = 80; // Turn on the LED over 80F

OneWire oneWire(ONE_WIRE_BUS);       // Prepare the OneWire connection
DallasTemperature sensors(&oneWire); // Declare the temp sensor object
```

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

  // Initialize the sensor
  sensors.begin();
}

void loop(void)
{

  sensors.requestTemperatures(); // Request a temperature reading

  // Retrieve the temperature reading in F and C
  float fahrenheit = sensors.getTempFByIndex(0);
  float celsius    = sensors.getTempCByIndex(0);

  // Display the temperature readings in a Serial Plotter-friendly format
  Serial.print("C:"); Serial.print(celsius);
  Serial.print(",");
  Serial.print("F:"); Serial.println(fahrenheit);

  if (fahrenheit > threshold){    // is the temperature over the threshold?
    digitalWrite(ledPin, HIGH);  // if it is turn on the LED
  } else {
    digitalWrite(ledPin, LOW);   // if it isn't turn the LED off
  }
  delay(1000);

}
```

The sketch pulls in the header files for each library, and initializes the data structures needed to work with the 1-Wire protocol and with the sensor. Inside the loop, the sketch requests a temperature reading, and then reads the temperature in Celsius, then Fahrenheit. Note that you do not need to perform any arithmetic conversion on the results you get from the sensor. Everything is handled by the library. Note also that you do not need to make any code changes (but make sure you wire the sensor's power to 3.3V, not 5V) when you use a 3.3V board.

## See Also

The TMP36 datasheet

The DS18B20 datasheet

# 6.10 Reading RFID (NFC) Tags

**Problem**
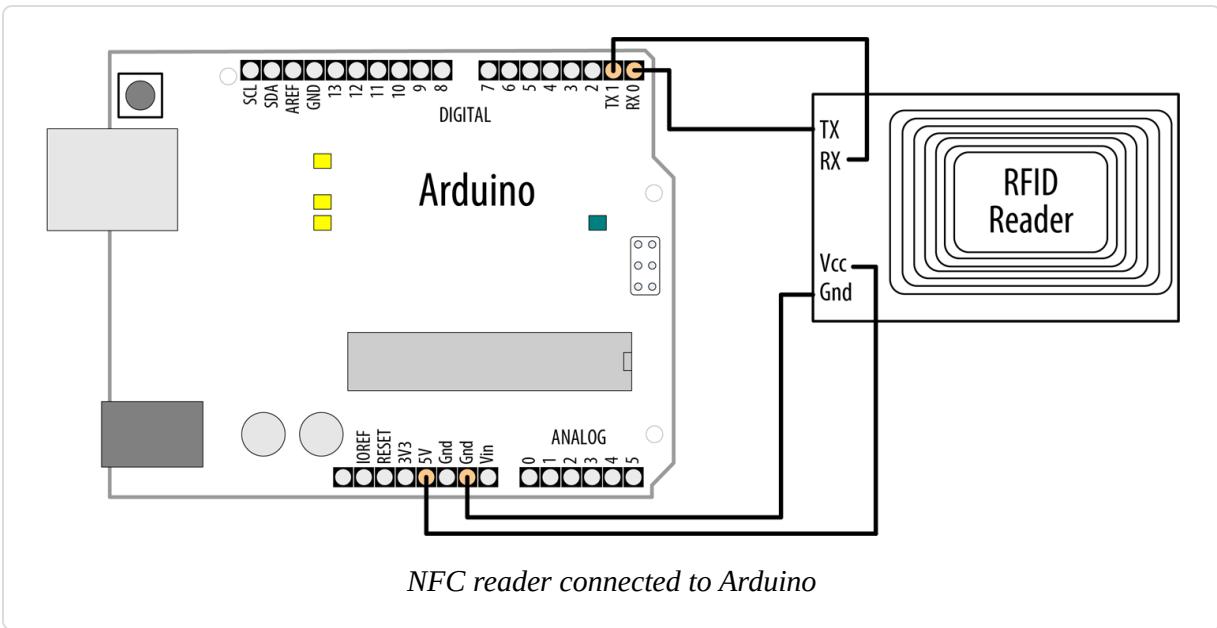
You want to read an RFID/NFC tag and respond to specific IDs.

## Solution

Figure 6-15 shows a PN532 NFC reader connected to Arduino over serial pins (TX and RX). PN532 NFC readers are available from a number of suppliers. The Seeed Studio Grove NFC reader (part 113020006) is connected as shown in the diagram. You can also find a PN532 reader in a shield form factor (SeeedStudio part 113030001, Adafruit part 789). You will need to install the SeeedStudio Seeed_Arduino_NFC library (see Recipe 16.2). The Seeed library includes a from modified version of the NDEF library so you do not need to install that library.

---

**NOTE**

PN532 readers work with 13.56 MHz MIFARE Classic and MIFARE Ultralight tags. If you are using a different reader, check the documentation for information on wiring the reader to Arduino and for example code.

---

*NFC reader connected to Arduino*

The sketch reads an NFC tag and displays its unique ID:

```
/* NFC Tag Scanner - Serial
 * Look for an NFC tag and display its unique identifier.
 */

#include <NfcAdapter.h>
#include <PN532/PN532/PN532.h>
#include <PN532/PN532_HSU/PN532_HSU.h>

PN532_HSU pn532hsu(Serial1);
NfcAdapter nfc(pn532hsu);

void setup()
{
  Serial.begin(9600);
  nfc.begin(); // Initialize the NFC reader
}

void loop()
{
  Serial.println("Waiting for a tag");
  if (nfc.tagPresent()) // If the reader sees an NFC tag
  {
    NfcTag tag = nfc.read(); // read the NFC tag
    Serial.println(tag.getUidString()); // Display its id
  }
  delay(500);
}
```

## Discussion

NFC (Near-Field Communication) is a specialized variant of RFID (Radio Frequency Identification) technology that operates at a frequency of 13.56 MHz and supports a data format called NDEF (NFC Data Exchange Format). NDEF provides a variety of structured messages you can store on a *tag*, a small electronic device that can be embedded in cards, stickers, keychain fobs, and other objects. The tag consists of a relatively large antenna that receives signals from an RFID/NFC reader. The reader can be embedded in a computer or a mobile phone, or can be a module that you connect to your Arduino (as with the PN532 module). When the tag receives the signal, it harvests enough energy from it to energize the circuitry on the tag, which responds to the signal by transmitting the information contained in its memory. There are also tags that have their own power, such as a motor vehicle transponder used in automated toll payment systems. Such tags are known as *active tags*, while the energy-harvesting type is called a *passive tag*.

An NDEF tag transmits a collection of data when it is activated by a reader. This data includes information that identifies the tag, along with any information stored on the tag. The Solution uses Don Coleman's NDEF library to simplify reading the tag data.

The code shown in the Solution will work with the Seeed Studio Grove NFC module connected over `Serial1`. It uses the USB serial connection to send information that you can view in the Serial Monitor. `Serial1` is not present on the Arduino Uno (see "Serial Hardware"), which means you would need to use SoftwareSerial with this module because on the Uno (and compatible boards based on the ATmega328), USB Serial and the TX/RX pins are shared, so these boards cannot talk to a serial device and over the USB Serial connection at the same time. See Recipe 4.11 for information on SoftwareSerial. You can also reconfigure the Grove NFC module to use I2C.

The Seeed Studio NFC shield communicates over SPI. If you want to use it with the Seeed Studio NFC shield, change the lines at the top of the sketch to:

```
#include <SPI.h>
#include <NfcAdapter.h>
#include <PN532/PN532/PN532.h>
#include <PN532/PN532_SPI/PN532_SPI.h>

PN532_SPI pn532spi(SPI, 10);
NfcAdapter nfc = NfcAdapter(pn532spi);
```

If you want to use it with the Adafruit shield or the Grove NFC module in
I2C mode, change the lines at the top of the sketch to:

```
#include <Wire.h>
#include <NfcAdapter.h>
#include <PN532/PN532/PN532.h>
#include <PN532/PN532_I2C/PN532_I2C.h>

PN532_I2C pn532i2c(Wire);
NfcAdapter nfc = NfcAdapter(pn532i2c);
```

You can also read any message on that tag and write your own message
(assuming the tag has not been locked) using the NDEF library. If you
replace the `loop` function with the following, the sketch will read the tag,
and then use the `NfcTag` object's `print` function to display the tag ID and
any message on it. It will then display a countdown. If you leave the tag in
place, it will write a URL to the tag. If you have an NFC-enabled mobile
phone, you can hold the tag up to the phone and it should open the URL in
a web browser:

```
void loop()
{
  Serial.println("Waiting for a tag");
  if (nfc.tagPresent()) // If the reader sees an NFC tag
  {
    NfcTag tag = nfc.read(); // read the NFC tag
    tag.print(); // print whatever is currently on it

    // Give the user time to avoid writing to the tag
    Serial.print("Countdown to writing the tag: 3");
    for (int i = 2; i >= 0; i--) {
      delay(1000);
      Serial.print("..."); Serial.print(i);
    }
```

```
    Serial.println();

    // Write a message to the tag
    NdefMessage message = NdefMessage();
    message.addUriRecord("http://oreilly.com");
    bool success = nfc.write(message);
    if (!success)
      Serial.println("Write failed.");
    else
      Serial.println("Success.");
  }
  delay(500);
}
```

# 6.11 Tracking Rotary Movement

## Problem

You want to measure and display the rotation of something to track its speed and/or direction.

## Solution

To sense rotary motion you can use a rotary encoder that is attached to the object you want to track. Connect the encoder as shown in Figure 6-16:

```
/*
 * Read a rotary encoder
 * This simple version polls the encoder pins
 * The position is displayed on the Serial Monitor
 */

const int encoderPinA = 3;
const int encoderPinB = 2;
const int encoderStepsPerRevolution=16;
int angle = 0;

int encoderPos = 0;
bool encoderALast = LOW;  // remembers the previous pin state

void setup()
{
  Serial.begin (9600);
  pinMode(encoderPinA, INPUT_PULLUP);
```
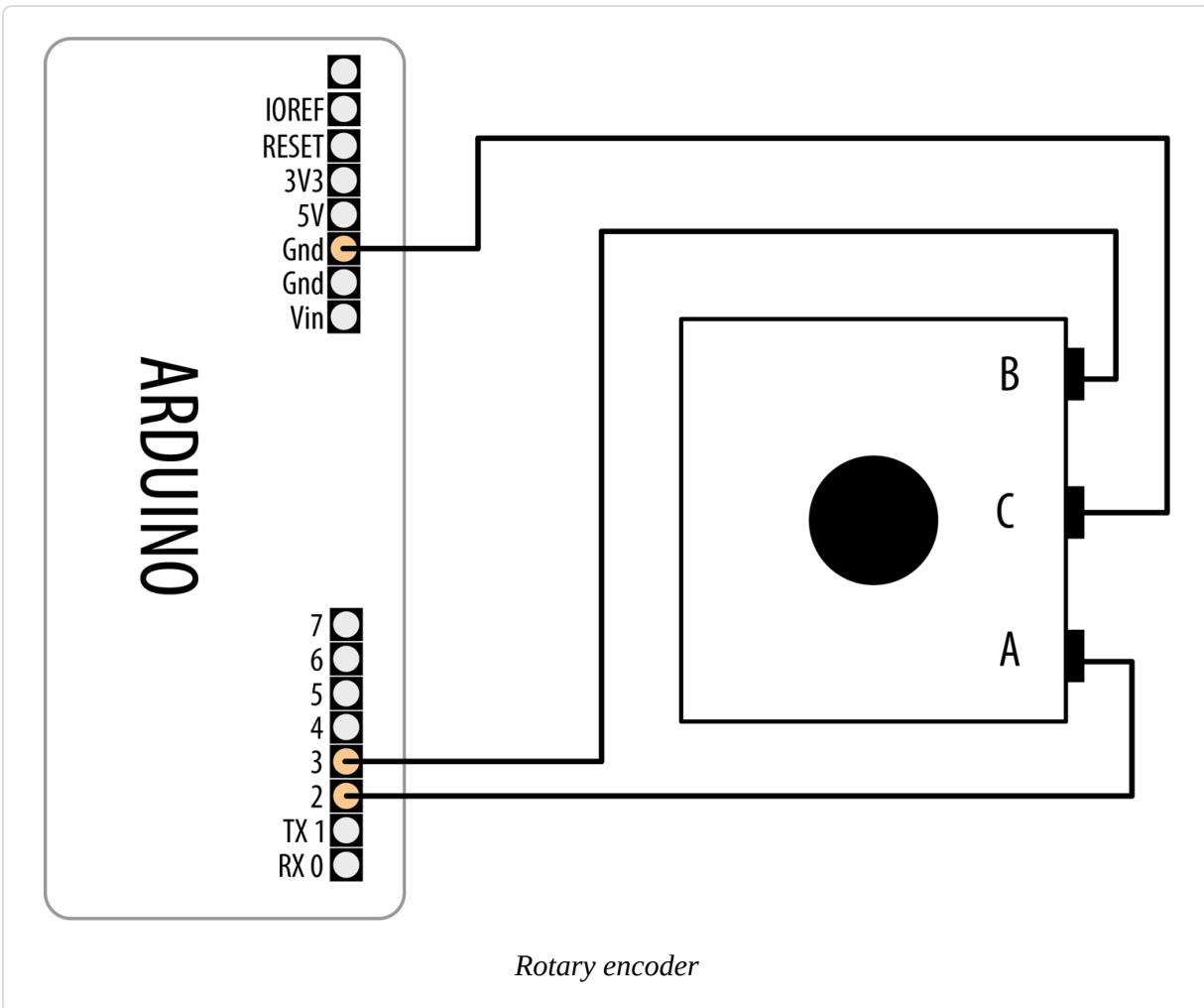
```
    pinMode(encoderPinB, INPUT_PULLUP);
}

void loop()
{
  bool encoderA = digitalRead(encoderPinA);

  if ((encoderALast == HIGH) && (encoderA == LOW))
  {
    if (digitalRead(encoderPinB) == LOW)
    {
      encoderPos--;
    }
    else
    {
      encoderPos++;
    }
    angle=(encoderPos %
encoderStepsPerRevolution)*360/encoderStepsPerRevolution;
    Serial.print (encoderPos);
    Serial.print (" ");
    Serial.println (angle);
  }

  encoderALast = encoderA;
}
```

*Rotary encoder*

## Discussion

A rotary encoder produces two signals as it is turned. Both signals alternate between HIGH and LOW as the shaft is turned, but the signals are slightly out of phase with each other. If you detect the point where one of the signals changes from HIGH to LOW, the state of the other pin (whether it is HIGH or LOW) will tell you which way the shaft is rotating.

So, the first line of code in the loop function reads one of the encoder pins:

```
int encoderA = digitalRead(encoderPinA);
```

Then it checks this value and the previous one to see if the value has just changed to LOW:

```
if ((encoderALast == HIGH) && (encoderA == LOW))
```

If it has not, the code doesn't execute the following block; it goes to the bottom of `loop`, saves the value it has just read in `encoderALast`, and goes back around to take a fresh reading.

When the following expression is `true`:

```
if ((encoderALast == HIGH) && (encoderA == LOW))
```

the code reads the other encoder pin and increments or decrements `encoderPos` depending on the value returned. It calculates the angle of the shaft (taking 0 to be the point the shaft was at when the code started running). It then sends the values down the serial port so that you can see it in the Serial Monitor.

Encoders come in different resolutions, quoted as *steps per revolution*. This indicates how many times the signals alternate between `HIGH` and `LOW` for one revolution of the shaft. Values can vary from 16 to 1,000. The higher values can detect smaller movements, and these encoders cost much more money. The value for the encoder is hardcoded in the code in the following line:

```
const int encoderStepsPerRevolution=16;
```

If your encoder is different, you need to change that to get the correct angle values.

If you get values out that don't go up and down, but increase regardless of the direction you turn the encoder, try changing the test to look for a rising edge rather than a falling one. Swap the `LOW` and `HIGH` values in the line that checks the values so that it looks like this:

```
if ((encoderALast == LOW) && (encoderA == HIGH))
```

Rotary encoders just produce an increment/decrement signal; they cannot directly tell you the shaft angle. The code calculates this, but it will be

relative to the start position each time the code runs. The code monitors the pins by *polling* (continuously checking the value of) them. There is no guarantee that the pins have not changed a few times since the last time the code looked, so if the code does lots of other things as well, and the encoder is turned very quickly, it is possible that some of the steps will be missed. For high-resolution encoders this is more likely, as they will send signals much more often as they are turned.

To work out the speed, you need to count how many steps are registered in one direction in a set time.

# 6.12 Tracking Rotary Movement in a Busy Sketch with Interrupts

## Problem

As you extend your code and it is doing other things in addition to reading the encoder, or if you want to read more than one encoder, you will find that your readings from the encoder start to get unreliable. This problem is particularly bad if the shaft rotates quickly.

## Solution

The circuit is the same as the one for Recipe 6.11. We will use a library that is optimized for reading rotary encoders. It uses Arduino's interrupt capabilities (Recipe 18.2) to respond quickly to changes in the pin states. Use the Library Manager to install the Encoder library by Paul Stoffregen (see Recipe 16.2), and run the following sketch:

```
/* Rotary Encoder library sketch
 * Read the rotary encoder with a library that uses interrupts
 * to process the encoder's activity
 */

#include <Encoder.h>

Encoder myEnc(2, 3); // On MKR boards, use pins 6, 7
```

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

long lastPosition = -999;

void loop()
{
  long currentPosition = myEnc.read();
  if (currentPosition != lastPosition) { // If the position changed
    lastPosition = currentPosition;   // Save the last position
    Serial.println(currentPosition); // print it to the Serial monitor
  }
}
```

## Discussion

With the Solution from Recipe 6.11, as your code has more things to do, the encoder pins will be checked less often. If the pins go through a whole step change before getting read, the Arduino will simply not detect that step. Moving the shaft quickly will cause more errors, as the steps will be happening more quickly.

To make sure the code responds every time a step happens, you need to use interrupts. When the interrupt condition happens (such as a pin changing state), the code jumps from wherever it is, handles the interrupt, and then returns to where it was and carries on. The Encoder library will perform best with pins that support hardware interrupts, but it will do its best with pins that do not.

On the Arduino Uno and for other boards based on the ATmega328, only two pins can be used as interrupts: pins 2 and 3. See this list of which pins are supported on specific boards. You declare and initialize a rotary encoder with the following line of code:

```
Encoder myEnc(2, 3);
```

The parameters to the Encoder initialization are the two pins the encoder is attached to. If you find that the encoder value is decreasing when you

expect it to increase, you can swap the arguments or swap your wiring. Once you've initialized an encoder, whenever you spin the encoder it will interrupt the sketch briefly to keep track of the movement. You can read the value at any time with `myEnc.read()`.

You can create as many encoders as you have pins, but whenever possible, use pins that support interrupts. The following sketch will handle two encoders, and will work optimally on a board that can handle interrupts on the selected pins such as the SAMD21-based M0 boards (Adafruit Metro M0, SparkFun RedBoard Turbo, and Arduino Zero). If you are using a different board, you may need to use different pins. The Uno and other ATmega328-based boards only support interrupts on pins 2 and 3, so the quality of readings will be diminished on the second encoder no matter which pins you choose with one of those boards:

```cpp
#include <Encoder.h>

Encoder myEncA(2, 3); // MKR boards use pins 4, 5
Encoder myEncB(6, 7); // Mega boards use pins 18, 19

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

long lastA  = -999;
long lastB  = -999;

void loop()
{
  long currentA = myEncA.read();
  long currentB = myEncB.read();
  if (currentA != lastA || currentB != lastB) { // If either position changed
    lastA = currentA; // Save both positions
    lastB = currentB;

    // Print the positions to the Serial Monitor (or Serial Plotter)
    Serial.print("A:"); Serial.print(currentA);
    Serial.print(" ");
    Serial.print("B:"); Serial.println(currentB);
  }
}
```

## See Also

The Arduino MKR Vidor 4000 includes an FPGA that is capable of reading a rotary encoder with much more accuracy than with an Arduino alone.

# 6.13 Using a Mouse

## Problem

You want to detect movements of a PS/2-compatible mouse and respond to changes in the *x* and *y* coordinates.

## Solution

This solution uses LEDs to indicate mouse movement. The brightness of the LEDs changes in response to mouse movement in the *x* (left and right) and *y* (nearer and farther) directions. Clicking the mouse buttons sets the current position as the reference point (Figure 6-17 shows the connections).

To use this sketch, you will need to install the PS/2 library. As of this writing, you will need to use a text editor to open the *ps2.h* file in the *ps2* directory and change `#include "WProgram.h"` to `#include "Arduino.h"`:

```
/*
  Mouse
  an arduino sketch using ps2 mouse library
  from http://www.arduino.cc/playground/ComponentLib/Ps2mouse
*/

#include <ps2.h>

const int dataPin =  5;
const int clockPin = 6;

const int xLedPin  = 9; // Use pin 8 on the MKR boards
const int yLedPin  = 10;

const int mouseRange = 255;  // the maximum range of x/y values

char x;                   // values read from the mouse
```

```
char y;
byte status;

int xPosition = 0;       // values incremented and decremented when mouse moves
int yPosition = 0;
int xBrightness = 128;   // values increased and decreased based on mouse
position
int yBrightness = 128;

const byte REQUEST_DATA = 0xeb; // command to get data from the mouse

PS2 mouse(clockPin, dataPin); // Declare the mouse object

void setup()
{
  mouseBegin(); // Initialize the mouse
}

void loop()
{
  // get a reading from the mouse
  mouse.write(REQUEST_DATA); // ask the mouse for data
  mouse.read();        // ignore ack

  status = mouse.read(); // read the mouse buttons
  if(status & 1) // this bit is set if the left mouse btn pressed
    xPosition = 0; // center the mouse x position
  if(status & 2) // this bit is set if the right mouse btn pressed
    yPosition = 0; // center the mouse y position

  x = mouse.read();
  y = mouse.read();
  if( x != 0 || y != 0)
  {
    // here if there is mouse movement

    xPosition = xPosition + x; // accumulate the position
    xPosition = constrain(xPosition,-mouseRange,mouseRange);

    xBrightness = map(xPosition, -mouseRange, mouseRange, 0,255);
    analogWrite(xLedPin, xBrightness);

    yPosition = constrain(yPosition + y, -mouseRange,mouseRange);
    yBrightness = map(yPosition, -mouseRange, mouseRange, 0,255);
    analogWrite(yLedPin, yBrightness);
  }
}
```
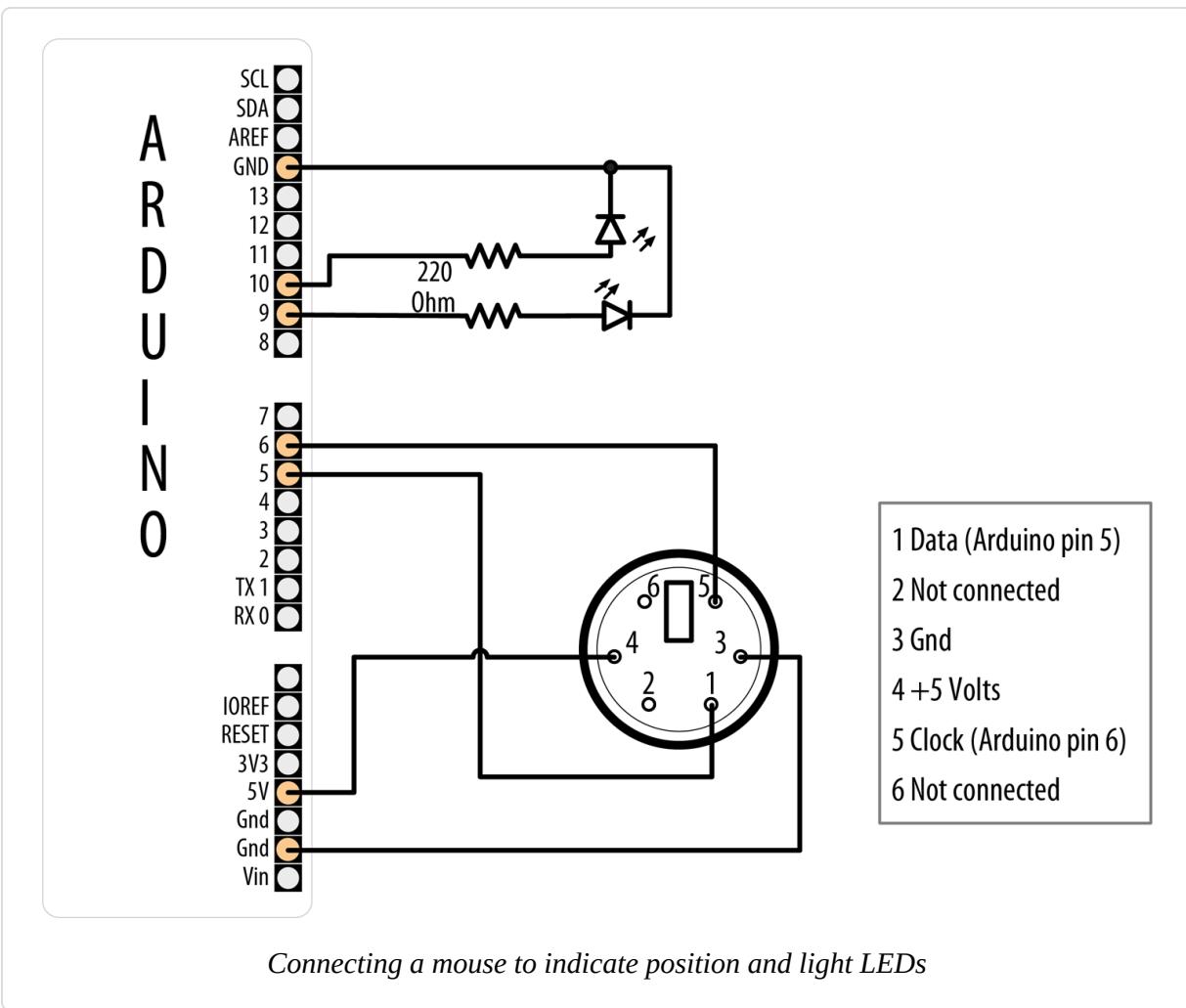
```
void mouseBegin()
{
  // reset and initialize the mouse
  mouse.write(0xff);         // reset
  delayMicroseconds(100);
  mouse.read();              // ack byte
  mouse.read();              // blank
  mouse.read();              // blank
  mouse.write(0xf0);         // remote mode
  mouse.read();              // ack
  delayMicroseconds(100);
}
```



| | |
|---|---|
| 1 Data (Arduino pin 5) | |
| 2 Not connected | |
| 3 Gnd | |
| 4 +5 Volts | |
| 5 Clock (Arduino pin 6) | |
| 6 Not connected | |

*Connecting a mouse to indicate position and light LEDs*

> **WARNING**
>
> If you are using a 3.3V board, you will either need to add a voltage divider to both the clock and data pins, or you may try powering the mouse from 3.3V instead of 5V (which may or may not work, depending on your mouse). See Recipe 5.11 for a discussion of voltage dividers.

Figure 6-17 shows a female PS/2 connector (the socket you plug the mouse into) from the front. If you don't have a female connector and don't mind chopping the end off your mouse, you can note which wires connect to each of these pins and solder to pin headers that plug directly into the correct Arduino pins. A continuity test from a pin to a wire will let you quickly determine which wires go to which pins, but if you are testing the pins from the male plug end that you cut off of your mouse, you need to reverse the diagram left to right.

## Discussion

Connect the mouse signal (clock and data) and power leads to Arduino, as shown in Figure 6-17. This solution only works with PS/2-compatible devices, so you will need to find an older mouse—most mice with the round PS/2 connector should work.

The `mouseBegin` function initializes the mouse to respond to requests for movement and button status. The PS/2 library handles the low-level communication. The `mouse.write` command is used to instruct the mouse that data will be requested. The first call to `mouse.read` gets an acknowledgment (which is ignored in this example). The next call to `mouse.read` gets the button status, and the last two `mouse.read` calls get the $x$ and $y$ movement that has taken place since the previous request.

The sketch tests to see which bits are `HIGH` in the `status` value to determine if the left or right mouse button was pressed. The two rightmost bits will be `HIGH` when the left and right buttons are pressed, and these are checked in the following lines:

```
status = mouse.read(); // read the mouse buttons
  if(status & 1) // rightmost bit is set if the left mouse btn pressed
    xPosition = 0; // center the mouse x position
  if(status & 2) // this bit is set if the right mouse btn pressed
    yPosition = 0; // center the mouse y position
```

The x and y values read from the mouse represent the movement since the previous request, and these values are accumulated in the variables xPosition and yPosition.

The values of x and y will be positive if the mouse moves right or away from you, and negative if it moves left or toward you.

The sketch ensures that the accumulated value does not exceed the defined range (mouseRange) using the constrain function:

```
xPosition = xPosition + x; // accumulate the position
  xPosition = constrain(xPosition,-mouseRange,mouseRange);
```

The yPosition calculation shows a shorthand way to do the same thing; here the calculation for the y value is done within the call to constrain:

```
yPosition = constrain(yPosition + y,-mouseRange,mouseRange);
```

The xPosition and yPosition variables are reset to zero if the left and right mouse buttons are pressed.

LEDs are illuminated to correspond to position using analogWrite—half brightness in the center, and increasing and decreasing in brightness as the mouse position increases and decreases. You must use a PWM-capable pin in order for this to work correctly. If your board does not support PWM on pins 9 and 10 (most do), you will see the lights turn on and off instead of dimming. On the MKR family of boards, pin 9 does not support PWM so you need to change the wiring and the code to use a pin that does.

The position can be graphed on the Serial Plotter by adding the following line just after the second call to analogWrite():

```
printValues(); // show button and x and y values on Serial Monitor/Plotter
```

You'll also need to add this line to `setup()`:

```
Serial.begin(9600);
```

Add the following function to the end of the sketch to print or plot the current position of the mouse:

```
void printValues()
{
  Serial.print("X:");
  Serial.print(xPosition);
  Serial.print(",Y:");
  Serial.print(yPosition);
  Serial.println();
}
```

## See Also

The Adafruit site has a suitable PS/2 connector with built-in wires.

# 6.14 Getting Location from a GPS

## Problem

You want to determine location using a GPS module.

## Solution

A number of Arduino-compatible GPS units are available today. Most use a familiar serial interface to communicate with their host microcontroller using a protocol known as *NMEA 0183*. This industry standard provides for GPS data to be delivered to *listener* devices such as Arduino as human-readable ASCII *sentences*. For example, the following NMEA sentence:

```
$GPGLL,4916.45,N,12311.12,W,225444,A,*1D
```

describes, among other things, a location on the globe at 49° 16.45' north latitude by 123° 11.12' west longitude.

To establish location, your Arduino sketch must parse these strings and convert the relevant text to numeric form. Writing code to manually extract data from NMEA sentences can be tricky and cumbersome in the Arduino's limited address space, but fortunately there is a useful library that does this work for you: Mikal Hart's TinyGPS++. Download it from Mikal's GitHub site and install it. (For instructions on installing third-party libraries, see Recipe 16.2.)

The general strategy for using a GPS is as follows:

1. Physically connect the GPS device to the Arduino.

2. Read serial NMEA data from the GPS device.

3. Process the data to determine location.

Using TinyGPSPlus, you do the following:

1. Physically connect the GPS device to the Arduino.

2. Create a TinyGPSPlus object.

3. Read serial NMEA data from the GPS device.

4. Process each byte with TinyGPSPlus's `encode()` method.

5. Periodically query TinyGPSPlus's `get_position()` method to determine location.

The following sketch illustrates how you can acquire data from a GPS attached to Arduino's serial port. Every five seconds, it blinks the built-in LED once if the device is in the southern hemisphere and twice if it is in the northern hemisphere. If your Arduino's TX and RX pins are associated with another serial device such as `Serial1`, change the definition of `GPS_SERIAL` (see Table 4-1):

```
/* GPS sketch
 * Indicate which hemisphere your GPS is in with the built-in LED.
 */

#include "TinyGPS++.h"

// Change this to the serial port your GPS uses (Serial, Serial1, etc.)
#define GPS_SERIAL Serial
```

```
TinyGPSPlus gps; // create a TinyGPS++ object

#define HEMISPHERE_PIN LED_BUILTIN

void setup()
{
  GPS_SERIAL.begin(9600); // GPS devices frequently operate at 9600 baud
  pinMode(HEMISPHERE_PIN, OUTPUT);
  digitalWrite(HEMISPHERE_PIN, LOW); // turn off LED to start
}

void loop()
{
  while (GPS_SERIAL.available())
  {
    // encode() each byte; if encode() returns "true",
    // check for new position.
    if (gps.encode(GPS_SERIAL.read()))
    {
      if (gps.location.isValid())
      {
        if (gps.location.lat() < 0) // Southern Hemisphere?
          blink(HEMISPHERE_PIN, 1);
        else
          blink(HEMISPHERE_PIN, 2);
      } else // panic
        blink(HEMISPHERE_PIN, 5);
      delay(5000); // Wait 5 seconds
    }
  }
}

void blink(int pin, int count)
{
  for (int i = 0; i < count; i++)
  {
    digitalWrite(pin, HIGH);
    delay(250);
    digitalWrite(pin, LOW);
    delay(250);
  }
}
```

Start serial communications using the rate required by your GPS. See Chapter 4 if you need more information on using Arduino serial

communications.

A 9,600-baud connection is established with the GPS. Once bytes begin flowing, they are processed by `encode()`, which parses the NMEA data. A `true` return from `encode()` indicates that TinyGPSPlus has successfully parsed a complete sentence and that fresh position data may be available. This is a good time to check whether the position is valid with a call to `gps.location.isValid()`.

TinyGPSPlus's `gps.location.lat()` returns the most recently observed latitude, which this sketch examines; if it is less than zero (that is, south of the equator), the LED blinks once. If it is greater than zero (at or north of the equator), it blinks twice. If the GPS is unable to get a valid fix, it blinks five times.

## Discussion

Attaching a GPS unit to an Arduino is usually as simple as connecting two or three data lines from the GPS to input pins on the Arduino as shown in Table 6-2. If you are using a 5V board such as the Uno, you can use either a 3.3V or 5V GPS module. If you are using a board that is not 5V tolerant, such as a SAMD-based board like the Arduino Zero, Adafruit Metro M0/M4, or SparkFun Redboard Turbo, you must use a 3.3V GPS module.

*GPS pin connections*

| GPS line | Arduino pin |
| --- | --- |
| GND | GND |
| 5V or 3.3V | 5V or 3.3V |
| RX | TX (pin 1) |
| TX | RX (pin 0) |

The code in the Solution assumes that the GPS is connected directly to Arduino's built-in serial pins. On an ATmega328-based board like the Arduino Uno, this is not usually the most convenient design because RX and TX (pins 0 and 1) are shared with the USB serial connection. In many projects, you'll use the hardware serial port to communicate with a host PC or other peripheral, which means that port cannot be used by the GPS. In cases like this, select another pair of digital pins and use a serial port emulation ("soft serial") library to talk to the GPS instead.

With the Arduino and GPS powered down, move the GPS's TX line to Arduino pin 2 and RX line to pin 3 to free up the hardware serial port for debugging (see Figure 4-8). With the USB cable connected to the host PC, try the following sketch to get a detailed glimpse of TinyGPS in action through the Arduino's Serial Monitor:

```
/* GPS sketch with logging
 */

#include "TinyGPS++.h"

// Delete the next four lines if your board has a separate hardware serial
port
#include "SoftwareSerial.h"
#define GPS_RX_PIN 2
#define GPS_TX_PIN 3
SoftwareSerial softserial(GPS_RX_PIN, GPS_TX_PIN); // create soft serial
object

// If your board has a separate hardware serial port,
// change "softserial" to that port
#define GPS_SERIAL softserial

TinyGPSPlus gps; // create a TinyGPSPlus object
```

```
void setup()
{
  Serial.begin(9600); // for debugging
  GPS_SERIAL.begin(9600); // Use Soft Serial object to talk to GPS
}
void loop()
{
  while (GPS_SERIAL.available())
  {
    int c = GPS_SERIAL.read();
    Serial.write(c); // display NMEA data for debug

    // Send each byte to encode()
    // Check for new position if encode() returns "True"
    if (gps.encode(c))
    {
      Serial.println();
      float lat = gps.location.lat();
      float lng = gps.location.lng();
      unsigned long fix_age = gps.date.age();

      if (!gps.location.isValid())
        Serial.println("Invalid fix");
      else if (fix_age > 2000)
        Serial.println("Stale fix");
      else
        Serial.println("Valid fix");

      Serial.print("Lat: ");
      Serial.print(lat);
      Serial.print(" Lon: ");
      Serial.println(lng);
    }
  }
}
```

For a more detailed discussion on software serial, see Recipes 4.11 and 4.12.

Note that you can use a different baud rate for connection to the Serial Monitor and the GPS.

This new sketch behaves the same as the earlier example (but for brevity, omits the LED blinking code) but is much easier to debug. At any time, you can connect a serial LCD (see Recipe 4.11) to the built-in serial port to

watch the NMEA sentences and TinyGPSPlus data scrolling by. You could also connect to the serial port using Arduino's Serial Monitor.

When power is turned on, a GPS unit begins transmitting NMEA sentences. However, the sentences containing valid location data are only transmitted after the GPS establishes a fix, which requires the GPS antenna to have visibility of the sky and can take up to two minutes or more. Stormy weather or the presence of buildings or other obstacles may also interfere with the GPS's ability to pinpoint location. So, how does the sketch know whether TinyGPSPlus is delivering valid position data? The answer lies in the return value from the `gps.location.isValid()` function. A false value means TinyGPS has not yet parsed any valid sentences containing position data. In this case, you'll know that the returned latitude and longitude are invalid as well.

You can also check how old the fix is. The `gps.date.age()` function returns the number of milliseconds since the last fix. The sketch stores its value in `fix_age`. Under normal operation, you can expect to see quite low values for `fix_age`. Modern GPS devices are capable of reporting position data as frequently as one to five times per second or more, so a `fix_age` in excess of 2,000 ms or so suggests that there may be a problem. Perhaps the GPS is traveling through a tunnel or a wiring flaw is corrupting the NMEA data stream, invalidating the checksum (a calculation to check that the data is not corrupted). In any case, a large `fix_age` indicates that the coordinates returned by `get_position()` are stale.

## See Also

For a deeper understanding of the NMEA protocol, read the Wikipedia articles.

Several shops sell GPS modules that interface well with TinyGPS and Arduino. These differ mostly in power consumption, voltage, accuracy, physical interface, and whether they support serial NMEA. Adafruit sells a variety of modules as does SparkFun.

GPS technology has inspired lots of creative Arduino projects. A very popular example is the GPS data logger, in which a moving device records location data at regular intervals to the Arduino EEPROM or other onboard storage. See the *https://oreil.ly/w0asL* for an example. Adafruit makes a popular GPS data logging shield.

Other interesting GPS projects include hobby airplanes and helicopters that maneuver themselves to preprogrammed destinations under Arduino software control. Mikal Hart has built a GPS-enabled "treasure chest" with an internal latch that cannot be opened until the box is physically moved to a certain location. See his post about this project.

# 6.15 Detecting Rotation Using a Gyroscope

## Problem

You want to respond to the rate of rotation. This can be used to keep a vehicle or robot moving in a straight line or turning at a desired rate.

## Solution

Gyroscopes provide an output related to rotation rate (as opposed to an accelerometer, which indicates rate of change of velocity). In the early days of Arduino, most low-cost gyroscopes used an analog voltage proportional to rotation rate. Now, with the ubiquitous use of gyroscopes and accelerometers in smartphones, it is cheaper and easier to find gyroscopes and accelerometers combined using the I2C protocol. See Chapter 13 for more on using I2C.

> **NOTE**
>
> The Arduino Nano 33 BLE Sense board has a gyroscope and accelerometer built onto the board. See Recipe 6.1 for more information.

The MPU-9250 inertial measurement unit is a relatively inexpensive nine degrees of freedom (9DOF) sensor that works well with Arduino. It is available on a breakout board from many suppliers, including SparkFun (part number SEN-13762). There are several libraries available that support the MPU-9250. The following sketch uses the Bolder Flight Systems MPU9250 library that you can install using the Arduino Library Manager. (For instructions on installing third-party libraries, see Recipe 16.2.) Connect the sensor as shown in Figure 6-18:

```
/* Gyro sketch
 * Read a gyro and display rotation in degrees/sec
 */

#include "MPU9250.h"

//  I2C address of IMU. If this doesn't work, try 0x69.
#define IMU_ADDRESS 0x68

MPU9250 IMU(Wire, IMU_ADDRESS); // Declare the IMU object

void setup() {

  Serial.begin(9600);
  while(!Serial);

  // Initialize the IMU
  int status = IMU.begin();
  if (status < 0) {
    Serial.println("Could not initialize the IMU.");
    Serial.print("Error value: "); Serial.println(status);
    while(1); // halt the sketch
  }

  // Set the full range of the gyro to +/- 500 degrees/sec
  status = IMU.setGyroRange(MPU9250::GYRO_RANGE_500DPS);
  if (status < 0) {
    Serial.println("Could not change gyro range.");
    Serial.print("Error value: "); Serial.println(status);
  }
}

void loop() {

  IMU.readSensor();
```
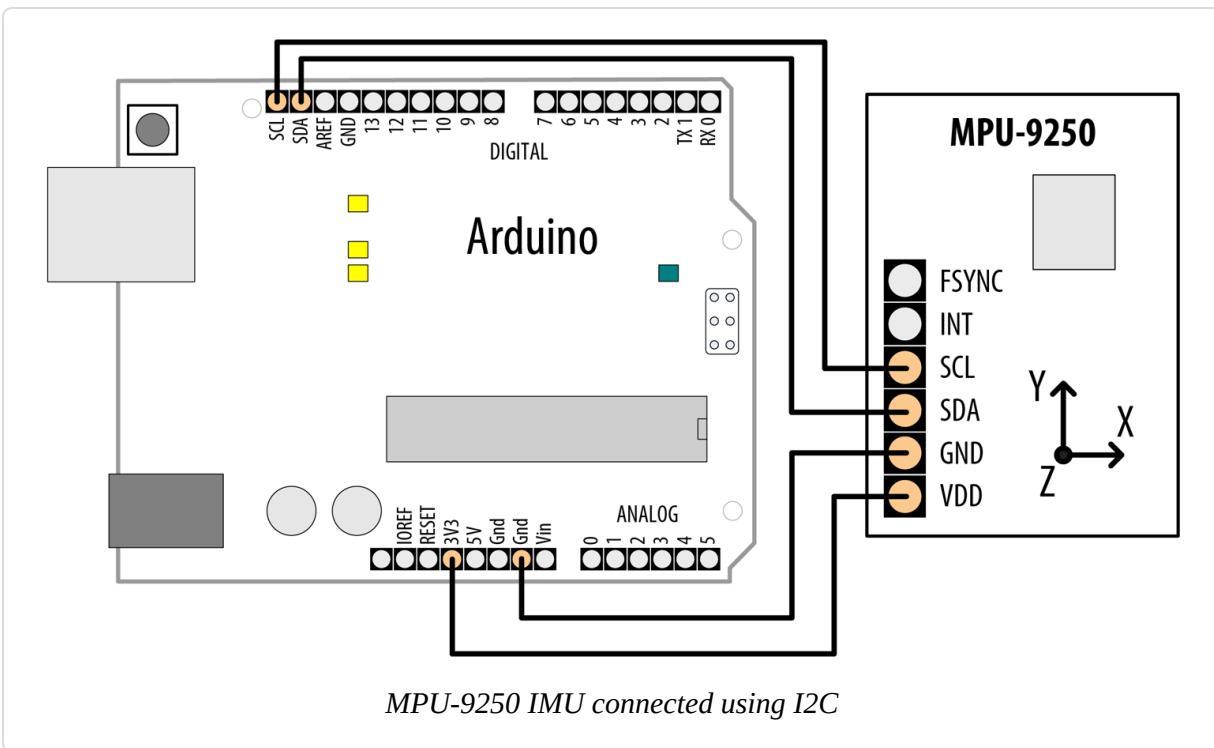
```
// Obtain the rotational velocity in rads/second
float gx = IMU.getGyroX_rads();
float gy = IMU.getGyroY_rads();
float gz = IMU.getGyroZ_rads();

// Display velocity in degrees/sec
Serial.print("gx:");
Serial.print(gx * RAD_TO_DEG, 4);
Serial.print(",gy:");
Serial.print(gy * RAD_TO_DEG, 4);
Serial.print(",gz:");
Serial.print(gz * RAD_TO_DEG, 4);
Serial.println();
delay(100);
}
```

## WARNING

The MPU-9250 is a 3.3V I2C device, so if you are not using a 3.3V Arduino board you will need a logic-level converter to protect the gyro's SCL and SDA pins. See the introduction to Chapter 13 for more on I2C and using 3.3V devices.



*MPU-9250 IMU connected using I2C*

## Discussion

The sketch starts out by including the MPU9250 library and declaring an object to represent the IMU. Within `setup()`, it attempts to initialize the IMU. If this fails, you may need to change the `IMU_ADDRESS` definition to 0x69 or check your wiring. After the IMU is initialized, the sketch changes the gyro's full range to +/– 500 degrees per second.

Within `loop`, the sketch reads the sensor and obtains the rotational velocity in radians per second. It then uses the `RAD_TO_DEG` Arduino constant to convert this to degrees per second. The output of the sketch is readable in either the Serial Monitor or Serial Plotter.

## See Also

See Chapter 13 for more about I2C.

See "Using 3.3-Volt Devices with 5-Volt Boards" for more about connecting 3.3V devices to 5V boards.

Try the SparkFun tutorial for the MPU-9250. This tutorial uses a different library, but the concepts are the same.

# 6.16 Detecting Direction

### Problem

You want your sketch to determine direction from an electronic compass.

### Solution

This recipe uses the magnetometer in the MPU-9250 nine degrees of freedom (9DOF) inertial measurement unit (IMU) from Recipe 6.15. Connect the sensor as shown in Figure 6-18. Each of the MPU-9250's three primary sensors (gyro, magnetometer, and accelerometer) read values in three dimensions (x, y, z), which is where the nine degrees of freedom come from:

> **NOTE**
>
> Before you use the magnetometer, you must calibrate it. You can find a calibration sketch in this GitHub issue. That sketch will store the calibration values in your microcontroller board's nonvolatile EEPROM memory. You will need to load the calibration values any time you want to work with the magnetometer, as shown in the next sketch. If you use the sensor with a different microcontroller board, you'll need to run the calibration sketch again. Also, if you store anything else in the EEPROM, you'll need to make sure you don't store it in the same location as the calibration values.

```
/* Magnetometer sketch
   Read a magnetometer and display magnetic field strengths
*/

#include "MPU9250.h"
#include <math.h>
#include "EEPROM.h"

//  I2C address of IMU. If this doesn't work, try 0x69.
#define IMU_ADDRESS 0x68

// Change this to the declination for your location.
// See https://www.ngdc.noaa.gov/geomag/calculators/magcalc.shtml
#define DECLINATION (-14)

MPU9250 IMU(Wire, IMU_ADDRESS); // Declare the IMU object

void setup() {

  int status;

  Serial.begin(9600);
  while (!Serial);

  // Initialize the IMU
  status = IMU.begin();
  if (status < 0)
  {
    Serial.println("Could not initialize the IMU.");
    Serial.print("Error value: "); Serial.println(status);
    while (1); // halt the sketch
  }

  load_calibration();
}
```

```cpp
void loop() {

  IMU.readSensor();

  // Obtain the magnetometer values across each axis in units of microTesla
  float mx = IMU.getMagX_uT();
  float my = IMU.getMagY_uT();
  float mz = IMU.getMagZ_uT();

  // From https://github.com/bolderflight/MPU9250/issues/33
  // Normalize the magnetometer data.
  float m = sqrtf(mx * mx + my * my + mz * mz);
  mx /= m;
  my /= m;
  mz /= m;

  // Display the magnetometer values
  Serial.print("mx:");
  Serial.print(mx, 4);
  Serial.print(",my:");
  Serial.print(my, 4);
  Serial.print(",mz:");
  Serial.print(mz, 4);
  Serial.println();

  float constrained =
      constrainAngle360(atan2f(-my, mx) + (DECLINATION * DEG_TO_RAD));
  float calcAngle = constrained * RAD_TO_DEG;
  Serial.print(calcAngle);
  Serial.println(" degrees");
  delay(100);
}

// From https://github.com/bolderflight/MPU9250/issues/33
float constrainAngle360(float dta) {
  dta = fmod(dta, 2.0 * PI);
  if (dta < 0.0)
    dta += 2.0 * PI;
  return dta;
}

// Load the calibration from the eeprom
// From https://github.com/bolderflight/MPU9250/issues/33
void load_calibration() {
  float hxb, hxs, hyb, hys, hzb, hzs;

  uint8_t eeprom_buffer[24];
```

```
    for (unsigned int i = 0; i < sizeof(eeprom_buffer); i++ ) {
      eeprom_buffer[i] = EEPROM.read(i);
    }
    memcpy(&hxb, eeprom_buffer, sizeof(hxb));
    memcpy(&hyb, eeprom_buffer + 4, sizeof(hyb));
    memcpy(&hzb, eeprom_buffer + 8, sizeof(hzb));
    memcpy(&hxs, eeprom_buffer + 12, sizeof(hxs));
    memcpy(&hys, eeprom_buffer + 16, sizeof(hys));
    memcpy(&hzs, eeprom_buffer + 20, sizeof(hzs));
    IMU.setMagCalX(hxb, hxs);
    IMU.setMagCalY(hyb, hys);
    IMU.setMagCalZ(hzb, hzs);
  }
```

---

### WARNING

If you want to use the IMU with a 5-volt Arduino board, see "Using 3.3-Volt Devices with 5-Volt Boards" for details on how to use a logic-level converter.

---

## Discussion

The compass module provides magnetic field intensities on three axes (x, y, and z). These values vary as the compass orientation is changed with respect to the Earth's magnetic field (magnetic north).

As with the sketch shown in Recipe 6.15, this sketch configures and initializes the IMU, but instead of showing gyro data, it reads magnetometer readings in units of microTesla and converts them to a compass bearing. (Another big difference is that it loads the calibration data from the EEPROM.) For this sketch to work properly, the IMU must be on a level surface. You must also set the declination for your geographic location by changing the value of DECLINATION at the top of the sketch (use a negative number for a west declination, positive for east). For more, refer to the NGDC declination lookup tool.

The magnetometer readings are normalized by then dividing each reading by the square root of the sum of the squares (RSS) of all of the readings. The angle to magnetic north is calculated by adding the declination (in radians) to the following formula: radians = arctan2(-my, mx),

constrained to 360 degrees (2 * pi radians) by the `constrainAngle360`
function. That result is converted to degrees by multiplying it by the
`RAD_TO_DEG` constant. Zero degrees indicates magnetic north.

To make a servo follow the compass direction over the first 180 degrees,
use the techniques shown in "Servos", but use `calcAngle` to move the
servo as shown:

```
angle = constrain(calcAngle, 0, 180);
myservo.write(calcAngle);
```

# 6.17 Reading Acceleration

## Problem

You want to respond to acceleration; for example, to detect when something
starts or stops moving. Or you want to detect how something is oriented
with respect to the Earth's surface (measure acceleration due to gravity).

## Solution

This recipe uses the accelerometer in the MPU-9250 nine degrees of
freedom (9DOF) inertial measurement unit (IMU) from Recipe 6.15.
Connect the sensor as shown in Figure 6-18.

> ### WARNING
>
> If you want to use the IMU with a 5-volt Arduino board, see "Using 3.3-Volt Devices with 5-Volt
> Boards" for details on how to use a logic-level converter.

The simple sketch here uses the MPU-9250 to display the acceleration in
the x-, y-, and z-axes:

```
/* Accelerometer sketch
 * Read an accelerometer and display acceleration in m/s/s
 */
```

```arduino
#include "MPU9250.h"

//  I2C address of IMU. If this doesn't work, try 0x69.
#define IMU_ADDRESS 0x68

MPU9250 IMU(Wire, IMU_ADDRESS); // Declare the IMU object

void setup() {

  Serial.begin(9600);
  while(!Serial);

  // Initialize the IMU
  int status = IMU.begin();
  if (status < 0) {
    Serial.println("Could not initialize the IMU.");
    Serial.print("Error value: "); Serial.println(status);
    while(1); // halt the sketch
  }

}

void loop() {

  IMU.readSensor();

  // Obtain the rotational velocity in rads/second
  float ax = IMU.getAccelX_mss();
  float ay = IMU.getAccelY_mss();
  float az = IMU.getAccelZ_mss();

  // Display velocity in degrees/sec
  Serial.print("ax:");  Serial.print(ax, 4);
  Serial.print(",ay:"); Serial.print(ay, 4);
  Serial.print(",az:"); Serial.print(az, 4);
  Serial.println();
  delay(100);
}
```

## Discussion

This sketch is similar to the gyro sketch from Recipe 6.15, except that it displays acceleration along each axis in meters per second squared (m/s/s). Even when stationary, you'll notice that the z acceleration hovers around −9.8 m/s/s. At least that's what you'll see if you're running this sketch on

Earth, where gravity is roughly 9.8 m/s/s. If you see a value of 0 along the z-axis, then the sensor is in free fall. The force that causes the 9.8 m/s/s acceleration is the mechanical force of whatever is keeping the sensor from falling (your hand, a table, the floor). Although the object appears to have no acceleration from your viewpoint, it is accelerating relative to free fall, which is the condition that would apply if there was nothing (no floor, no table, no hand) between your sensor and the center of the Earth. If there was nothing between your sensor and the center of the Earth, that would be a somewhat unusual and certainly undesirable configuration of Earth's mass, at least from the viewpoint of Earth's life forms.

You can use techniques from the previous recipes to extract information from the accelerometer readings. You might need to check for a threshold to work out movement (see Recipe 6.7 for an example of threshold detection). You may find it useful to apply a moving average formula to the incoming data.

If the accelerometer is reading horizontally, you can use the values directly to work out movement. If it is reading vertically, you will need to take into account the effects of gravity on the values. This is similar to the DC offset in Recipe 6.8, but it can be complicated, as the accelerometer may be changing orientation so that the effect of gravity is not a constant value for each reading.

The data produced by accelerometers can be difficult to work with, particularly trying to make decisions about movement over time—detecting gestures, not just positions. Machine learning techniques are starting to be used to process live sensor data and recognize how they relate to example sets of data produced before. These approaches currently need to run on a computer, and are still quite fiddly to set up, but can produce very useful results.

## See Also

An excellent example that is integrated with Arduino boards is the Example-based Sensor Prediction system by David Mellis, built on top of the Gesture Recognition Toolkit.

Also worth looking at is Wekinator.

SparkFun's advanced library for the MPU-9250 includes pedometer, tap, and orientation direction. It requires a SAMD-based Arduino or Arduino compatible.

# Visual Output

## 7.0 Introduction

Visual output lets the Arduino convey information to users, and toward that end, the Arduino supports a broad range of LED devices. (Arduino can also display information with graphical display panels, which is covered in Chapter 11.) Before delving into the recipes in this chapter, we'll discuss Arduino digital and analog output and explain how Arduino works with light-emitting diodes (LEDs). This introduction will be a good starting point if you are not yet familiar with using digital and analog outputs (`digitalWrite` and `analogWrite`) or with using LEDs in a circuit. The recipes in this chapter cover everything from simple single-LED displays to creating the illusion of motion (Recipe 7.7) and showing shapes (Recipe 7.9).

## Digital Output

All the pins that can be used for digital input can also be used for digital output. Chapter 5 provided an overview of the Arduino pin layout; you may want to look through the introduction section in that chapter if you are unfamiliar with connecting things to Arduino pins.

Digital output causes the voltage on a pin to be either high (5 or 3.3 volts depending on board) or low (0 volts). Use the `digitalWrite(outputPin, value)` function to turn something on or off. The function has two parameters: `outputPin` is the pin to control, and `value` is either `HIGH` (5 or 3.3 volts) or `LOW` (0 volts).
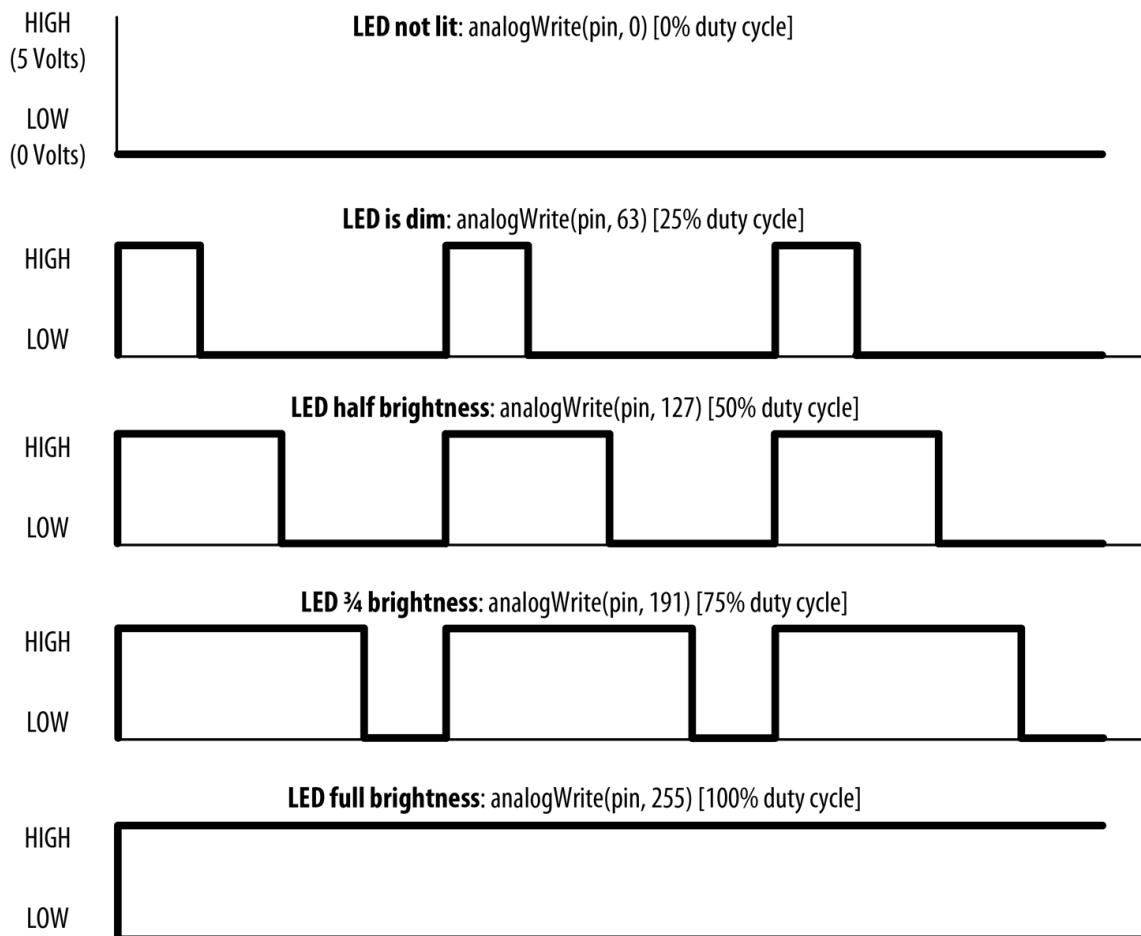
For the pin voltage to respond to this command, the pin must have been set in *output* mode using the `pinMode(outputPin, OUTPUT)` command. The sketch in Recipe 7.1 provides an example of how to use digital output.

## Analog Output

*Analog* refers to levels that can be gradually varied up to their maximum level (think of light dimmers and volume controls). Arduino has an `analogWrite` function that can be used to control such things as the intensity of an LED connected to the Arduino.

The `analogWrite` function is not truly analog, although it can behave like analog, as you will see. `analogWrite` uses a technique called Pulse Width Modulation (PWM) that emulates an analog signal using digital pulses.

PWM works by varying the proportion of the pulses' on time to off time, as shown in Figure 7-1. Low-level output is emulated by producing pulses that are on for only a short period of time. Higher-level output is emulated with pulses that are on more than they are off. When the pulses are repeated quickly enough (almost five hundred times per second or faster on Arduino boards), the pulsing cannot be detected by human senses, and the output from things such as LEDs looks like it is being smoothly varied as the pulse rate is changed.

*PWM output for various analogWrite values*

Arduino has a limited number of pins that can be used for PWM output. On the Arduino Uno and compatible boards based on the ATmega328, you can use pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega board, you can use pins 2 through 13 and 44 through 46 for PWM output. The Nano board has only five PWM outputs, while the Zero, SparkFun RedBoard Turbo, and Adafruit Metro Express M0 support PWM on every digital pin except 2 and 7. Many of the recipes that follow use pins that can be used for both digital and PWM to minimize rewiring if you want to try out different recipes. If you want to select different pins for PWM output, remember to choose one of the supported `analogWrite` pins (other pins will not give any output). The Zero, RedBoard Turbo, and Metro Express M0 boards all have a DAC pin (A0) that produces a true analog signal. This is not intended for

controlling things like the brightness or motor speeds (PWM works better), but is really useful for producing audio signals, as shown in Recipe 1.8.

# Controlling Light

Controlling light using digital or analog output is a versatile, effective, and widely used method for providing user interaction. Single LEDs, arrays, and numeric displays are covered extensively in the recipes in this chapter. LCD text and graphical displays require different techniques and are covered in Chapter 11.

### LED specifications

An LED is a semiconductor device (diode) with two leads, an *anode* and a *cathode*. When the voltage on the anode is more positive than that on the cathode (by an amount called the *forward voltage*), the device emits light (photons). The anode is usually the longer lead, and there is often a flat spot on the housing to indicate the cathode (see Figure 7-2). The LED color and the exact value of the forward voltage depend on the construction of the diode.

A typical red LED has a forward voltage of around 1.8 volts. If the voltage on the anode is not 1.8 volts more positive than the cathode, no current will flow through the LED and no light will be produced. When the voltage on the anode becomes 1.8 volts more positive than that on the cathode, the LED "turns on" (conducts) and effectively becomes a short circuit. You must limit the current with a resistor, or the LED will (sooner or later) burn out. Recipe 7.1 shows you how to calculate values for current-limiting resistors.

You may need to consult an LED datasheet to select the correct LED for your application, particularly to determine values for forward voltage and maximum current. Tables 7-1 and 7-2 show the most important fields you should look for on an LED datasheet.

*Key datasheet specifications: absolute maximum ratings*

| Parameter | Symbol | Rating | Units | Comment |
|---|---|---|---|---|
| Forward current | $I_F$ | 25 | mA | The maximum continuous current for this LED |
| Peak forward current (1/10 duty @ 1 kHz) | $I_{FP}$ | 160 | mA | The maximum pulsed current (given here for a pulse that is 1/10 on and 9/10 off) |

*Key datasheet specifications: electro-optical characteristics*

| Parameter | Symbol | Rating | Units | Comment |
|---|---|---|---|---|
| Luminous intensity | $I_V$ | 2 | mcd | If = 2 mA – brightness with 2 mA current |
|  | $I_V$ | 40 | mcd | If = 20 mA – brightness with 20 mA current |
| Viewing angle |  | 120 | degrees | The beam angle |
| Wavelength |  | 620 | nm | The dominant or peak wavelength (color) |
| Forward voltage | $V_F$ | 1.8 | volts | The voltage across the LED when on |

Arduino pins on Uno, Leonardo, and Mega boards can supply up to 40 mA of current. This is plenty for a typical medium-intensity LED, but not enough to drive the higher-brightness LEDs or multiple LEDs connected to a single pin. Recipe 7.3 shows how to use a transistor to increase the current through the LED.

The 3.3-volt boards have a lower current capacity; check the datasheet for your board to ensure that you do not exceed the maximum ratings.

Multicolor LEDs consist of two or more LEDs in one physical package. These may have more than two leads to enable separate control of the different colors. There are many package variants, so you should check the datasheet for your LED to determine how to connect the leads.

### Multiplexing

Applications that need to control many LEDs can use a technique called *multiplexing.* Multiplexing works by switching groups of LEDs (usually

arranged in rows or columns) in sequence. Recipe 7.12 shows how 32 individual LEDs (eight LEDs per digit, including decimal point) with four digits can be driven with just 12 pins. Eight pins drive a digit segment for all the digits and four pins select which digit is active. Scanning through the digits quickly enough (at least 25 times per second) creates the impression that the lights remain on rather than pulsing, through the phenomenon of *persistence of vision*.

*Charlieplexing* uses multiplexing along with the fact that LEDs have *polarity* (they only illuminate when the anode is more positive than the cathode) to switch between two LEDs by reversing the polarity.

## Maximum pin current

LEDs can draw more power than the Arduino chip is designed to handle. The datasheet gives the absolute maximum ratings for the Arduino Uno chip (ATmega328P) as 40 mA per pin. The chip is capable of sourcing and sinking 200 mA overall, so you must also ensure that the total current is less than this; for example, five pins providing a `HIGH` output (sourcing) and five `LOW` (sinking) with each pin at 40 mA. It is good practice to design your applications to operate well within the absolute maximum ratings for best reliability, so best to keep current at or below 30 mA to provide a large comfort margin. For hobby use where more pin current is wanted and reduced reliability is acceptable, you can drive a pin with up to 40 mA as long as the 200 mA source and 200 mA sink limits per chip are not exceeded.

See the Discussion section of Recipe 7.3 for a tip on how to get increased current without using external transistors.
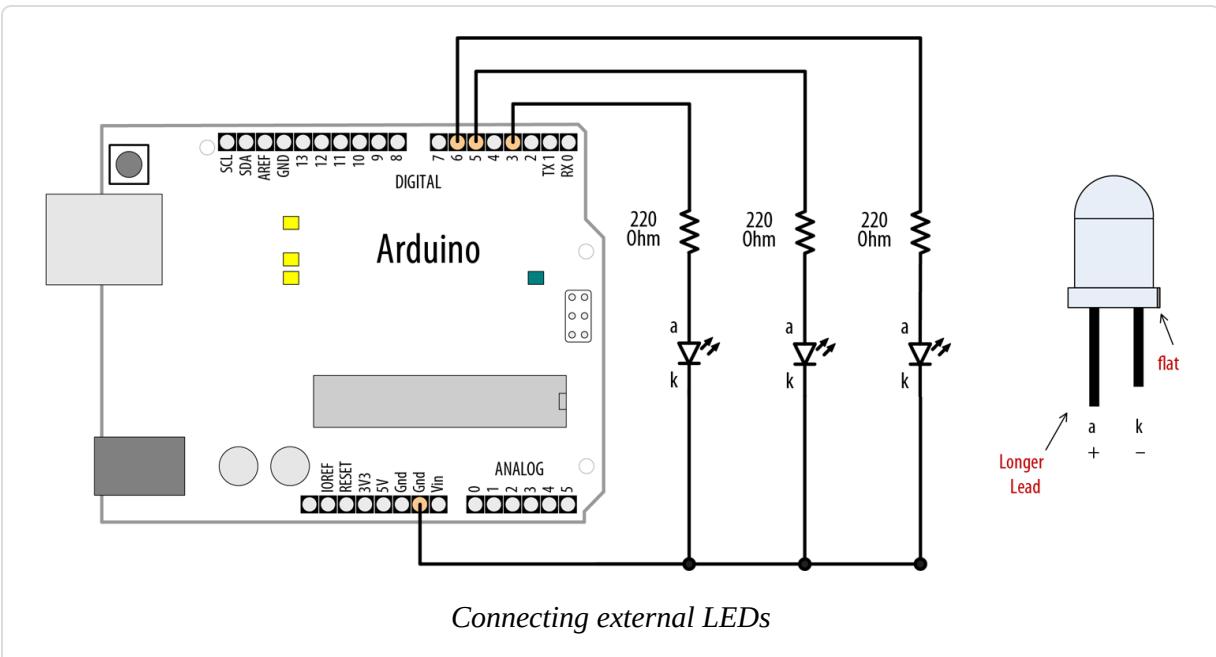
# 7.1 Connecting and Using LEDs

## Problem

You want to control one or more LEDs and select the correct current-limiting resistor so that you do not damage the LEDs.

## Solution

Turning an LED on and off is easy to do with Arduino, and some of the recipes in previous chapters have included this capability (see Recipe 5.1 for an example that controls the built-in LED on pin 13). The recipe here provides guidance on choosing and using external LEDs. Figure 7-2 shows the wiring for three LEDs, but you can run this sketch with just one or two.

*Connecting external LEDs*

---

**TIP**

The schematic symbol for the cathode (the negative pin) is *k*, not *c*. The schematic symbol *c* is used for a capacitor.

---

The following sketch lights up three LEDs connected to pins 3, 5, and 6 in sequence for one second:

```
/*
 * LEDs sketch
 * Blink three LEDs each connected to a different digital pin
 */

const int firstLedPin  = 3; // choose the pin for each of the LEDs
const int secondLedPin = 5;
const int thirdLedPin  = 6;

void setup()
{
  pinMode(firstLedPin, OUTPUT);  // declare LED pins as output
  pinMode(secondLedPin, OUTPUT); // declare LED pins as output
  pinMode(thirdLedPin, OUTPUT);  // declare LED pins as output
}
```

```
void loop()
{
  // flash each of the LEDs for 1000 ms (1 second)
  blinkLED(firstLedPin,  1000);
  blinkLED(secondLedPin, 1000);
  blinkLED(thirdLedPin,  1000);
}


// blink the LED on the given pin for the duration in milliseconds
void blinkLED(int pin, int duration)
{
  digitalWrite(pin, HIGH);     // turn LED on
  delay(duration);
  digitalWrite(pin, LOW);      // turn LED off
  delay(duration);
}
```
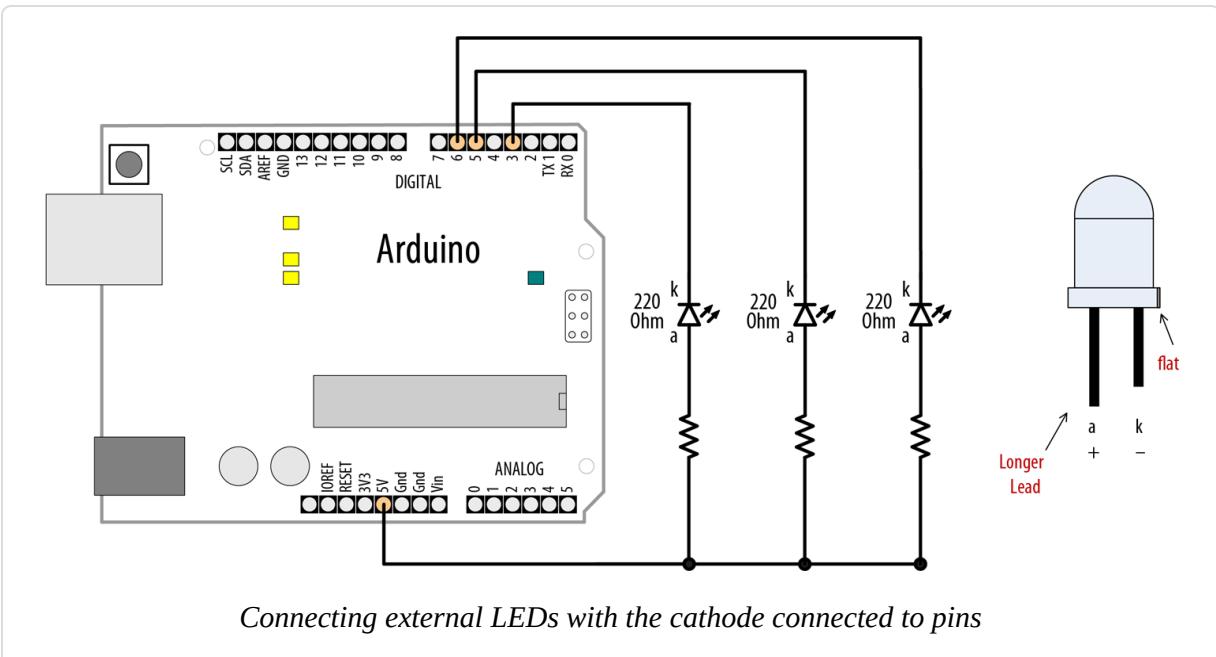
The sketch sets the pins connected to LEDs as output in the `setup` function.
The `loop` function calls `blinkLED` to flash the LED for each of the three
pins. `blinkLED` sets the indicated pin `HIGH` for one second (1,000 ms).

## Discussion

Because the anodes are connected to Arduino pins and the cathodes are
connected to ground, the LEDs will light when the pin goes `HIGH` and will
be off when the pin is `LOW`. You can illuminate the LED when the pin is `LOW`
by connecting the cathodes to the pins and the anodes to ground (the
resistors can be used on either side of the LED).

When LEDs are connected with the anode connected to +5V, as shown in
Figure 7-3, the LEDs light when the pin goes `LOW` (the visual effect would
reverse—one of the LEDs would turn off for a second while the other two
would be lit).

*Connecting external LEDs with the cathode connected to pins*

## NOTE

LEDs require a series resistor to control the current or they can quickly burn out. External LEDs need to be connected through a series resistor on either the anode or the cathode.

A resistor in series with the LED is used to control the amount of current that will flow when the LED conducts. To calculate the resistor value, you need to know the input power supply voltage ($V_S$, usually 5 volts), the LED forward voltage ($V_F$), and the amount of current (I) that you want to flow through the LED.

The formula for the resistance in ohms (known as Ohm's law) is:

- $R = (V_S - V_F) / I$

For example, driving an LED with a forward voltage of 1.8 volts with 15 mA of current using an input supply voltage of 5 volts would use the following values:

- $V_S = 5$ (for a 5V Arduino board)
- $V_F = 1.8$ (the forward voltage of the LED)

- I = 0.015 (1 milliamp [mA] is one one-thousandth of an amp, so 15 mA is 0.015 amps)

The voltage across the LED when it is on ($V_s - V_F$) is 5 – 1.8, which is 3.2 volts.

Therefore, the calculation for the series resistor is 3.2 / 0.015, which is 213 ohms.

The value of 213 ohms is not a standard resistor value, so you can round this up to 220 ohms.

The resistor is shown in Figure 7-2 connected between the cathode and ground, but it can be connected to the other side of the LED instead (between the voltage supply and the anode).

---

### WARNING

Arduino Uno and Mega pins have a specified maximum current of 40 mA. If your LED needs more current than your board can supply, see Recipe 7.3.

---

## See Also

Recipe 7.3

# 7.2 Adjusting the Brightness of an LED

### Problem

You want to control the intensity of one or more LEDs from your sketch.

### Solution

Connect each LED to an analog (PWM) output. Use the wiring shown in Figure 7-2. The sketch will fade the LED(s) from off to maximum intensity and back to off, with each cycle taking around five seconds:

```
/*
 * LedBrightness sketch
 * controls the brightness of LEDs on analog output ports
 */

const int firstLed   = 3; // specify the pin for each of the LEDs
const int secondLed  = 5;
const int thirdLed   = 6;

int brightness = 0;
int increment  = 1;

void setup()
{
   // pins driven by analogWrite do not need to be declared as outputs
}

void loop()
{
  if(brightness > 255)
  {
    increment = -1; // count down after reaching 255
  }
  else if(brightness < 1)
  {
    increment =  1; // count up after dropping back down to 0
  }
  brightness = brightness + increment; // increment (or decrement sign is
minus)

  // write the brightness value to the LEDs
  analogWrite(firstLed, brightness);
  analogWrite(secondLed, brightness);
  analogWrite(thirdLed, brightness );

  delay(10); // 10 ms for each step change means 2.55 secs to fade up or down
}
```

## Discussion

This uses the same wiring as the previous sketch, but here the pins are controlled using `analogWrite` instead of `digitalWrite`. `analogWrite` uses PWM to control the power to the LED; see this chapter's introduction for more on analog output.

The sketch fades the light level up and down by increasing (on fade up) or decreasing (on fade down) the value of the `brightness` variable in each pass through the loop. This value is given to the `analogWrite` function for the three connected LEDs. The minimum value for `analogWrite` is 0—this keeps the voltage on the pin at 0. The maximum value is 255 (5V on a 5V board, 3.3V on a 3.3V board).

> **TIP**
>
> It is good practice to restrict the range of values to the range of 0–255; values outside that range can give unexpected results. See Recipe 3.5.

When the `brightness` variable reaches the maximum value, it will start to decrease, because the sign of the `increment` is changed from +1 to –1 (adding –1 to a value is the same as subtracting 1 from that value).

## See Also

This chapter's Recipe 7.0 describes how Arduino analog output works.

Boards such as the Due, Zero, and MKR1000 can have the PWM range to a maximum of 4,095, although they all default to the standard 255. If you need to use this higher resolution you can set this using the `analogWriteResolution` function.
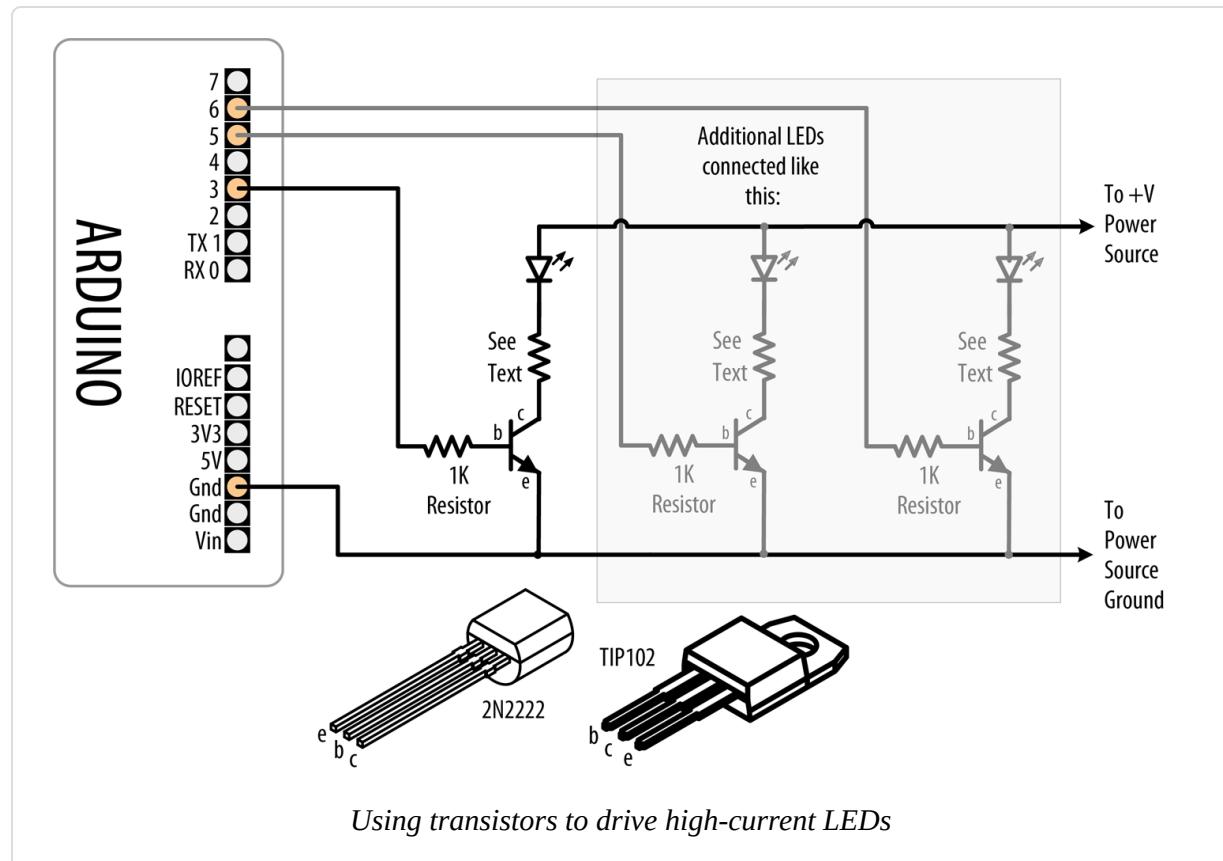
# 7.3 Driving High-Power LEDs

## Problem

You need to switch or control the intensity of LEDs that need more power than the Arduino pins can provide. Arduino Uno and Mega chips can only handle current up to 40 mA per pin.

## Solution

Use a transistor to switch on and off the current flowing through the LEDs. Connect the LED as shown in Figure 7-4. You can use the same code as shown in Recipes 7.1 and 7.2 (just make sure the pins connected to the transistor base match the pin number used in your sketch).



*Using transistors to drive high-current LEDs*

## Discussion

Figure 7-4 has an arrow indicating a +V power source. This can be the Arduino +5V power pin, which can supply up to 400 mA or so if powered from USB. The available current when powered through the external power socket is dependent on the current rating and voltage of your DC power supply (the regulator dissipates excess voltage as heat—check that the onboard regulator, a 3-pin chip usually near the DC input socket, is not too hot to the touch). If more current is required than the Arduino +5V can provide, you need a power source separate from the Arduino to drive the LEDs. See Appendix C for information on using an external power supply.
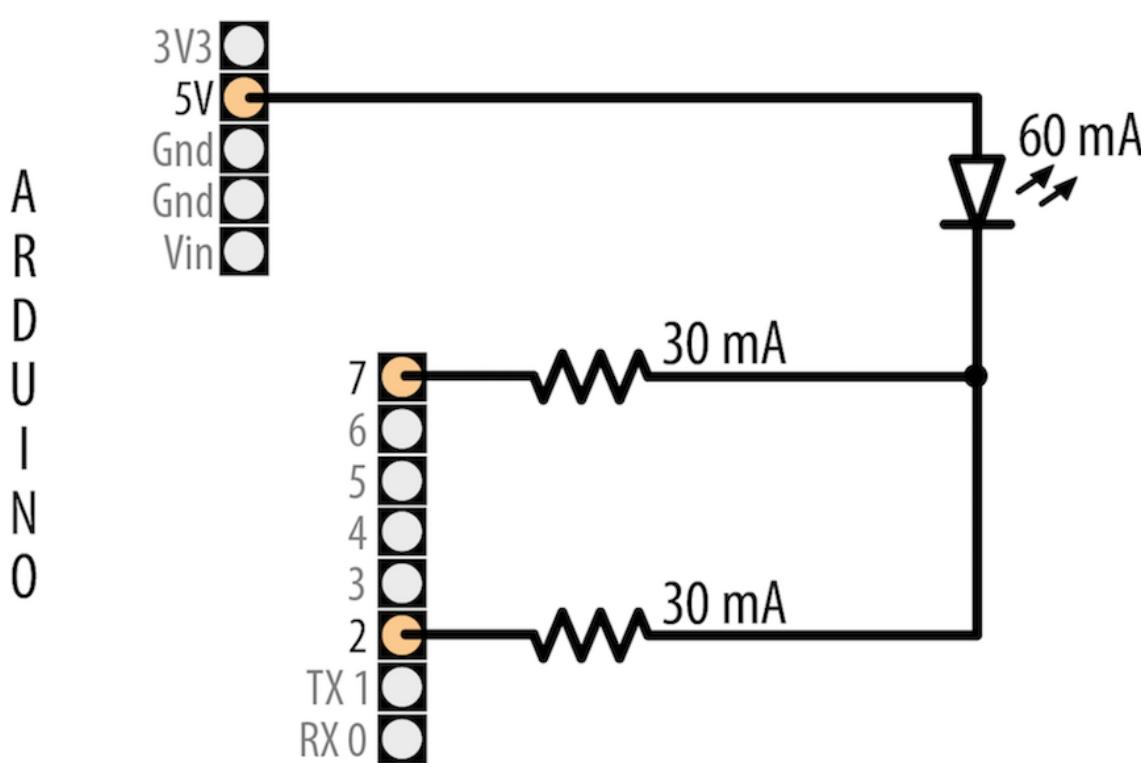
Current is allowed to flow from the collector to the emitter when the transistor is switched on. No significant current flows when the transistor is off. The Arduino can turn a transistor on by making the voltage on a pin `HIGH` with `digitalWrite`. A resistor is necessary between the pin and the transistor base to prevent too much current from flowing—1K ohms is a typical value (this provides 5 mA of current to the base of the transistor). See Appendix B for advice on how to read a datasheet and pick and use a transistor. You can also use specialized integrated circuits such as the ULN2003A for driving multiple outputs. These contain seven high-current (0.5 amp) output drivers.

The resistor used to limit the current flow through the LED is calculated using the technique given in Recipe 7.1, but you may need to take into account that the source voltage will be reduced slightly because of the small voltage drop through the transistor. This will usually be less than three-fourths of a volt (the actual value can be found by looking at collector-emitter saturation voltage; see Appendix B). High-current LEDs (1 watt or more) are best driven using a constant current source (a circuit that actively controls the current) to manage the current through the LED.

### How to exceed 40 mA on an ATmega chip

If your board uses an ATmega chip you can also connect multiple pins in parallel to increase current beyond the 40 mA per pin rating (see "Maximum pin current").

Figure 7-5 shows how to connect an LED that can be driven with 60 mA through two pins. This shows the LED connecting the resistors to ground through pins 2 and 7—both pins need to be `LOW` for the full 60 mA to flow through the LED. The separate resistors are needed; don't try to use a single resistor to connect the two pins.

*How to exceed 40 mA*

This technique can also be used to source current. For example, flip the LED around—connect the lead that was going to the resistors (cathode) to GND and the other end (anode) to the resistors—and you illuminate the LED by setting both pins to `HIGH`.

It is best if you use pins that are not adjacent to minimize stress on the chip. This technique works for any pin using `digitalWrite`; it does not work with `analogWrite`—if you need more current for analog outputs (PWM), you will need to use transistors as explained previously.

This technique is not recommended for use on 32-bit boards.

## See Also

The Web reference for constant current drivers
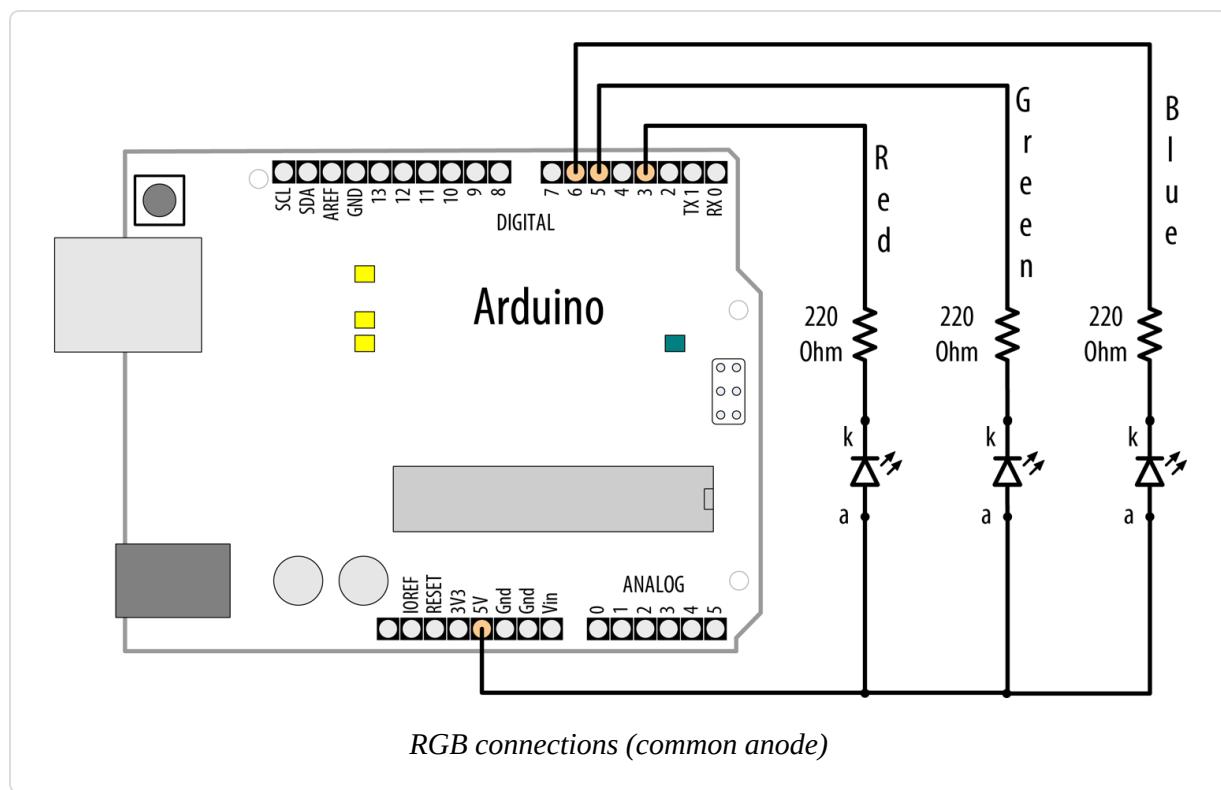
# 7.4 Adjusting the Color of an LED

## Problem

You want to control the color of an RGB LED under program control.

## Solution

RGB LEDs have red, green, and blue elements in a single package, with either the anodes connected together (known as *common anode*) or the cathodes connected together (known as *common cathode*). Use the wiring in Figure 7-6 for common anode (the anodes are connected to +5 volts and the cathodes are connected to pins). Use Figure 7-2 if your RGB LEDs are common cathode.



*RGB connections (common anode)*

This sketch continuously fades through the color spectrum by varying the intensity of the red, green, and blue elements:

```
/*
 * RGB_LEDs sketch
 * RGB LEDs driven from PWM output ports
```

```
*/

const int redPin   = 3;   // choose the pin for each of the LEDs
const int greenPin = 5;
const int bluePin  = 6;
const bool invert = true; // set true if common anode, false if common cathode

int color = 0; // a value from 0 to 255 representing the hue
int R, G, B;   // the Red Green and Blue color components

void setup()
{
  // pins driven by analogWrite do not need to be declared as outputs
}

void loop()
{
  int brightness = 255;       // 255 is maximum brightness

  hueToRGB(color, brightness); // call function to convert hue to RGB

  // write the RGB values to the pins
  analogWrite(redPin,   R);
  analogWrite(greenPin, G);
  analogWrite(bluePin,  B);

  color++; // increment the color
  if (color > 255)
    color = 0;
  delay(10);
}

// function to convert a color to its Red, Green, and Blue components.
//
void hueToRGB(int hue, int brightness)
{
  unsigned int scaledHue = (hue * 6);

  // segment 0 to 5 around the color wheel
  unsigned int segment = scaledHue / 256;

  // position within the segment
  unsigned int segmentOffset = scaledHue - (segment * 256);

  unsigned int complement = 0;
  unsigned int prev = (brightness * ( 255 -  segmentOffset)) / 256;
  unsigned int next = (brightness *  segmentOffset) / 256;
  if (invert)
```

```
  {
    brightness = 255 - brightness;
    complement = 255;
    prev = 255 - prev;
    next = 255 - next;
  }

  switch (segment) {
    case 0:    // red
      R = brightness;
      G = next;
      B = complement;
      break;
    case 1:    // yellow
      R = prev;
      G = brightness;
      B = complement;
      break;
    case 2:    // green
      R = complement;
      G = brightness;
      B = next;
      break;
    case 3:    // cyan
      R = complement;
      G = prev;
      B = brightness;
      break;
    case 4:    // blue
      R = next;
      G = complement;
      B = brightness;
      break;
    case 5:    // magenta
    default:
      R = brightness;
      G = complement;
      B = prev;
      break;
  }
}
```

## Discussion

The color of an RGB LED is determined by the relative intensity of its red, green, and blue elements. The core function in the sketch (hueToRGB)

handles the conversion of a hue value ranging from 0 to 255 into a corresponding color ranging from red to blue. The spectrum of visible colors is often represented using a color wheel consisting of the primary and secondary colors with their intermediate gradients. The spokes of the color wheel representing the six primary and secondary colors are handled by six `case` statements. The code in a `case` statement is executed if the segment variable matches the case number, and if so, the RGB values are set as appropriate for each. Segment 0 is red, segment 1 is yellow, segment 2 is green, and so on.

If you also want to adjust the brightness, you can reduce the value of the `brightness` variable. The following shows how to adjust the brightness with a variable resistor or sensor connected as shown in Figures 7-14 and 7-18:

```
int brightness = map(analogRead(A0),0,1023,0,255);
```

The `brightness` variable will range in value from 0 to 255 as the analog input ranges from 0 to 1,023, causing the LED to increase brightness as the value increases.

### See Also

Recipe 2.16

# 7.5 Controlling Lots of Color LEDs

### Problem

You want to control the color of many LEDs using a single pin.

### Solution

This recipe shows how to use smart RGB LEDs with a tiny controller built into each LED that enables many LEDs to be controlled from a single digital pin. This sketch uses the Adafruit Neopixels library (installed using

the Arduino Library Manager) to change LED colors based on readings from an analog pin. Figure 7-7 shows the connection for a NeoPixel ring and a potentiometer to control the color:

```
/*
 * SimplePixel sketch
 * LED color changes with sensor value
 */

#include <Adafruit_NeoPixel.h>

const int sensorPin = A0; // analog pin for sensor
const int ledPin = 6;     // the pin the LED strip is connected to
const int count  = 8;     // how many LEDs in the strip

// declare LED strip
Adafruit_NeoPixel leds = Adafruit_NeoPixel(count, ledPin, NEO_GRB +
NEO_KHZ800);

void setup()
{
  leds.begin(); // initialize LED strip
  for (int i = 0; i < count; i++) {
    leds.setPixelColor(i, leds.Color(0,0,0)); // turn each LED off
  }
  leds.show();  // refresh the strip with the new pixel values (all off)
}

void loop()
{
  static unsigned int last_reading = -1;

  int reading =  analogRead(sensorPin);
  if (reading != last_reading) { // If the value has changed

    // Map the analog reading to the color range of the NeoPixel
    unsigned int mappedSensorReading = map(reading, 0, 1023, 0, 65535);

    // Update the pixels with a slight delay to create a sweeping effect
    for (int i = 0; i < count; i++) {
      leds.setPixelColor(i, leds.gamma32(
              leds.ColorHSV(mappedSensorReading, 255, 128)));
      leds.show();
      delay(25);
    }
    last_reading = reading;
```
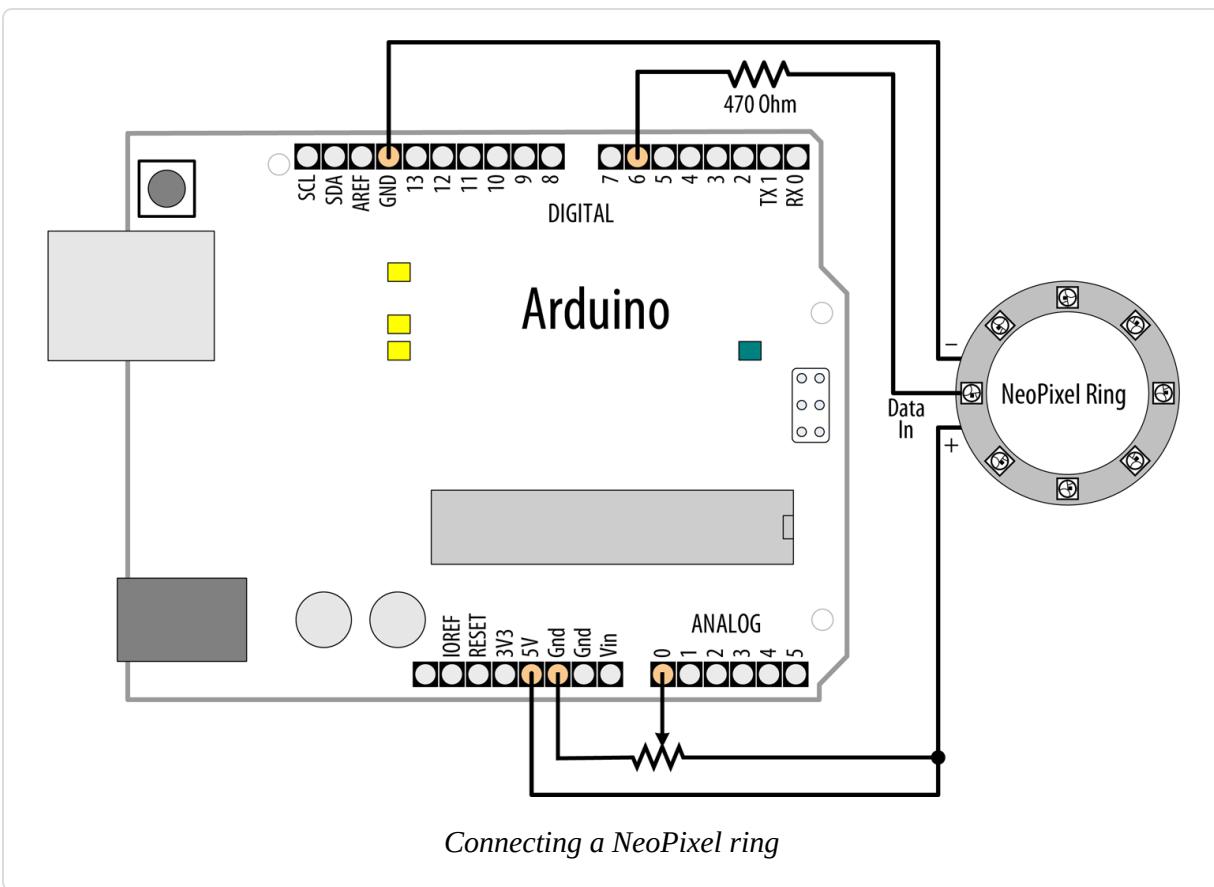
```
      }
    }
```



*Connecting a NeoPixel ring*

---

**WARNING**

If you are using a 3.3V board, you will need to connect both the potentiometer and NeoPixel's positive lead to 3.3V instead of 5V.

---

## Discussion

This sketch drives a stick, a strand, or a group of chained Adafruit NeoPixels that have eight RGB LEDs. You can change the variable `numOfLeds` if you connect a different number of LEDs, but bear in mind that each LED could consume up to 60 mA (if you set to white at full brightness). A USB port can power up to eight, but above that you will need to attach the strip's power connectors to a higher-current 5V power supply,

but you must connect the power supply's ground to Arduino ground. If you are using a 3.3V board, you should not power the NeoPixels with more than 3.7V (such as from a lithium ion polymer battery), because NeoPixels require a data signal that's close to their supply voltage. When using an external power supply, you should also connect a 1,000 uf capacitor between the positive and negative supply pins to protect the pixels (check the polarity on the capacitor and make sure you are connecting it correctly).

The `leds` variable is declared with this line of code:

```
Adafruit_NeoPixel leds = Adafruit_NeoPixel(count, ledPin, NEO_GRB +
NEO_KHZ800);
```

This creates the memory structure to store the color of each LED and communicate with the strip. You specify the number of LEDs in the strip (`count`), the Arduino pin the data line is connected to (`ledPin`), and the type of LED strip you are using (in this case: `NEO_GRB+NEO_KHZ800`). You will need to check the documentation for the library and your strip to see if you need a different setting, but you won't do any harm to try all the options listed with the library to find one that works.

To set the color of an individual LED you use the `led.setPixelColor` method. You need to specify the number of the LED (starting at 0 for the first one) and the desired color. To transfer data to the LEDs you need to call `led.show`. You can alter multiple LEDs' values before calling `led.show` to make them change together. Values not altered will remain at their previous settings. When you create the `Adafruit_NeoPixel` object, all the values are initialized to 0.

The NeoPixel library includes its own function for converting a hue to an RGB value: `ColorHSV`. The first argument is the hue, the second is the color saturation, and the third is brightness. The `gamma32` function performs a conversion on the output of `ColorHSV` to compensate between the way that computers represent colors and the way that humans perceive them.

Each LED "pixel" has connections for data input and output, power, and ground. Arduino drives the data input of the first pixel, the data output of

which is connected to the data input of the next in the chain. You can buy individual pixels or strips that come pre-connected.

---

### If Your Strip Is Not Supported by the Adafruit Library

Early LED strips used the WS2811 chip. There have been various other versions since: WS2812, WS2812B, and APA102, for example. If your LEDs are not supported by the Adafruit library, then try the Fast LED library.

---

The Arduino-compatible Teensy 3.x and higher can control eight strips on different pins and uses a combination of high-speed hardware and software to enable very high-quality animation.

The LEDs are available individually, but also on flexible strips in rolls, at different spacings (specified in LEDs/meter or foot). Adafruit produces a wide range of PCB form factors including circles of LEDs, short strips, and panels under the brand name NeoPixel.

## See Also

The Adafruit NeoPixel Uber Guide

Teensy library, which also has some good pictures of wiring for power supplies with large numbers of LEDs, and a processing program that extracts the data from video for you to add to code to display the video on the strands.
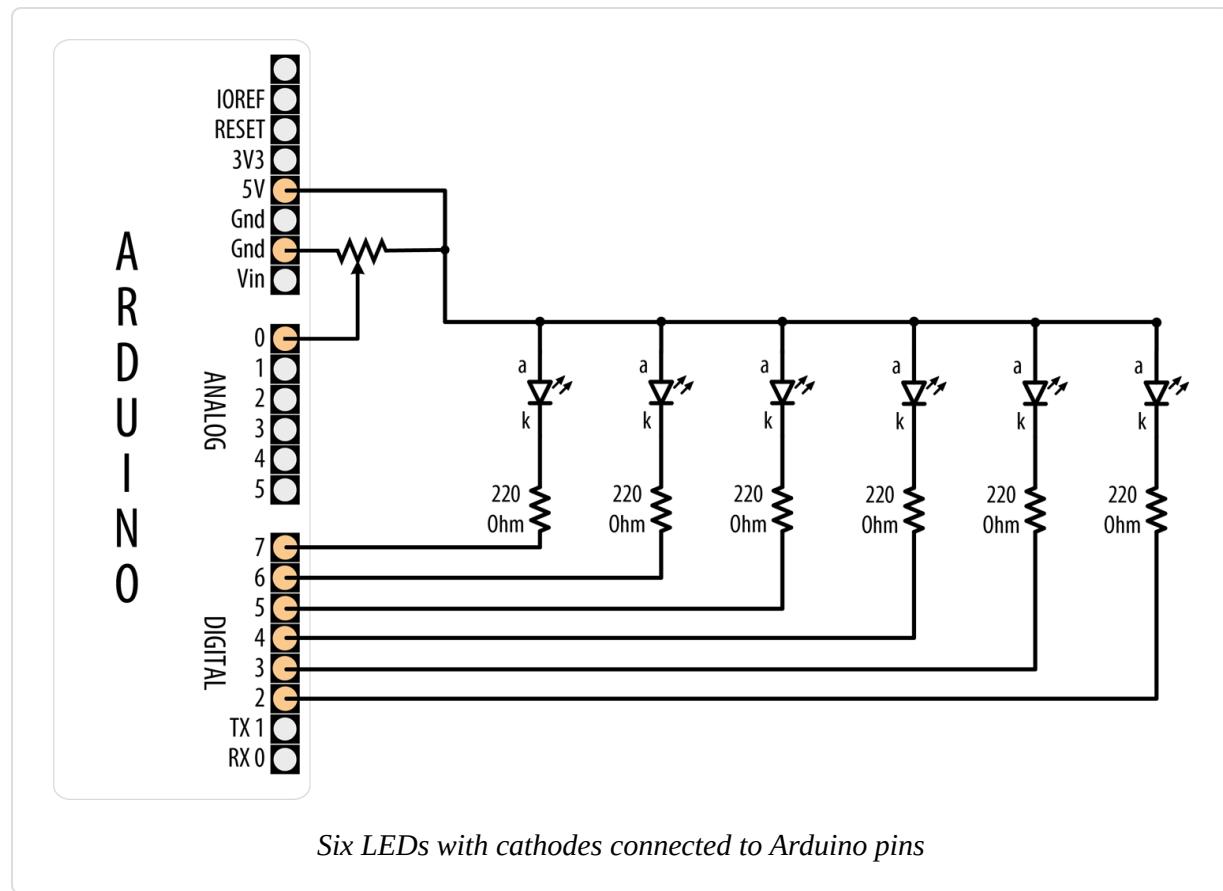
# 7.6 Sequencing Multiple LEDs: Creating a Bar Graph

## Problem

You want an LED bar graph that lights LEDs in proportion to a value in your sketch or a value read from a sensor.

## Solution

You can connect the LEDs as shown in Figure 7-2 (using additional pins if you want more LEDs). Figure 7-8 shows six LEDs connected on consecutive pins.



*Six LEDs with cathodes connected to Arduino pins*

The following sketch turns on a series of LEDs, with the number being proportional to the value of a sensor connected to an analog input port (see Figure 7-14 or Figure 7-18 to see how a sensor is connected):

```
/*
 * Bargraph sketch
 *
 * Turns on a series of LEDs proportional to a value of an analog sensor.
 * Six LEDs are controlled but you can change the number of LEDs by changing
 * the value of NbrLEDs and adding the pins to the ledPins array
 */

const int NbrLEDs = 6;
const int ledPins[] = { 2, 3, 4, 5, 6, 7 };
```

```
const int analogInPin = A0; // Analog input pin connected to variable resistor

// Swap values of the following two #defines if cathodes are connected to Gnd
#define LED_ON   LOW
#define LED_OFF HIGH

int sensorValue = 0;  // value read from the sensor
int ledLevel = 0;     // sensor value converted into LED 'bars'

void setup()
{
  for (int led = 0; led < NbrLEDs; led++)
  {
    pinMode(ledPins[led], OUTPUT);  // make all the LED pins outputs
  }
}

void loop()
{
  sensorValue = analogRead(analogInPin);           // read the analog in
value
  ledLevel = map(sensorValue, 10, 1023, 0, NbrLEDs); // map to the number of
LEDs
  for (int led = 0; led < NbrLEDs; led++)
  {
    if (led < ledLevel) {
      digitalWrite(ledPins[led], LED_ON);  // turn on pins below the level
    }
    else {
      digitalWrite(ledPins[led], LED_OFF); // turn off pins higher than the
level
    }
  }
}
```

## Discussion

The pins connected to LEDs are held in the array `ledPins`. To change the number of LEDs, you can add (or remove) elements from this array, but make sure the variable `NbrLEDs` is the same as the number of elements (which should be the same as the number of pins). You can have the compiler calculate the value of `NbrLEDs` for you by replacing this line:

```
const int NbrLEDs = 6;
```

with this line:

```
const int NbrLEDs = sizeof(ledPins) / sizeof(ledPins[0]);
```

The `sizeof` function returns the size (number of bytes) of a variable—in this case, the number of bytes in the `ledPins` array. Because it is an array of integers (with two bytes per element), the total number of bytes in the array is divided by the size of one element (`sizeof(ledPins[0])`) and this gives the number of elements.

The Arduino `map` function is used to calculate the number of LEDs that should be lit as a proportion of the sensor value. The code loops through each LED, turning it on if the proportional value of the sensor is greater than the LED number. For example, if the sensor value is below `10`, no pins are lit; if the sensor is at half value, half are lit. In an ideal world, a potentiometer at its lowest setting will return zero, but it's likely to drift in the real world. When the sensor is at maximum value, all the LEDs are lit. If you find that the last LED flickers when the potentiometer is at its maximum value, try lowering the second argument to `map` from `1023` to `1000` or so.

Figure 7-8 shows all the anodes connected together (known as *common anode*) and the cathodes connected to the pins; the pins need to be `LOW` for the LED to light. If the LEDs have the anodes connected to pins (as shown in Figure 7-2) and the cathodes are connected together (known as *common cathode*), the LED is lit when the pin goes `HIGH`. The sketch in this recipe uses the constant names `LED_ON` and `LED_OFF` to make it easy to select common anode or common cathode connections. To change the sketch for common cathode connection, swap the values of these constants as follows:

```
const bool LED_ON = HIGH;  // HIGH is on when using common cathode connection
const bool LED_OFF = LOW;
```

You may want to slow down the *decay* (rate of change) in the lights; for example, to emulate the movement of the indicator of a sound volume

meter. Here is a variation on the sketch that slowly decays the LED bars when the level drops:

```
/*
 *  LED bar graph - decay version
 */

const int ledPins[] = {2, 3, 4, 5, 6, 7};
const int NbrLEDs = sizeof(ledPins) / sizeof(ledPins[0]);
const int analogInPin = A0; // Analog input pin connected to variable resistor
const int decay = 10;        // increasing this reduces decay rate of
storedValue

// Swap values of the following two #defines if cathodes are connected to Gnd
#define LED_ON  LOW
#define LED_OFF HIGH

// the stored (decaying) sensor value
int storedValue = 0;

void setup()
{
  for (int led = 0; led < NbrLEDs; led++)
  {
    pinMode(ledPins[led], OUTPUT);  // make all the LED pins outputs
  }
}

void loop()
{
  int sensorValue = analogRead(analogInPin);       // read the analog in value
  storedValue = max(sensorValue, storedValue);     // use sensor value if
higher
  int ledLevel = map(storedValue, 10, 1023, 0, NbrLEDs);// map to number of
LEDs

  for (int led = 0; led < NbrLEDs; led++)
  {
    if (led < ledLevel ) {
      digitalWrite(ledPins[led], LED_ON);  // turn on pins less than the level
    }
    else {
      digitalWrite(ledPins[led], LED_OFF);   // turn off pins higher
                                             // than the level

    }
  }
```

```
    storedValue = storedValue - decay;        // decay the value
    delay(10);                                // wait 10 ms before next loop
}
```

The decay is handled by the line that uses the `max` function. This returns either the sensor value or the stored decayed value, whichever is higher. If the sensor is higher than the decayed value, this is saved in `storedValue`. Otherwise, the level of `storedValue` is reduced by the constant `decay` each time through the loop (set to 10 ms by the `delay` function). Increasing the value of the `decay` constant will reduce the time for the LEDs to fade to all off.

You could implement this bar graph using NeoPixels, as mentioned in the previous recipe. The code for this would be:

```
/*
 * PixelBarGraph.ino
 * Sensor value determines how many LEDs to light
 */

#include <Adafruit_NeoPixel.h>

const int sensorPin = A0; // analog pin for sensor

const int ledsPin = 2;    // the pin the LED strip is connected to
const int numOfLeds = 16; // how many LEDs in the strip

//used to automatically map sensor values
const int minReading = 0;
const int maxReading = 1023;

//declare LED strip
Adafruit_NeoPixel leds =
  Adafruit_NeoPixel(numOfLeds, ledsPin, NEO_GRB + NEO_KHZ800);

void setup()
{
  leds.begin();    //initialize led strip
  leds.setBrightness(25);
}

void loop()
{
```

```
    int sensorReading =  analogRead(A0);
    int nbrLedsToLight = map(sensorReading, minReading, maxReading, 0,
numOfLeds);

    for (int i = 0; i < numOfLeds; i++)
    {
      if ( i < nbrLedsToLight)
        leds.setPixelColor(i, leds.Color(0, 0, 255)); // blue
      else
        leds.setPixelColor(i, leds.Color(0, 255, 0)); // green
    }
    leds.show();
}
```

## See Also

Recipe 3.6 explains the `max` function.

Recipe 5.6 has more on reading a sensor with the `analogRead` function.

Recipe 5.7 describes the `map` function.

See Recipes 12.2 and 12.1 if you need greater precision in your decay times. The total time through the loop is actually greater than 10 ms because it takes an additional millisecond or so to execute the rest of the loop code.

# 7.7 Sequencing Multiple LEDs: Making a Chase Sequence

## Problem

You want to light LEDs in a "chasing lights" sequence. This sequence was used in special effects on the TV shows *Knight Rider* and *Battlestar Galactica*, both created by Glen A. Larson, so this effect is also called a "Larson Scanner."

## Solution

You can use the same connection as shown in Figure 7-8:

```
/* Chaser
 */

const int NbrLEDs = 6;
const int ledPins[] = {2, 3, 4, 5, 6, 7};
const int wait_time = 30;

// Swap values of the following two #defines if cathodes are connected to Gnd
#define LED_ON  LOW
#define LED_OFF HIGH

void setup()
{
  for (int led = 0; led < NbrLEDs; led++)
  {
    pinMode(ledPins[led], OUTPUT);
  }
}

void loop()
{
  for (int led = 0; led < NbrLEDs - 1; led++)
  {
    digitalWrite(ledPins[led], LED_ON);
    delay(wait_time);
    digitalWrite(ledPins[led + 1], LED_ON);
    delay(wait_time);
    digitalWrite(ledPins[led], LED_OFF);
    delay(wait_time * 2);
  }
  for (int led = NbrLEDs - 1; led > 0; led--) {
    digitalWrite(ledPins[led], LED_ON);
    delay(wait_time);
    digitalWrite(ledPins[led - 1], LED_ON);
    delay(wait_time);
    digitalWrite(ledPins[led], LED_OFF);
    delay(wait_time * 2);
  }
}
```

## Discussion

This code is similar to the code in Recipe 7.6, except the pins are turned on and off in a fixed sequence rather than depending on a sensor level. There are two `for` loops; the first produces the left-to-right pattern by lighting up

LEDs from left to right. This loop starts with the first (leftmost) LED and steps through adjacent LEDs until it reaches and illuminates the rightmost LED. The second `for` loop lights the LEDs from right to left by starting at the rightmost LED and decrementing (decreasing by one) the LED that is lit until it gets to the first (rightmost) LED. The delay period is set by the `wait` variable and can be chosen to provide the most pleasing appearance.

# 7.8 Controlling an LED Matrix Using Multiplexing

## Problem

You have a matrix of LEDs and want to minimize the number of Arduino pins needed to turn LEDs on and off.

## Solution

This sketch uses an LED matrix of 64 LEDs, with anodes connected in rows and cathodes in columns (as in the Jameco 2132349). Figure 7-9 shows the connections. (Dual-color LED displays may be easier to obtain, and you can drive just one of the colors if that is all you need.)

> **WARNING**
>
> This is a relatively power-hungry solution, and is only suitable for the Arduino Uno and other boards based on the ATmega328. The Uno WiFi Rev2 and Nano Every, as well as most (if not all) 32-bit boards, cannot safely deliver enough current to drive all these LEDs. See Recipes 7.10 or 7.14 for a suitable Solution.

```
/*
 * matrixMpx sketch
 *
 * Sequence LEDs starting from first column and row until all LEDs are lit
 * Multiplexing is used to control 64 LEDs with 16 pins
 */

const int columnPins[] = {2, 3, 4, 5, 6, 7, 8, 9};
```

```
const int rowPins[]     = {10,11,12,A1,A2,A3,A4,A5};

int pixel       = 0; // 0 to 63 LEDs in the matrix
int columnLevel = 0; // pixel value converted into LED column
int rowLevel    = 0; // pixel value converted into LED row

void setup()
{
  for (int i = 0; i < 8; i++)
  {
    pinMode(columnPins[i], OUTPUT);  // make all the LED pins outputs
    pinMode(rowPins[i], OUTPUT);
  }
}

void loop()
{
  pixel = pixel + 1;
  if(pixel > 63)
      pixel = 0;

  columnLevel = pixel / 8;  // map to the number of columns
  rowLevel = pixel % 8;     // get the fractional value

  for (int column = 0; column < 8; column++)
  {
    digitalWrite(columnPins[column], LOW);     // connect this column to GND
    for(int row = 0; row < 8; row++)
    {
      if (columnLevel > column)
      {
        digitalWrite(rowPins[row], HIGH);  // connect all LEDs in row to +V
      }
      else if (columnLevel == column && rowLevel >= row)
      {
          digitalWrite(rowPins[row], HIGH);
      }
      else
      {
        digitalWrite(columnPins[column], LOW); // turn off all LEDs in this
row
      }
      delayMicroseconds(300);    // delay gives frame time of 20 ms for 64
LEDs
      digitalWrite(rowPins[row], LOW);  // turn off LED
    }

    // disconnect this column from Ground
```
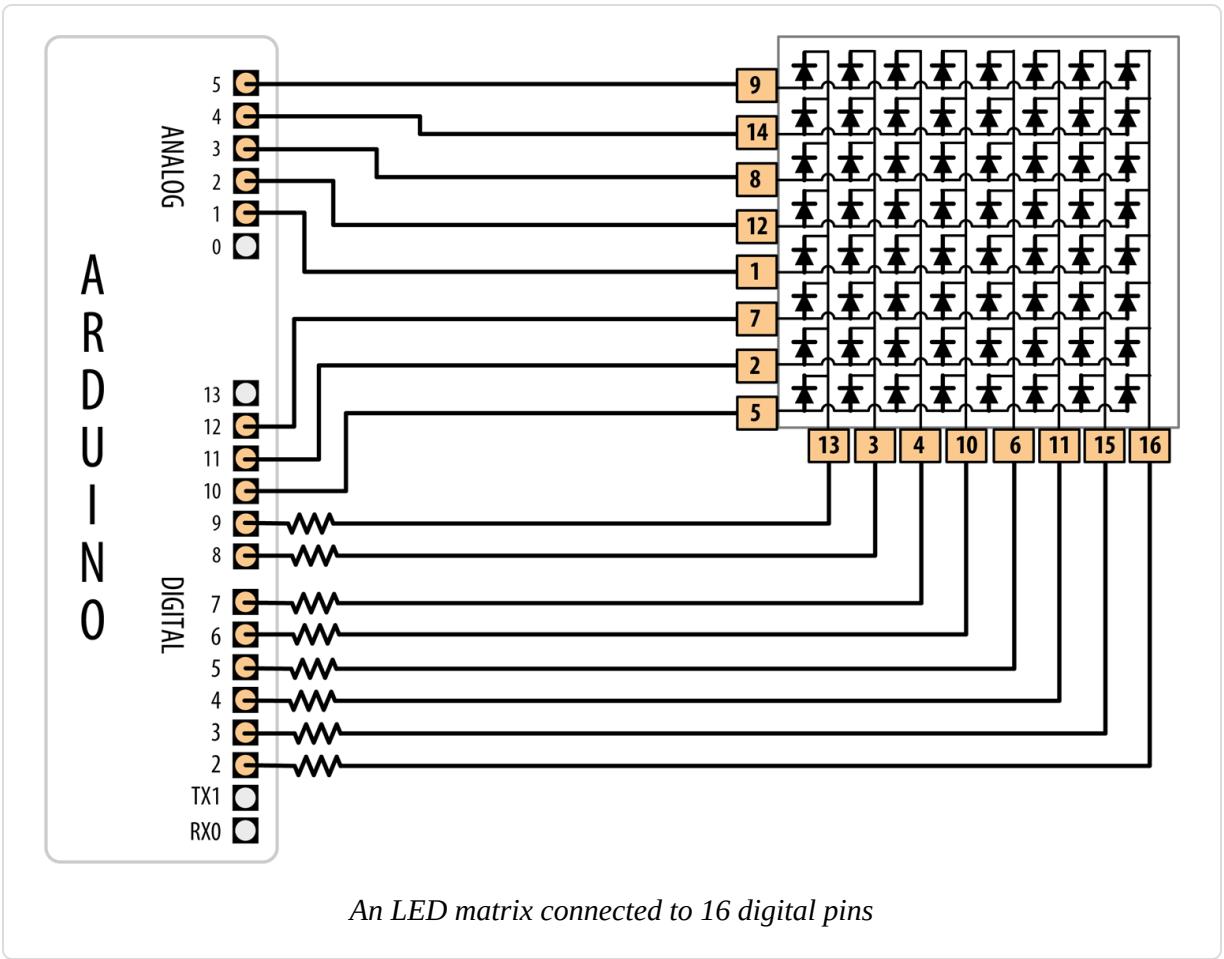
```
    digitalWrite(columnPins[column], HIGH);
  }
}
```



*An LED matrix connected to 16 digital pins*

### WARNING

The wiring shown is based on Jameco part number 2132349, which represents a common form factor for this kind of array. But LED matrix displays do not have a standard pinout, so you must check the datasheet for your display. Wire the rows of anodes and columns of cathodes as shown in Figure 7-16 or Figure 7-17, but use the LED pin numbers in your datasheet.

Figure 7-9 illustrates the logical arrangement of the pins as they relate to columns and rows. The numbers of the pins in the diagram correspond to the physical layout. Generally, the pin numbering follows a U-shaped pattern starting in the top left (1–8 from the top of the left column of pins to

the bottom; 9–16 from the bottom of the right column of pins to the top). The trick is orienting the part so that pin 1 is in the upper left. You will need to look for an indentation, often in the shape of a small dot, that indicates which pin is 1. It is probably on the side, directly on the casing of the component. When in doubt, check the datasheet.

## Discussion

The resistor's value must be chosen to ensure that the maximum current through a pin does not exceed 40 mA on the Arduino Uno (and other boards based on the ATmega328; do not use this Solution with a 3.3V board or any board that cannot handle 40 mA per pin). Because the current for up to eight LEDs can flow through each column pin, the maximum current for each LED must be one-eighth of 40 mA, or 5 mA. Each LED in a typical small red matrix has a forward voltage of around 1.8 volts. Calculating the resistor that results in 5 mA with a forward voltage of 1.8 volts gives a value of 680 ohms. Check your datasheet to find the forward voltage of the matrix you want to use. Each column of the matrix is connected through the series resistor to a digital pin. When the column pin goes low and a row pin goes high, the corresponding LED will light. For all LEDs where the column pin is high or its row pin is low, no current will flow through the LED and it will not light.

The `for` loop scans through each row and column and turns on sequential LEDs until all LEDs are lit. The loop starts with the first column and row and increments the row counter until all LEDs in that row are lit; it then moves to the next column, and so on, lighting another LED with each pass through the loop until all the LEDs are lit.

You can control the number of lit LEDs in proportion to the value from a sensor (see Recipe 5.6 for connecting a sensor to the analog port) by making the following changes to the sketch.

Comment out or remove these three lines from the beginning of the loop:

```
pixel = pixel + 1;
  if(pixel > 63)
```

```
        pixel = 0;
```

Replace them with the following lines that read the value of a sensor on pin 0 and map this to a number of pixels ranging from 0 to 63:

```
int sensorValue = analogRead(0);                // read the analog in value
  pixel = map(sensorValue, 0, 1023, 0, 63);     // map sensor value to pixel
(LED)
```

You can test this with a variable resistor connected to analog input pin 0 connected as shown in Figure 5-7 in Chapter 5. The number of LEDs lit will be proportional to the value of the sensor.

You don't have to light an entire row at once. The following sketch will light one LED at a time as it goes through the sequence:

```
/*
 * matrixMpx sketch, one at a time
 *
 * Sequence LEDs starting from first column and row, one at a time
 * Multiplexing is used to control 64 LEDs with 16 pins
 */

const int columnPins[] = {2, 3, 4, 5, 6, 7, 8, 9};
const int rowPins[]    = {10,11,12,A1,A2,A3,A4,A5};

int pixel = 0; // 0 to 63 LEDs in the matrix

void setup()
{
  for (int i = 0; i < 8; i++)
  {
    pinMode(columnPins[i], OUTPUT);  // make all the LED pins outputs
    pinMode(rowPins[i], OUTPUT);
    digitalWrite(columnPins[i], HIGH);
  }
}

void loop()
{
  pixel = pixel + 1;
  if(pixel > 63)
      pixel = 0;
```

```
    int column = pixel / 8;   // map to the number of columns
    int row = pixel % 8;      // get the fractional value

    digitalWrite(columnPins[column], LOW);   // Connect this column to GND
    digitalWrite(rowPins[row], HIGH);        // Take this row HIGH

    delay(125);   // pause briefly

    digitalWrite(rowPins[row], LOW);         // Take the row low
    digitalWrite(columnPins[column], HIGH); // Disconnect the column from GND

  }
```

# 7.9 Displaying Images on an LED Matrix

## Problem

You want to display one or more images on an LED matrix, perhaps
creating an animation effect by quickly alternating multiple images.

## Solution

This Solution can use the same wiring as in Recipe 7.8. The sketch creates
the effect of a heart beating by briefly lighting LEDs arranged in the shape
of a heart. A small heart followed by a larger heart is flashed for each
heartbeat (the images look like Figure 7-10):

```
  /*
   * matrixMpxAnimation sketch
   * animates two heart images to show a beating heart
   */

  // the heart images are stored as bitmaps - each bit corresponds to an LED
  // a 0 indicates the LED is off, 1 is on
  byte bigHeart[] = {
    B01100110,
    B11111111,
    B11111111,
    B11111111,
    B01111110,
    B00111100,
    B00011000,
    B00000000};
```

```arduino
byte smallHeart[] = {
  B00000000,
  B00000000,
  B00010100,
  B00111110,
  B00111110,
  B00011100,
  B00001000,
  B00000000};

const int columnPins[] = { 2, 3, 4, 5, 6, 7, 8, 9};
const int rowPins[]    = {10,11,12,A1,A2,A3,A4,A5};

void setup() {
  for (int i = 0; i < 8; i++)
  {
    pinMode(rowPins[i], OUTPUT);        // make all the LED pins outputs
    pinMode(columnPins[i], OUTPUT);
    digitalWrite(columnPins[i], HIGH);  // disconnect column pins from Ground
  }
}

void loop() {
  int pulseDelay = 800 ; // milliseconds to wait between beats

  show(smallHeart, 80);   // show the small heart image for 80 ms
  show(bigHeart, 160);    // followed by the big heart for 160 ms
  delay(pulseDelay);      // show nothing between beats
}

// Show a frame of an image stored in the array pointed to by the image
// parameter. The frame is repeated for the given duration in milliseconds.
void show(byte * image, unsigned long duration)
{
 unsigned long start = millis();      // begin timing the animation
 while (start + duration > millis()) // loop until the duration has passed
  {
    for(int row = 0; row < 8; row++)
    {
      digitalWrite(rowPins[row], HIGH);        // connect row to +5 volts
      for(int column = 0; column < 8; column++)
      {
        bool pixel = bitRead(image[row],column);
        if(pixel == 1)
        {
          digitalWrite(columnPins[column], LOW);  // connect column to Gnd
        }
```
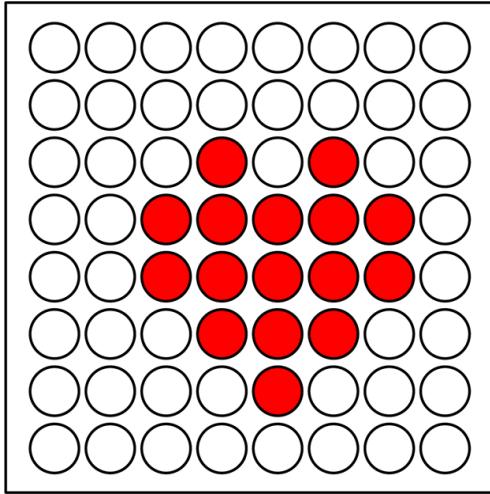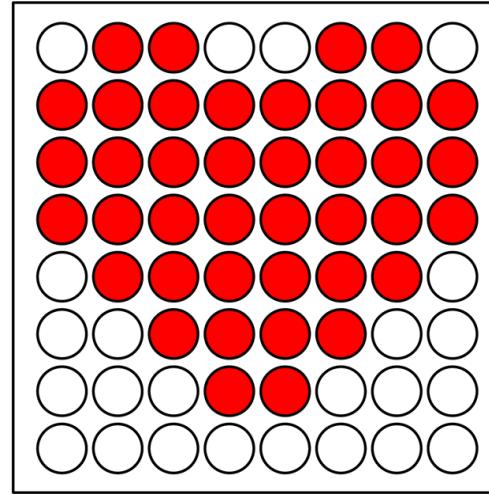
```
        delayMicroseconds(300);                  // a small delay for each
    LED
        digitalWrite(columnPins[column], HIGH);  // disconnect column from
    Gnd
      }
      digitalWrite(rowPins[row], LOW);           // disconnect LEDs
    }
  }
}
```



*The two heart images displayed on each beat*

## Discussion

Columns and rows are multiplexed (switched) similar to Recipe 7.8, but here the value written to the LED is based on images stored in the `bigHeart` and `smallHeart` arrays. Each element in the array represents a *pixel* (a single LED) and each array row represents a row in the matrix. A row consists of eight bits represented using binary format (as designated by the capital *B* at the start of each row). A bit with a value of `1` indicates that the corresponding LED should be on; a `0` means off. The animation effect is created by rapidly switching between the arrays.

The `loop` function waits a short time (800 ms) between beats and then calls the `show` function, first with the `smallHeart` array and then followed by the `bigHeart` array. The `show` function steps through each element in all the rows and columns, lighting the LED if the corresponding bit is 1. The `bitRead` function (see Recipe 2.20) is used to determine the value of each bit.

A short delay of 300 microseconds between each pixel allows the eye enough time to perceive the LED. The timing is chosen to allow each image to repeat quickly enough (50 times per second) so that blinking is not perceptible.

Here is a variation that changes the rate at which the heart beats, based on the value from a sensor. You can test this using a variable resistor connected to analog input pin 0, as shown in Recipe 5.6. Use the wiring and code shown earlier, except replace the `loop` function with this code:

```
void loop() {
  int sensorValue = analogRead(A0);              // read the analog in value
  int pulseRate =  map(sensorValue,0,1023,40,240); // convert to beats /
minute
  int pulseDelay = (60000 / pulseRate);   // milliseconds to wait between beats

  show(smallHeart, 80);              // show the small heart image for 100 ms
  show(bigHeart, 160);               // followed by the big heart for 200 ms
  delay(pulseDelay);                 // show nothing between beats
}
```

This version calculates the delay between pulses using the `map` function (see Recipe 5.7) to convert the sensor value into beats per minute. The calculation does not account for the time it takes to display the heart, but you can subtract 240 ms (80 ms plus 160 ms for the two images) if you want more accurate timing.

## See Also

See Recipes 7.13 and 7.14 for information on how to use shift registers to drive LEDs if you want to reduce the number of Arduino pins needed for

driving an LED matrix.

Recipes 12.2 and 12.1 provide more information on how to manage time using the `millis` function.

# 7.10 Controlling a Matrix of LEDs: Charlieplexing

## Problem

You have a matrix of LEDs and you want to minimize the number of pins needed to turn any of them on and off.

## Solution

*Charlieplexing* is a special kind of multiplexing that increases the number of LEDs that can be driven by a group of pins. This sketch sequences through six LEDs using just three pins. Figure 7-11 shows the connections (to calculate the correct resistor value for the LED connections, see Recipe 7.1):

```
/*
 * Charlieplexing sketch
 * light six LEDs in sequence that are connected to pins 2, 3, and 4
 */

int pins[] = {2,3,4};  // the pins that are connected to LEDs

// the next two lines calculate the number of pins and LEDs from the above
array
const int NUMBER_OF_PINS = sizeof(pins)/ sizeof(pins[0]);
const int NUMBER_OF_LEDS = NUMBER_OF_PINS * (NUMBER_OF_PINS-1);

byte pairs[NUMBER_OF_LEDS/2][2] = { {2,1}, {1,0}, {2,0} }; // maps pins to
LEDs

void setup()
{
    // nothing needed here
}
```
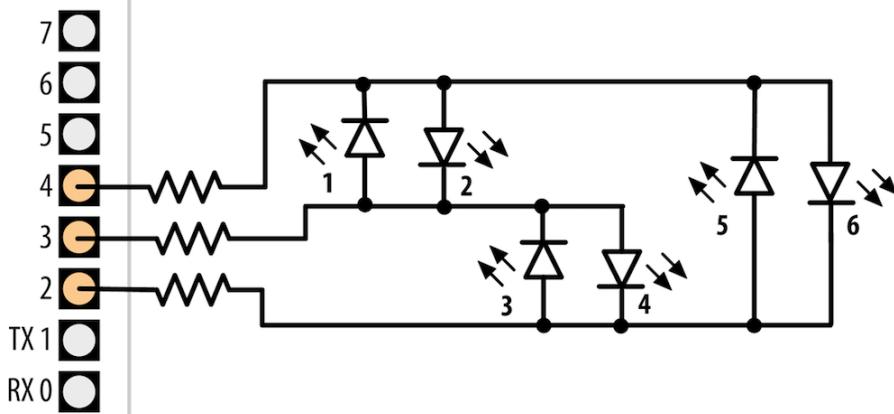
```arduino
void loop(){
    for(int i=0; i < NUMBER_OF_LEDS; i++)
    {
        lightLed(i);   // light each LED in turn
        delay(1000);
    }
}

// this function lights the given LED, the first LED is 0
void lightLed(int led)
{
  // the following four lines convert LED number to pin numbers
  int indexA = pairs[led/2][0];
  int indexB = pairs[led/2][1];
  int pinA = pins[indexA];
  int pinB = pins[indexB];

  // turn off all pins not connected to the given LED
  for(int i=0; i < NUMBER_OF_PINS; i++)
  {
    if(pins[i] != pinA && pins[i] != pinB)
    {  // if this pin is not one of our pins
        pinMode(pins[i], INPUT);    // set the mode to input
        digitalWrite(pins[i],LOW); // make sure pull-up is off
    }
  }
  // now turn on the pins for the given LED
  pinMode(pinA, OUTPUT);
  pinMode(pinB, OUTPUT);
  if( led % 2 == 0)
  {
    digitalWrite(pinA,LOW);
    digitalWrite(pinB,HIGH);
  }
  else
  {
    digitalWrite(pinB,LOW);
    digitalWrite(pinA,HIGH);
  }
}
```

*Six LEDs driven through three pins using Charlieplexing*
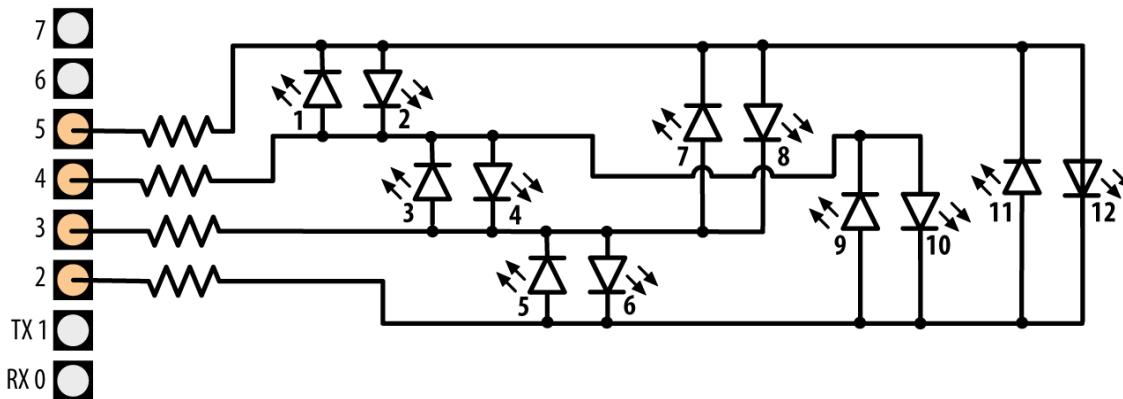
## Discussion

The term *Charlieplexing* comes from Charlie Allen (of Microchip Technology, Inc.), who published the method. The technique is based on the fact that LEDs only turn on when connected the "right way" around (with the anode more positive than the cathode). Here is the table showing the LED number (see Figure 7-9) that is lit for the valid combinations of the three pins. L is LOW, H is HIGH, and i is INPUT mode. Setting a pin in INPUT mode effectively disconnects it from the circuit:

```
Pins        LEDs
4  3  2     1  2  3  4  5  6
L  L  L     0  0  0  0  0  0
L  H  i     1  0  0  0  0  0
H  L  i     0  1  0  0  0  0
i  L  H     0  0  1  0  0  0
i  H  L     0  0  0  1  0  0
```

```
L  i  H     0  0  0  0  1  0
H  i  L     0  0  0  0  0  1
```

You can double the number of LEDs to 12 using just one more pin. The first six LEDs are connected in the same way as in the preceding example; add the additional six LEDs so that the connections look like Figure 7-12.



*Charlieplexing using four pins to drive 12 LEDs*

Modify the preceding sketch by adding the extra pin to the `pins` array:

```
byte pins[] = {2,3,4,5};  // the pins that are connected to LEDs
```

Add the extra entries to the `pairs` array so that it reads as follows:

```
byte pairs[NUMBER_OF_LEDS/2][2] = { {0,1}, {1,2}, {0,2}, {2,3}, {1,3}, {0,3}
};
```

Everything else can remain the same, so the loop will sequence through all 12 LEDs because the code determines the number of LEDs from the number of entries in the `pins` array.

Because Charlieplexing works by controlling the Arduino pins so that only a single LED is turned on at a time, it is more complicated to create the impression of lighting multiple LEDs. But you can light multiple LEDs using a multiplexing technique modified for Charlieplexing.

This sketch creates a bar graph by lighting a sequence of LEDs based on the value of a sensor connected to analog pin 0:

```
byte pins[] = {2,3,4};
const int NUMBER_OF_PINS = sizeof(pins)/ sizeof(pins[0]);
const int NUMBER_OF_LEDS = NUMBER_OF_PINS * (NUMBER_OF_PINS-1);

byte pairs[NUMBER_OF_LEDS/2][2] = { {2,1}, {1,0}, {2,0} }; // maps pins to
LEDs

int ledStates = 0; //holds states for up to 15 LEDs
int refreshedLed;  // the LED that gets refreshed

void setup()
{
   // nothing here
}

void loop()
{
const int analogInPin = 0; // Analog input pin connected to the variable
resistor

  // here is the code from the bargraph recipe
  int sensorValue = analogRead(analogInPin);         // read the analog in
value
  // map to the number of LEDs
  int ledLevel = map(sensorValue, 0, 1023, 0, NUMBER_OF_LEDS);
  for (int led = 0; led < NUMBER_OF_LEDS; led++)
  {
    if (led < ledLevel ) {
      setState(led, HIGH);     // turn on pins less than the level
    }
    else {
      setState(led, LOW);      // turn off pins higher than the level
    }
  }
  ledRefresh();

}

void setState( int led, bool state)
{
   bitWrite(ledStates,led, state);
}
```

```
void ledRefresh()
{
    // refresh a different LED each time this is called.
    if( refreshedLed++ > NUMBER_OF_LEDS) // increment to the next LED
        refreshedLed = 0; // repeat from the first LED if all have been
refreshed

    if( bitRead(ledStates, refreshedLed ) == HIGH)
        lightLed( refreshedLed );
    else
        if(refreshedLed == 0) // Turn them all off if pin 0 is off
          for(int i=0; i < NUMBER_OF_PINS; i++)
            digitalWrite(pins[i],LOW);
}

// this function is identical to the one from the sketch in the Solution
// it lights the given LED, the first LED is 0
void lightLed(int led)
{
  // the following four lines convert LED number to pin numbers
  int indexA = pairs[led/2][0];
  int indexB = pairs[led/2][1];
  int pinA = pins[indexA];
  int pinB = pins[indexB];

  // turn off all pins not connected to the given LED
  for(int i=0; i < NUMBER_OF_PINS; i++)
  {
    if(pins[i] != pinA && pins[i] != pinB)
    {  // if this pin is not one of our pins
        pinMode(pins[i], INPUT);   // set the mode to input
        digitalWrite(pins[i],LOW); // make sure pull-up is off
    }
  }
  // now turn on the pins for the given LED
  pinMode(pinA, OUTPUT);
  pinMode(pinB, OUTPUT);
  if( led % 2 == 0)
  {
    digitalWrite(pinA,LOW);
    digitalWrite(pinB,HIGH);
  }
  else
  {
    digitalWrite(pinB,LOW);
    digitalWrite(pinA,HIGH);
  }
}
```

This sketch uses the value of the bits in the variable `ledStates` to represent the state of the LEDs (0 if off, 1 if on). The `refresh` function checks each bit and lights the LEDs for each bit that is set to 1. The `refresh` function must be called quickly and repeatedly, or the LEDs will appear to blink.

> ### WARNING
>
> Adding delays into your code can interfere with the *persistence of vision* effect that creates the illusion that hides the flashing of the LEDs.

You can use an interrupt to service the `refresh` function in the background (without needing to explicitly call the function in `loop`). Timer interrupts are covered in Chapter 18, but here is a preview of one approach for using an interrupt to service your LED refreshes. This uses a third-party library called FrequencyTimer2, available from the Library Manager, to create the interrupt (for instructions on installing third-party libraries, see Recipe 16.2):

```
#include <FrequencyTimer2.h>  // include this library to handle the refresh

byte pins[] = {2,3,4};
const int NUMBER_OF_PINS = sizeof(pins)/ sizeof(pins[0]);
const int NUMBER_OF_LEDS = NUMBER_OF_PINS * (NUMBER_OF_PINS-1);

byte pairs[NUMBER_OF_LEDS/2][2] = { {2,1}, {1,0}, {2,0} };

int ledStates = 0; //holds states for up to 15 LEDs
int refreshedLed;  // the LED that gets refreshed

void setup()
{
  FrequencyTimer2::setPeriod(20000/NUMBER_OF_LEDS); // set the period
  // the next line tells FrequencyTimer2 the function to call (ledRefresh)
  FrequencyTimer2::setOnOverflow(ledRefresh);
  FrequencyTimer2::enable();
}

void loop()
{
```

```
const int analogInPin = 0; // Analog input pin connected to the variable
resistor

  // here is the code from the bargraph recipe
  int sensorValue = analogRead(analogInPin);          // read the analog in
value
  // map to the number of LEDs
  int ledLevel = map(sensorValue, 0, 1023, 0, NUMBER_OF_LEDS);
  for (int led = 0; led < NUMBER_OF_LEDS; led++)
  {
    if (led < ledLevel ) {
      setState(led, HIGH);      // turn on pins less than the level
    }
    else {
      setState(led, LOW);       // turn off pins higher than the level
    }
  }
   // the LED is no longer refreshed in loop, it's handled by FrequencyTimer2
}

  // the remaining code is the same as the previous example
```

The FrequencyTimer2 library has the period set to 1,666 microseconds (20 ms divided by 12, the number of LEDs). The `FrequencyTimer2setOnOverflow` method gets the function to call (`ledRefresh`) each time the timer "triggers." The FrequencyTimer2 library is compatible with a limited number of boards: Arduino Uno (and likely most ATmega328-based compatibles), the Arduino Mega, and several Teensy variants. This PJRC page has more details on the library.

## See Also

Chapter 18 provides more information on timer interrupts.

# 7.11 Driving a 7-Segment LED Display

## Problem

You want to display numerals using a 7-segment numeric display.

## Solution

The following sketch displays numerals from 0 to 9 on a single-digit, 7-segment display. Figure 7-13 shows the connections for a common anode display. Your pin assignments may be different so check the datasheet for your display. If yours is a common cathode, connect the common cathode connection to GND. The output is produced by turning on combinations of segments that represent the numerals:

```
/*
 * SevenSegment sketch
 * Shows numerals ranging from 0 through 9 on a single-digit display
 * This example counts seconds from 0 to 9
 */

// bits representing segments A through G (and decimal point) for numerals 0-9
const byte numeral[10] = {
 //ABCDEFG+dp
  B11111100,   // 0
  B01100000,   // 1
  B11011010,   // 2
  B11110010,   // 3
  B01100110,   // 4
  B10110110,   // 5
  B00111110,   // 6
  B11100000,   // 7
  B11111110,   // 8
  B11100110,   // 9
};

// pins for decimal point and each segment
//                        dp,G,F,E,D,C,B,A
const int segmentPins[8] = { 5,8,9,7,6,4,3,2};

void setup()
{
  for(int i=0; i < 8; i++)
  {
    pinMode(segmentPins[i], OUTPUT); // set segment and DP pins to output
  }
}

void loop()
{
  for(int i=0; i <= 10; i++)
```
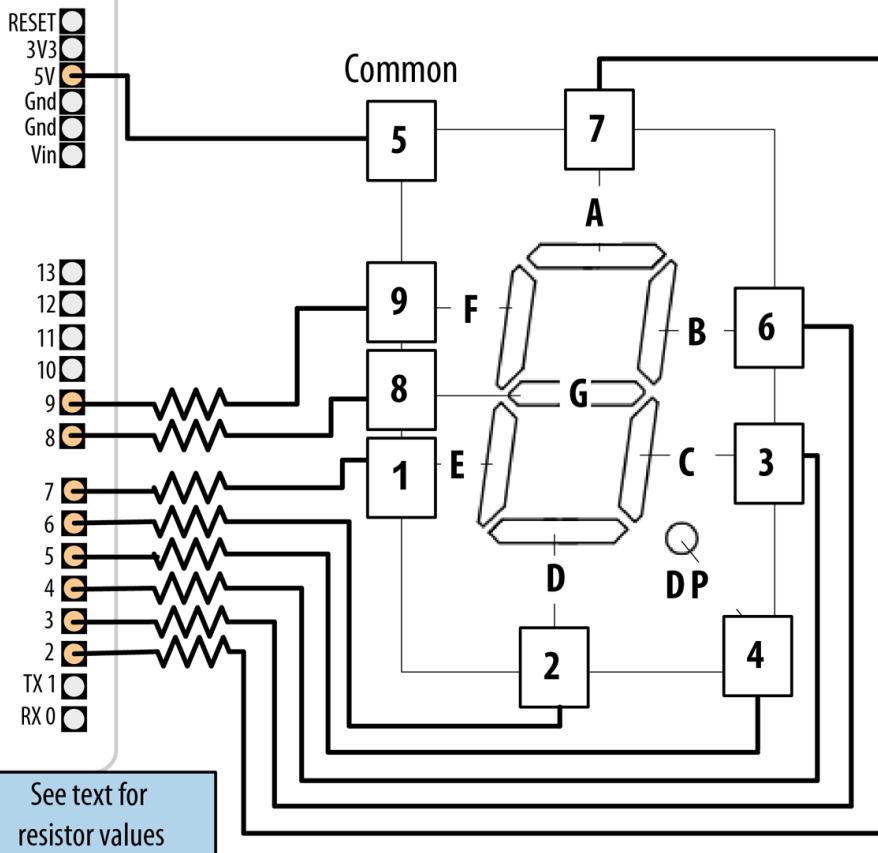
```
  {
    showDigit(i);
    delay(1000);
  }
  // the last value if i is 10 and this will turn the display off
  delay(2000);  // pause two seconds with the display off
}

// Displays a number from 0 through 9 on a 7-segment display
// any value not within the range of 0-9 turns the display off
void showDigit(int number)
{
  bool isBitSet;

  for(int segment = 1; segment < 8; segment++)
  {
    if( number < 0 || number > 9){
      isBitSet = 0;    // turn off all segments
    }
    else{
      // isBitSet will be true if given bit is 1
      isBitSet = bitRead(numeral[number], segment);
    }
    isBitSet = ! isBitSet; // remove this line if common cathode display
    digitalWrite(segmentPins[segment], isBitSet);
  }
}
```

*Connecting a 7-segment display*

## Discussion

The segments to be lit for each numeral are held in the array called `numeral`. There is one byte per numeral where each bit in the byte represents one of seven segments (or the decimal point).

The array called `segmentPins` holds the pins associated with each segment. The `showDigit` function checks that the number ranges from 0 to 9, and if valid, looks at each segment bit and turns on the segment if the bit is set (equal to 1). See Recipe 3.12 for more on the `bitRead` function.

As mentioned in Recipe 7.4, a pin is set `HIGH` when turning on a segment on a common cathode display, and it's set `LOW` when turning on a segment on a common anode display. The code here is for a common anode display, so it inverts the value (sets 0 to 1 and 1 to 0) as follows:

```
isBitSet = ! isBitSet; // remove this line if common cathode display
```

The `!` is the negation operator—see Recipe 2.20. If your display is a common cathode display (all the cathodes are connected together; see the datasheet if you are not sure), you can remove that line.

# 7.12 Driving Multidigit, 7-Segment LED Displays: Multiplexing

**Problem**

You want to display numbers using a 7-segment display that shows two or more digits.

**Solution**

Multidigit, 7-segment displays usually use multiplexing. In earlier recipes, multiplexed rows and columns of LEDs were connected together to form an array; here, corresponding segments from each digit are connected together (Figure 7-14 shows the connection for a Lite-On LTC-2623, but you will need to check the datasheet for your display if it's different):

> **NOTE**
>
> The wiring diagram shown is for a Lite-On LTC-2623 display. If you are using a different display, you can use the same Arduino pins, but you'll need to look up the corresponding pins on your display's datasheet. The LTC-2623 is a common anode display. If yours is a common cathode, you'll need to change two things: first, connect the transistors differently: connect all the emitters together and to ground, and each transistor's collector to the corresponding pin on the display. Second, comment out or delete this line from the sketch: `isBitSet = ! isBitSet;`.

```
/*
 * SevenSegmentMpx sketch
 * Shows numbers ranging from 0 through 9999 on a four-digit display
 * This example displays the value of a sensor connected to an analog input
 */

// bits representing segments A through G (and decimal point) for numerals 0-9
const int numeral[10] = {
  //ABCDEFG /dp
  B11111100,  // 0
  B01100000,  // 1
  B11011010,  // 2
  B11110010,  // 3
  B01100110,  // 4
  B10110110,  // 5
  B00111110,  // 6
  B11100000,  // 7
  B11111110,  // 8
  B11100110,  // 9
};

// pins for decimal point and each segment
                      // dp,G,F,E,D,C,B,A
const int segmentPins[] = { 4, 7,8,6,5,3,2,9};

const int nbrDigits= 4;   // the number of digits in the LED display
                          //dig  1  2  3  4
const int digitPins[nbrDigits] = { 10,11,12,13};

void setup()
{
  for(int i=0; i < 8; i++)
    pinMode(segmentPins[i], OUTPUT); // set segment and DP pins to output

  for(int i=0; i < nbrDigits; i++)
    pinMode(digitPins[i], OUTPUT);
}

void loop()
{
  int value = analogRead(0);
  showNumber(value);
}

void showNumber(int number)
{
  if(number == 0)
    showDigit( 0, nbrDigits-1) ; // display 0 in the rightmost digit
```
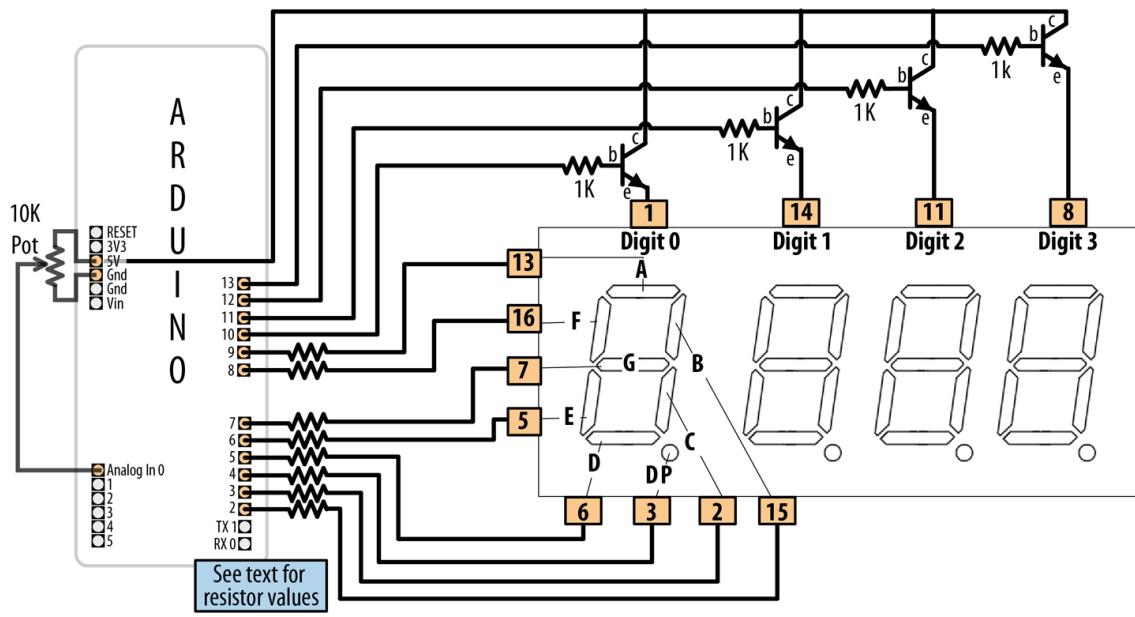
```cpp
        else
    {
      // display the value corresponding to each digit
      // leftmost digit is 0, rightmost is one less than the number of places
      for( int digit = nbrDigits-1; digit >= 0; digit--)
      {
        if(number > 0)
        {
          showDigit( number % 10, digit)  ;
          number = number / 10;
        }
      }
    }
}

// Displays given number on a 7-segment display at the given digit position
void showDigit( int number, int digit)
{
  digitalWrite( digitPins[digit], HIGH );
  for(int segment = 1; segment < 8; segment++)
  {
    bool isBitSet = bitRead(numeral[number], segment);
    // isBitSet will be true if given bit is 1
    isBitSet = ! isBitSet; // remove this line if common cathode display
    digitalWrite( segmentPins[segment], isBitSet);
  }
  delay(5);
  digitalWrite( digitPins[digit], LOW );
}
```

*Connecting a multidigit, 7-segment display (LTC-2623)*

## Discussion

This sketch has a `showDigit` function similar to that discussed in Recipe 7.11. Here the function is given the numeral and the digit place. The logic to light the segments to correspond to the numeral is the same, but in addition, the code sets the pin corresponding to the digit place `HIGH`, so only that digit will be written (see the earlier multiplexing explanations).

# 7.13 Driving Multidigit, 7-Segment LED Displays with the Fewest Pins

## Problem

You want to control multiple 7-segment displays, but you want to minimize the number of required Arduino pins.

## Solution

This Solution uses an HT16K33-based breakout board to control four-digit common cathode displays, such as the LuckyLight KW4-56NXBA-P or the Betlux BL-Q56C-43. The HT16K33 provides a simpler solution than Recipe 7.12, because it handles multiplexing and digit decoding in hardware. You can obtain HT16K33-based boards from a variety of sources. Adafruit makes the 7-Segment LED Matrix Backpack (part number 877) that is designed to work with the four-digit 7-segment displays that it stocks, but it also carries these boards with the displays included, and in a variety of colors.

This sketch will display a number between 0 and 9,999 (Figure 7-15 shows the connections):

```
/*
  HT16K33 seven segment display sketch
*/

#include <Wire.h>
#include <Adafruit_GFX.h>
#include "Adafruit_LEDBackpack.h"

Adafruit_7segment matrix = Adafruit_7segment();

const int numberOfDigits = 4; // change 4 to the # of digits wired up
const int maxCount       = 9999;

int number = 0;

void setup()
{
  Serial.begin(9600);
  matrix.begin(0x70);      // Initialize the display
  matrix.println(number); // Send the number (0 initially) to the display
  matrix.writeDisplay();   // Update the display
}

void loop()
{
  // display a number from serial port terminated by end of line character
  if (Serial.available())
  {
    char ch  = Serial.read();
    if ( ch == '\n')
```
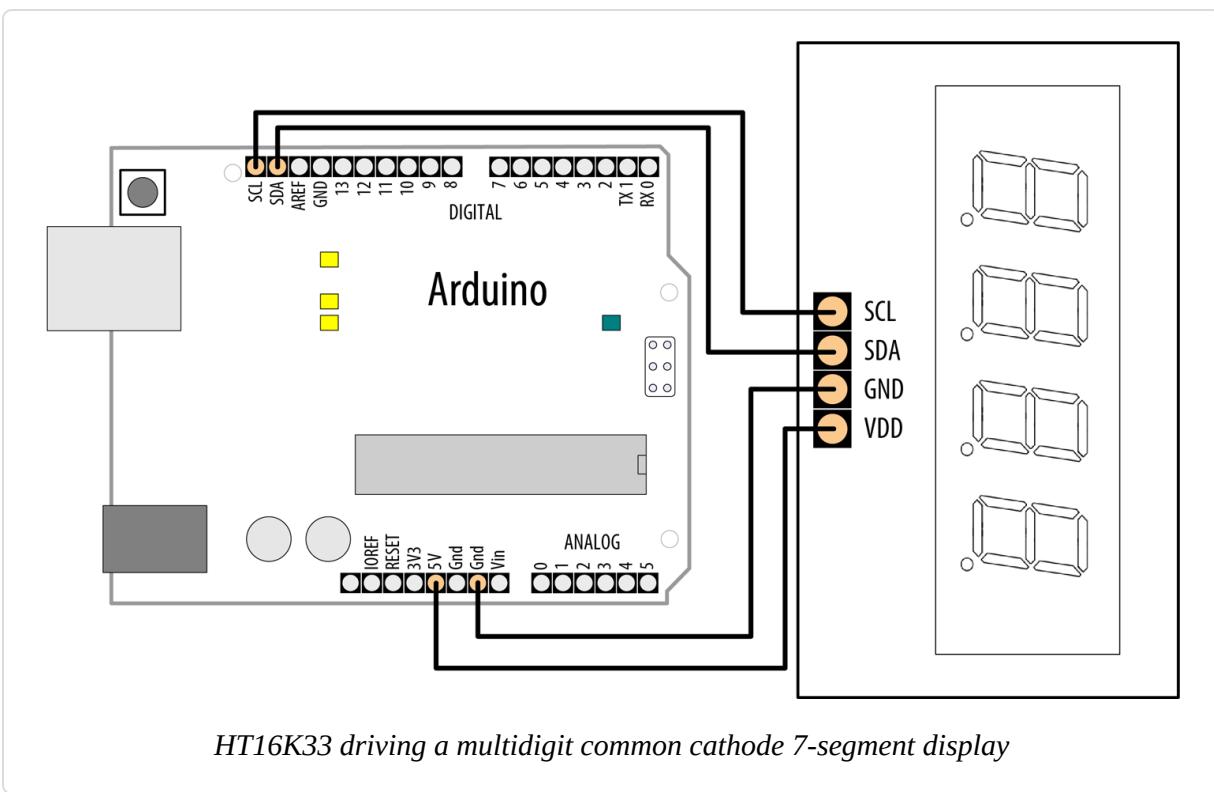
```
  {
    if (number <= maxCount)
    {
      matrix.println(number);
      matrix.writeDisplay();
      number = 0; // Reset the number to 0
    }
  }
  else
    number = (number * 10) + ch - '0'; // see Chapter 4 for details
  }
}
```



*HT16K33 driving a multidigit common cathode 7-segment display*

## Solution

This recipe uses Arduino I2C communication to talk to the HT16K33 chip. The Adafruit_LEDBackpack library provides an interface to the hardware, via an instance of the `Adafruit_7segment` object (in this sketch, it's called `matrix`). Chapter 13 covers I2C in more detail.

This sketch displays a number if up to four digits are received on the serial port—see Chapter 4 for an explanation of the serial code in `loop`. The

`matrix.println` function sends values to the HT16K33, and the `matrix.writeDisplay` function updates the display with the latest value sent to it.

The breakout board uses a four-digit, 7-segment display, but if you buy a more generic HT16K33 breakout board (such as Adafruit part number 1427), you can use it with single- or dual-digit displays for up to eight digits. When combining multiple displays, each corresponding segment pin should be connected together. (Recipe 13.1 shows the connections for a common dual-digit display.) You will need to consult the datasheet for your segment display(s), as well as for whichever HT16K33 breakout you use.
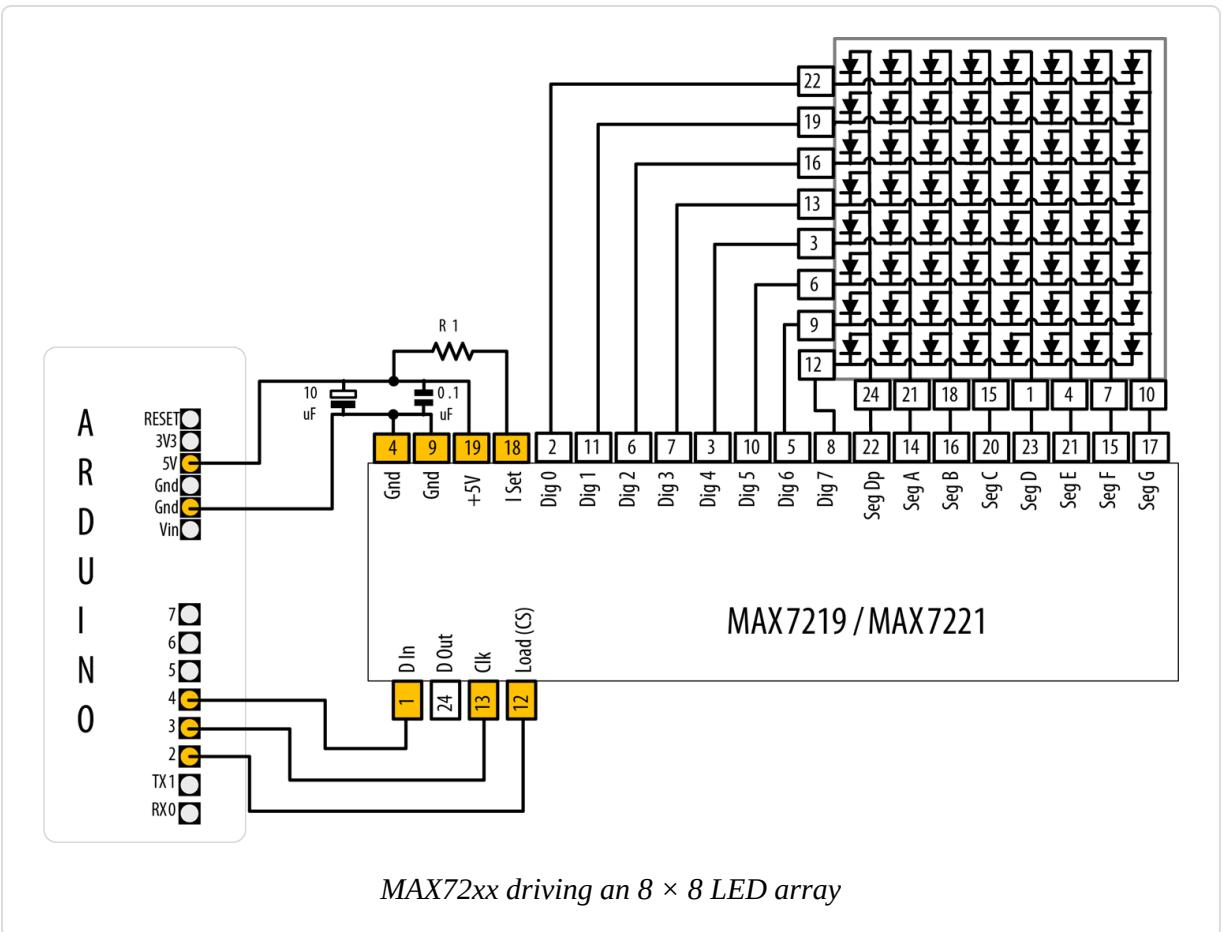
# 7.14 Controlling an Array of LEDs by Using MAX72xx Shift Registers

## Problem

You have an 8 × 8 array of LEDs to control, and you want to minimize the number of required Arduino pins.

## Solution

As in Recipe 7.13, you can use a shift register to reduce the number of pins needed to control an LED matrix. This Solution uses the popular MAX7219 or MAX7221 LED driver chip to provide this capability. Connect your Arduino, matrix, and MAX72xx as shown in Figure 7-16.

*MAX72xx driving an 8 × 8 LED array*

This sketch is based on the powerful MD_MAX72XX library, which can display text, draw objects on the display, and perform various transformations on the display. You can find the library in the Arduino Library Manager (see Recipe 16.2):

```
/*
    7219 Matrix demo
*/

#include <MD_MAX72xx.h>

// Pins to control 7219
#define LOAD_PIN 2
#define CLK_PIN  3
#define DATA_PIN 4

// Configure the hardware
#define MAX_DEVICES  1
#define HARDWARE_TYPE MD_MAX72XX::PAROLA_HW
```

```cpp
MD_MAX72XX mx = MD_MAX72XX(HARDWARE_TYPE, DATA_PIN, CLK_PIN,
  LOAD_PIN, MAX_DEVICES);

void setup()
{
  mx.begin();
}

void loop()
{
  mx.clear(); // Clear the display

  // Draw rows and columns
  for (int r = 0; r < 8; r++)
  {
    for (int c = 0; c < 8; c++) {
      mx.setPoint(r, c, true); // Light each LED
      delay(50);
    }

    // Cycle through available brightness levels
    for (int k = 0; k <= MAX_INTENSITY; k++) {
      mx.control(MD_MAX72XX::INTENSITY, k);
      delay(100);
    }
  }
}
```

## Discussion

A matrix is created by passing the hardware type, pin numbers for the data, load, and clock pins, and also the maximum number of devices (in case you are chaining modules). `loop` clears the display, then uses the `setPoint` method to turn pixels on. After the sketch draws a row, it cycles through the available brightness intensities and moves on to the next row.

The pin numbers shown here are for the green LEDs in the dual-color 8 × 8 matrix, available from Adafruit (part number 458). If you are using a different LED matrix, consult the datasheet to determine which pins correspond to each row and column. This sketch will work with a single-color matrix as well, since it only uses one of the two colors. If you find that your matrix is displaying text backward or not in the orientation you expect, you can try changing the hardware type in the line `#define`

HARDWARE_TYPE MD_MAX72XX::PAROLA_HW from PAROLA_HW to one of GENERIC_HW, ICSTATION_HW, or FC16_HW. There is a test sketch included in the MD_MAX72xx library's examples, MD_MAX72xx_HW_Mapper, which will run a test and help you decide the right hardware type to use.

The resistor (marked R1 in Figure 7-16) is used to control the maximum current that will be used to drive an LED. The MAX72xx datasheet has a table that shows a range of values (see Table 7-3).

*Table of resistor values (from MAX72xx datasheet)*

| LED forward voltage | | | | | |
|---|---|---|---|---|---|
| Current | 1.5V | 2.0V | 2.5V | 3.0V | 3.5V |
| 40 mA | 12 kΩ | 12 kΩ | 11 kΩ | 10 kΩ | 10 kΩ |
| 30 mA | 18 kΩ | 17 kΩ | 16 kΩ | 15 kΩ | 14 kΩ |
| 20 mA | 30 kΩ | 28 kΩ | 26 kΩ | 24 kΩ | 22 kΩ |
| 10 mA | 68 kΩ | 64 kΩ | 60 kΩ | 56 kΩ | 51 kΩ |

The green LED in the LED matrix shown in Figure 7-16 has a forward voltage of 2 volts and a forward current of 20 mA. Table 7-3 indicates 28K ohms, but to add a little safety margin, a resistor of 30K or 33K would be a suitable choice. The capacitors (0.1 uf and 10 uf) are required to prevent noise spikes from being generated when the LEDs are switched on and off —see "Using Capacitors for Decoupling" in Appendix C if you are not familiar with connecting decoupling capacitors.

## See Also

The MAX72xx datasheet

# 7.15 Increasing the Number of Analog Outputs Using PWM Extender Chips

## Problem

You want to have individual control of the intensity of more LEDs than Arduino can support.

## Solution

The PCA9685 chip drives up to 16 LEDs using only the two I2C data pins. Adafruit makes a breakout board that can drive multiple servos or LEDs (Adafruit part number 815). Figure 7-17 shows the connections. This sketch is based on Adafruit's Adafruit_PWMServoDriver library, which you can install using the Arduino Library Manager (see Recipe 16.2):

```
/*
   PCA9685 sketch
   Create a Knight Rider-like effect on LEDs plugged into all the PCA9685
outputs
   this version assumes one PCA9685 with 16 LEDs
*/

#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>

Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();

void setup()
{
  pwm.begin(); // Initialize the PWM board
}

int channel = 0;
int channel_direction = 1;
int intensity = 4095;        // Maximum brightness
int dim = intensity / 4;     // Intensity for a dim LED

void loop()
{
  channel += channel_direction; // increment (or decrement) the channel number

  // Turn off all the pins
  for (int pin = 0; pin < 16; pin++) {
    pwm.setPin(pin, 0);
  }
```
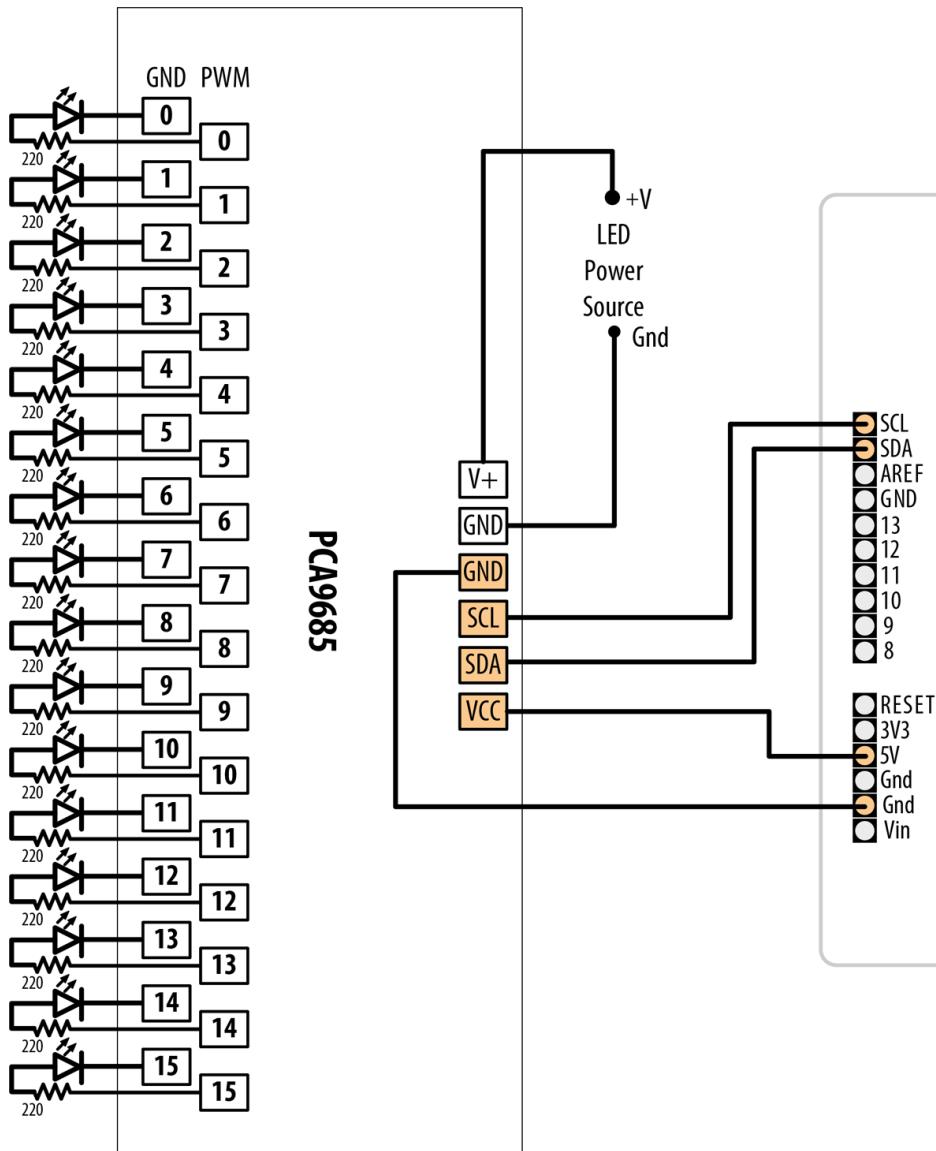
```
  // If we've hit channel 0, set direction to 1
  if (channel == 0) {
    channel_direction = 1;
  }
  else { // If we're at channel 1 or higher, set its previous neighbor to dim
    pwm.setPin(channel - 1, dim);
  }

  // Set this channel to maximum brightness
  pwm.setPin(channel, intensity);

  if (channel < 16) { // If we're below channel 16, set the next channel to
dim
    pwm.setPin(channel + 1, dim);
  }
  else { // If we've hit channel 16, set direction to -1
    channel_direction = -1;
  }

  delay(75);
}
```

*Sixteen LEDs driven using external PWM*

## Discussion

This sketch loops through each channel (LED), setting the previous LED to dim, the current channel to full intensity, and the next channel to dim. The LEDs are controlled through a few core methods. The sketch assumes that the PCA9685 is configured with the default I2C address of 0x40.

The `Adafruit_PWMServoDriver.begin` method initializes the driver prior to any other function. The `pwm.setPin` method sets the duty cycle of a given channel given as a number of ticks from 0 to 4,095. The first argument is the channel number followed by the brightness. Each pulse-width modulation cycle is divided into 4,096 ticks. The value you supply for the brightness indicates the number of ticks in which the LED should be on. You can change the PWM frequency with the `pwm.setPWMFreq` method (supply the value in hertz, from 40 to 1,600).

## See Also

More functions are available in the library.

You can chain multiple driver boards by chaining their pins together. Each board must have a unique address, which you set by soldering the pads labeled A0 through A5. You can chain up to 62 driver boards.

# 7.16 Using an Analog Panel Meter as a Display

## Problem

You would like to control the pointer of an analog panel meter from your sketch. Fluctuating readings are easier to interpret on an analog meter, and analog meters add a cool retro look to a project.

## Solution

Connect the meter through a series resistor (5K ohms for the typical 1 mA meter) and connect to an analog (PWM) output (see Figure 7-18).

The pointer movement corresponds to the position of a pot (variable resistor):

```
/*
 * AnalogMeter sketch
 * Drives an analog meter through an Arduino PWM pin
```
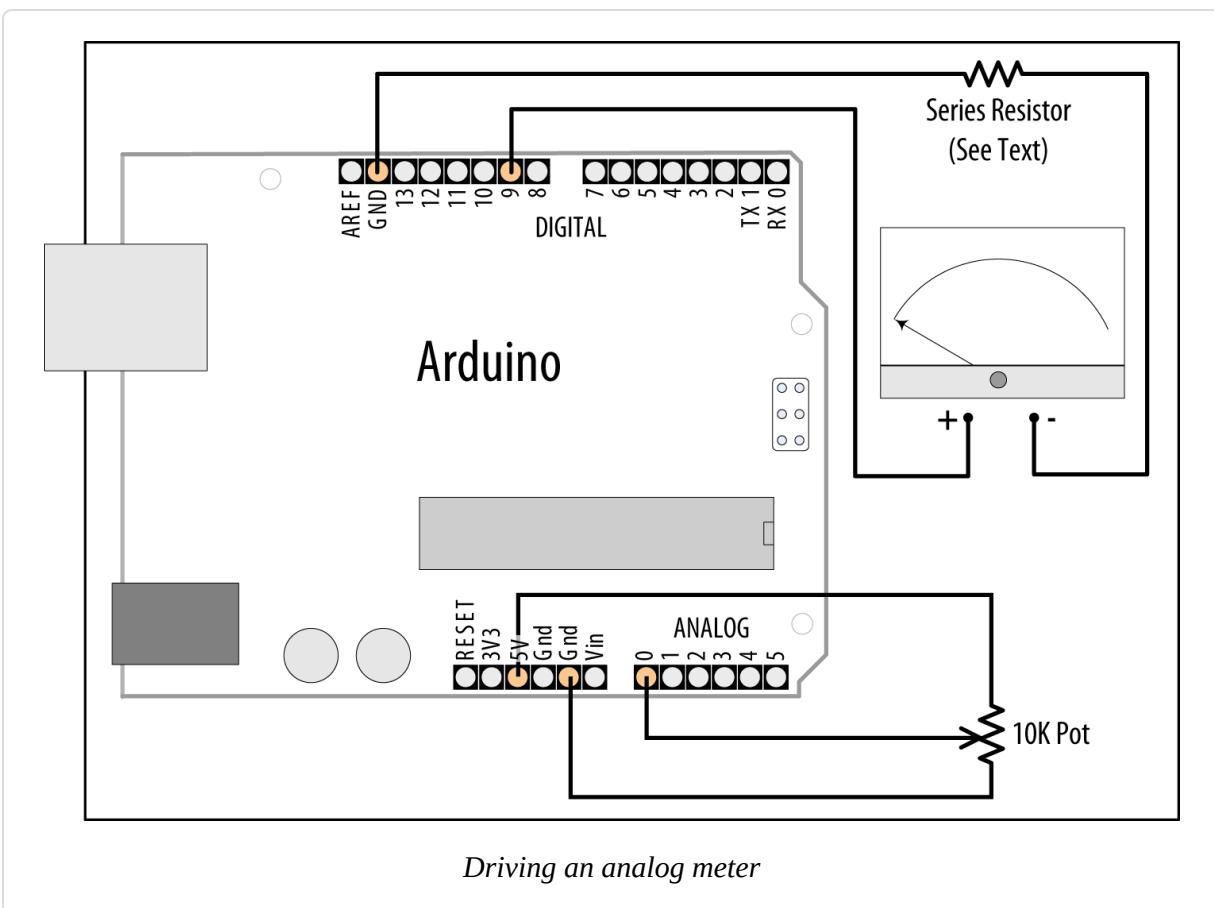
```
 * The meter level is controlled by a variable resistor on an analog input pin
 */

const int analogInPin = A0;   // Analog input connected to variable resistor
const int analogMeterPin = 9; // Analog output pin connected to the meter

int sensorValue = 0;        // value read from the pot
int outputValue = 0;        // value output to the PWM (analog out)

void setup()
{
  // nothing in setup
}

void loop()
{
  sensorValue = analogRead(analogInPin);            // read the analog in
value
  outputValue = map(sensorValue, 0, 1023, 0, 255);  // scale for analog out
  analogWrite(analogMeterPin, outputValue);         // write the analog out
value
}
```

*Driving an analog meter*

## Discussion

In this variation on Recipe 7.2, the Arduino `analogWrite` output drives a panel meter. Panel meters are usually much more sensitive than LEDs; a resistor must be connected between the Arduino output and the meter to drop the current to the level for the meter.

The value of the series resistor depends on the sensitivity of the meter; 5K ohms give full-scale deflection with a 1 mA meter. You can use 4.7K resistors, as they are easier to obtain than 5K, although you will probably need to reduce the maximum value given to `analogWrite` to 240 or so. Here is how you can change the range in the `map` function if you use a 4.7K ohm resistor with a 1 mA meter:

```
outputValue = map(sensorValue, 0, 1023, 0, 240); // map to meter's range
```

If your meter has a different sensitivity than 1 mA, you will need to use a different value series resistor. The resistor value in ohms is:

- resistor = 5,000 / mA

So, a 500 milliamp meter (0.5 mA) is 5,000 / 0.5, which is 10,000 (10K) ohms. A 10 mA meter requires 500 ohms; 20 mA requires 250 ohms.

Some surplus meters already have an internal series resistor—you may need to experiment to determine the correct value of the resistor, but be careful not to apply too much voltage to your meter.

## See Also

Recipe 7.2

# Physical Output

## 8.0 Introduction

You can make things move by controlling motors with Arduino. Different types of motors suit different applications, and this chapter shows how Arduino can drive many different kinds of motors. You'll see how to work with servos, which are motors that have circuits within them to allow moving to a specific motor position or to spin at a specific speed. You'll also learn about brushed and brushless motors, which use different designs to drive a motor that spins at varying speeds and directions. There are recipes in this chapter for stepper motors, which allow you to move a motor a specific number of steps in one direction or the other. In addition to motors that generate rotary motion, there are recipes for working with relays and solenoids.
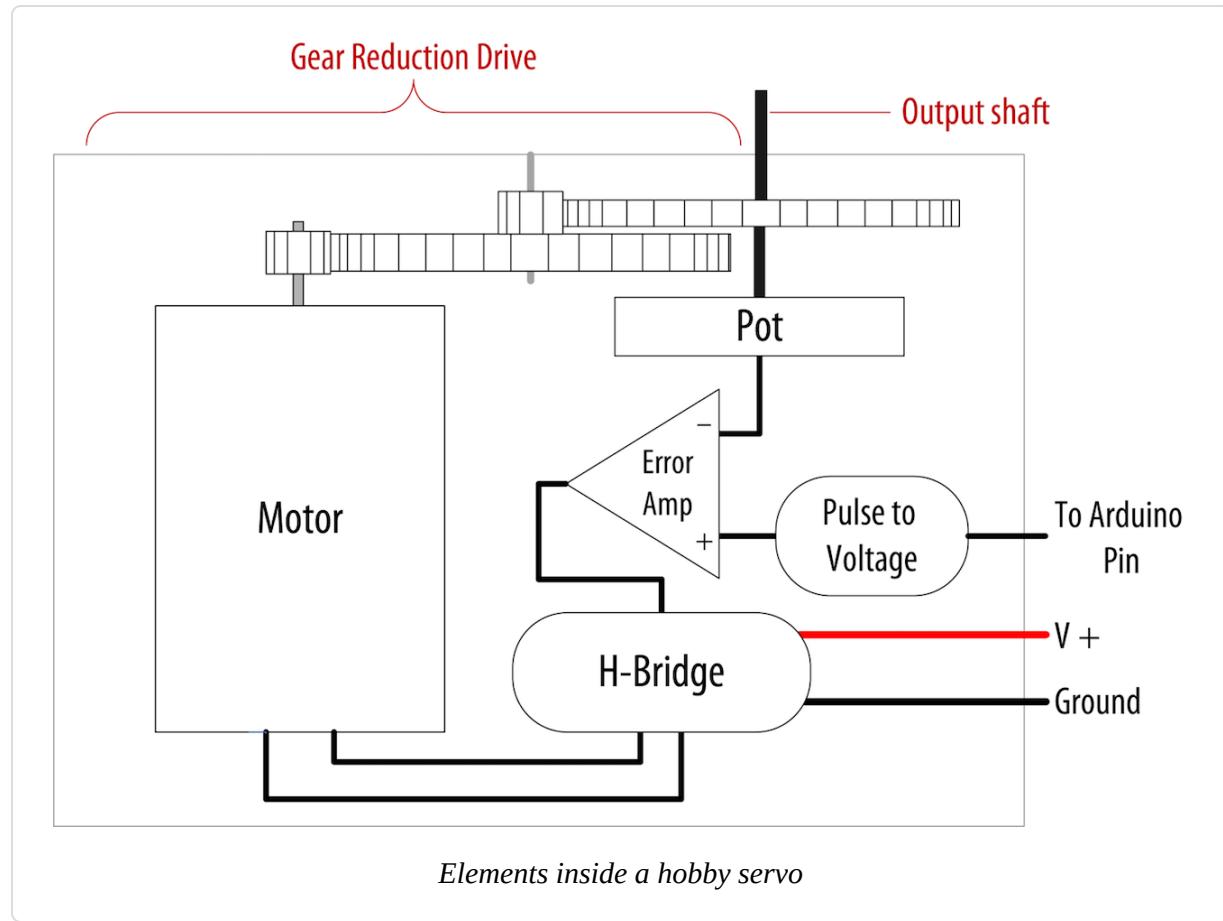
## Servos

Servos enable you to accurately control physical movement because they generally move to a position instead of continuously rotating. They are ideal for making something rotate over a range of 0 to 180 degrees. Servos are easy to connect and control because the motor driver is built into the servo.

Servos contain a small motor connected through gears to an output shaft. The output shaft drives a servo arm and is also connected to a potentiometer to provide position feedback to an internal control circuit (see Figure 8-1).

You can get continuous rotation servos that have the positional feedback disconnected so that you can instruct the servo to rotate continuously clockwise and counterclockwise with some control over the speed. These function a little like the brushed motors covered in Recipe 8.9, except that continuous rotation servos use the servo library code instead of `analogWrite` and don't require a motor shield.
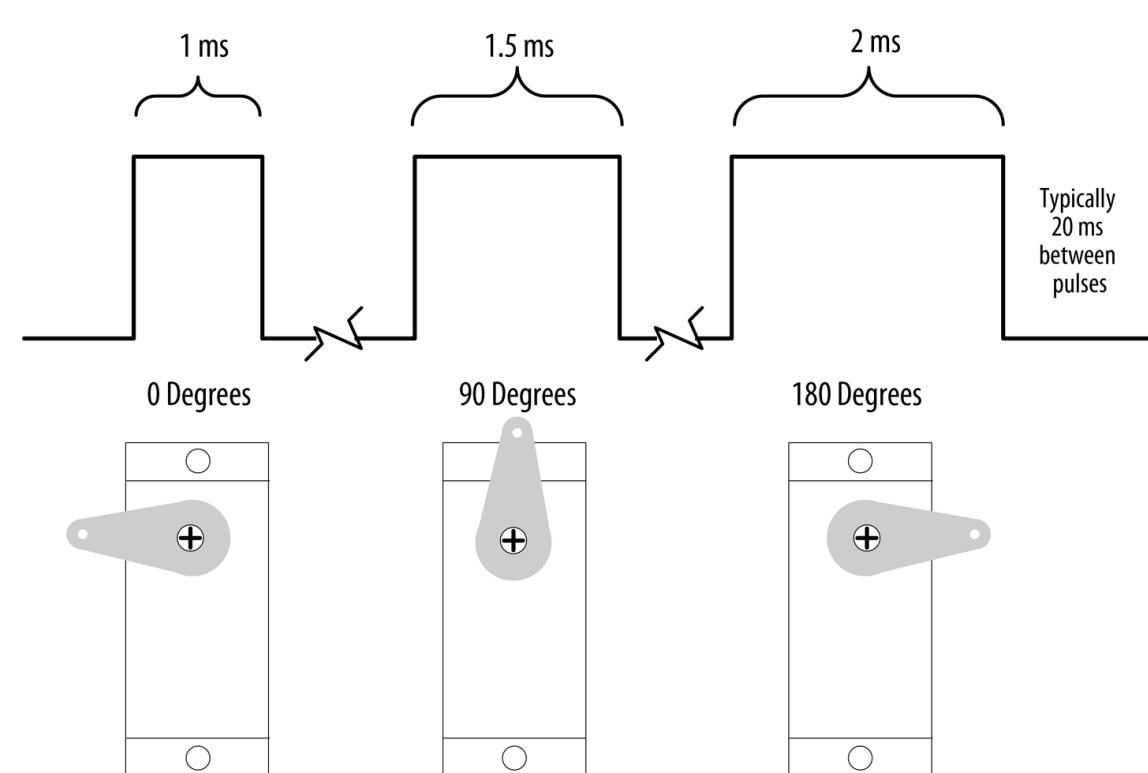
Continuous rotation servos are easy to use because they don't need a motor shield—the motor drivers are inside the servo. The disadvantages are that the speed and power choices are limited compared to external motors, and the precision of speed control is usually not as good as with a motor shield (the electronics are designed for accurate positioning, not linear speed control). See Recipe 8.3 for more on using continuous rotation servos.



*Elements inside a hobby servo*

Servos respond to changes in the duration of a pulse. A short pulse of 1 ms or less will cause the servo to rotate to one extreme; a pulse duration of 2 ms or so will rotate the servo to the other extreme (see Figure 8-2). Pulses ranging between these values will rotate the servo to a position proportional to the pulse width. There is no standard for the exact relationship between pulses and position, and you may need to tinker with the commands in your sketch to adjust for the range of your servos.

*Relationship between the pulse width and the servo angle; the servo output arm moves proportionally as the pulse width increases from 1 ms to 2 ms*

## Solenoids and Relays

Although most motors produce rotary motion, a solenoid produces linear movement when powered. A solenoid has a metallic core that is moved by a magnetic field created when current is passed through a coil. A mechanical relay is a type of solenoid that connects or disconnects electrical contacts (it's a solenoid operating a switch). Relays are controlled just like solenoids. Relays and solenoids, like most motors, require more current than an

Arduino pin can safely provide, and the recipes in this chapter show how you can use a transistor or external circuit to drive these devices.

## Brushed and Brushless Motors

Most low-cost direct current (DC) motors are simple devices with two leads connected to brushes (contacts) that control the magnetic field of the coils that drives a metallic core (armature). The direction of rotation can be reversed by reversing the polarity of the voltage on the contacts. DC motors are available in many different sizes, but even the smallest (such as vibration motors used in cell phones) require a transistor or other external control to provide adequate current. The recipes that follow show how to control motors using a transistor or an external control circuit called an H-Bridge.

The primary characteristic in selecting a motor is *torque*. Torque determines how much work the motor can do. Typically, higher-torque motors are larger and heavier and draw more current than lower-torque motors.

Brushless motors usually are more powerful and efficient for a given size than brushed motors, but they require more complicated electronic control. Where the performance benefit of a brushless motor is desired, components called *electronic speed controllers* intended for hobby radio control use can be easily controlled by Arduino because they are controlled much like a servo motor.

## Stepper Motors

Steppers are motors that rotate a specific number of degrees in response to control pulses. The number of degrees in each step is motor-dependent, ranging from one or two degrees per step to 30 degrees or more.

Two types of steppers are commonly used with Arduino: bipolar (typically with four leads attached to two coils) and unipolar (five or six leads attached to two coils). The additional wires in a unipolar stepper are internally connected to the center of the coils (in the five-lead version, each coil has a center tap and both center taps are connected together). The

recipes covering bipolar and unipolar steppers have diagrams illustrating these connections.

> **NOTE**
>
> The most common cause of problems when connecting devices that require external power is neglecting to connect all the grounds together. Your Arduino ground must be connected to the external power supply ground and the grounds of external devices being powered.

# 8.1 Controlling Rotational Position with a Servo

## Problem

You want to control rotation using an angle calculated in your sketch. For example, you want a sensor on a robot to swing through an arc or move to a position you select.

## Solution

Use the Servo library distributed with Arduino. Connect the servo power and ground to a suitable power supply (a single hobby servo can usually be powered from the Arduino 5V line). You can connect the servo signal leads to any Arduino digital pin.

Here is the example Sweep sketch distributed with Arduino; Figure 8-3 shows the connections:

```
/*
 * Servo rotation sketch
 */
#include <Servo.h>

Servo myservo;  // create servo object to control a servo

int angle = 0;  // variable to store the servo position

void setup()
```
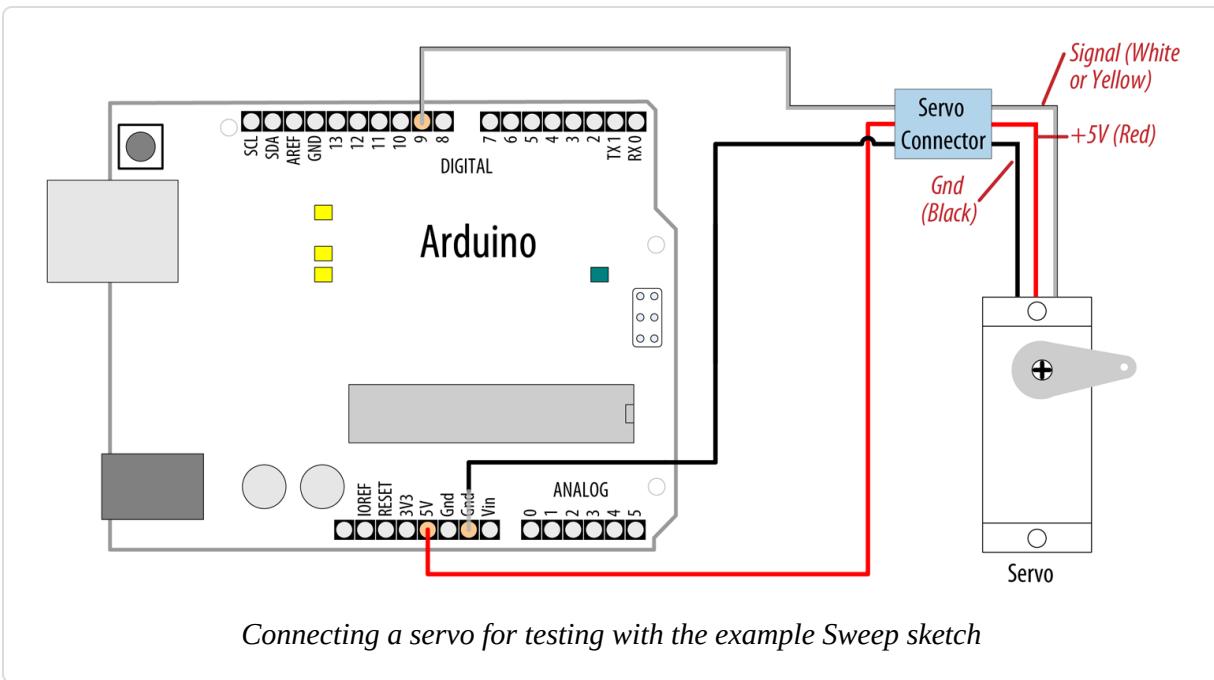
```
{
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop()
{
  for(angle = 0; angle < 180; angle += 1)  // goes from 0 degrees to 180
degrees
  {                                         // in steps of 1 degree
    myservo.write(angle);     // tell servo to go to position in variable
'angle'
    delay(20);                // waits 20 ms between servo commands
  }
  for(angle = 180; angle >= 1; angle -= 1) // goes from 180 degrees to 0
degrees
  {
    myservo.write(angle);    // move servo in opposite direction
    delay(20);               // waits 20 ms between servo commands
  }
}
```



*Connecting a servo for testing with the example Sweep sketch*

## Discussion

This example sweeps the servo between 0 and 180 degrees. You may need to tell the library to adjust the minimum and maximum positions so that you

get the range of movement you want. Calling `Servo.attach` with optional arguments for minimum and maximum positions will adjust the movement:

```
myservo.attach(9,1000,2000);  // use pin 9, set min to 1000 us, max to 2000 us
```

Because typical servos respond to pulses measured in microseconds and not degrees, the arguments following the pin number inform the Servo library how many microseconds to use when 0 degrees or 180 degrees are requested. Not all servos will move over a full 180-degree range, so you may need to experiment with yours to get the range you want.

The parameters for `servo.attach(`*pin, min, max*`)` are the following:

*pin*

> The pin number that the servo is attached to (you can use any digital pin)

*min* (optional)

> The pulse width, in microseconds, corresponding to the minimum (0-degree) angle on the servo (defaults to 544)

*max* (optional)

> The pulse width, in microseconds, corresponding to the maximum (180-degree) angle on the servo (defaults to 2,400)

---

**NOTE**

The Servo library can handle up to 12 servos on most Arduino boards, but 48 on the Arduino Mega. On the Uno and other boards based on the ATmega328, you will give up `analogWrite()` (PWM) on pins 9 and 10, even if you're not connecting a servo to those pins. The Arduino Mega is an exception, and you will likely find that some 32-bit boards do not have this limitation either. See the Servo library reference for more information.

---

Power requirements vary depending on the servo and how much torque is needed to rotate the shaft.

# 8.2 Controlling Servo Rotation with a Potentiometer or Sensor

## Problem

You want to control rotational direction and position with a potentiometer. For example, you want to control the pan and tilt of a camera or sensor. This recipe can work with any variable voltage from a sensor that can be read from an analog input.

## Solution

You can use the same library as in Recipe 8.1, with the addition of code to read the voltage on a potentiometer. This value is scaled so that the position of the pot (from 0 to 1023) is mapped to a range of 0 to 180 degrees. The only difference in the wiring is the addition of the potentiometer; see Figure 8-4:

```
/*
 * Servo With Sensor sketch
 * Control a servo with a sensor.
 */

#include <Servo.h>

Servo myservo;  // create servo object to control a servo

int potpin = A0; // analog pin used to connect the potentiometer
int val;        // variable to read the value from the analog pin

void setup()
{
  myservo.attach(9); // attaches the servo on pin 9 to the servo object
```
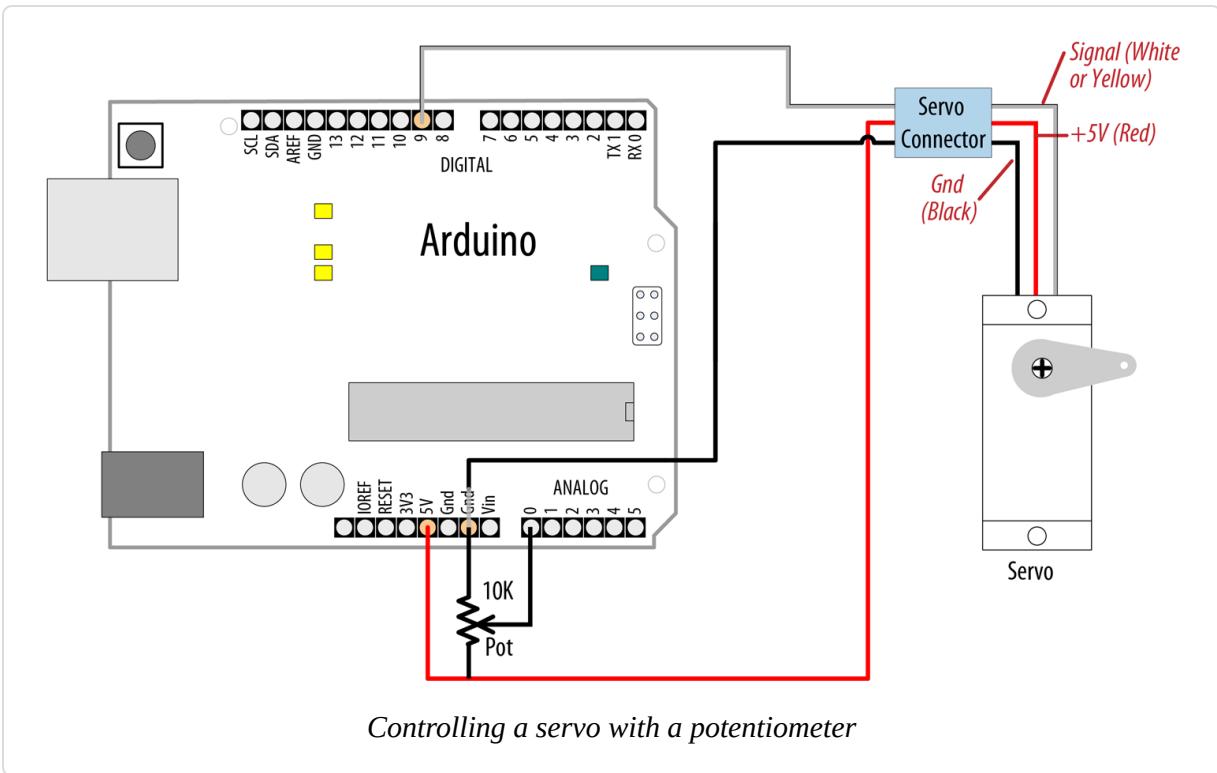
```
  }

  void loop()
  {
    val = analogRead(potpin);         // reads the value of the potentiometer
    val = map(val, 0, 1023, 0, 180);  // scale it to use it with the servo
    myservo.write(val);               // sets position to the scaled value
    delay(15);                        // waits for the servo to get there
  }
```



*Controlling a servo with a potentiometer*

## NOTE

Hobby servos have a cable with a three-pin female connector that can be directly plugged into a "servo" header fitted to some shields, such as the Adafruit Motor Shield. The physical connector is compatible with the Arduino connectors so you can use the same wire jumpers as those used to connect Arduino pins. Bear in mind that the color of the signal lead is not standardized; yellow is sometimes used instead of white. Red is always in the middle and the ground lead is usually black or brown.

# Discussion

Anything that can be read from `analogRead` (see Chapters 5 and 6) can be used—for example, the gyro and accelerometer recipes in Chapter 6 can be used, so that the angle of the servo is controlled by the yaw of the gyro or angle of the accelerometer.

> **TIP**
>
> Not all servos will rotate over the full range of the Servo library. If your servo buzzes due to hitting an end stop at an extreme of movement, try reducing the output range in the `map` function until the buzzing stops. For example:
>
> ```
> val=map(val,0,1023,10,170);
> ```
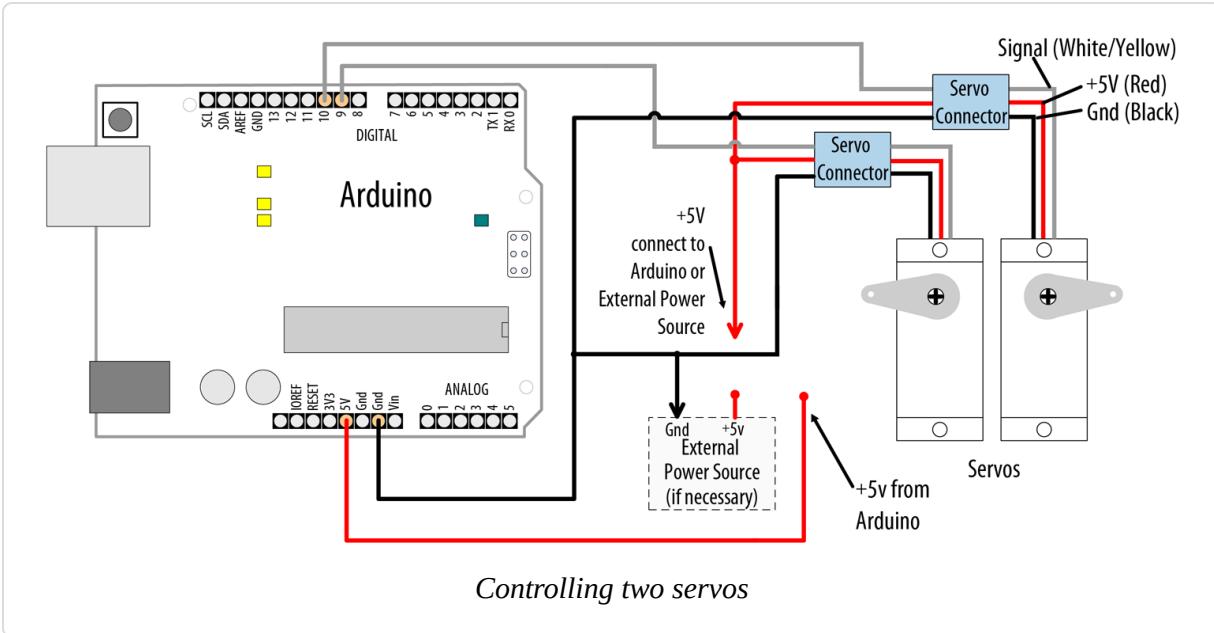
# 8.3 Controlling the Speed of Continuous Rotation Servos

## Problem

You want to control the rotational direction and speed of servos modified for continuous rotation. For example, you are using two continuous rotation servos to power a robot and you want the speed and direction to be controlled by your sketch.

## Solution

Continuous rotation servos are a form of gear-reduced motor with forward and backward speed adjustment. Control of continuous rotation servos is similar to normal servos. The servo rotates in one direction as the angle is increased from 90 degrees; it rotates in the other direction when the angle is decreased from 90 degrees. The actual direction forward or backward depends on how you have the servos attached. Figure 8-5 shows the connections for controlling two servos.

*Controlling two servos*

Servos are usually powered from a 4.8V to 6V source. Heavier-duty servos may require more current than the Arduino board can provide through the +5V pin and these will require an external power source. Four 1.2V rechargeable batteries can be used to power Arduino and the servos. If you are thinking of powering your Arduino from these batteries as well, bear in mind that you are in something of a gray area: you could, in theory, power the Arduino by connecting the batteries' positive lead to Arduino's 5V pin. However, this bypasses the voltage regulator and is not a great idea. The other option is to supply power to Arduino's VIN pin, which requires a minimum of 6 volts. But that's not great either, because anything less than 7 volts could make the Arduino unstable. You'll need to balance out these constraints with the power needs of your servo motors to strike the right balance.

The sketch sweeps the servos from 90 to 180 degrees, which translates to a variable speed with continuous rotation servos. So, if the servos were connected to wheels, the vehicle would move forward at a slowly increasing pace and then slow down to a stop. Because the servo control code is in `loop`, this will continue for as long as there is power:

```
/*
 * Continuous rotation
```

```
  */
#include <Servo.h>

Servo myservoLeft;   // create servo object to control a servo
Servo myservoRight;  // create servo object to control a servo

int angle = 0;     // variable to store the servo position

void setup()
{
  myservoLeft.attach(9);   // attaches left servo on pin 9 to servo object
  myservoRight.attach(10); // attaches right servo on pin 10 to servo object
}

void loop()
{
  for(angle = 90; angle < 180; angle += 1)  // goes from 90 to 180 degrees
  {                                          // in steps of 1 degree.
                                             // 90  degrees is stopped.

    myservoLeft.write(angle);        // rotate servo at speed given by 'angle'
    myservoRight.write(180-angle);   // go in the opposite direction

    delay(20);                       // waits 20 ms between servo commands
  }
  for(angle = 180; angle >= 90; angle -= 1) // goes from 180 to 90 degrees
  {
    myservoLeft.write(angle);        // rotate at a speed given by 'angle'
    myservoRight.write(180-angle);   // other servo goes in opposite direction
  }
}
```

## Discussion

You can use similar code for continuous rotation and normal servos, but be aware that continuous rotation servos may not stop rotating when writing exactly 90 degrees. Some servos have a small potentiometer you can trim to adjust for this, or you can add or subtract a few degrees to stop the servo. For example, if the left servo stops rotating at 92 degrees, you can change the lines that write to the servos as follows:

```
myservoLeft.write(angle+TRIM); // declare int TRIM=2; at beginning of sketch
```

# 8.4 Controlling Servos Using Computer Commands

## Problem

You want to provide commands to control servos from the serial port. Perhaps you want to control servos from a program running on your computer.

## Solution

You can use software to control the servos. This has the advantage that any number of servos can be supported. However, your sketch needs to constantly attend to refreshing the servo position, so the logic can get complicated as the number of servos increases if your project needs to perform a lot of other tasks.

This recipe drives four servos according to commands received on the serial port. The commands are of the following form:

- `180a` writes 180 to servo a

- `90b` writes 90 to servo b

- `0c` writes 0 to servo c

- `17d` writes 17 to servo d

Here is the sketch that drives four servos connected on pins 7 through 10:

```
/*
 * Computer Control of Servos sketch
 */
#include <Servo.h>  // the servo library

#define SERVOS 4 // the number of servos
int servoPins[SERVOS] = {7,8,9,10}; // servos on pins 7 through 10

Servo myservo[SERVOS];

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

```
    for(int i=0; i < SERVOS; i++)
    {
      myservo[i].attach(servoPins[i]);
    }
  }

  void loop()
  {
    serviceSerial();
  }

  // serviceSerial checks the serial port and updates position with received
  data
  // it expects servo data in the form:
  //
  //   "180a" writes 180 to servo a
  //   "90b writes 90 to servo b
  //
  void serviceSerial()
  {
    if ( Serial.available())
    {
      int pos = Serial.parseInt();
      char ch = Serial.read();
      if(ch >= 'a' && ch < 'a' + SERVOS) // If ch is a valid servo letter
      {
        Serial.print("Servo "); Serial.print(ch - 'a');
        Serial.print(" set to "); Serial.println(pos);
        myservo[ch - 'a'].write(pos);  // write position to corresponding servo
      }
    }
  }
```

## Discussion

Connecting the servos is similar to the previous recipes. Each servo line
wire gets connected to a digital pin. All servo grounds are connected to
Arduino ground. The servo power lines are connected together, and you
may need an external 5V or 6V power source if your servos require more
current than the Arduino power supply can provide.

An array named myservo (see Recipe 2.4) is used to hold references for the
four servos. A for loop in setup attaches each servo in the array to
consecutive pins defined in the servoPins array.

The sketch uses `parseInt` to collect integer values over the serial port. If the character is the letter *a*, the position is written to the first servo in the array (the servo connected to pin 7). The letters *b*, *c*, and *d* control the subsequent servos.

## See Also

See Chapter 4 for more on handling values received over serial.

# 8.5 Driving a Brushless Motor (Using a Hobby Speed Controller)
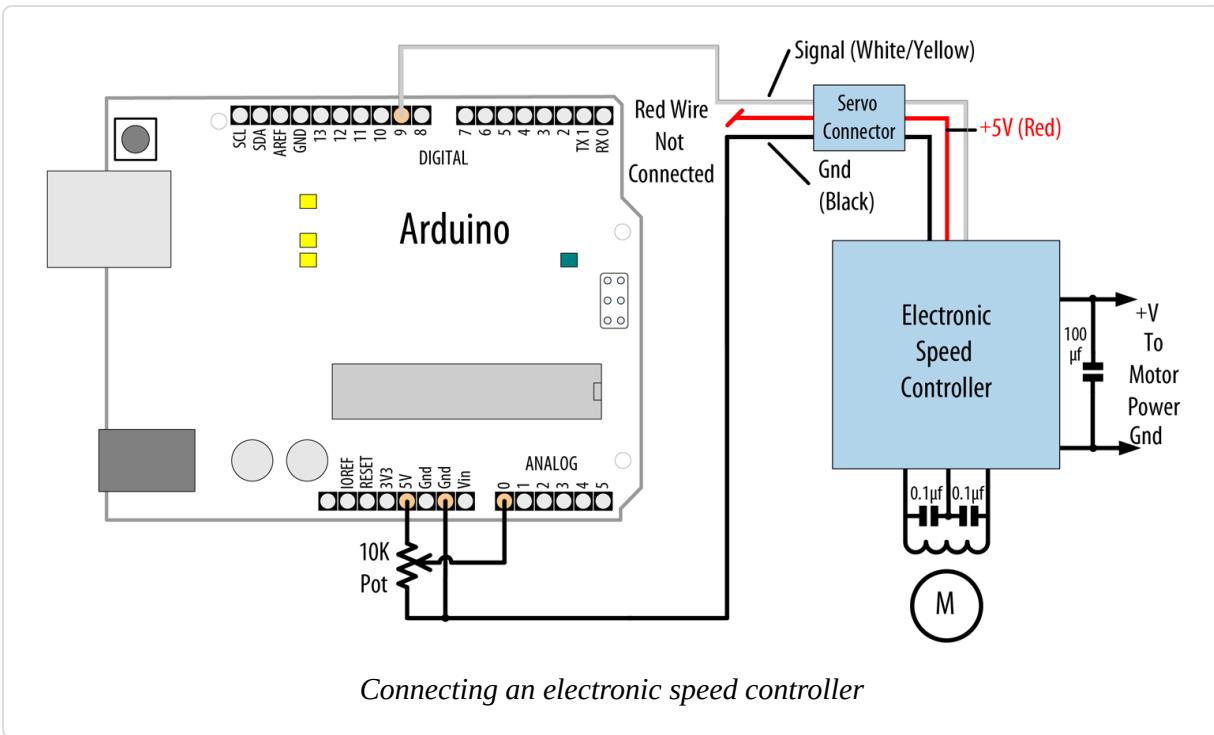
## Problem

You have a hobby brushless motor and you want to control its speed.

## Solution

This sketch uses the same code as Recipe 8.2. The wiring is similar, except for the speed controller and motor. A hobby electronic speed controller (ESC) is a device used to control brushless motors in radio-controlled vehicles. Because these items are mass produced, they are a cost-effective way to drive brushless motors. You can find a selection by typing "esc" into the search field of your favorite hobby store website or searching for "electronic speed controller" on Amazon, eBay, or AliExpress.

Brushless motors have three windings and these should be connected following the instructions for your speed controller (see Figure 8-6). Check the documentation for your ESC to see if there are any special considerations when working with Arduino. For example, the ESC from RC Electric Parts suggests initializing the Servo library with `Servo.attach(`*pin,* ` 1000, 2000)`.

*Connecting an electronic speed controller*

## Discussion

Consult the documentation for your speed controller to confirm that it is suitable for your brushless motor and to verify the wiring. Brushless motors have three connections for the three motor wires and two connections for power. Many speed controllers provide power on the center pin of the servo-style connector. Unless you want to power the Arduino board from the speed controller, you must disconnect or cut this center wire.

### WARNING

If your speed controller has a feature that provides 5V power to servos and other devices (called a *battery eliminator circuit* or *BEC* for short), do not connect this wire to your Arduino (see Figure 8-6).

# 8.6 Controlling Solenoids and Relays

## Problem

You want to activate a solenoid or relay under program control. Solenoids are electromagnets that convert electrical energy into mechanical movement. An electromagnetic relay is a switch that is activated by a solenoid.

## Solution

Most solenoids require more power than an Arduino pin can provide, so a transistor is used to switch the current needed to activate a solenoid. Activating the solenoid is achieved by using `digitalWrite` to set the pin `HIGH`.

This sketch turns on a transistor connected as shown in Figure 8-7. The solenoid will be activated for one second every hour:
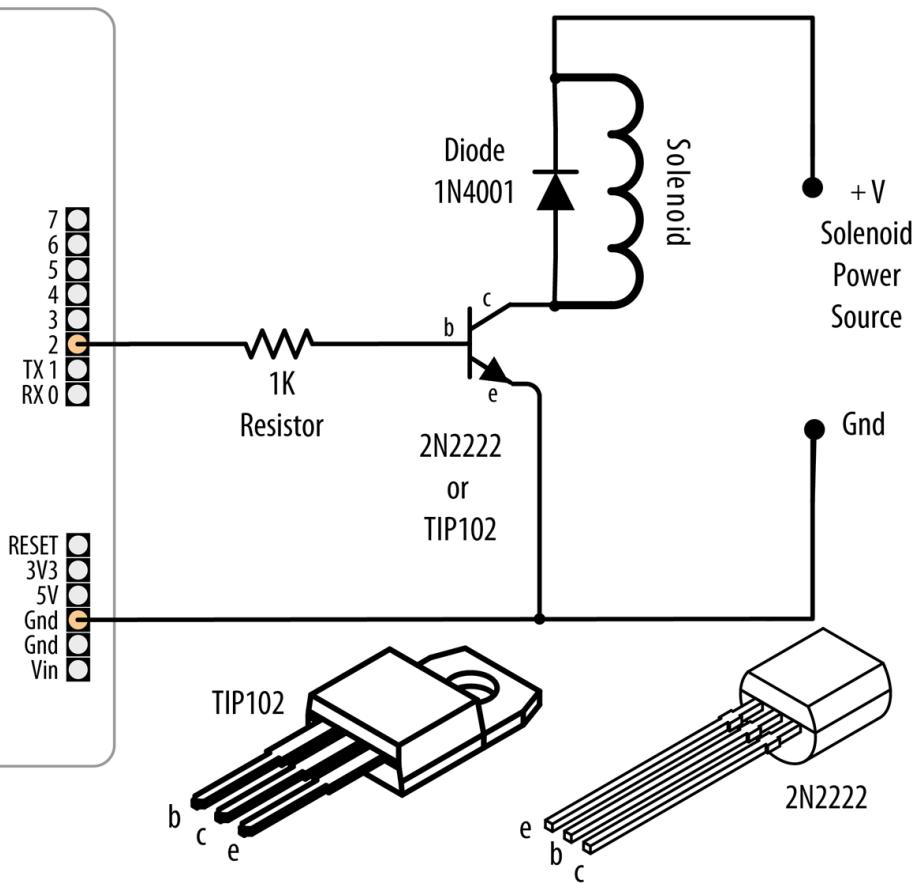
```
/*
 * Solenoid sketch
 */
int solenoidPin = 2;              // Solenoid connected to transistor on pin 2

void setup()
{
  pinMode(solenoidPin, OUTPUT);
}

void loop()
{
  long interval = 1000 * 60 * 60 ;   // interval = 60 minutes

  digitalWrite(solenoidPin, HIGH); // activates the solenoid
  delay(1000);                     // waits for a second
  digitalWrite(solenoidPin, LOW);  // deactivates the solenoid
  delay(interval);                 // waits one hour
}
```

*Driving a solenoid with a transistor*

## Discussion

The choice of transistor is dependent on the amount of current required to activate the solenoid or relay. The datasheet may specify this in milliamperes (mA) or as the resistance of the coil. To find the current needed by your solenoid or relay, divide the voltage of the coil by its resistance in ohms. For example, a 12V relay with a coil of 185 ohms draws 65 mA: 12 (volts) / 185 (ohms) = 0.065 amps, which is 65 mA. If your transistor is not capable of handling the current you drive through it, it will overheat and may burn out.

Small transistors such as the 2N2222 are sufficient for solenoids requiring up to a few hundred milliamps. Larger solenoids will require a higher-power transistor, like the TIP102/TIP120 or similar. There are many

suitable transistor alternatives; see Appendix B for help reading a datasheet and choosing transistors.

The purpose of the diode is to prevent reverse EMF from the coil from damaging the transistor (*reverse EMF* is a voltage produced when current through a coil is switched off). The polarity of the diode is important; there is a colored band indicating the cathode—this should be connected to the solenoid positive power supply.

Electromagnetic relays are activated just like solenoids. A special relay called a *solid state relay* (SSR) has internal electronics that can be driven directly from an Arduino pin without the need for the transistor. Check the datasheet for your relay to see what voltage and current it requires; anything more than 40 mA at 5 volts will require a circuit such as the one shown in Figure 8-7.
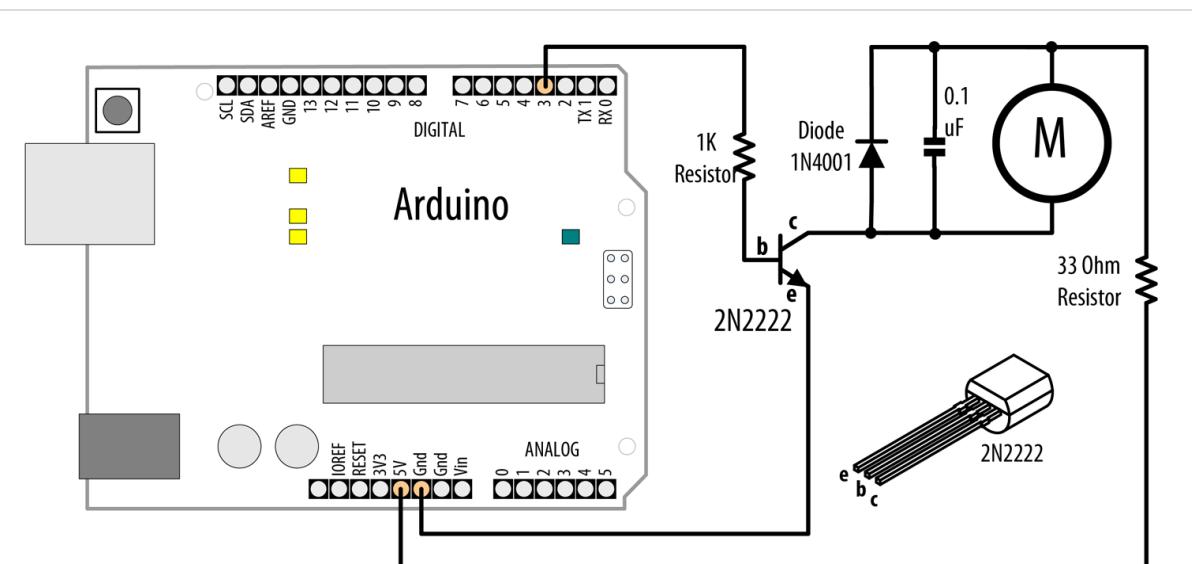
# 8.7 Making an Object Vibrate

## Problem

You want something to vibrate under Arduino control. For example, you want your project to shake for one second every minute.

## Solution

Connect a vibration motor as shown in Figure 8-8. You can use a ceramic capacitor for the 0.1 uF capacitor, but if you use an electrolytic capacitor, make sure the positive lead goes to the 33 ohm resistor that's connected to +5V.

*Connecting a vibration motor*

The following sketch will turn on the vibration motor for one second each minute:

```
/*
 * Vibrate sketch
 * Vibrate for one second every minute
 */

const int motorPin  = 3;  // vibration motor transistor is connected to pin 3

void setup()
{
  pinMode(motorPin, OUTPUT);
}

void loop()
{
  digitalWrite(motorPin,  HIGH);  // vibrate
  delay(1000); // delay one second
  digitalWrite(motorPin,  LOW);   // stop vibrating
  delay(59000); // wait 59 seconds.
}
```

## Discussion

This recipe uses a motor designed to vibrate, such as the SparkFun ROB-08449. If you have an old cell phone you no longer need, it may contain tiny vibration motors that would be suitable. Vibration motors require more power than an Arduino pin can provide, so a transistor is used to switch the motor current on and off. Almost any NPN transistor can be used; Figure 8-8 shows the common 2N2222. A 1K ohm resistor connects the output pin to the transistor base; the value is not critical, and you can use values up to 4.7K ohm or so (the resistor prevents too much current flowing through the output pin). The diode absorbs (or *snubs*—it's sometimes called a *snubber diode*) voltages produced by the motor windings as it rotates. The capacitor absorbs voltage spikes produced when the *brushes* (contacts connecting electric current to the motor windings) open and close. The 33 ohm resistor is needed to limit the amount of current flowing through the motor.

This sketch sets the output pin `HIGH` for one second (1,000 ms) and then waits for 59 seconds. The transistor will turn on (conduct) when the pin is `HIGH`, allowing current to flow through the motor.

Here is a variation of this sketch that uses a sensor to make the motor vibrate. The wiring is similar to that shown in Figure 8-8, with the addition of a photocell connected to analog pin 0 (see Recipe 6.3):

```
/*
 * Vibrate_Photocell sketch
 * Vibrate when photosensor detects light above ambient level
 */

const int motorPin  = 3;  // vibration motor transistor is connected to pin 3
const int sensorPin = A0; // Photodetector connected to analog input 0
int sensorAmbient = 0;               // ambient light level (calibrated in
setup)
const int thresholdMargin = 100;   // how much above ambient needed to vibrate

void setup()
{
  pinMode(motorPin, OUTPUT);
  sensorAmbient = analogRead(sensorPin); // get startup light level
}

void loop()
{
```

```
      int sensorValue = analogRead(sensorPin);
      if( sensorValue > sensorAmbient + thresholdMargin)
      {
        digitalWrite(motorPin,  HIGH);  // vibrate
      }
      else
      {
        digitalWrite(motorPin,  LOW);   // stop vibrating
      }
    }
```

Here the output pin is turned on when a light shines on the photocell. When the sketch starts, the background light level on the sensor is read and stored in the variable `sensorAmbient`. Light levels read in `loop` that are higher than this will turn on the vibration motor.

# 8.8 Driving a Brushed Motor Using a Transistor

## Problem

You want to turn a motor on and off. You may want to control its speed. The motor only needs to turn in one direction.

## Solution

This sketch turns the motor on and off and controls its speed from commands received on the serial port. Connect the components as shown in Figure 8-9. You can use a ceramic capacitor for the 0.1 uF capacitor, but if you use an electrolytic capacitor, make sure the positive lead goes to +5V:

```
/*
 * SimpleBrushed sketch
 * commands from serial port control motor speed
 * digits '0' through '9' are valid where '0' is off, '9' is max speed
 */

const int motorPin = 3; // motor driver is connected to pin 3

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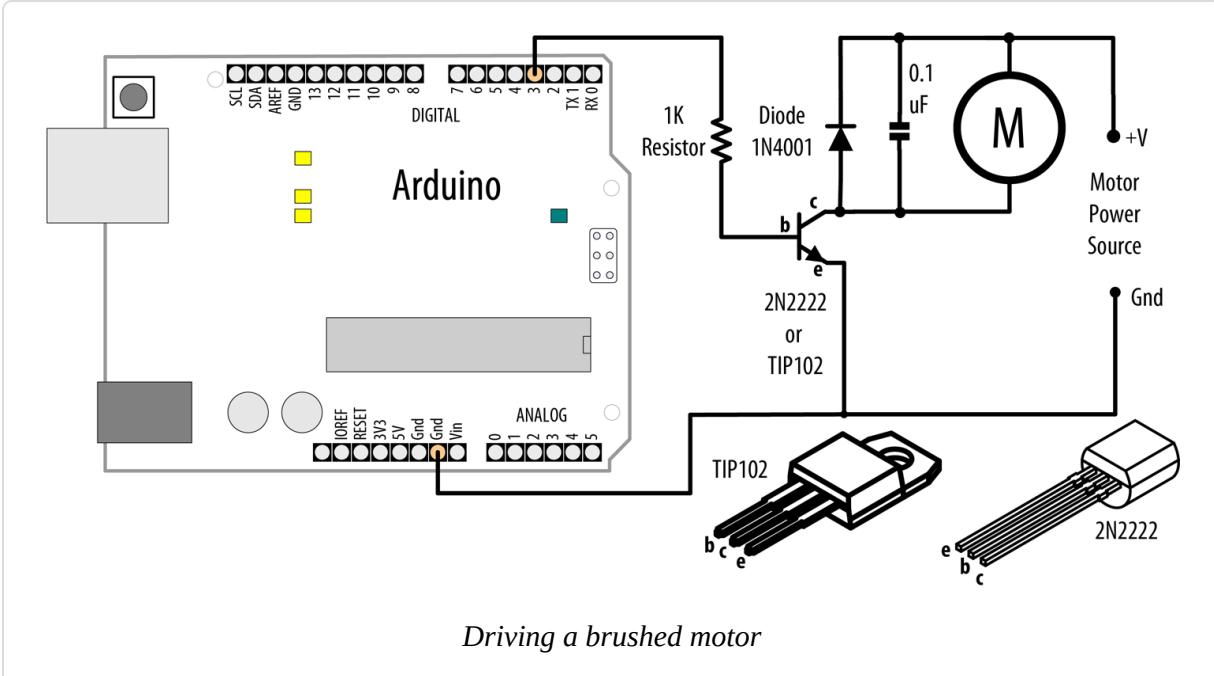
```
{
  Serial.begin(9600);
}

void loop()
{
  if (Serial.available())
  {
    char ch = Serial.read();

    if(isDigit(ch))   // is ch a number?
    {
      int speed = map(ch, '0', '9', 0, 255);
      analogWrite(motorPin, speed);
      Serial.println(speed);
    }
    else
    {
      Serial.print("Unexpected character ");
      Serial.println(ch);
    }
  }
}
```



*Driving a brushed motor*

## Discussion

This recipe is similar to Recipe 8.7; the difference is that `analogWrite` is used to control the speed of the motor. See "Analog Output" for more on `analogWrite` and Pulse Width Modulation (PWM).

# 8.9 Controlling the Direction of a Brushed Motor with an H-Bridge

## Problem

You want to control the direction of a brushed motor—for example, you want to cause a motor to rotate in one direction or the other from serial port commands.

## Solution
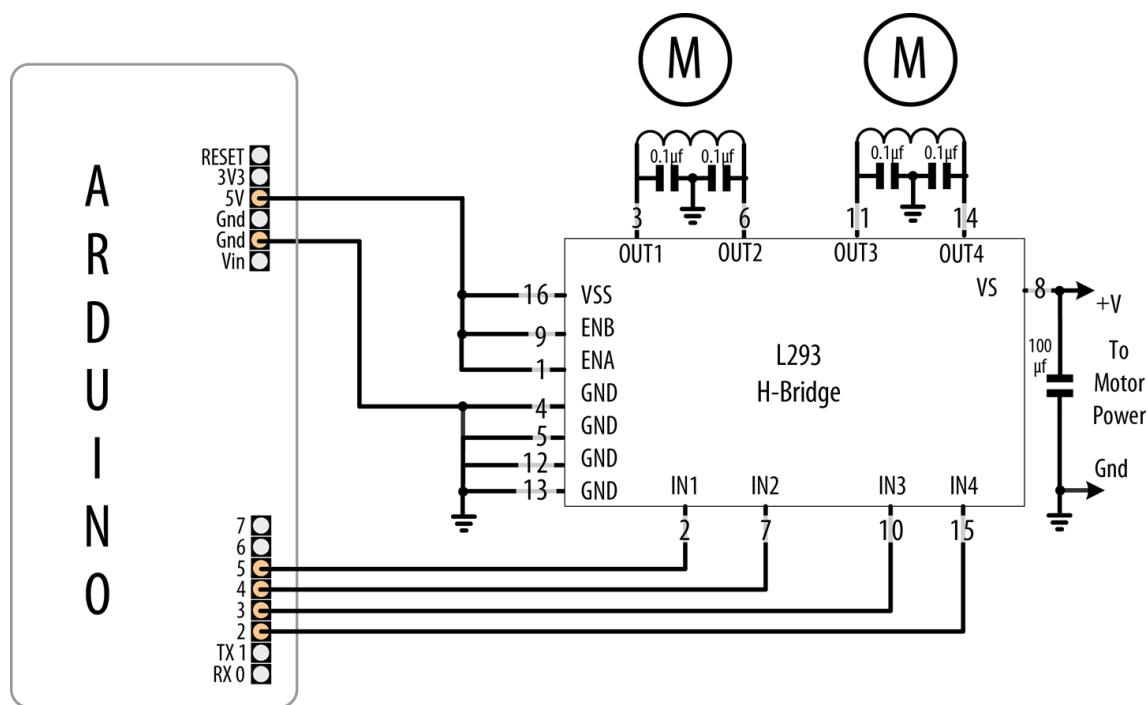
An H-Bridge is a component that can reverse the polarity of a motor, or stop it completely. Its name derives from the shape of the schematic representation of an H-Bridge circuit, but this recipe uses an integrated circuit version of an H-Bridge that can control two brushed motors. Figure 8-10 shows the connections for the L293D H-Bridge IC; you can also use the SN754410, which has the same pin layout. You should use ceramic capacitors for the 0.1 uF capacitors. Here's the sketch:

```
/*
 * Brushed_H_Bridge_simple sketch
 * commands from serial port control motor direction
 * + or - set the direction, any other key stops the motor
 */

const int in1Pin = 5;   // H-Bridge input pins
const int in2Pin = 4;

void setup()
{
  Serial.begin(9600);
  pinMode(in1Pin, OUTPUT);
  pinMode(in2Pin, OUTPUT);
  Serial.println("+ - to set direction, any other key stops motor");
}
```

```
void loop()
{
  if ( Serial.available()) {
    char ch = Serial.read();
    if (ch == '+')
    {
      Serial.println("CW");
      digitalWrite(in1Pin,LOW);
      digitalWrite(in2Pin,HIGH);
    }
    else if (ch == '-')
    {
      Serial.println("CCW");
      digitalWrite(in1Pin,HIGH);
      digitalWrite(in2Pin,LOW);
    }
    else if (ch != '\n' && ch != '\r') // ignore cr or lf
    {
      Serial.print("Stop motor");
      digitalWrite(in1Pin,LOW);
      digitalWrite(in2Pin,LOW);
    }
  }
}
```

*Connecting two brushed motors using an L293D H-Bridge*

## Discussion

Table 8-1 shows how the values on the H-Bridge input affect the motor. In the sketch in this recipe's Solution, a single motor is controlled using the IN1 and IN2 pins; the EN pin is permanently `HIGH` because it is connected to +5V.

*Logic table for H-Bridge*

| EN | IN1 | IN2 | Function |
|---|---|---|---|
| HIGH | LOW | HIGH | Turn clockwise |
| HIGH | HIGH | LOW | Turn counterclockwise |
| HIGH | LOW | LOW | Motor stop |
| HIGH | HIGH | HIGH | Motor stop |
| LOW | Ignored | Ignored | Motor stop |

Figure 8-10 shows how a second motor can be connected. The following sketch controls both motors together:

```
/*
 * Brushed_H_Bridge_simple2 sketch
 * commands from serial port control motor direction
 * + or - set the direction, any other key stops the motors
 */

const int in1Pin = 5;  // H-Bridge input pins
const int in2Pin = 4;

const int in3Pin = 3;  // H-Bridge pins for second motor
const int in4Pin = 2;

void setup()
{
  Serial.begin(9600);
  pinMode(in1Pin, OUTPUT);
  pinMode(in2Pin, OUTPUT);
  pinMode(in3Pin, OUTPUT);
  pinMode(in4Pin, OUTPUT);
  Serial.println("+ - sets direction of motors, any other key stops motors");
}

void loop()
{
  if ( Serial.available()) {
    char ch = Serial.read();
    if (ch == '+')
    {
      Serial.println("CW");
      // first motor
      digitalWrite(in1Pin,LOW);
      digitalWrite(in2Pin,HIGH);
      //second motor
      digitalWrite(in3Pin,LOW);
      digitalWrite(in4Pin,HIGH);
    }
    else if (ch == '-')
    {
      Serial.println("CCW");
      digitalWrite(in1Pin,HIGH);
      digitalWrite(in2Pin,LOW);

      digitalWrite(in3Pin,HIGH);
```

```
      digitalWrite(in4Pin,LOW);
    }
    else if (ch != '\n' && ch != '\r') // ignore cr or lf
    {
      Serial.print("Stop motors");
      digitalWrite(in1Pin,LOW);
      digitalWrite(in2Pin,LOW);
      digitalWrite(in3Pin,LOW);
      digitalWrite(in4Pin,LOW);
    }
  }
}
```
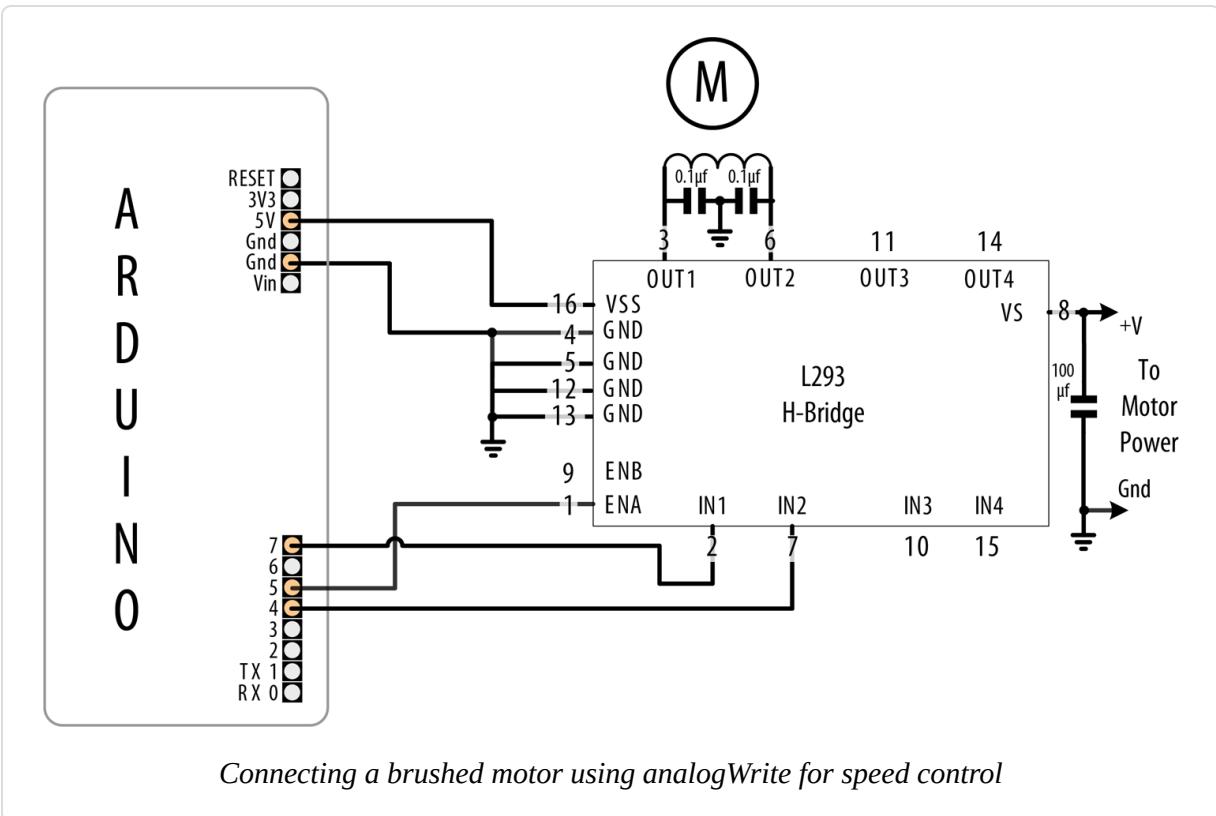
# 8.10 Controlling the Direction and Speed of a Brushed Motor with an H-Bridge

## Problem

You want to control the direction and speed of a brushed motor. This extends the functionality of Recipe 8.9 by controlling both motor direction and speed through commands from the serial port.

## Solution

Connect a brushed motor to the output pins of the H-Bridge as shown in Figure 8-11. You should use ceramic capacitors for the 0.1 uF capacitors.

*Connecting a brushed motor using analogWrite for speed control*

This sketch uses commands from the Serial Monitor to control the speed and direction of the motor. Sending 0 will stop the motor, and the digits 1 through 9 will control the speed. Sending "+" and "-" will set the motor direction:

```
/*
 * Brushed_H_Bridge sketch with speed control
 * commands from serial port control motor speed and direction
 * digits '0' through '9' are valid where '0' is off, '9' is max speed
 * + or - set the direction
 */

const int enPin  = 5;  // H-Bridge enable pin
const int in1Pin = 7;  // H-Bridge input pins
const int in2Pin = 4;

void setup()
{
  Serial.begin(9600);
  pinMode(in1Pin, OUTPUT);
  pinMode(in2Pin, OUTPUT);
  Serial.println("Speed (0-9) or + - to set direction");
```

```
    }

    void loop()
    {
      if ( Serial.available())
      {
        char ch = Serial.read();

        if(isDigit(ch)) // is ch a number?
        {
          int speed = map(ch, '0', '9', 0, 255);
          analogWrite(enPin, speed);
          Serial.println(speed);
        }
        else if (ch == '+')
        {
          Serial.println("CW");
          digitalWrite(in1Pin,LOW);
          digitalWrite(in2Pin,HIGH);
        }
        else if (ch == '-')
        {
          Serial.println("CCW");
          digitalWrite(in1Pin,HIGH);
          digitalWrite(in2Pin,LOW);
        }
        else if (ch != '\n' && ch != '\r') // ignore cr or lf
        {
          Serial.print("Unexpected character ");
          Serial.println(ch);
        }
      }
    }
```

## Discussion

This recipe is similar to Recipe 8.9, in which motor direction is controlled
by the levels on the IN1 and IN2 pins. But in addition, speed is controlled
by the `analogWrite` value on the EN pin (see Chapter 7 for more on
PWM). Writing a value of `0` will stop the motor; writing 255 will run the
motor at full speed. The motor speed will vary in proportion to values
within this range.
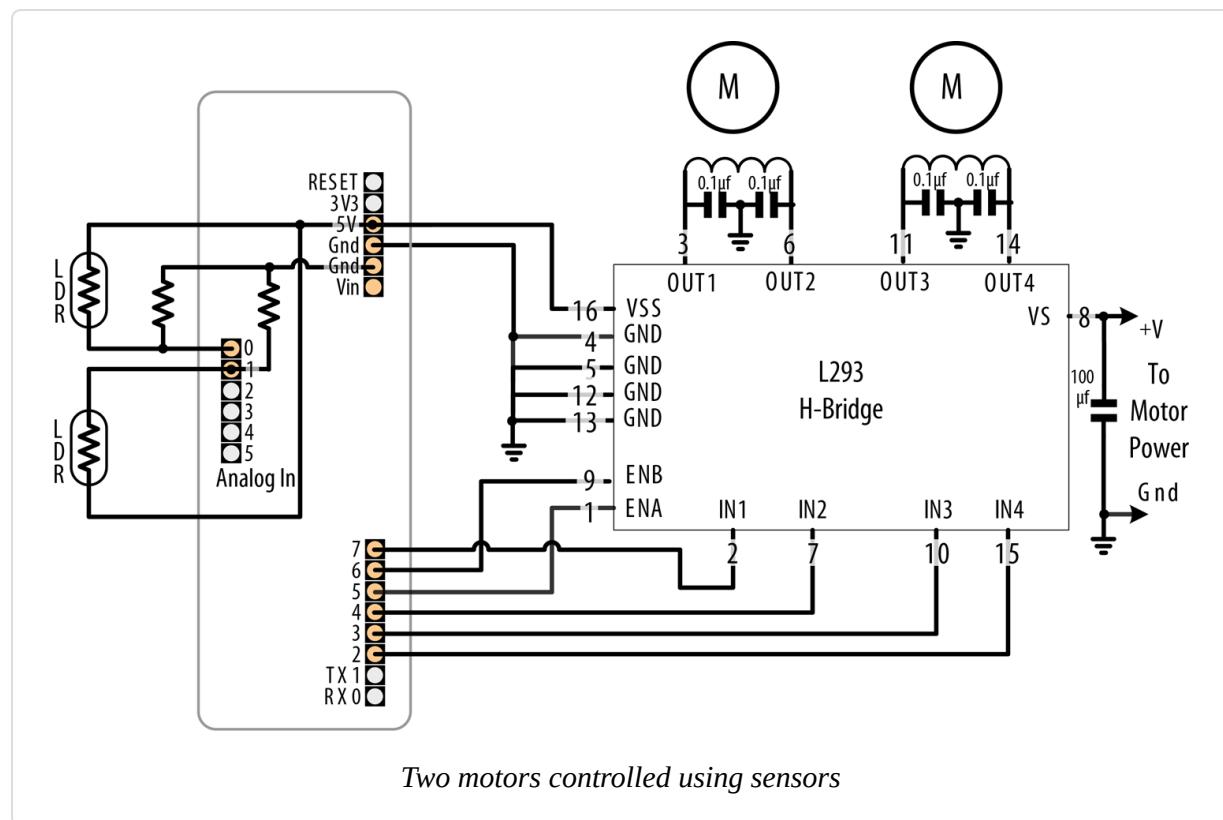
# 8.11 Using Sensors to Control the Direction and Speed of Brushed Motors

## Problem

You want to control the direction and speed of brushed motors with feedback from sensors. For example, you want two photo sensors to control motor speed and direction to cause a robot to move toward a beam of light.

## Solution

This Solution uses similar motor connections to those shown in Figure 8-10, but with the addition of two photoresistors (or phototransitors; see Recipe 1.6 for details), as shown in Figure 8-12. You should use ceramic capacitors for the 0.1 uF capacitors.



*Two motors controlled using sensors*

The sketch monitors the light level on the sensors and drives the motors to steer toward the sensor detecting the brighter light level:

```
/*
 * Brushed_H_Bridge_Direction sketch
 * uses photo sensors to control motor direction
 * robot moves in the direction of a light
 */

int leftPins[]  = {5,7,4};   // on pin for PWM, two pins for motor direction
int rightPins[] = {6,3,2};

const int MIN_PWM        = 64;   // this can range from 0 to MAX_PWM
const int MAX_PWM        = 128;  // this can range from around 50 to 255
const int leftSensorPin  = A0;   // analog pins with sensors
const int rightSensorPin = A1;

int sensorThreshold = 0;  // must have this much light on a sensor to move

void setup()
{
  for(int i=1; i < 3; i++)
  {
    pinMode(leftPins[i], OUTPUT);
    pinMode(rightPins[i], OUTPUT);
  }
}

void loop()
{
  int leftVal = analogRead(leftSensorPin);
  int rightVal = analogRead(rightSensorPin);

  if(sensorThreshold == 0) // have the sensors been calibrated?
  {
    // if not, calibrate sensors to something above the ambient average
    sensorThreshold = ((leftVal + rightVal) / 2) + 100 ;
  }

  if( leftVal > sensorThreshold || rightVal > sensorThreshold)
  {
    // if there is adequate light to move ahead
    setSpeed(rightPins, map(rightVal,0,1023, MIN_PWM, MAX_PWM));
    setSpeed(leftPins,  map(leftVal ,0,1023, MIN_PWM, MAX_PWM));
  }
}

void setSpeed(int pins[], int speed )
{
  if(speed < 0)
  {
```

```
      digitalWrite(pins[1],HIGH);
      digitalWrite(pins[2],LOW);
      speed = -speed;
    }
    else
    {
      digitalWrite(pins[1],LOW);
      digitalWrite(pins[2],HIGH);
    }
    analogWrite(pins[0], speed);
  }
```

## Discussion

This sketch controls the speed of two motors in response to the amount of light detected by two photocells. The photocells are arranged so that an increase in light on one side will increase the speed of the motor on the other side. This causes the robot to turn toward the side with the brighter light. Light shining equally on both cells makes the robot move forward in a straight line. Insufficient light causes the robot to stop.

---
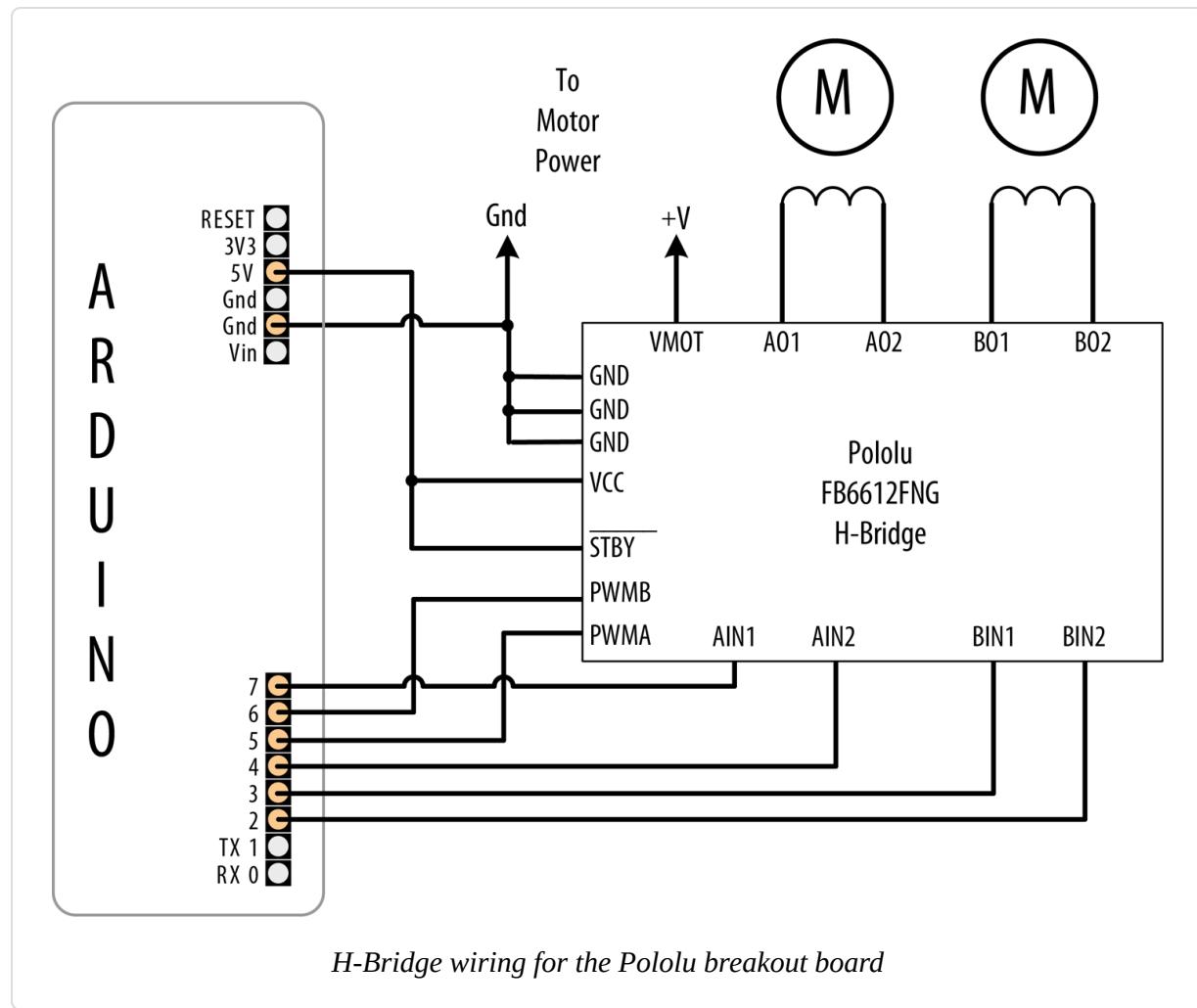
**TIP**

If you build this into a robot and have inadvertently created a light-fearing, rather than light-following, robot, try reversing the polarity of both motors. If your robot spins in place when it should be moving forward, try reversing the polarity of just one of the motors.

---

Light is sensed through analog inputs 0 and 1 using `analogRead` (see Recipe 6.3). When the program starts, the ambient light is measured and this threshold is used to determine the minimum light level needed to move the robot. A margin of 100 is added to the average level of the two sensors so the robot won't move for small changes in ambient light level. Light level as measured with `analogRead` is converted into a PWM value using the `map` function. Set `MIN_PWM` to the approximate value that enables your robot to move (low values will not provide sufficient torque; find this through trial and error with your robot). Set `MAX_PWM` to a value (up to 255) to determine the fastest speed you want the robot to move.

Motor speed is controlled in the `setSpeed` function. Two pins are used to control the direction for each motor, with another pin to control speed. The pin numbers are held in the `leftPins` and `rightPins` arrays. The first pin in each array is the speed pin; the other two pins are for direction.

An alternative to the L293 is the Toshiba TB6612FNG. This can be used in any of the recipes showing the L293D. Figure 8-13 shows the wiring for the TB6612 as used on the Pololu breakout board (Pololu item 713).



*H-Bridge wiring for the Pololu breakout board*

You can reduce the number of pins needed by adding additional hardware to control the direction pins. This is done by using only one pin per motor for direction, with a transistor or logic gate to invert the level on the other H-Bridge input. You can find circuit diagrams for this in the Arduino wiki, but if you want something already wired up, you can use an H-Bridge shield

such as the Arduino Motor Shield (7630049200371) or the Ardumoto from SparkFun (DEV-09213). Both are based on the L298, an alternative to the L293 that can drive more current. These shields plug directly into Arduino and only require connections to the motor power supply and windings.

Here is the sketch revised for the Arduino Motor Shield (analog pins 0 and 1 are used for current sensing, so the sketch uses A2 and A3):

```
/*
 * Brushed_H_Bridge_Direction sketch for motor shield
 * uses photo sensors to control motor direction
 * robot moves in the direction of a light
 */

int leftPins[]  = {3,12};    // one pin for PWM, one pin for motor direction
int rightPins[] = {11,13};

const int MIN_PWM       = 64;   // this can range from 0 to MAX_PWM
const int MAX_PWM       = 128;  // this can range from around 50 to 255
const int leftSensorPin = A2;   // analog pins with sensors
const int rightSensorPin = A3;

int sensorThreshold = 0; // must have this much light on a sensor to move

void setup()
{
  pinMode(leftPins[1], OUTPUT);
  pinMode(rightPins[1], OUTPUT);
}

void loop()
{
  int leftVal = analogRead(leftSensorPin);
  int rightVal = analogRead(rightSensorPin);
  if(sensorThreshold == 0) // have the sensors been calibrated?
  {
     // if not, calibrate sensors to something above the ambient average
     sensorThreshold = ((leftVal + rightVal) / 2) + 100 ;
  }

  if( leftVal > sensorThreshold || rightVal > sensorThreshold)
  {
    // if there is adequate light to move ahead
    setSpeed(rightPins, map(rightVal,0,1023, MIN_PWM, MAX_PWM));
    setSpeed(leftPins,  map(leftVal, 0,1023, MIN_PWM, MAX_PWM));
```

```
    }
  }

  void setSpeed(int pins[], int speed )
  {
    if(speed < 0)
    {
      digitalWrite(pins[1], HIGH);
      speed = -speed;
    }
    else
    {
      digitalWrite(pins[1], LOW);
    }
    analogWrite(pins[0], speed);
  }
```

The `loop` function is identical to the preceding sketch. `setSpeed` has less code because hardware on the shield allows a single pin to control motor direction.
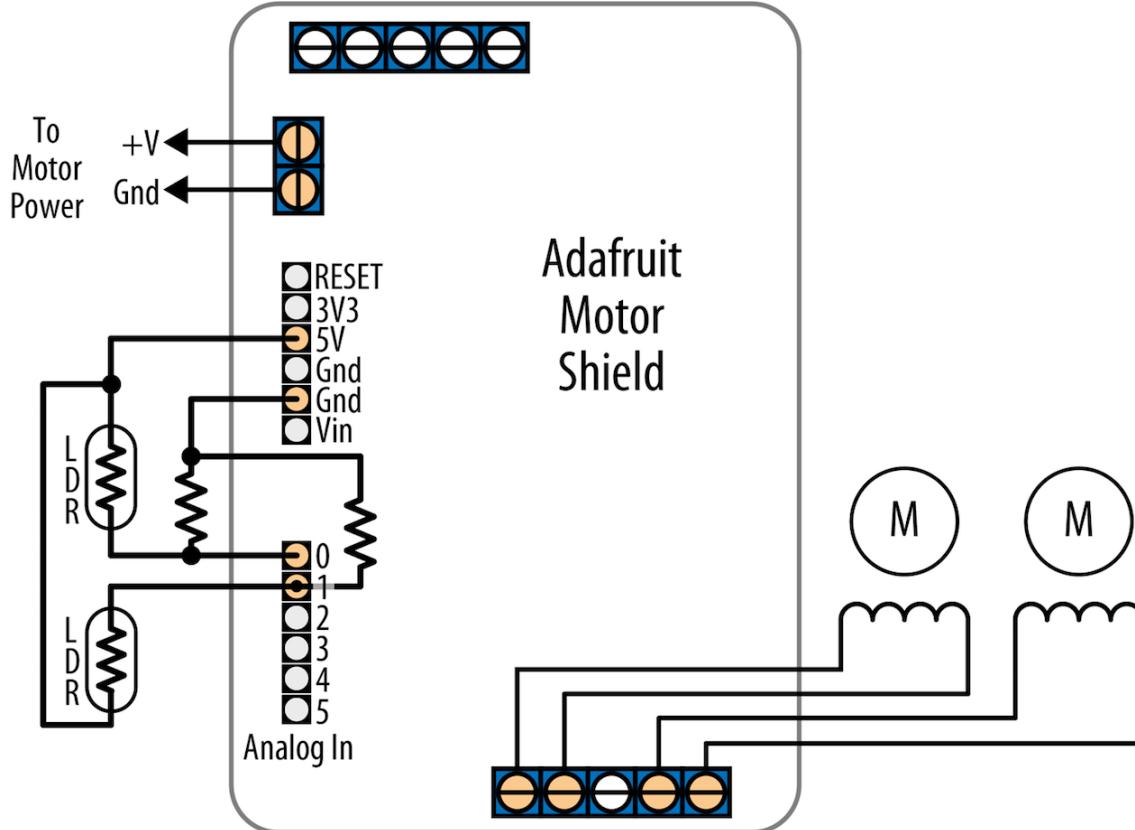
The Ardumoto Shield uses different pins, so you'd need to modify the code as shown:

```
int leftPins[]  = {3, 2};   // one pin for PWM, one pin for motor direction
int rightPins[] = {11, 4};
```

Here is the same functionality implemented using the Adafruit Motor Shield V2; see Figure 8-14. This uses a library named `Adafruit_MotorShield` that you can install using the Library Manager.

*Using the Adafruit Motor Shield*

The Adafruit shield supports four connections for motor windings; the sketch that follows has the motors connected to connectors 3 and 4:

```
/*
 * Brushed_H_Bridge_Direction sketch for Adafruit Motor shield
 * uses photo sensors to control motor direction
 * robot moves in the direction of a light
 */

#include <Wire.h>
#include <Adafruit_MotorShield.h> // Adafruit motor shield library

// Create an object for the shield
Adafruit_MotorShield AFMS = Adafruit_MotorShield();

Adafruit_DCMotor *leftMotor = AFMS.getMotor(1);
Adafruit_DCMotor *rightMotor = AFMS.getMotor(2);

const int MIN_PWM      = 64;  // this can range from 0 to MAX_PWM
```

```
const int MAX_PWM        = 128; // this can range from around 50 to 255
const int leftSensorPin  = A0;  // analog pins with sensors
const int rightSensorPin = A1;

int sensorThreshold = 0;   // must be more light than this on sensors to move

void setup()
{
  AFMS.begin();   // create with the default frequency 1.6KHz
}

void loop()
{
  int leftVal  = analogRead(leftSensorPin);
  int rightVal = analogRead(rightSensorPin);

  if(sensorThreshold == 0) // have the sensors been calibrated?
  {
    // if not, calibrate sensors to something above the ambient average
    sensorThreshold = ((leftVal + rightVal) / 2) + 100 ;
  }

  if( leftVal > sensorThreshold || rightVal > sensorThreshold)
  {
    // if there is adequate light to move ahead
    setSpeed(rightMotor, map(rightVal,0,1023, MIN_PWM, MAX_PWM));
    setSpeed(leftMotor,  map(leftVal ,0,1023, MIN_PWM, MAX_PWM));
  }
}

void setSpeed(Adafruit_DCMotor *motor, int speed )
{
  if(speed < 0)
  {
    motor->run(BACKWARD);
    speed = -speed;
  }
  else
  {
    motor->run(FORWARD);
  }
  motor->setSpeed(speed);
}
```

If you have a different shield than the ones mentioned in this recipe, you will need to refer to the datasheet and make sure the values in the sketch

match the pins used for PWM and direction.

## See Also

The datasheet for the Pololu board

The product page for the Ardumoto shield

The documentation for the Arduino Motor Shield

The Adafruit Motor Shield V2 documentation and library

# 8.12 Driving a Bipolar Stepper Motor

## Problem

You have a bipolar (four-wire) stepper motor and you want to step it under program control using an H-Bridge.

## Solution

This sketch steps the motor in response to serial commands. A numeric value followed by a + steps in one direction; a - steps in the other. For example, "24+" steps a 24-step motor through one complete revolution in one direction, and "12-" steps half a revolution in the other direction. Connect the components as shown in Figure 8-15. You should use ceramic capacitors for the 0.1 uF capacitors. Here's the sketch:

```
/*
 * Stepper_bipolar sketch
 * stepper is controlled from the serial port.
 * a numeric value followed by '+' or '-' steps the motor
 */

#include <Stepper.h>

// change this to the number of steps on your motor
#define STEPS 24

// create an instance of the stepper class, specifying
// the number of steps of the motor and the pins it's
```
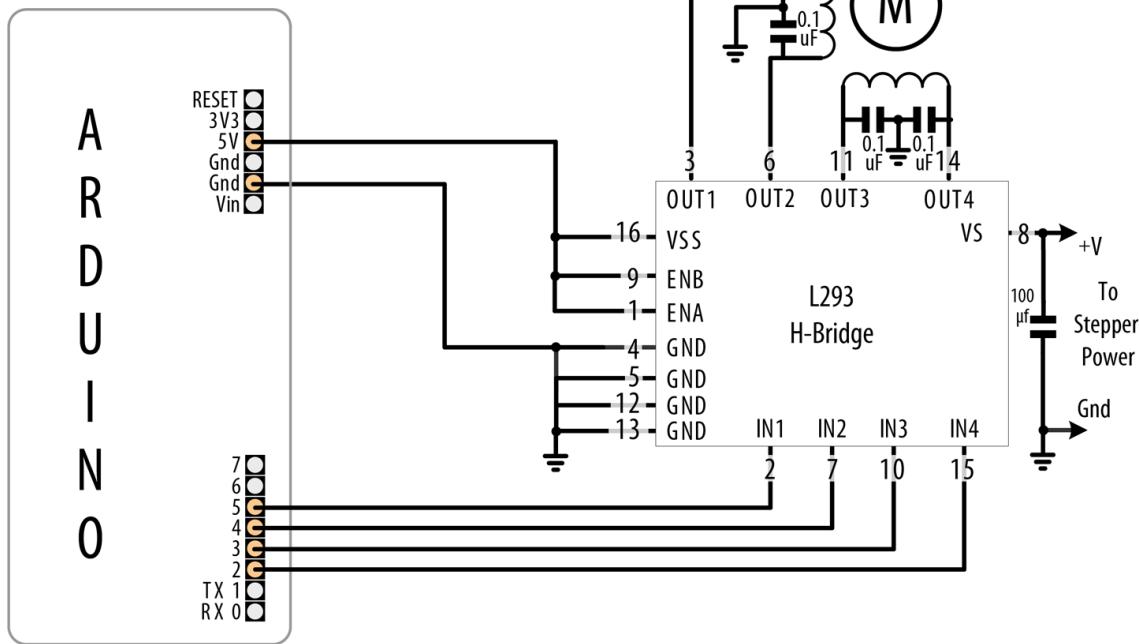
```
// attached to
Stepper stepper(STEPS, 2, 3, 4, 5);

int steps = 0;

void setup()
{
  // set the speed of the motor to 30 RPM
  stepper.setSpeed(30);
  Serial.begin(9600);
}

void loop()
{
  if ( Serial.available())
  {
    char ch = Serial.read();

    if(isDigit(ch)) // is ch a number?
    {
      steps = steps * 10 + ch - '0'; // yes, accumulate the value
    }
    else if(ch == '+')
    {
      stepper.step(steps);
      steps = 0;
    }
    else if(ch == '-')
    {
      stepper.step(steps * -1);
      steps = 0;
    }
  }
}
```

**Four-pin wiring to L293 H-Bridge**
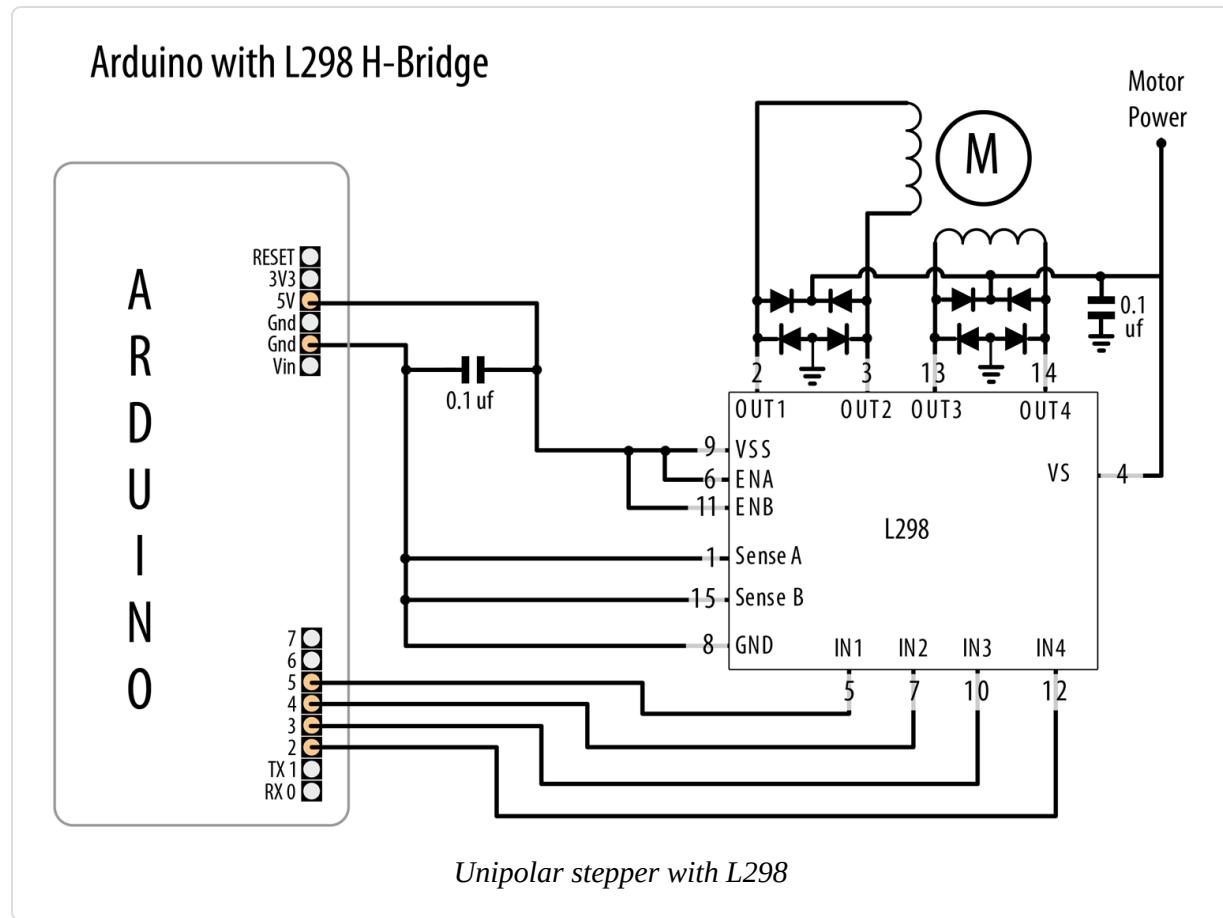
*Four-wire bipolar stepper using L293 H-Bridge*

## Discussion

This type of motor has two discrete groups of coils, and each group makes up a *phase* of the stepper motor. When a phase is energized in a particular direction, the motor turns a step in that direction. By pulsing the phases alternately, the motor can move several steps. There are four wires, and each pair of wires corresponds to a phase. You must consult the datasheet or other documentation for your motor to ensure that you are using the correct voltage and the correct number of steps (`#define STEPS`) with it, but if you don't know the wiring arrangement of it, there is a simple test you can do with a multimeter. Measure the resistance between different pairs of wires: you will find two pairs of wires that have the same resistance, and all the other pairs will have infinite resistance because there is no connection between them.

If your stepper requires a higher current than the L293 can provide (600 mA for the L293D), you can use the SN754410 chip for up to 1 amp with the

same wiring and code as the L293. For current up to 2 amps, you can use the L298 chip. The L298 can use the same sketch as shown in this recipe's Solution, and it should be connected as shown in Figure 8-16. You should use a ceramic capacitor for the 0.1 uF capacitor.



*Unipolar stepper with L298*

A simple way to connect an L298 to Arduino is to use the Arduino Motor Shield (7630049200371). This plugs on top of an Arduino board and only requires external connection to the motor windings; the motor power comes from the Arduino Vin (external Voltage Input) pin. In1/2 is controlled by pin 12, and ENA is pin 3. In3/4 is connected to pin 13, and ENB is on pin 11. Make the following changes to the code to use the preceding sketch with the Arduino Motor Shield:

```
Stepper stepper(STEPS, 12, 13);
```

Replace all the code inside of `setup()` with the following:

```
  pinMode(3, OUTPUT);
  digitalWrite(3, HIGH);    // enable A, use LOW to turn off the motor

  pinMode(11, OUTPUT);
  digitalWrite(11, HIGH);   // enable B, use LOW to turn off the motor

  stepper.setSpeed(60);    // set the speed of the motor to 60 rpm

  Serial.begin(9600);
```

The `loop` code is the same as the previous sketch.

> ### WARNING
>
> Stepper motors can draw a substantial amount of current, including when they are not moving. If you (gently; do not actually turn it) try to turn an energized stepper motor, you will feel resistance. An L293 or even L298 on a breadboard will get very hot over time, quite possibly too hot for the plastic composition of the breadboard. For this reason, we strongly suggest you use a motor shield, described next. Motor shields will include the appropriate heat sink and thermal resilience for this application. If you want to save power, you can turn off the stepper motors when you are not actively using them by taking the ENA and ENB pins low when not in use. This often defeats the purpose of using a stepper motor (that is, it will hold its position in between active stepping). See Recipe 8.13 for an example that uses a timeout to turn off motors when not in use.

## See Also

For more on stepper motor wiring, see Tom Igoe's stepper motor notes.

Documentation for the Stepper library

The Adafruit stepper motor tutorial
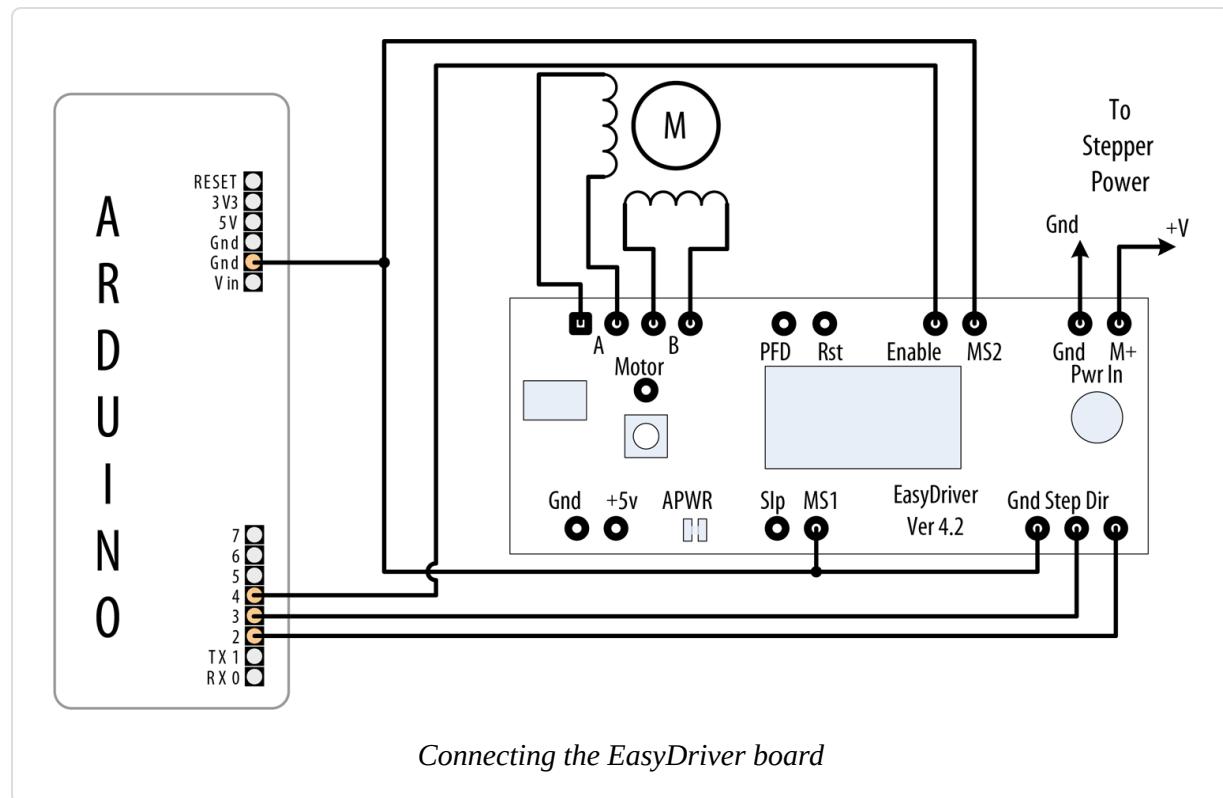
# 8.13 Driving a Bipolar Stepper Motor (Using the EasyDriver Board)

## Problem

You have a bipolar (four-wire) stepper motor and you want to step it under program control using the EasyDriver board.

# Solution

This Solution is similar to Recipe 8.12, and uses the same serial command protocol described there, but it uses the popular EasyDriver board. Figure 8-17 shows the connections.



*Connecting the EasyDriver board*

The following sketch controls the step direction and count from the serial port. Unlike the code in Recipe 8.12, it does not require the Stepper library, because the EasyDriver board handles the control of the motor coils in hardware:

```
/*
 * Stepper_Easystepper sketch
 * stepper is controlled from the serial port.
 * a numeric value followed by '+' or '-' steps the motor
 * a numeric value followed by 's' changes the speed
 */

const int dirPin  = 2;
const int stepPin = 3;
const int enPin   = 4;
```

```arduino
int speed = 100;      // desired speed in steps per second
int steps = 0;        // the number of steps to make
long last_step = millis();
long timeout = 30 * 1000; // turn off the motor after 30 secs of inactivity

void setup()
{
  pinMode(dirPin, OUTPUT);
  pinMode(stepPin, OUTPUT);
  pinMode(enPin, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  if (millis() > last_step + timeout)
  {
    digitalWrite(enPin,HIGH); // Turn off the motor
  }

  if ( Serial.available())
  {
    char ch = Serial.read();

    if(isDigit(ch)) // is ch a number?
    {
      steps = steps * 10 + ch - '0'; // yes, accumulate the value
    }
    else if(ch == '+')
    {
      step(steps);
      steps = 0;
    }
    else if(ch == '-')
    {
      step(-steps);
      steps = 0;
    }
    else if(ch == 's')
    {
      speed = steps;
      Serial.print("Setting speed to "); Serial.println(steps);
      steps = 0;
    }
  }
}

void step(int steps)
```

```
{
  int stepDelay = 1000 / speed;  //delay in ms for speed given as steps per
sec
  int stepsLeft;

  digitalWrite(enPin,LOW); // Enable the motor
  last_step = millis();

  // determine direction based on whether steps_to_mode is + or -
  if (steps > 0)
  {
    digitalWrite(dirPin, HIGH);
    stepsLeft = steps;
  }
  if (steps < 0)
  {
    digitalWrite(dirPin, LOW);
    stepsLeft = -steps;
  }
  // decrement the number of steps, moving one step each time
  while(stepsLeft > 0)
  {
    digitalWrite(stepPin,HIGH);
    delayMicroseconds(1);
    digitalWrite(stepPin,LOW);
    delay(stepDelay);
    stepsLeft--; // decrement the steps left
  }
}
```

## Discussion

The EasyDriver board is powered through the pins marked M+ and GND (shown in the upper right of Figure 8-17). The board operates with voltages between 8 volts and 30 volts; check the specifications of your stepper motor for the correct operating voltage. If you are using a 5V stepper, you must provide 5 volts to the pins marked GND and +5V (these pins are on the lower left of the EasyDriver board) and cut the jumper on the printed circuit board marked APWR (this disconnects the onboard regulator and powers the motor and EasyDriver board from an external 5V supply).

This sketch reduces power consumption when the motor has not moved for more than 30 seconds by setting the Enable pin HIGH to disable output (a

LOW value enables output). You can adjust this timeout by changing the last_step variable.

Stepping options are selected by connecting the MS1 and MS2 pins to +5V (HIGH) or GND (LOW), as shown in Table 8-2. With the board connected as shown in Figure 8-17, it will use full-step resolution (MS1 and MS2 are both LOW). Additionally, note that the reset pin is in its default state when not wired to GND (HIGH). Pulling it LOW will turn off stepper control.

*Microstep options*

| Resolution | MS1 | MS2 |
|---|---|---|
| Full step | LOW | LOW |
| Half step | HIGH | LOW |
| Quarter step | LOW | HIGH |
| Eighth step | HIGH | HIGH |

You can modify the code so that the speed value determines the revolutions per second as follows:

```
// use the following for speed given in RPM
int speed = 100;     // desired speed in RPM
int stepsPerRevolution = 200;  // this line sets steps for one revolution
```

Change the step function so that the first line is as follows:

```
int stepDelay = 60L * 1000L / stepsPerRevolution / speed; // speed as RPM
```

Everything else can remain the same, but now the speed command you send will be the RPM of the motor when it steps.

# 8.14 Driving a Unipolar Stepper Motor with the ULN2003A Driver Chip

## Problem

You have a unipolar (five- or six-wire) stepper motor and you want to control it using a ULN2003A Darlington driver chip.

## Solution

Connect a unipolar stepper as shown in Figure 8-18. The +V connection goes to a power supply rated for the voltage and current needed by your motor. You should use a ceramic capacitor for the 0.1 uF capacitor.

The following sketch steps the motor using commands from the serial port. A numeric value followed by a + steps in one direction; a - steps in the other:
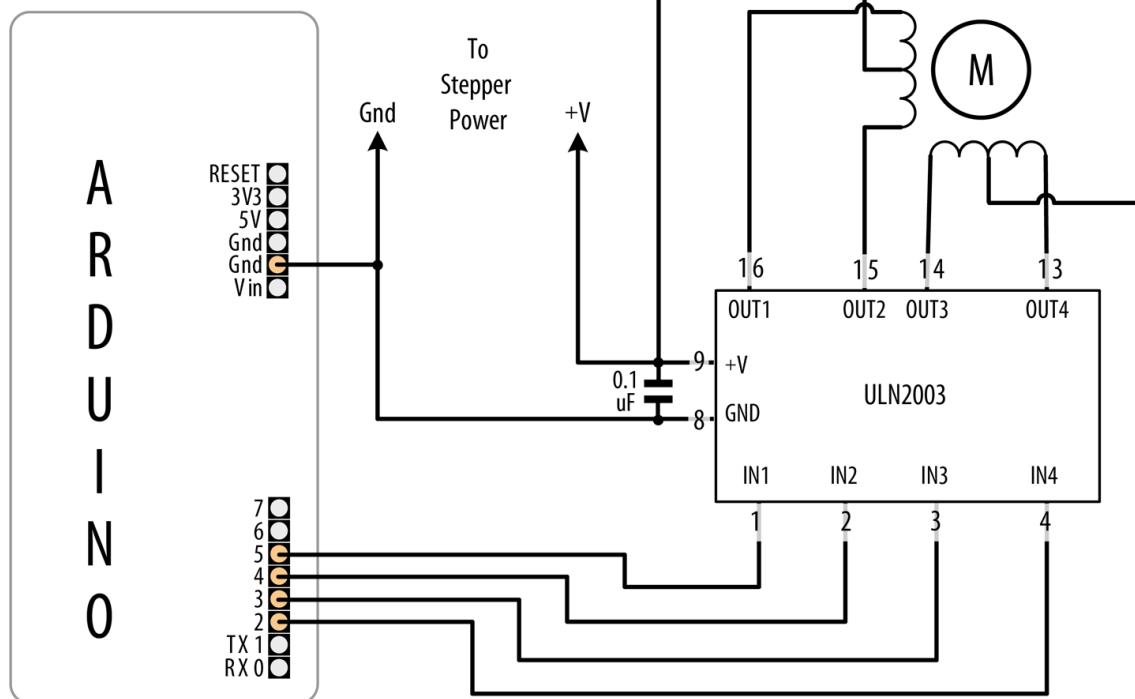
```
/*
 * Stepper sketch
 * stepper is controlled from the serial port.
 * a numeric value followed by '+' or '-' steps the motor
 */

#include <Stepper.h>

// change this to the number of steps on your motor
#define STEPS 24

// create an instance of the stepper class, specifying
// the number of steps of the motor and the pins it's
// attached to
Stepper stepper(STEPS, 2, 3, 4, 5);

int steps = 0;

void setup()
{
  stepper.setSpeed(30);    // set the speed of the motor to 30 RPMs
  Serial.begin(9600);
}

void loop()
{
  if ( Serial.available())
  {
    char ch = Serial.read();
```

```cpp
    if(isDigit(ch)) // is ch a number?
    {
      steps = steps * 10 + ch - '0';          // yes, accumulate the value
    }
    else if(ch == '+')
    {
      stepper.step(steps);
      steps = 0;
    }
    else if(ch == '-')
    {
      stepper.step(steps * -1);
      steps = 0;
    }
    else if(ch == 's')
    {
      stepper.setSpeed(steps);
      Serial.print("Setting speed to "); Serial.println(steps);
      steps = 0;
    }
  }
}
```

Unipolar Stepper – 4 wire

*Unipolar stepper connected using ULN2003 driver*

## Discussion

This type of motor has two coils, and each coil has a connection to the center. Motors with only five wires have both center connections brought out on a single wire. If the connections are not marked, you can identify the wiring using a multimeter. Measure the resistance across pairs of wires to find the two pairs of wires that have the maximum resistance. The center tap wire should have half the resistance of the full coil. A step-by-step procedure is available.

## See Also

For more on stepper motor wiring, see Tom Igoe's stepper motor notes.

Documentation for the Stepper library

# Audio Output

## 9.0 Introduction

There are millions of Google results for "Arduino music project" and there is only space in this chapter to introduce Arduino audio techniques and some examples to get you started. Arduino wasn't built to be a sophisticated audio synthesizer, but it can certainly produce sound through an output device such as a speaker.

This chapter shows how to create noise, play prerecorded sounds and create some simple output, and even experiment with sound synthesis.

If you want inspiration for your project, look through some of the many websites featuring music projects:

- Arduino blog entry with 172 music projects
- Arduino blog entries about music

Here are just two excellent examples of what can be achieved with Arduino, some hardware, and lots of creativity:

- A laser harp
- A theremin (See Recipe 9.6 for a very simple version of this early electronic instrument, but the real thing can be made with this information.)

Sound is produced by vibrating air. A sound has a distinctive pitch if the vibration repeats regularly. The Arduino can create sound by driving a loudspeaker or *Piezo device* (a small ceramic transducer that produces sound when pulsed), converting electronic vibrations into speaker pulses that vibrate the air. The pitch (frequency) of the sound is determined by the time it takes to pulse the speaker in and out; the shorter the amount of time, the higher the frequency.

The unit of frequency is measured in hertz, and it refers to the number of times the signal goes through its repeating cycle in one second. The range of human hearing is from around 20 hertz (Hz) up to 20,000 hertz (although it varies by person and changes with age).

The Arduino software includes a `tone` function for producing sound. Recipes 9.1 and 9.2 show how to use this function to make sounds and tunes. The `tone` function uses hardware timers. On a standard Arduino board (the Uno and similar boards), only one tone can be produced at a time. When you use the `tone` function, it will tie up the timer used for `analogWrite` on pins 3 and 11 so you'll need to choose different pins if you need analog output. To overcome this limitation, Recipe 9.3 shows how to use an enhanced tone library for multiple tones, and Recipe 9.4 shows how sound can be produced without using the `tone` function or hardware timers.

The sound that can be produced by pulsing a speaker is limited and does not sound very musical. The output is a square wave (see Figure 9-1), which sounds harsh and more like an antique computer game than a musical instrument.

It is difficult for basic boards like the Arduino Uno to produce more musically complex sounds without external hardware. You can add a shield that extends the Uno's capabilities such as the Adafruit Wave Shield, to play back audio files from a memory card on the shield.

Some of the more recent Arduino boards have a digital-to-analog converter output (DAC), which can produce high quality audio—either playing sound files from SD cards, or synthesizing sound in software (see Recipe 1.8). Another option in newer boards is I2S (Inter-IC Sound). This is digital
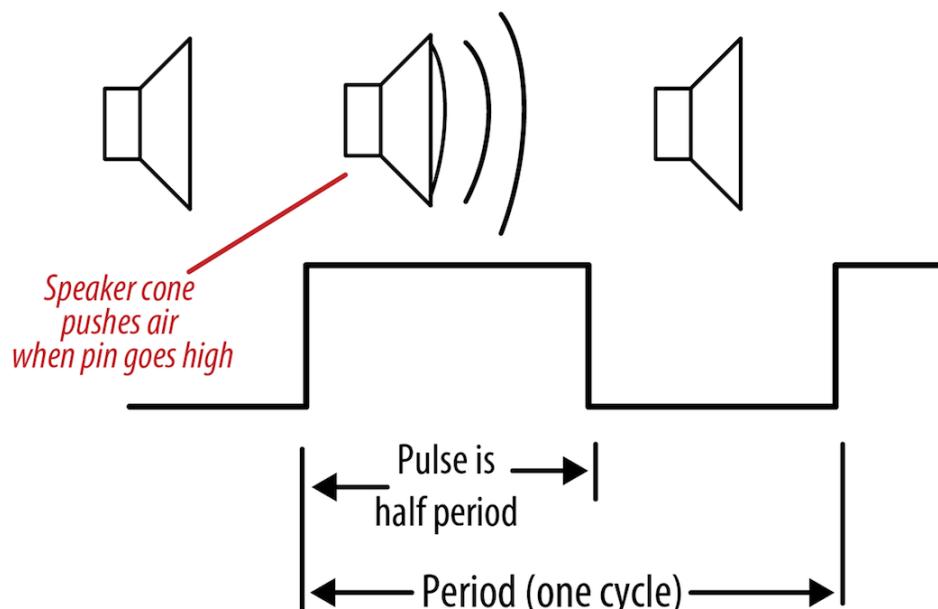
interface for communicating with external chips produces high-quality stereo audio interfaces—both input and output.

You can also use Arduino to control an external device that is built to make sound. Recipe 9.5 shows how to send Musical Instrument Digital Interface (MIDI) messages to a MIDI device. These devices produce high-quality sounds of a huge variety of instruments and can produce the sounds of many instruments simultaneously. The sketch in Recipe 9.5 shows how to generate MIDI messages to play a musical scale.



Frequency is the number of cycles per second.
Period is the time for one cycle.
Period = 1 / Frequency

Speaker cone pushes air when pin goes high

Pulse is half period

Period (one cycle)

*Generating sound using digital pulses*

Recipe 9.6 provides an overview of an application called Auduino that uses complex software processing to synthesize sound. Recipe 9.7 shows a more advanced audio synthesis library.
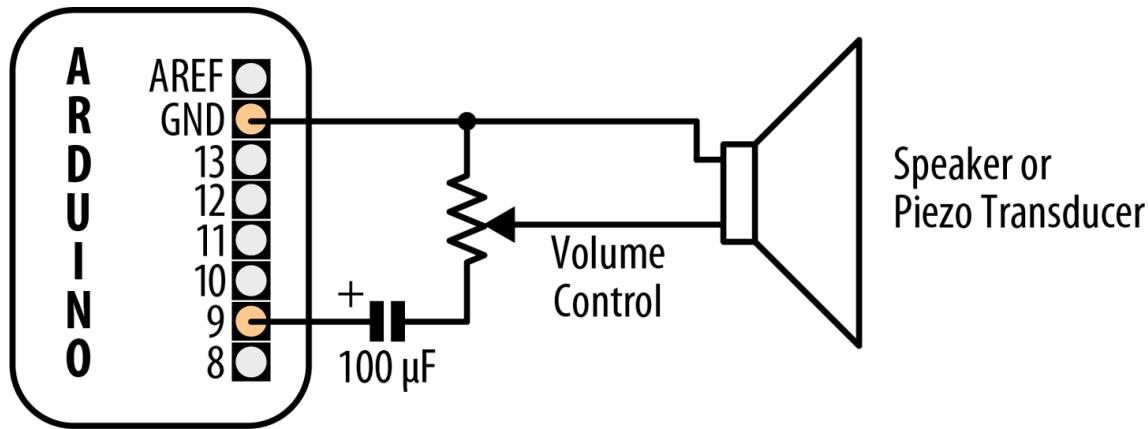
If you want to explore sophisticated musical applications, the 32-bit Teensy boards from PJRC are a good choice. There is also a well-developed audio library that makes use of the DSP capabilities of the chip on the Teensy to provide complex synthesis and audio effects using the DAC that is also built in, giving true analog audio output. An audio shield is available for these boards that has a microSD card reader and an I2S audio chip that produces stereo 16-bit 44.1 kHz audio output, as well as stereo audio in. The Teensy can also serve as a native USB MIDI, and an audio input and output device.

SparkFun offers a range of audio modules, including an Audio-Sound Breakout (part number WIG-11125) and MP3 player shield (part number DEV-12660).

The Arduino Zero, MKR1000, and MKRZero have a DAC pin, and if you install the experimental AudioZero library they can play a WAV file from an SD card.

This chapter covers the many ways you can generate sound electronically. If you want to make music by getting Arduino to play acoustic instruments (such as glockenspiels, drums, and acoustic pianos), you can employ actuators such as solenoids and servos that are covered in Chapter 8.

Many of the recipes in this chapter will drive a small speaker or Piezo device. The circuit for connecting one of these to an Arduino pin is shown in Figure 9-2.

*Connecting to an audio transducer*

The volume control is a variable resistor; the value is not critical and anything from 200 to 500 ohms would work. The capacitor is a 100 microfarad electrolytic with the positive end connected to the Arduino pin. A speaker will work regardless of which wire is attached to ground, but a Piezo is polarized, so connect the negative wire (usually black) to the GND pin.

Alternatively, you can connect the output to an external audio amplifier. Recipe 9.6 shows how an output pin can be connected to an audio jack.

> ### WARNING
>
> If you connect one of these circuits to headphones using an audio jack, make sure you set the volume to a safe level before putting the headphones on. Depending on which board you are using, and how you have wired it, the sound could be quite loud.

# 9.1 Playing Tones

## Problem

You want to produce audio tones through a speaker or other audio transducer. You want to specify the frequency and duration of the tone.

## Solution

Use the Arduino `tone` function. This sketch plays a tone with the frequency set by a variable resistor (or other sensor) connected to analog input 0 (see Figure 9-3):

```
/*
 * Tone sketch
 *
 * Plays tones through a speaker on digital pin 9
 * Frequency determined by values read from analog pin
 */

const int speakerPin = 9;     // connect speaker to pin 9
const int pitchPin = A0;      // pot that will determine the frequency of the
tone

void setup()
{
}

void loop()
{
  int sensor0Reading = analogRead(pitchPin);    // read input to set frequency

  // map the analog readings to a meaningful range
  int frequency = map(sensor0Reading, 0, 1023, 100, 5000); // 100 Hz to 5 kHz

  int duration = 250;    // how long the tone lasts
  tone(speakerPin, frequency, duration); // play the tone
  delay(1000); // pause one second
}
```
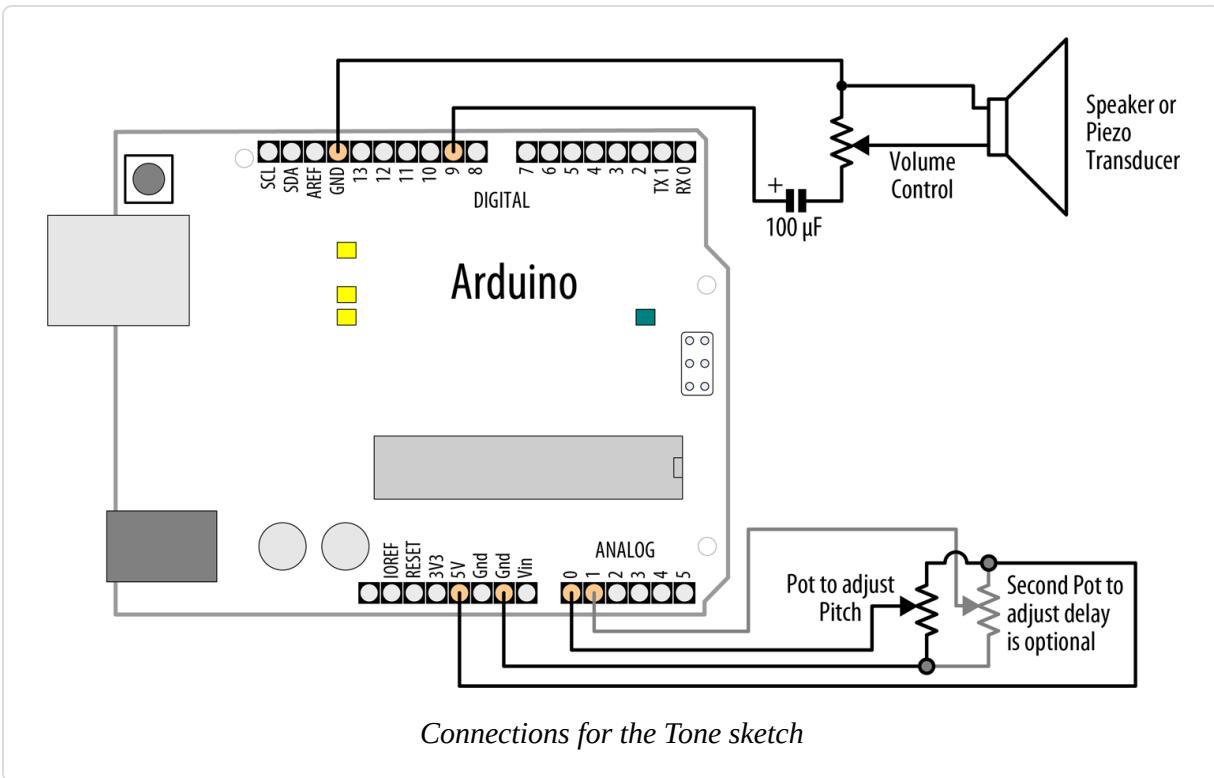
*Connections for the Tone sketch*

The `tone` function can take up to three parameters: the pin attached to the speaker, the frequency to play (in hertz), and the length of time (in milliseconds) to play the note. The third parameter is optional. If it is omitted, the note will continue until there is another call to `tone`, or a call to `noTone`. The value for the frequency is mapped to sensible values for audio frequencies in the following line:

```
int frequency  = map(sensor0Reading, 0, 1023, 100, 5000); //100 Hz to 5 kHz
```

This variation uses a second variable resistor (the bottom-right pot in Figure 9-3) to set the duration of the tone:

```
const int speakerPin = 9;    // connect speaker to pin 9
const int pitchPin = A0;     // input that determines frequency of the tone
const int durationPin = A1;  // input to determine the duration of the tone

void setup()
{
}
```

```
void loop()
{
  int sensor0Reading = analogRead(pitchPin);    // read input for frequency
  int sensor1Reading = analogRead(durationPin); // read input for duration

  // map the analog readings to a meaningful range
  int frequency  = map(sensor0Reading, 0, 1023, 100, 5000); // 100Hz to 5kHz
  int duration = map(sensor1Reading, 0, 1023, 100, 1000);    // dur 0.1-1
second
  tone(speakerPin, frequency, duration); // play the tone
  delay(duration); // wait for the tone to finish
}
```

Another variation is to add a switch so that tones are generated only when the switch is pressed.

Enable an input with pull-up resistors in `setup` with the following line (see Recipe 5.2 for a connection diagram and explanation). You'll also need to define `inputPin` to specify the pin you want to use:

```
pinMode(inputPin, INPUT_PULLUP);
```

Modify the `loop` code so that the `tone` and `delay` functions are only called when the switch is pressed:

```
if (digitalRead(inputPin) == LOW)  // read input value
{
  tone(speakerPin, frequency, duration); // play the tone
  delay(duration); //wait for the tone to finish
}
```

You can use almost any audio transducer to produce sounds with Arduino. Small speakers work very well. Piezo transducers also work and are inexpensive, robust, and easily salvaged from old audio greeting cards. Piezos draw little current (they are high-resistance devices), so they can be connected directly to the pin. Speakers are usually of much lower resistance and need a resistor to limit the current flow. The components to build the circuit pictured in Figure 9-3 should be easy to find.

## See Also

You can achieve enhanced functionality using the Tone library by Brett Hagman that is described in .

# 9.2 Playing a Simple Melody

## Problem

You want Arduino to play a simple melody.

## Solution

You can use the `tone` function described in to play sounds corresponding to notes on a musical instrument. This sketch uses `tone` to play a string of notes—the "Hello world" of learning the piano, "Twinkle, Twinkle Little Star":

```
/*
 * Twinkle sketch
 * Plays "Twinkle, Twinkle Little Star"
 * Speaker is connected to digital pin 9
 */

const int speakerPin = 9; // connect speaker to pin 9

char noteNames[] = {'C','D','E','F','G','a','b'};
unsigned int frequencies[] = {262,294,330,349,392,440,494};
const byte noteCount = sizeof(noteNames); // number of notes (7 here)

//notes, a space represents a rest
char score[] = "CCGGaaGFFEEDDC GGFFEEDGGFFEED CCGGaaGFFEEDDC ";
const byte scoreLen = sizeof(score); // the number of notes in the score

void setup()
{
}

void loop()
{
  for (int i = 0; i < scoreLen; i++)
  {
```

```
      int duration = 333;  // each note lasts for a third of a second
      playNote(score[i], duration); // play the note
      delay(duration/10); // slight pause to separate the notes
    }

    delay(4000); // wait four seconds before repeating the song
  }

  void playNote(char note, int duration)
  {
    // play the tone corresponding to the note name
    for (int i = 0; i < noteCount; i++)
    {
      // try and find a match for the noteName to get the index to the note
      if (noteNames[i] == note) // find a matching note name in the array
        tone(speakerPin, frequencies[i], duration); //  play the note
    }
    // if there is no match then the note is a rest, so just do the delay
    delay(duration);
  }
```

noteNames is an array of characters to identify notes in a score. Each entry in the array is associated with a frequency defined in the notes array. For example, note C (the first entry in the noteNames array) has a frequency of 262 Hz (the first entry in the notes array).

score is an array of notes representing the note names you want to play (lowercase notes are an octave higher than the uppercase notes):

```
  // a space represents a rest
  char score[] = "CCGGaaGFFEEDDC GGFFEEDGGFFEED CCGGaaGFFEEDDC ";
```

Each character in the score that matches a character in the noteNames array will make the note play. The space character is used as a rest, but any character not defined in noteNames will also produce a rest (no note playing).

The sketch calls playNote with each character in the score and a duration for the notes of one-third of a second.

The `playNote` function does a lookup in the `noteNames` array to find a match and uses the corresponding entry in the `frequencies` array to get the frequency to sound.

Every note has the same duration. If you want to specify the length of each note, you can add the following code to the sketch:

```
byte beats[scoreLen] = {1,1,1,1,1,1,2,  1,1,1,1,1,1,2,1,
                        1,1,1,1,1,1,2,  1,1,1,1,1,1,2,1,
                        1,1,1,1,1,1,2,  1,1,1,1,1,1,2};
byte beat = 180; // beats per minute for eighth notes
unsigned int speed = 60000 / beat; // the time in ms for one beat
```

`beats` is an array showing the length of each note: 1 is an eighth note, 2 a quarter note, and so on.

`beat` is the number of beats per minute.

`speed` is the calculation to convert beats per minute into a duration in milliseconds.

The only change to the `loop` code is to set the duration to use the value in the `beats` array. Change:

```
int duration = 333;  // each note lasts for a third of a second
```

to:

```
int duration = beats[i] * speed; // use beats array to determine duration
```

# 9.3 Generating More than One Simultaneous Tone

## Problem

You want to play two tones at the same time. The Arduino Tone library only produces a single tone on a standard board, and you want two simultaneous

tones. Note that the Mega board has more timers and can produce up to six tones.

## Solution

The Arduino Tone library is limited to a single tone because a different timer is required for each tone, and although a standard Arduino board has three timers, one is used for the `millis` function and another for servos. This recipe uses a library written by Brett Hagman, the author of the Arduino `tone` function. The library enables you to generate multiple simultaneous tones. You can download it or simply install it with the Library Manager.

This is an example sketch that plays part of Twinkle, Twinkle Little Star with the same notes across two octaves:

```
/*
 * Dual Tones
 * Plays Twinkle, Twinkle Little Star over two octaves.
 */
#include <Tone.h>

int notes1[] = {NOTE_C3, NOTE_C3, NOTE_G3, NOTE_G3, NOTE_A4, NOTE_A4,
                NOTE_G3, NOTE_F3, NOTE_F3, NOTE_E3, NOTE_E3, NOTE_D3,
                NOTE_D3, NOTE_C3 };
int notes2[] = {NOTE_C3, NOTE_C3, NOTE_G3, NOTE_G3, NOTE_A4, NOTE_A4,
                NOTE_G3, NOTE_F3, NOTE_F3, NOTE_E3, NOTE_E3, NOTE_D3,
                NOTE_D3, NOTE_C3 };
const byte scoreLen = sizeof(notes1)/sizeof(notes1[0]); // number of notes

// You can declare the tones as an array
Tone notePlayer[2];

void setup(void)
{
  notePlayer[0].begin(11);
  notePlayer[1].begin(12);
}

void loop(void)
{
  for (int i = 0; i < scoreLen; i++)
  {
```

```
        notePlayer[0].play(notes1[i]);
        delay(100); // Slight delay before starting the next note
        notePlayer[1].play(notes2[i]);
        delay(400);
        notePlayer[0].stop();
        notePlayer[1].stop();
        delay(30);
    }
    delay(1000);
}
```

## Discussion

To mix the output of the two tones to a single speaker, use 500 ohm resistors from each output pin and tie them together at the speaker. The other speaker lead connects to GND, as shown in the previous sketches.

On a standard Arduino board, the first tone will use timer 2 (so PWM on pins 9 and 10 will not be available); the second tone uses timer 1 (preventing the Servo library and PWM on pins 11 and 12 from working). On a Mega board, each simultaneous tone will use timers in the following order: 2, 3, 4, 5, 1, 0. At the time of this writing, this library uses features that are specific to the AVR architecture and is not supported on ARM boards or MegaAVR boards such as the Uno WiFi R2 or the Nano Every.

When you play two notes that are the same frequency, or are the same note from a different octave, you may notice an effect known as a *beat* that is similar to a tremelo. This happens because the two channels are not in perfect sync. This effect is used to tune guitar strings manually: the beat ceases to be heard once the string is in tune with the reference note.

> ### WARNING
>
> Playing three simultaneous notes on a standard Uno Arduino board, or more than six on a Mega, is possible, but `millis` and `delay` will no longer work properly. It is safest to use only two simultaneous tones (or five on a Mega).

# 9.4 Generating Audio Tones Without Interfering with PWM

## Problem

You want to produce sounds through a speaker or other audio transducer, and you need to generate the tone in software instead of with a timer; for example, if you need to use `analogWrite` on pin 3 or 11.

## Solution

The `tone` function discussed in earlier recipes is easy to use, but it requires a hardware timer, which may be needed for other tasks such as `analogWrite`. This code does not use a timer, but it will not do anything else while the note is played. Unlike the Arduino `tone` function, the `playTone` function described here will block—it will not return until the note has finished.

The sketch plays six notes, each one twice the frequency of (an octave higher than) the previous one. The `playTone` function generates a tone for a specified duration on a speaker or Piezo device connected to a digital output pin and ground; see Figure 9-4:

```
/*
 * Tone and fade sketch
 * Plays tones while fading an LED
 */
byte speakerPin = 9;
byte ledPin = 3;

void setup()
{
  pinMode(speakerPin, OUTPUT);
}

void playTone(int period, int duration)
{
  // period is one cycle of tone
  // duration is how long the pulsing should last in milliseconds
  int pulse = period / 2;
  for (long i = 0; i < duration * 1000L; i += period )
```

```
  {
    digitalWrite(speakerPin, HIGH);
    delayMicroseconds(pulse);
    digitalWrite(speakerPin, LOW);
    delayMicroseconds(pulse);
  }
}

void fadeLED(){
  // These two static variables are assigned initial values
  // only the first time the function is called.
  static int brightness = 0;
  static int changeval  = 5;

  analogWrite(ledPin, brightness);

  // If we've exceeded the limits of analogWrite
  brightness += changeval;
  if (brightness >= 255 || brightness <= 0)
    changeval *= -1; // Change direction

  delay(2);
}

void loop()
{
  // a note with period of 15289 is deep C (second lowest C note on piano)
  for(int period=15289; period >= 477;  period=period / 2)  // play 6 octaves
  {
    playTone(period, 200); // play tone for 200 ms
    fadeLED();
  }
}
```
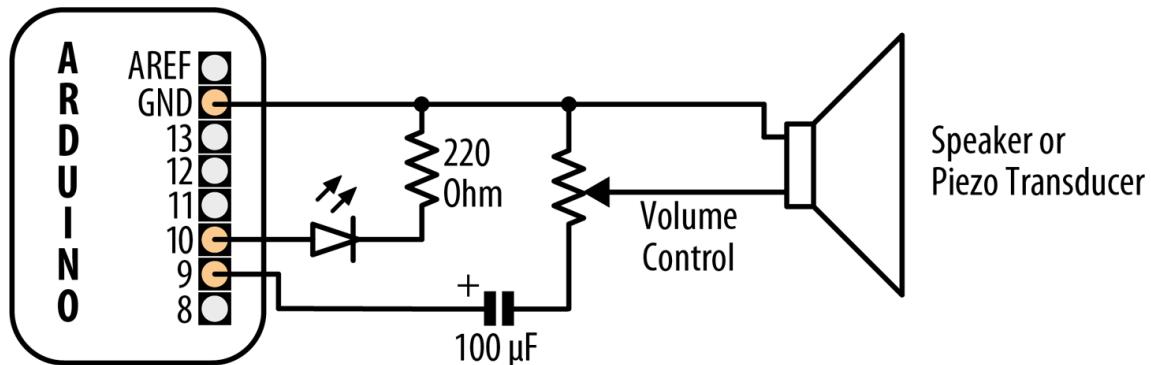
*Connections for speaker and LED*

## Discussion

Two values are used by `playTone`: `period` and `duration`. The variable `period` represents the time for one cycle of the tone to play. The speaker is pulsed high and then low for the number of microseconds given by `period`. The `for` loop repeats the pulsing for the number of milliseconds given in the `duration` argument.

If you prefer to work in frequency rather than period, you can use the reciprocal relationship between frequency and period; period is equal to 1 divided by frequency. You need the period value in microseconds; because there are 1 million microseconds in one second, the period is calculated as 1000000L / frequency (the "L" at the end of that number tells the compiler that it should calculate using long integer math to prevent the calculation from exceeding the range of a normal integer—see the explanation of long integers in Recipe 2.2):

```
void playFrequency(int frequency, int duration)
{
  int period = 1000000L / frequency;
  int pulse = period / 2;
```

The rest of the code is the same as `playTone`:

```
for (long i = 0; i < duration * 1000L; i += period )
 {
   digitalWrite(speakerPin, HIGH);
   delayMicroseconds(pulse);
   digitalWrite(speakerPin, LOW);
   delayMicroseconds(pulse);
 }
}
```

The code in this recipe stops and waits until a tone has completed before it can do any other processing. It is possible to produce the sound in the background (without waiting for the sound to finish) by putting the sound generation code in an interrupt handler. The source code for the `tone` function that comes with the Arduino distribution shows how this is done.

## See Also

Recipe 9.6

Some examples of more complex audio synthesis you can do with the Arduino:

Pulse-Code Modulation

> PCM allows you to approximate analog audio using digital signaling. This Arduino wiki article explains how to produce 8-bit PCM using a timer.

Pocket Piano shield

> Modern Device's Fluxamasynth Shield is a 64-voice polyphonic synthesizer shield for Arduino.
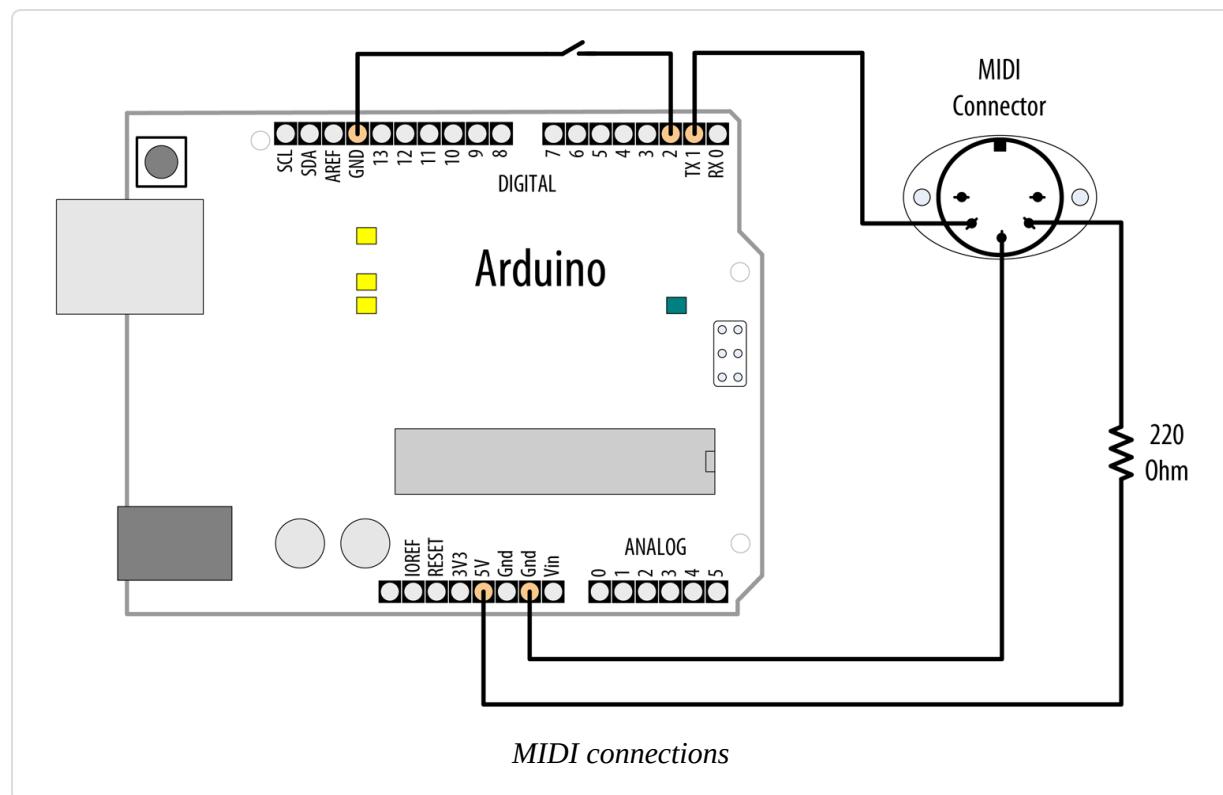
# 9.5 Controlling MIDI

## Problem

You want to get a MIDI synthesizer to play music using Arduino.

## Solution

To connect to a MIDI device, you need a five-pin DIN plug or socket. If you use a socket, you will also need a lead to connect to the device. Connect the MIDI connector to Arduino using a 220 ohm resistor, as shown in Figure 9-5.



*MIDI connections*

To upload the code onto Arduino, you should disconnect the MIDI device, as it may interfere with the upload. After the sketch is uploaded, connect a MIDI sound device to the Arduino output. A musical scale will play each time you press the button connected to pin 2:

```
/*
 * midiOut sketch
 * Sends MIDI messages to play a scale on a MIDI instrument
 * each time the switch on pin 2 is pressed
 */

// these numbers specify which note to play
const byte notes[8] = {60, 62, 64, 65, 67, 69, 71, 72};
const int num_notes = sizeof(notes)/ sizeof(notes[0]);

const int switchPin = 2;
```

```
const int ledPin = LED_BUILTIN;

void setup() {
  Serial.begin(31250);
  pinMode(switchPin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  if (digitalRead(switchPin) == LOW)
  {
    for (byte noteNumber = 0; noteNumber < num_notes; noteNumber++)
    {
      // Play the note
      playMidiNote(1, notes[noteNumber], 127);
      digitalWrite(ledPin, HIGH);
      delay(70); // Hold the note

      // Stop playing the note (velocity of 0)
      playMidiNote(1, notes[noteNumber], 0);
      digitalWrite(ledPin, HIGH);
      delay(30);
    }
  }
}

void playMidiNote(byte channel, byte note, byte velocity)
{
  byte midiMessage= 0x90 + (channel - 1);
  Serial.write(midiMessage);
  Serial.write(note);
  Serial.write(velocity);
}
```

## Discussion

This sketch uses the serial TX pin to send MIDI information. The circuit connected to pin 1 may interfere with uploading code to the board. Remove the wire from pin 1 while you upload, and plug it back in afterward.

MIDI was originally used to connect digital musical instruments together so that one could control another. The MIDI specification describes the electrical connections and the messages you need to send.

MIDI is actually a serial connection (at a nonstandard serial speed, 31,250 baud), so Arduino can send and receive MIDI messages using its serial port hardware from pins 0 and 1. Because the serial port is occupied by MIDI messages, you can't print messages to the Serial Monitor, so the sketch flashes the onboard LED each time it sends a note.

Each MIDI message consists of at least one byte. This byte specifies what is to be done. Some commands need no other information, but other commands need data to make sense. The message in this sketch is *note on,* which needs two pieces of information: which note and how loud. Both of these bits of data are in the range of zero to 127.

The sketch initializes the serial port to a speed of 31,250 baud; the other MIDI-specific code is in the function `playMidiNote`:

```
void playMidiNote(byte channel, byte note, byte velocity)
{
  byte midiMessage= 0x90 + (channel - 1);
  Serial.write(midiMessage);
  Serial.write(note);
  Serial.write(velocity);
}
```

This function takes three parameters and calculates the first byte to send using the channel information.

MIDI information is sent on different channels between 1 and 16. Each channel can be set to be a different instrument, so multichannel music can be played. The command for *note on* (to play a sound) is a combination of 0x90 (the top four bits at b1001), with the bottom four bits set to the numbers between b0000 and b1111 to represent the MIDI channels. The byte represents channels using 0 to 15 for channels 1 to 16, so 1 is subtracted first.

Then the note value and the volume (referred to as *velocity* in MIDI, as it originally related to how fast the key was moving on a keyboard) are sent.

The serial `write` statements specify that the values must be sent as bytes (rather than as the ASCII value). `println` is not used because a line return

character would insert additional bytes into the signal that are not wanted.

The sound is turned off by sending a similar message, but with velocity set to 0.

This recipe works with MIDI devices having five-pin DIN MIDI in connectors. If your MIDI device only has a USB connector, this will not work. It will not enable the Arduino to control MIDI music programs running on your computer without additional hardware (a MIDI-to-USB adapter). Although Arduino has a USB connector, your computer recognizes it as a serial device, not a MIDI device.

## See Also

To send and receive MIDI, have a look at the MIDI library.

MIDI messages are detailed on this MIDI Association page.

The SparkFun MIDI shield is a kit that includes MIDI in and out connectors, some buttons, and an optoisolator to keep the MIDI device and the Arduino electrically isolated.

The Teensy board is also able to be programmed to be a native USB MIDI device.

# 9.6 Making an Audio Synthesizer

## Problem

You want to generate complex sounds similar to those used to produce electronic music.

## Solution

The simulation of audio oscillators used in a sound synthesizer is complex, but Peter Knight has created a sketch called Auduino that enables Arduino to produce more complex and interesting sounds. Because it uses many low-level capabilities, Auduino is unlikely to run on anything other than 8-bit boards based on the ATmega such as the Uno.

Download the sketch by following this link.

Connect five 4.7K ohm linear potentiometers to analog pins 0 through 4, as shown in Figure 9-6. Potentiometers with full-size shafts are better than small presets because you can easily twiddle the settings. Pin 3 is used for audio output and is connected to an amplifier using a jack plug.
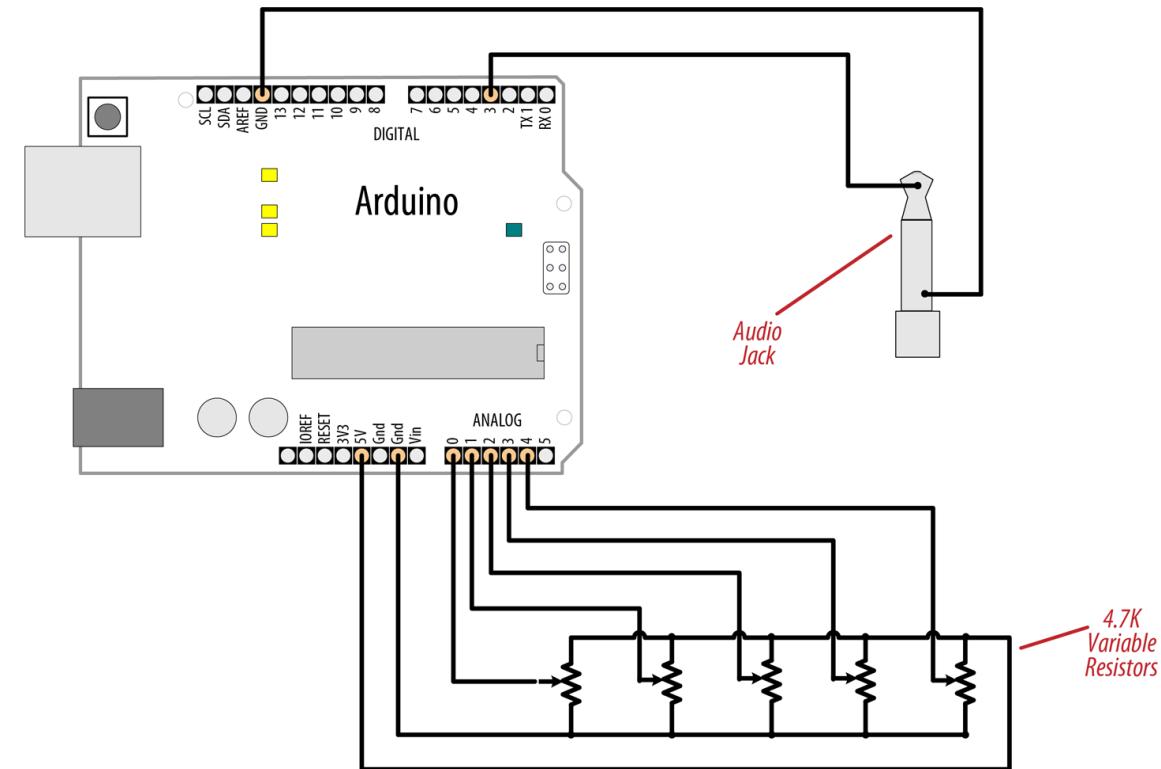
> **WARNING**
>
> The voltage level (5 volts) is higher than audio amplifiers expect, so you may need to use a 4.7K variable resistor to reduce the voltage (connect one end to pin 9 and the other end to ground; then connect the slider to the tip of the jack plug. The barrel of the jack plug is connected to ground).

## Discussion

The sketch code is complex because it is directly manipulating hardware timers to generate the desired frequencies, which are transformed in software to produce the audio effects. It is not included in the text because you do not need to understand the code to use Auduino.

Auduino uses a technique called *granular synthesis* to generate the sound. It uses two electronically produced sound sources (called *grains*). The variable resistors control the frequency and decay of each grain (inputs 0 and 2 for one grain and inputs 3 and 1 for the other). Input 4 controls the synchronization between the grains.

*Wiring diagram for controlling and listening to Auduino*

If you want to tweak the code, you can change the scale used to calculate the frequency. The default setting is pentatonic, but you can comment that out and uncomment another option to use a different scale.

Be careful when adding code to the main loop, because the sketch is highly optimized and additional code could slow things down too much, causing the audio synthesis to not work well.

You can replace any of the pots with sensors that can produce an analog voltage signal (see Chapter 6). For example, a photoresistor (see Recipe 6.3) or a distance sensor (the analog output described toward the end of Recipe 6.5) connected to one of the frequency inputs (pin 0 or 3) would enable you to control the pitch by moving your hand closer to or farther from the sensor (look up "theremin" in Wikipedia or on Google to read more about this musical instrument that is played by sensing hand movement).

## See Also

# 9.7 Attain High-Quality Audio Synthesis

## Problem

You want to generate higher-quality audio than is possible with the Tone library on an 8-bit board, which can generally produce only square wave signals without additional hardware. For example, you want to play music with sine waves, play back WAV files, or perform advanced sound synthesis without additional hardware.

## Solution

As you saw in Recipe 1.8, SAMD-based boards have a digital-to-analog converter (DAC) on one pin that can generate true voltages between 0V and the operating voltage (3.3V) of the board. This, coupled with the higher speed of 32-bit boards, allows you to generate complex waveforms. Connect your board to a transducer as shown in Figure 9-2, but instead of connecting your transducer to pin 9, use the DAC pin (analog 0 on the Arduino Zero, Adafruit Metro M0, and SparkFun RedBoard Turbo). Install Mozzi (see Recipe 16.2) and run the following script:

```
/*
 * Mozzi Melody
 * Play the Chimes of Big Ben
 */

#include <MozziGuts.h>
#include <Oscil.h> // oscillator template
#include <tables/sin2048_int8.h> // sine table for oscillator
#include <EventDelay.h>
#include <mozzi_midi.h>
```

```cpp
#define CONTROL_RATE 64 // Control rate in Hz, use powers of 2

enum notes
{
  E3 = 52, B4 = 59, E4 = 64, F4S = 66, G4S = 68, REST = 0
};

int score[] = { E4, G4S, F4S, B4, REST,
                E4, F4S, G4S, E4, REST,
                G4S, E4, F4S, B4, REST,
                B4, F4S, G4S, E4, REST,
                E3, REST, E3, REST, E3, REST, E3, REST };
const byte scoreLen = sizeof(score) / sizeof(score[0]);

byte beats[scoreLen] = {2, 2, 2, 2, 2,
                        2, 2, 2, 2, 2,
                        2, 2, 2, 2, 2,
                        2, 2, 2, 2, 10,
                        4, 4, 4, 4, 4, 4, 4, 10};
unsigned int beat_ms = 60000 / 180; // the time in ms for 1/8 note

const int pauseTime = 200; // pause between notes
int currNote = 0; // index of the note we're playing
bool pausing = false; // Mini-rest between beats

Oscil <SIN2048_NUM_CELLS, AUDIO_RATE> aSin(SIN2048_DATA); // Sine wave
EventDelay kChangeNoteDelay; // Delay object

void setup() {
  startMozzi(CONTROL_RATE);
  kChangeNoteDelay.start();
}

void updateControl() {

  if (kChangeNoteDelay.ready()) { // Is the delay up?
    pausing = !pausing; // Toggle rest state
    if (pausing) {
      aSin.setFreq(0); // set the frequency
      kChangeNoteDelay.set(pauseTime); // Hold the rest for 200 ms
    } else {
      if (currNote >= scoreLen) {
        currNote = 0; // Go back to the beginning when done
      }
      int duration = beats[currNote] * beat_ms; // get duration from array
      kChangeNoteDelay.set(duration - pauseTime); // Set the note duration
      aSin.setFreq(mtof(score[currNote])); // set the frequency
      currNote++;
```

```
    }
    kChangeNoteDelay.start();
  }
}

int updateAudio() {
  return aSin.next();
}

void loop() {
  audioHook(); // You must have this in your loop
}
```

## Discussion

Mozzi is an advanced audio synthesis library for Arduino and compatible boards. Although it supports 8-bit boards, it uses PWM, so the sound quality is limited. But on 32-bit boards such as SAMD-based boards and the Teensy, it excels. Mozzi is a powerful framework with many features, and it includes dozens of sample programs.

The sketch sets things up by including the Mozzi header files and defining the Mozzi control rate, which determines how often the update functions are called. The sketch then defines an enum of the notes used in this sketch. For convenience, it uses MIDI notes (see Recipe 9.5). Because MIDI notes can be represented as integers, this allows you to use them in an enum, which in turn makes it easy to use their symbolic names in an array of integers. You could accomplish the same thing with #defines, and in that case, you could use floating-point frequencies instead of MIDI numbers.

Similar to the sketch in Recipe 9.2, this sketch uses an array to represent the score, and an array to represent how many beats each note should be held. It then initializes some variables and objects that are used later in the sketch.

In setup(), the sketch initializes the Mozzi system and starts a timer. This is where things become noticeably different from a typical Arduino sketch. Instead of using delay() to wait until it's time to play the next note, all the changes take place in the updateControl() function. When the timer is up, the sketch toggles the pausing state, which is used to determine when to

take a brief pause between notes. If `pausing` is true, the sketch goes silent for 200 ms. Otherwise, the sketch advances to the next note in the `score` array, calculates the duration, and sets the frequency by converting the MIDI number to a frequency in hertz. It then sets the timer to the duration of the note (subtracting the pause time so as to keep closer to standard musical timing).

## See Also

The ArduTouch board and software is an Arduino-compatible music synthesizer kit that has similar capabilities to Mozzi. You can also run the ArduTouch software on an Arduino Uno.

The Teensy Audio library supports high-quality audio for Teensy boards. It includes polyphonic capabilities, as well as audio recording, synthesis, filtering, and more.

# Remotely Controlling External Devices

## 10.0 Introduction

The Arduino can interact with almost any device that uses some form of remote control, including TVs, audio equipment, cameras, garage doors, appliances, and toys. Most remote controls work by sending digital data from a transmitter to a receiver using infrared light (IR) or wireless radio technology. Different protocols (signal patterns) are used to translate key presses into a digital signal, and the recipes in this chapter show you how to use commonly found remote controls and protocols.

An IR remote works by turning an LED on and off in patterns to produce unique codes. The codes are typically 12 to 32 bits (pieces of data). Each key on the remote is associated with a specific code that is transmitted when the key is pressed. If the key is held down, the remote usually sends the same code repeatedly, although some remotes (e.g., NEC) send a special repeat code when a key is held down. For Philips RC-5 or RC-6 remotes, a bit in the code is toggled each time a key is pressed; the receiver uses this toggle bit to determine when a key is pressed a second time. You can read more about the technologies used in IR remote controls on the SB-Projects site.

The IR recipes here use a low-cost IR receiver module to detect the signal and provide a digital output that the Arduino can read. The digital output is then decoded by a library called IRremote, which was written by Ken Shirriff and can be installed using the Arduino IDE Library Manager (see Chapter 16).

The same library is used in the recipes in which Arduino sends commands to act like a remote control.

Remote controls using wireless radio technology are more difficult to emulate than IR controls. However, the button contacts on these controls can be activated by Arduino. The recipes using wireless remotes simulate button presses by closing the button contacts inside the remote control. With wireless remotes, you may need to take apart the remote control and connect wires from the contacts to Arduino to be able to use these devices. Components called *optocouplers* are used to provide electrical separation between Arduino and the remote control. This isolation prevents voltages from Arduino from harming the remote control, and vice versa.

Optocouplers (also called *optoisolators*) enable you to safely control another circuit that may be operating at different voltage levels from Arduino. As the "isolator" part of the name implies, optoisolators provide a way to keep things electrically separated. These devices contain an LED, which can be controlled by an Arduino digital pin. The light from the LED in the optocoupler shines onto a light-sensitive transistor. Turning on the LED causes the transistor to conduct, closing the circuit between its two connections—the equivalent of pressing a switch.

# 10.1 Responding to an Infrared Remote Control

### Problem

You want to respond to any key pressed on a TV or other remote control.

### Solution

Arduino responds to IR remote signals using a device called an *IR receiver module*. Common devices are the TSOP38238, TSOP4838, PNA4602, and TSOP2438. The first three have the same connections, so the circuit is the same; the TSOP2438 has the +5V and GND pins reversed. Check the datasheet for your device to ensure that you connect it correctly.

This recipe uses the IRremote library, which you can install with the Library Manager. Connect the IR receiver module according to your

datasheet. The Arduino wiring in is for the TSOP38238/TSOP4838/PNA4602 devices.

This sketch will toggle the built-in LED when any button on an infrared remote control is pressed:

```cpp
/*
  IR_remote_detector sketch
  An IR remote receiver is connected to pin 2.
  The built-in LED toggles each time a button on the remote is pressed.
 */

#include <IRremote.h>              // adds the library code to the sketch

const int irReceiverPin = 2;    // pin the receiver is connected to
const int ledPin = LED_BUILTIN;

IRrecv irrecv(irReceiverPin);   // create an IRrecv object
decode_results decodedSignal;   // stores results from IR detector

void setup()
{
  pinMode(ledPin, OUTPUT);
  irrecv.enableIRIn();          // Start the receiver object
}

bool lightState = LOW;          // keep track of whether the LED is on
unsigned long last = millis();  // remember when we last received an IR
message

void loop()
{
  if (irrecv.decode(&decodedSignal) == true) // this is true if a message has
                                             // been received
  {
    if (millis() - last > 250) {      // has it been 1/4 sec since last
message?
      if (lightState == LOW)
        lightState = HIGH;
      else
        lightState = LOW;
      lightState = lightState ;             // Yes: toggle the LED
      digitalWrite(ledPin, lightState);
    }
    last = millis();
    irrecv.resume();                        // watch out for another message
```
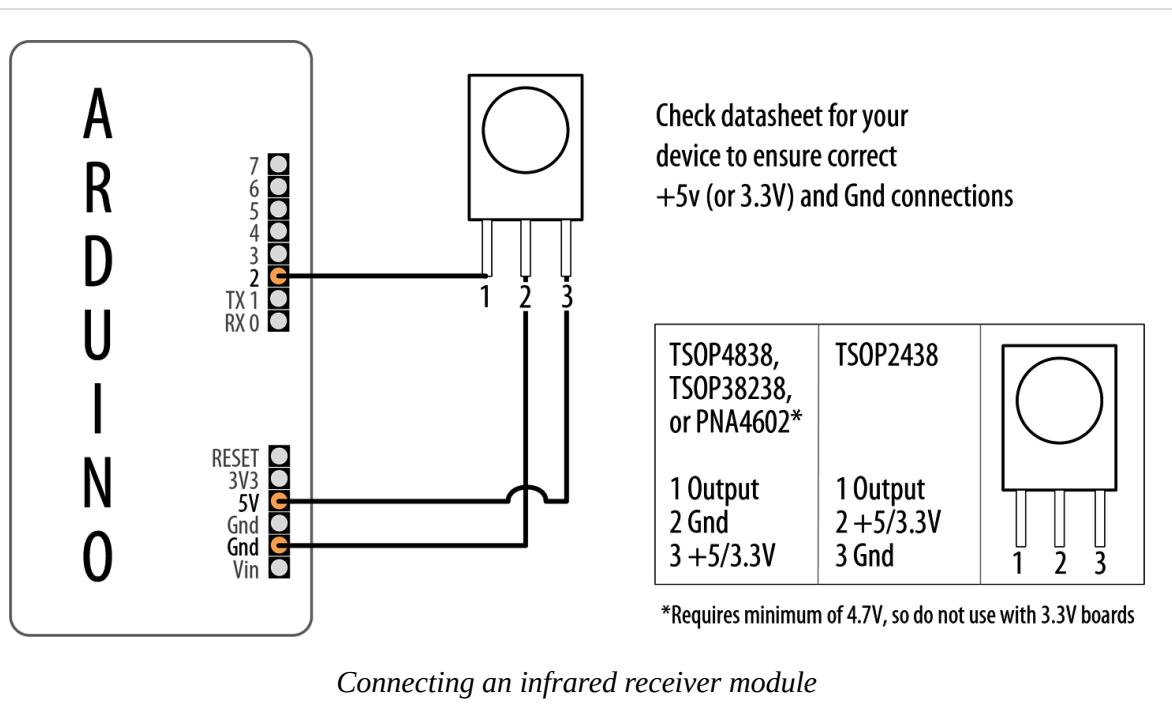
```
    }
  }
```



Check datasheet for your device to ensure correct +5v (or 3.3V) and Gnd connections

| TSOP4838, TSOP38238, or PNA4602* | TSOP2438 | |
|---|---|---|
| 1 Output<br>2 Gnd<br>3 +5/3.3V | 1 Output<br>2 +5/3.3V<br>3 Gnd | |

*Requires minimum of 4.7V, so do not use with 3.3V boards

*Connecting an infrared receiver module*

## WARNING

If you are using a 3.3V board that is not 5-volt tolerant, you should power the infrared receiver from 3.3V instead of 5V.

## Discussion

The IR receiver converts the IR signal to digital pulses. These are a sequence of ones and zeros that correspond to buttons on the remote. The IRremote library decodes these pulses and provides a numeric value for each key (the actual values that your sketch will receive are dependent on the specific remote control you use).

`#include <IRremote.h>` at the top of the sketch makes the library code available to your sketch, and the line `IRrecv irrecv(irReceiverPin);` creates an `IRrecv` object named `irrecv` to receive signals from an IR

receiver module connected to `irReceiverPin` (pin 2 in the sketch). Chapter 16 has more on using libraries.

You use the `irrecv` object to access the signal from the IR receiver. You can give it commands to look for and decode signals. The decoded responses provided by the library are stored in a variable named `decode_results`. The receiver object (`irrecv`) is started in `setup` with the line `irrecv.enableIRIn();`. The results are checked in `loop` by calling the function `irrecv.decode(&decodedSignal)`.

The `decode` function returns `true` if there is data, which will be placed in the `decodedSignal` variable. Recipe 2.11 explains how the ampersand symbol is used in function calls where parameters are modified so that information can be passed back.

If a remote message has been received, the code *toggles* the LED (flips its state) if it is more than one-quarter of a second since the last time it was toggled (otherwise, the LED will get turned on and off quickly by remotes that send codes more than once when you press the button, and may appear to be flashing randomly).

The `decodedSignal` variable will contain a value associated with a key. This value is ignored in this recipe (although it is used in the next recipe)—you can print the value by adding to the sketch the `Serial.println` line highlighted in the following code (you will also need to add `Serial.begin(9600);` in the `setup` function):

```
if (irrecv.decode(&decodedSignal) == true) // this is true if a message has
                                           // been received
{
 if (millis() - last > 250) {      // has it been 1/4 sec since last message
    Serial.println(decodedSignal.value);
```

The library needs to be told to continue monitoring for signals, and this is achieved with the line `irrecv.resume();`.

This sketch flashes an LED when any button on the remote control is pressed, but you can control other things—for example, you can use a

stepper motor to turn a knob on a lamp or stereo (for more on controlling physical devices, see Chapter 8).

## See Also

The Infrared category of Ken Shirriff's blog

# 10.2 Decoding Infrared Remote Control Signals

## Problem

You want to detect a specific key pressed on a TV or other remote control.

## Solution

This sketch uses remote control key presses to adjust the brightness of an LED. The code prompts for remote control keys 0 through 4 when the sketch starts. These codes are stored in Arduino memory (RAM), and the sketch then responds to these keys by setting the brightness of an LED to correspond with the button pressed, with 0 turning the LED off and 1 through 4 providing increased brightness:

```
/*
 * RemoteDecode sketch
 * Infrared remote control signals are decoded to control LED brightness.
 * The values for keys 0-4 are detected and stored when the sketch starts.
 * Key 0 turns the LED off; brightness increases in steps with keys 1-4.
 */

#include <IRremote.h>            // IR remote control library

const int irReceivePin = 2;      // pin connected to IR detector output
const int ledPin       = 9;      // LED is connected to a PWM pin

const int numberOfKeys = 5;      // 5 keys are learned (0 through 4)
long irKeyCodes[numberOfKeys];   // holds the codes for each key

IRrecv irrecv(irReceivePin);     // create the IR library
decode_results results;          // IR data goes here
```

```
void setup()
{
  Serial.begin(9600);
  while(!Serial); // Needed for Leonardo and ARM boards
  pinMode(irReceivePin, INPUT);
  pinMode(ledPin, OUTPUT);
  irrecv.enableIRIn();              // Start the IR receiver
  learnKeycodes();                  // learn remote control key  codes
  Serial.println("Press a remote key");
}

void loop()
{
  long key;
  int  brightness;

  if (irrecv.decode(&results))
  {
    // here if data is received
    irrecv.resume();
    key = convertCodeToKey(results.value);
    if(key >= 0)
    {
      Serial.print("Got key ");
      Serial.println(key);
      brightness = map(key, 0,numberOfKeys-1, 0, 255);
      analogWrite(ledPin, brightness);
    }
  }
}

/*
 * get remote control codes
 */
void learnKeycodes()
{
  while(irrecv.decode(&results))    // empty the buffer
    irrecv.resume();

  Serial.println("Ready to learn remote codes");
  long prevValue = -1;
  int i=0;
  while(i < numberOfKeys)
  {
    Serial.print("press remote key ");
    Serial.print(i);
    while(true)
```

```
    {
      if(irrecv.decode(&results))
      {
        if(results.value != -1 &&
           results.decode_type != UNKNOWN &&
           results.value != prevValue)
        {
          showReceivedData();
          Serial.println(results.value);
          irKeyCodes[i] = results.value;
          i = i + 1;
          prevValue = results.value;
          irrecv.resume(); // Receive the next value
          break;
        }
        irrecv.resume(); // Receive the next value
      }
    }
  }
  Serial.println("Learning complete");
}

/*
 * converts a remote protocol code to a logical key code
 * (or -1 if no digit received)
 */
int convertCodeToKey(long code)
{
  for(int i=0; i < numberOfKeys; i++)
  {
    if(code == irKeyCodes[i])
    {
      return i; // found the key so return it
    }
  }
  return -1;
}

/*
 * display the protocol type and value
 */
void showReceivedData()
{
  if (results.decode_type == UNKNOWN)
  {
    Serial.println("-Could not decode message");
  }
  else
```

```
  {
    if (results.decode_type == NEC) {
      Serial.print("- decoded NEC: ");
    }
    else if (results.decode_type == SONY) {
      Serial.print("- decoded SONY: ");
    }
    else if (results.decode_type == RC5) {
      Serial.print("- decoded RC5: ");
    }
    else if (results.decode_type == RC6) {
      Serial.print("- decoded RC6: ");
    }
    Serial.print("hex value = ");
    Serial.println( results.value, HEX);
  }
}
```

## Discussion

This solution is based on the IRremote library; see this chapter's introduction for details.

The sketch starts the remote control library with the following code:

```
irrecv.enableIRIn(); // Start the IR receiver
```

It then calls the `learnKeyCodes` function to prompt the user to press keys 0 through 4. The code for each key is stored in an array named `irKeyCodes`. After all the keys are detected and stored, the `loop` code waits for a key press and checks if this was one of the digits stored in the `irKeyCodes` array. If so, the value is used to control the brightness of an LED using `analogWrite`.

> **NOTE**
>
> See Recipe 5.7 for more on using the `map` function and `analogWrite` to control the brightness of an LED.

The library should be capable of working with most any IR remote control; it can discover and remember the timings and repeat the signal on command.

You can permanently store the key code values so that you don't need to learn them each time you start the sketch. Replace the declaration of `irKeyCodes` with the following lines to initialize the values for each key, and comment out the call to `learnKeycodes();`. Change the values to coincide with the ones for your remote (these will be displayed in the Serial Monitor when you press keys in the `learnKeyCodes` function):

```
long irKeyCodes[numberOfKeys] = {
  0x18E758A7,  //0 key
  0x18E708F7,  //1 key
  0x18E78877,  //2 key
  0x18E748B7,  //3 key
  0x18E7C837,  //4 key
  };
```

## See Also

Recipe 18.1 explains how you can store learned data in EEPROM (nonvolatile memory).

# 10.3 Imitating Remote Control Signals

## Problem

You want to use Arduino to control a TV or other remotely controlled appliance by emulating the infrared signal. This is the inverse of Recipe 10.2—it sends commands instead of receiving them.

## Solution

This sketch uses the remote control codes from Recipe 10.2 to control a device (your remote codes are likely to differ so make sure you run that recipe's Solution code and use the values unique to your remote). Five

buttons select and send one of five codes. Connect an infrared LED to send the signal as shown in Figure 10-2:

```
/*
 * irSend sketch
 * this code needs an IR LED connected to pin 3
 * and 5 switches connected to pins 6 - 10
 */

#include <IRremote.h>        // IR remote control library

const int numberOfKeys = 5;
const int firstKey = 6;    // the first pin of the 5 sequential pins connected
                           // to buttons
bool buttonState[numberOfKeys];
bool lastButtonState[numberOfKeys];
long irKeyCodes[numberOfKeys] = {
  0x18E758A7,  //0 key
  0x18E708F7,  //1 key
  0x18E78877,  //2 key
  0x18E748B7,  //3 key
  0x18E7C837,  //4 key
};

IRsend irsend;

void setup()
{
  for (int i = 0; i < numberOfKeys; i++) {
    buttonState[i] = true;
    lastButtonState[i] = true;
    int physicalPin=i + firstKey;
    pinMode(physicalPin, INPUT_PULLUP);  // turn on pull-ups
  }
  Serial.begin(9600);
}

void loop() {
  for (int keyNumber=0; keyNumber<numberOfKeys; keyNumber++)
  {
    int physicalPinToRead = keyNumber + firstKey;
    buttonState[keyNumber] = digitalRead(physicalPinToRead);
    if (buttonState[keyNumber] != lastButtonState[keyNumber])
    {
      if (buttonState[keyNumber] == LOW)
      {
```
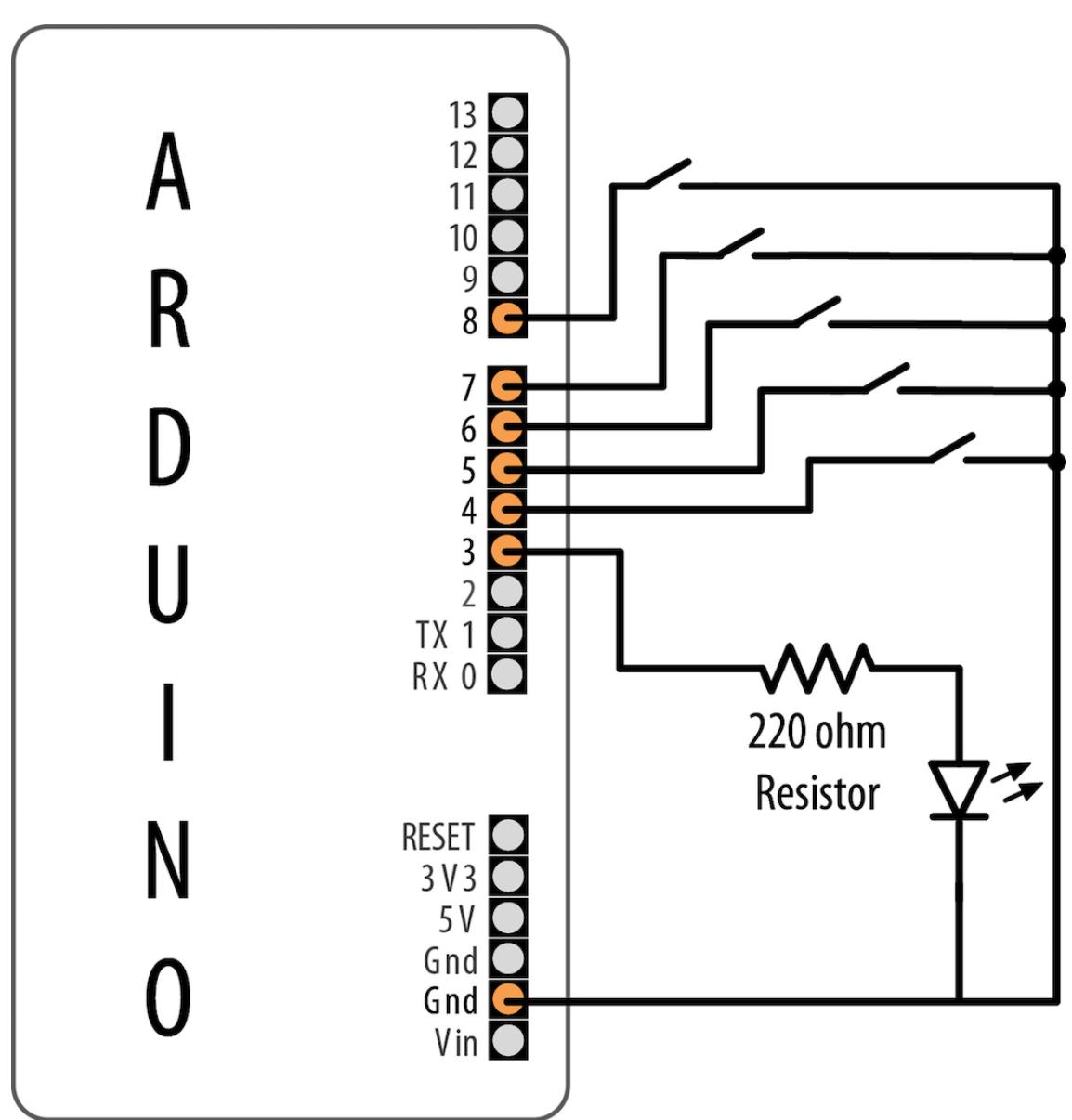
```
        irsend.sendSony(irKeyCodes[keyNumber], 32);
        Serial.println("Sending");
      }
      lastButtonState[keyNumber] = buttonState[keyNumber];
    }
  }
}
```



*Buttons and LED for IR sender*

## Discussion

Arduino controls the device by flashing an IR LED to duplicate the signal that would be sent from your remote control. This requires an IR LED. The specifications are not critical; see Appendix A for suitable components.

The IR library handles the translation from numeric code to IR LED flashes. You need to create an object for sending IR messages. The following line creates an `IRsend` object that will control the LED on pin 3 (you are not able to specify which pin to use; this is hardcoded within the library):

```
IRsend irsend;
```

**NOTE**

Depending on which board you are using, the IRremote library may require a different pin for the infrared LED. For example, Teensy 3.x boards use pin 5. See the README for the IRremote library. If your board uses a pin that overlaps with the range of pins used for the buttons (pins 6–10 in the Solution sketch), you may need to change the range of pins.

The code uses an array (see Recipe 2.4) called `irKeyCodes` to hold the range of values that can be sent. It monitors five switches to see which one has been pressed and sends the relevant code in the following line:

```
irsend.sendSony(irKeyCodes[keyNumber], 32);
```

The `irSend` object has different functions for various popular infrared code formats, so check the library documentation if you are using one of the

other remote control formats. You can use Recipe 10.2 if you want to display the format used in your remote control.

The sketch passes the code from the array, and the number after it tells the function how many bits long that number is. The 0x at the beginning of the numbers in the definition of `irKeyCodes` at the top of the sketch means the codes are written in hex (see Chapter 2 for details about hex numbers). Each character in hex represents a 4-bit value. The codes here use eight characters, so they are 32 bits long.

The LED is connected with a current-limiting resistor (see the introduction to Chapter 7).

If you need to increase the sending range, you can use multiple LEDs or select one with greater output.

## See Also

Chapter 7 provides more information on controlling LEDs.

Mitch Altman's TV-B-Gone is a clever remote control application.

# 10.4 Controlling a Digital Camera

### Problem

You want Arduino to control a digital camera to take pictures under program control. You may want to do time lapse photography or take pictures triggered by an event detected by the Arduino.

### Solution

There are a few ways to do this. If your camera has an infrared remote, use Recipe 10.2 to learn the relevant remote codes and Recipe 10.3 to get Arduino to send those codes to the camera.

If your camera doesn't have an infrared remote but does have a socket for a wired remote, you can use this recipe to control the camera.

You connect the Arduino to a suitable cable for your camera using optocouplers, as shown in Figure 10-3.

This sketch takes 10 pictures, one every 20 seconds:

```
/*
 * camera sketch
 * takes 20 pictures with a digital camera
 * using pin 4 to trigger focus
 * pin 3 to trigger the shutter
 */

int focus = 4;          // optocoupler attached to focus
int shutter = 3;        // optocoupler attached to shutter
long exposure = 250;    // exposure time in milliseconds
long interval = 10000;  // time between shots, in milliseconds

void setup()
{
  pinMode(focus, OUTPUT);
  pinMode(shutter, OUTPUT);
  for (int i=0; i<20; i++)  // camera will take 20 pictures
  {
    takePicture(exposure);      // takes picture
    delay(interval);            // wait to take the next picture
  }
}

void loop()
{
                  // once it's taken 20 pictures it is done,
                  // so loop is empty
                  // but loop still needs to be here or the
                  // sketch won't compile
}

void takePicture(long exposureTime)
```
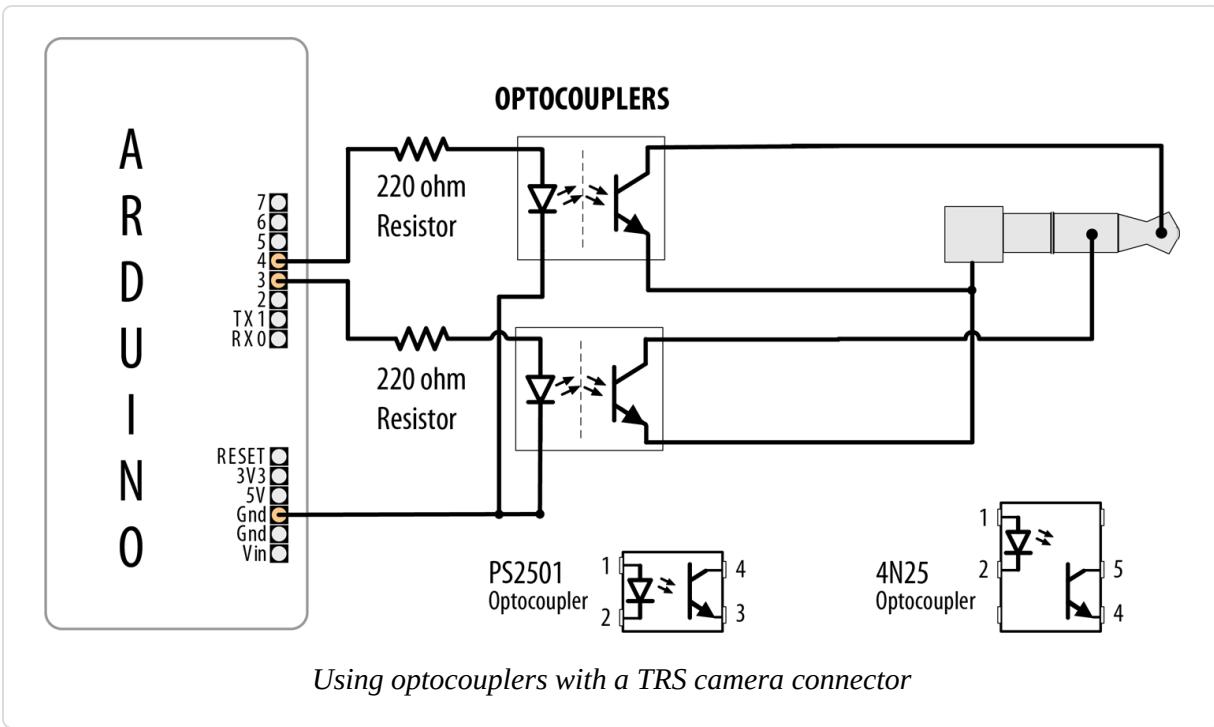
```
{
    int wakeup = 10;          // camera will take some time to wake up and focus
                              // adjust this to suit your camera

    digitalWrite(focus, HIGH);     // wake the camera and focus
    delay(wakeup);                 // wait for it to wake up and focus
    digitalWrite(shutter, HIGH);   // open the shutter
    delay(exposureTime);           // wait for the exposure time
    digitalWrite(shutter, LOW);    // release shutter
    digitalWrite(focus, LOW);      // release the focus
}
```



*Using optocouplers with a TRS camera connector*

## Discussion

It's not advisable to connect Arduino pins directly to a camera—the voltages may not be compatible and you risk damaging your Arduino or your camera. Optocouplers are used to isolate Arduino from your camera; see the introduction of this chapter for more about these devices.

You will need to check the user manual for your camera to identify the correct TRS connector to use.

You may need to change the order of the pins turning on and off in the `takePicture` function to get the behavior you want. For a Canon camera to do bulb exposures, you need to turn on the focus, then open the shutter without releasing the focus, then release the shutter, and then release the focus (as in the sketch). To take a picture and have the camera calculate the exposure, press the focus button, release it, and then press the shutter.

## See Also

If you want to control aspects of a camera's operation, have a look at the Canon Hack Development Kit.

Also see *The Canon Camera Hackers Manual: Teach Your Camera New Tricks* by Berthold Daum (Rocky Nook).

You can control a GoPro camera using an Arduino with WiFi using this library.

It is also possible to control video cameras in a similar fashion using LANC. This library supports this feature.

There is also an Arduino shield for controlling high-end Black Magic Design video equipment, including cameras using their open protocol.

# 10.5 Controlling AC Devices by Hacking a Remote-Controlled Switch

### Problem

You want to safely switch AC line currents on and off to control lights and appliances using a remote-controlled switch.

### Solution

Arduino can trigger the buttons of a remote-controlled switch using an optocoupler. This may be necessary for remotes that use wireless instead of infrared technology. This technique can be used for almost any remote control. Hacking a remote is particularly useful to isolate potentially
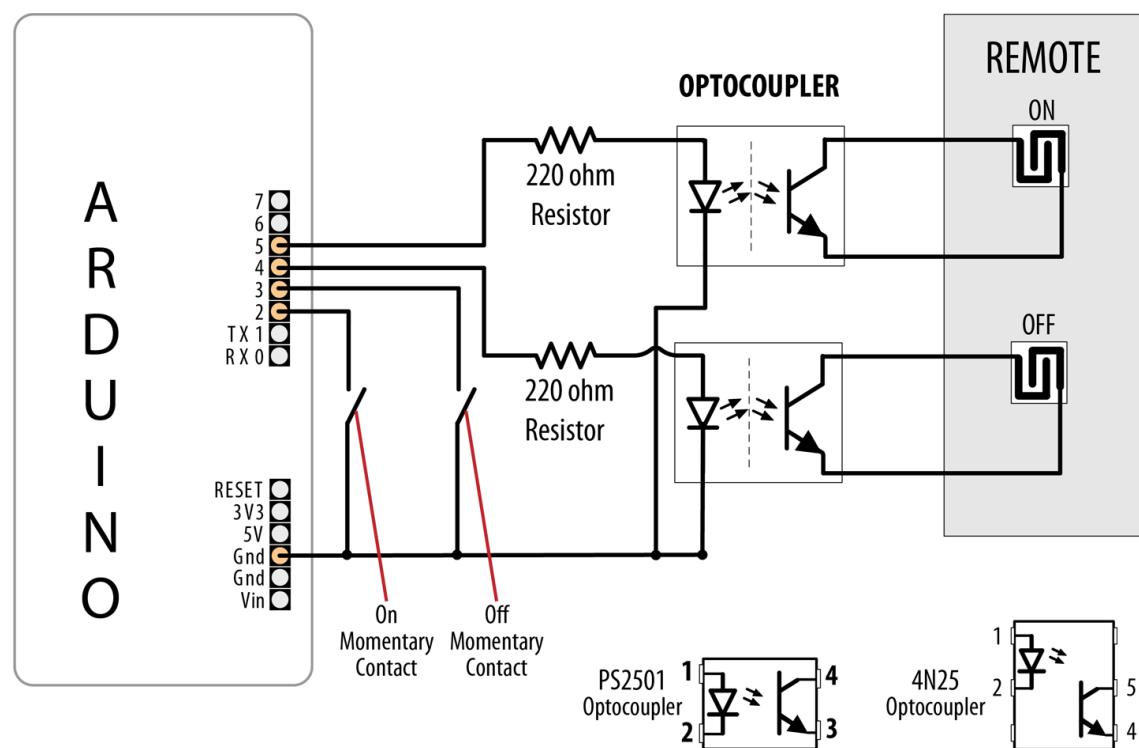
dangerous AC voltages from you and Arduino because only the battery-operated controller is modified.

> **WARNING**
>
> Opening the remote control will void the warranty and can potentially damage the device. The infrared recipes in this chapter are preferable because they avoid modifying the remote control.
>
> If you want to use this recipe to control a switch, but you want to keep using the remote control, consider purchasing a spare remote control for hacking. Most manufacturers will be happy to sell you a spare (but make sure you choose the right frequency for the variant of appliance, light, or outlet you want to control). After you receive the spare, you may need to configure the channel that it uses.

Open the remote control and connect the optocoupler so that the photo-emitter (pins 1 and 2 in Figure 10-4) is connected to Arduino and the photo transistor (pins 3 and 4) is connected across the remote control contacts.



*Optocouplers connected to remote control contacts*

This sketch uses momentary contact switches (push and release) to activate the remote's on and off buttons:

```
/*
 * OptoRemote sketch
 * Switches connected to pins 2 and 3 turn a remote device on and off
 * using optocouplers.
 * The outputs are pulsed for at least half a second when a switch is pressed
 */
const int onSwitchPin  = 2;          // input pin for the On switch
const int offSwitchPin = 3;          // input pin for the Off switch
const int remoteOnPin  = 4;          // output pin to turn the remote on
const int remoteOffPin = 5;          // output pin to turn the remote off
const int PUSHED       = LOW;        // value when button is pressed

void setup() {
  pinMode(remoteOnPin, OUTPUT);
  pinMode(remoteOffPin, OUTPUT);
  pinMode(onSwitchPin, INPUT_PULLUP);   // turn on internal pull-ups
  pinMode(offSwitchPin, INPUT_PULLUP);
}

void loop(){
  int val = digitalRead(onSwitchPin);   // read input value
  // if the switch is pushed then switch on if not already on
  if(val == PUSHED)
  {
    pulseRemote(remoteOnPin);
  }

  val = digitalRead(offSwitchPin);   // read input value
  // if the switch is pushed then switch off if not already off
  if(val == PUSHED)
  {
    pulseRemote(remoteOffPin);
  }
}

// turn the optocoupler on for half a second to blip the remote control button
void pulseRemote(int pin)
{
  digitalWrite(pin, HIGH);     // turn the optocoupler on
  delay(500);                  // wait half a second
  digitalWrite(pin, LOW);      // turn the optocoupler off
}
```

## Discussion

The switches in most remote controls consist of interleaved bare copper traces with a conductive button that closes a connection across the traces when pressed. Less common are controls that contain conventional push switches; these are easier to use as the legs of the switches provide a convenient connection point.

> **TIP**
>
> Though the original remote button and the optocoupler can be used together—the switching action is performed if either method is activated (pressing the button or turning on the optocoupler)—the wires tethered to Arduino can make this inconvenient.

The transistor in the optocoupler will only allow electricity to flow in one direction, so if it doesn't work the first time, try switching the transistor-side connections over and see if that fixes it.

Some remotes have one side of all of the switches connected together (usually to the ground of that circuit). You can trace the connections on the board to check for this or use a multimeter to see what the resistance is between the traces on different switches. If traces have common connections, it is only necessary to connect one wire to each common group. Fewer traces are easier because connecting the wires can be fiddly if the remote is small.

The remote control may have multiple contacts corresponding to each button. You may need more than one optocoupler for each button position to connect the contacts. Figure 10-5 shows three optocouplers controlled from a single Arduino pin.
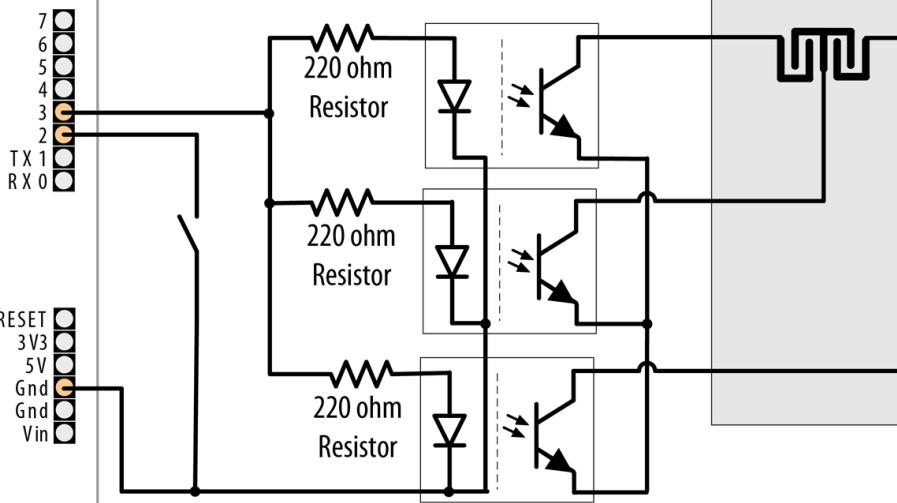
Another approach to controlling AC line currents is to use an isolated relay that can be switched on and off directly from Arduino pins, such as the Digital Loggers IoT Power Relay, available from Adafruit (part 2935) or SparkFun (part 14236). The IoT Power Relay is a great alternative to the now-discontinued PowerSwitch Tail.

*Multiple optocouplers connected to a single remote control button*

## See Also

Optocouplers are used in Recipe 10.4, so check out that recipe for an example of how they are used in a circuit.

# Using Displays

## 11.0 Introduction

Liquid crystal displays (LCDs) and LED displays offer a convenient and inexpensive way to provide a user interface for a project. This chapter explains how to connect and use common text and graphical LCD/LED panels with Arduino. By far the most popular LCD is the text panel based on the Hitachi HD44780 chip. This displays two or four lines of text, with 16 or 20 characters per line (32- and 40-character versions are available, but usually at higher prices). A library for driving text LCD displays is provided with Arduino, and you can print text on your LCD as easily as on the Serial Monitor (see Chapter 4), because LCD and serial share the same underlying `print` functions.

LCDs can do more than display simple text: words can be scrolled or highlighted and you can display a selection of special symbols and non-English characters.

You can create your own symbols and block graphics with a text LCD, but if you want fine graphical detail, you need a graphical display. Graphical LCD (GLCD) and graphical LED displays are available at a small price premium over text displays.

Graphical displays can have more wires connecting to Arduino than most other recipes in this book. Incorrect connections are the major cause of problems with graphical displays, so take your time wiring things up and triple-check that things are connected correctly. An inexpensive multimeter capable of measuring voltage and resistance is a big help for verifying that your wiring is correct. It can save you a lot of head scratching if nothing is being displayed. You don't need anything fancy, as even the cheapest multimeter will help you verify that the correct pins are connected and that the voltages are correct.

# 11.1 Connecting and Using a Text LCD Display

## Problem

You have a text LCD based on the industry-standard HD44780 or a compatible controller chip, and you want to display text and numeric values.

## Solution

The Arduino software includes the LiquidCrystal library for driving LCD displays based on the HD44780 chip such as the SparkFun LCD-00255 or Adafruit part number 181.

---

**TIP**
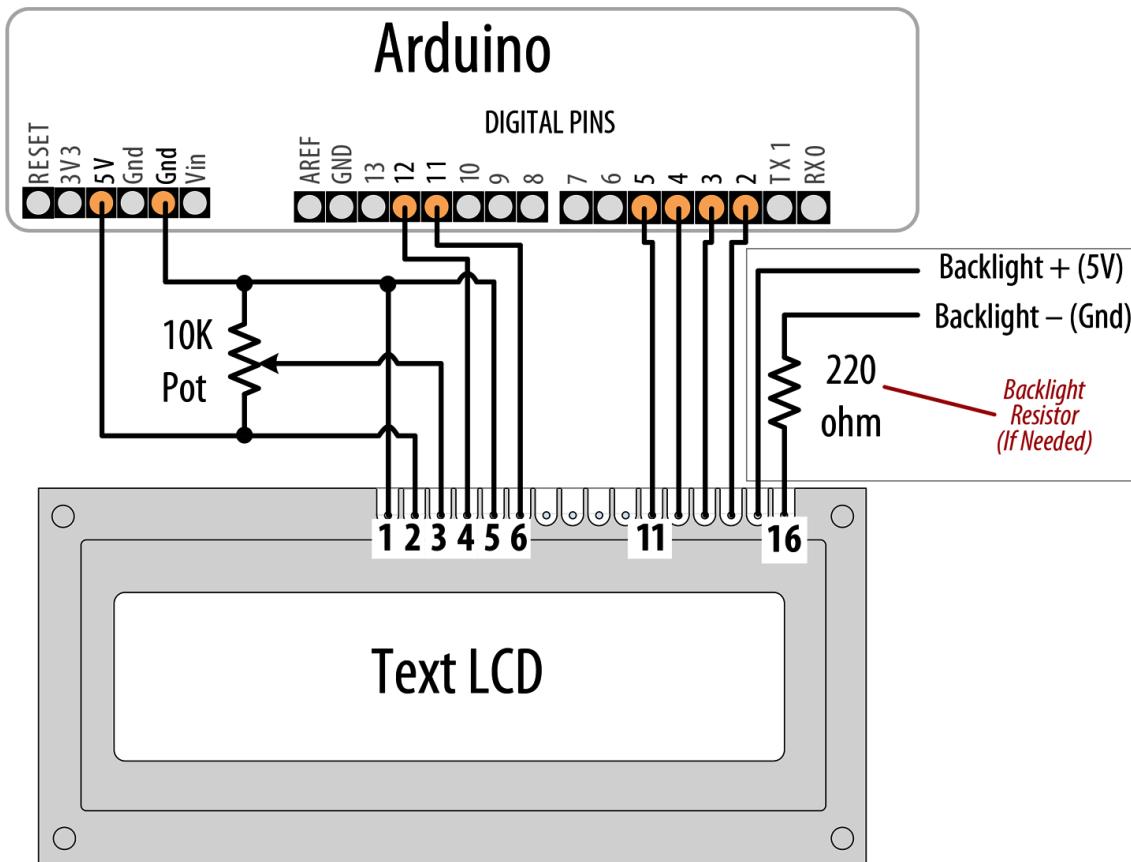
Most text LCDs supplied for use with Arduino will be compatible with the Hitachi HD44780 controller. If you are not sure about your controller, check the datasheet to see if it is a 44780 or compatible. If your LCD has a backpack-style controller board on it, you may be able to interface with it over a serial protocol with far fewer wires. See Recipe 4.11 for more details.

---

To get the display working, you need to wire the power, data, and control pins. Connect the data and status lines to digital output pins, and wire up a contrast potentiometer and connect the power lines. If your display has a backlight, this needs connecting, usually through a resistor.

Figure 11-1 shows the most common LCD connections. It's important to check the datasheet for your LCD to verify the pin connections. Table 11-1 shows the most common pin connections, but if your LCD uses different pins, make sure it is compatible with the Hitachi HD44780—this recipe will only work on LCD displays that are compatible with that chip. The LCD will have 16 pins (or 14 pins if there is no backlight)—make sure you identify pin 1 on your panel; it may be in a different position than shown in the figure.

*Connections for a text LCD*

*LCD pin connections*

| LCD pin | Function | Arduino pin |
|---|---|---|
| 1 | GND or 0V or Vss | GND |
| 2 | +5V or Vdd | 5V |
| 3 | Vo or contrast | |
| 4 | RS | 12 |
| 5 | R/W | GND |
| 6 | E | 11 |
| 7 | D0 | |
| 8 | D1 | |
| 9 | D2 | |
| 10 | D3 | |
| 11 | D4 | 5 |
| 12 | D5 | 4 |
| 13 | D6 | 3 |
| 14 | D7 | 2 |
| 15 | A or anode | |
| 16 | K or cathode | |

You will need to connect a 10K potentiometer to provide the contrast voltage to LCD pin 3. Without the correct voltage on this pin, you may not see anything displayed. In Figure 11-1, one side of the pot connects to GND (ground), the other side connects to Arduino +5V, and the center of the pot goes to LCD pin 3. The LCD is powered by connecting GND and +5V from Arduino to LCD pins 1 and 2.

Many LCD panels have an internal lamp called a *backlight* to illuminate the display. Your datasheet should indicate whether there is a backlight and if it

requires an external resistor—many do need this to prevent burning out the backlight LED assembly (if you are not sure, you can be safe by using a 220 ohm resistor). The backlight is polarized, so make sure pin 15 is connected to +5V and pin 16 to GND. (The resistor is shown connected between pin 16 and GND, but it can also be connected between pin 15 and +5V.)

Double-check the wiring before you apply power, as you can damage the LCD if you connect the power pins incorrectly. To run the HelloWorld sketch provided with Arduino, click the IDE Files menu item and navigate to Examples → Library → LiquidCrystal → HelloWorld.

The following code is modified slightly from the example. Change `numRows` and `numCols` to match the rows and columns in your LCD:

```
/*
 * LiquidCrystal Library - Hello World
 *
 * Demonstrates the use of a 16 × 2 LCD display.
 * https://www.arduino.cc/en/Tutorial/HelloWorld
 */

#include <LiquidCrystal.h> // include the library code

//constants for the number of rows and columns in the LCD
const int numRows = 2;
const int numCols = 16;

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup()
{
  lcd.begin(numCols, numRows);
  lcd.print("Hello, World!");  // Print a message to the LCD.
}

void loop()
{
  // set the cursor to column 0, line 1
  // (row numbering starts with 0, so line 1 is the second row):
  lcd.setCursor(0, 1);
  // print the number of seconds since the sketch started:
```

```
    lcd.print(millis()/1000);
  }
```

Run the sketch; you should see "hello world" displayed on the first line of your LCD. The second line will display a number that increases by one every second.

## Discussion

If you don't see any text and you have double-checked that all wires are connected correctly, you may need to adjust the contrast pot. With the pot shaft rotated to one side (usually the side connected to GND), you will have maximum contrast and should see blocks appear in all the character positions. With the pot rotated to the other extreme, you probably won't see anything at all. The correct setting will depend on many factors, including viewing angle and temperature—turn the pot until you get the best-looking display.

If you can't see blocks of pixels appear at any setting of the pot, check that the LCD is being driven on the correct pins.

Once you can see text on the screen, using the LCD in a sketch is easy. You use similar print commands to those for serial printing, covered in Chapter 4. The next recipe reviews the print commands and explains how to control text position.

## See Also

See the LiquidCrystal reference.

See Chapter 4 for details on `print` commands.

The datasheet for the Hitachi HD44780 LCD controller is the definitive reference for detailed, low-level functionality. The Arduino library insulates you from most of the complexity, but if you want to read about the raw capabilities of the chip, you can download the datasheet.

The LCD page in the Arduino Playground contains software and hardware tips and links.

# 11.2 Formatting Text

## Problem

You want to control the position of text displayed on the LCD screen; for example, to display values in specific positions.

## Solution

This sketch displays a countdown from 9 to 0. It then displays a sequence of digits in three columns of four characters. Change `numRows` and `numCols` to match the rows and columns in your LCD:

```
/*
 * LiquidCrystal Library - FormatText
 */

#include <LiquidCrystal.h> // include the library code:

//constants for the number of rows and columns in the LCD
const int numRows = 2;
const int numCols = 16;

int count;

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup()
{
  lcd.begin(numCols, numRows);
  lcd.print("Starting in ");  // this string is 12 characters long
  for(int i=9; i > 0; i--)    // count down from 9
  {
    // the top line is row 0
    lcd.setCursor(12,0); // move the cursor to the end of the string
    lcd.print(i);
    delay(1000);
  }
}

void loop()
{
  int columnWidth = 4;          //spacing for the columns
```

```
int displayColumns = 3;        //how many columns of numbers

lcd.clear();
for( int col=0; col < displayColumns; col++)
{
  lcd.setCursor(col * columnWidth, 0);
  count = count+ 1;
  lcd.print(count);
}
delay(1000);
}
```

## Discussion

The `lcd.print` functions are similar to `Serial.print`. In addition, the LCD library has commands that control the cursor location (the row and column where text will be printed).

The `lcd.print` statement displays each new character after the previous one. Text printed beyond the end of a line may not be displayed or may be displayed on another line. The `lcd.setCursor()` command enables you to specify where the next `lcd.print` will start. You specify the column and row position (the top-left corner is 0,0). Once the cursor is positioned, the next `lcd.print` will start from that point, and it will overwrite existing text. The sketch in this recipe's Solution uses this to print numbers in fixed locations.

For example, in `setup`:

```
lcd.setCursor(12,0); // move the cursor to the 13th position
lcd.print(i);
```

`lcd.setCursor(12,0)` ensures that each number is printed in the same position, the thirteenth column, first row, producing the digit shown at a fixed position, rather than each number being displayed after the previous number.

The following lines use `setCursor` to space out the start of each column to provide `columnwidth` spaces from the start of the previous column:

```
lcd.setCursor(col * columnWidth, 0);
count = count+ 1;
lcd.print(count);
```

`lcd.clear` clears the screen and moves the cursor back to the top-left corner:

```
lcd.clear();
```

Here is a variation on `loop` that displays numbers using all the rows of your LCD. Replace your `loop` code with the following (make sure you set `numRows` and `numCols` at the top of the sketch to match the rows and columns in your LCD):

```
void loop()
{
int columnWidth = 4;
int displayColumns = 3;

  lcd.clear();
  for(int row=0; row < numRows; row++)
  {
    for( int col=0; col < displayColumns; col++)
    {
      lcd.setCursor(col * columnWidth, row);
      count = count+ 1;
      lcd.print(count);
    }
  }
  delay(1000);
}
```

The first `for` loop steps through the available rows, and the second `for` loop steps through the columns.

To adjust how many numbers are displayed in a row to fit the LCD, calculate the `displayColumns` value rather than setting it. Change:

```
int displayColumns = 3;
```

to:

```
int displayColumns = numCols / columnWidth;
```

### See Also

The LiquidCrystal library tutorial

# 11.3 Turning the Cursor and Display On or Off

### Problem

You want to blink the cursor and turn the display on or off. You may also want to draw attention to a specific area of the display.

### Solution

This sketch shows how you can cause the cursor (a flashing block at the position where the next character will be displayed) to blink. It also illustrates how to turn the display on and off; for example, to draw attention by blinking the entire display:

```
/*
 * blink
 */

// include the library code:
#include <LiquidCrystal.h>
```

```
// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

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

void loop()
{
  lcd.setCursor(0, 1);

  lcd.print("cursor blink");
  lcd.blink();
  delay(2000);

  lcd.noBlink();
  lcd.print(" noBlink");
  delay(2000);

  lcd.clear();

  lcd.print("Display off ...");
  delay(1000);
  lcd.noDisplay();
  delay(2000);

  lcd.display();  // turn the display back on

  lcd.setCursor(0, 0);
  lcd.print(" display flash !");
  displayBlink(2, 250);  // blink twice
  displayBlink(2, 500);  // and again for twice as long

  lcd.clear();
}

void displayBlink(int blinks, int duration)
{
  while(blinks--)
  {
    lcd.noDisplay();
    delay(duration);
    lcd.display();
    delay(duration);
```

```
      }
  }
```

## Discussion

The sketch calls the `blink` and `noBlink` functions to toggle cursor blinking on and off.

The code to blink the entire display is in a function named `displayBlink` that makes the display flash a specified number of times. The function uses `lcd.display()` and `lcd.noDisplay()` to turn the display text on and off (without clearing it from the screen's internal memory).

# 11.4 Scrolling Text

### Problem

You want to scroll text; for example, to create a marquee that displays more characters than can fit on one line of the LCD display.

## Solution

This sketch demonstrates both `lcd.ScrollDisplayLeft` and `lcd.ScrollDisplayRight`.

It scrolls a line of text to the left when tilted and to the right when not tilted. Connect one side of a tilt sensor to pin 7 and the other pin to GND (see Recipe 6.2 if you are not familiar with tilt sensors):

```
/*
 * Scroll
 * this sketch scrolls text left when tilted
 * text scrolls right when not tilted.
 */

#include <LiquidCrystal.h>

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
const int numRows = 2;
```

```
const int numCols = 16;

const int tiltPin = 7; // pin connected to tilt sensor

const char textString[] = "tilt to scroll";
const int textLen = sizeof(textString) - 1; // the number of characters
bool isTilted = false;

void setup()
{
  // set up the LCD's number of columns and rows:
  lcd.begin(numCols, numRows);
  pinMode(tiltPin, INPUT_PULLUP);
  lcd.print(textString);
}

void loop()
{
  if(digitalRead(tiltPin) == LOW && isTilted == false)
  {
    // here if tilted left so scroll text left
    isTilted = true;
    for (int position = 0; position  < textLen; position++)
    {
      lcd.scrollDisplayLeft();
      delay(150);
    }
  }
  if(digitalRead(tiltPin) == HIGH && isTilted == true)
  {
    // here if previously tilted but now flat, so scroll text right
    isTilted = false;
    for (int position = 0; position  < textLen; position++)
    {
      lcd.scrollDisplayRight();
      delay(150);
    }
  }
}
```

## Discussion

The first half of the `loop` code handles the change from not tilted to tilted. The code checks to see if the tilt switch is closed (`LOW`) or open (`HIGH`). If it's `LOW` and the current state (stored in the `isTilted` variable) is not tilted,

the text is scrolled left. The delay in the `for` loop controls the speed of the scroll; adjust the delay if the text moves too fast or too slow.

The second half of the code uses similar logic to handle the change from tilted to not tilted.

A scrolling capability is particularly useful when you need to display more text than can fit on an LCD line.

This sketch has a `marquee` function that will scroll text up to 32 characters in length:

```
/*
 * Marquee
 * this sketch can scroll a very long line of text
 */

#include <LiquidCrystal.h>

LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
const int numRows = 2;
const int numCols = 16;

void setup()
{
  // set up the LCD's number of columns and rows:
  lcd.begin(numCols, numRows);
  marquee("This is a very long string of text that will scroll");
}

void loop()
{
}

// this function uses scrolling to display a message up to 32 bytes long
void marquee(char *text)
{
  lcd.print(text);
  delay(1000);
  for (int position = 0; position < strlen(text)-numCols; position++)
  {
    lcd.scrollDisplayLeft();
    delay(300);
  }
}
```

The sketch uses the `lcd.scrollDisplayLeft` function to scroll the display when the text is longer than the width of the screen.

The LCD chip has internal memory that stores the text. This memory is limited (32 bytes on most four-line displays). If you try to use longer messages, they may start to wrap over themselves. If you want to scroll longer messages (e.g., a tweet), or control scrolling more precisely, you need a different technique. The following function stores the text in RAM on Arduino and sends sections to the screen to create the scrolling effect. These messages can be any length that can fit into Arduino memory:

```
// this version of marquee uses manual scrolling for very long messages
void marquee(char *text)
{
  int length = strlen(text); // the number of characters in the text
  if(length < numCols)
    lcd.print(text);
  else
  {
    int pos;
    for(pos = 0; pos < numCols; pos++)
      lcd.print(text[pos]);
    delay(1000); // allow time to read the first line before scrolling
    pos=1;
    while(pos <= length - numCols)
    {
      lcd.setCursor(0,0);
      for(int i=0; i < numCols; i++)
        lcd.print(text[pos+i]);
      delay(300);
      pos = pos + 1;
    }
  }
}
```

# 11.5 Displaying Special Symbols

## Problem

You want to display special symbols: ° (degrees), ¢, ÷, π (pi), or any other symbol stored in the LCD character memory.

## Solution

Identify the character code you want to display by locating the symbol in the character pattern table in the LCD datasheet. This sketch prints some common symbols in `setup`. It then shows all displayable symbols in `loop`:

```
/*
 * LiquidCrystal Library - Special Chars
 */

#include <LiquidCrystal.h>

//set constants for number of rows and columns to match your LCD
const int numRows = 2;
const int numCols = 16;

// defines for some useful symbols
const byte degreeSymbol = B11011111;
const byte piSymbol     = B11110111;
const byte centsSymbol  = B11101100;
const byte sqrtSymbol   = B11101000;
const byte omegaSymbol  = B11110100;  // the symbol used for ohms

byte charCode = 32; // the first printable ascii character
int col;
int row;

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup()
{
  lcd.begin(numRows, numCols);

  showSymbol(degreeSymbol, "degrees");
  showSymbol(piSymbol,     "pi");
  showSymbol(centsSymbol,  "cents");
  showSymbol(sqrtSymbol,   "sqrt");
  showSymbol(omegaSymbol,  "ohms");
  lcd.clear();

}

void loop()
{
  lcd.write(charCode);
```

```
    calculatePosition();
    if(charCode == 255)
    {
      // finished all characters so wait another few seconds and start over
      delay(2000);
      lcd.clear();
      row = col = 0;
      charCode = 32;
    }
    charCode = charCode + 1;
  }

  void calculatePosition()
  {
    col = col + 1;
    if( col == numCols)
    {
      col = 0;
      row = row + 1;
      if( row == numRows)
      {
        row = 0;
        delay(2000); // pause
        lcd.clear();
      }
      lcd.setCursor(col, row);
    }
  }

  // function to display a symbol and its description
  void showSymbol(byte symbol, char *description)
  {
    lcd.clear();
    lcd.write(symbol);
    lcd.print(' '); // add a space before the description
    lcd.print(description);
    delay(3000);
  }
```

## Discussion

The datasheet for the LCD controller chip contains a table showing the available character patterns.

To use the table, locate the symbol you want to display. The code for that character is determined by combining the binary values for the column and

row for the desired symbol (see Figure 11-2).



Character Pattern Codes from Data Sheet

*Using the datasheet to derive character codes*

For example, the degree symbol (°) is the third-from-last entry at the bottom row of the table shown in Figure 11-2. Its column indicates the upper four bits are 1101 and its row indicates the lower four bits are 1111. Combining these gives the code for this symbol: B11011111. You can use this binary value or convert this to its hex value (0xDF) or decimal value (223). Note that Figure 11-2 shows only four of the 16 actual rows in the datasheet.

The LCD screen can also show any of the ASCII characters listed in the datasheet by passing the ASCII value to `lcd.write`.

The sketch uses a function named `showSymbol` to print the symbol and its description:

```
void showSymbol(byte symbol, char *description)
```

(See Recipe 2.6 if you need a refresher on using character strings and passing them to functions.)

## See Also

The datasheet for the Hitachi HD44780 display

# 11.6 Creating Custom Characters

## Problem

You want to define and display characters or symbols (glyphs) that you have created. The symbols you want are not predefined in the LCD character memory.

## Solution

Uploading the following code will create an animation of a face, switching between smiling and frowning:

```
/*
 * custom_char sketch
 * creates an animated face using custom characters
 */

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

byte happy[8] =
{
  B00000,
  B10001,
  B00000,
  B00000,
  B10001,
  B01110,
  B00000,
  B00000
};

byte saddy[8] =
{
  B00000,
  B10001,
  B00000,
  B00000,
  B01110,
  B10001,
  B00000,
  B00000
};
```
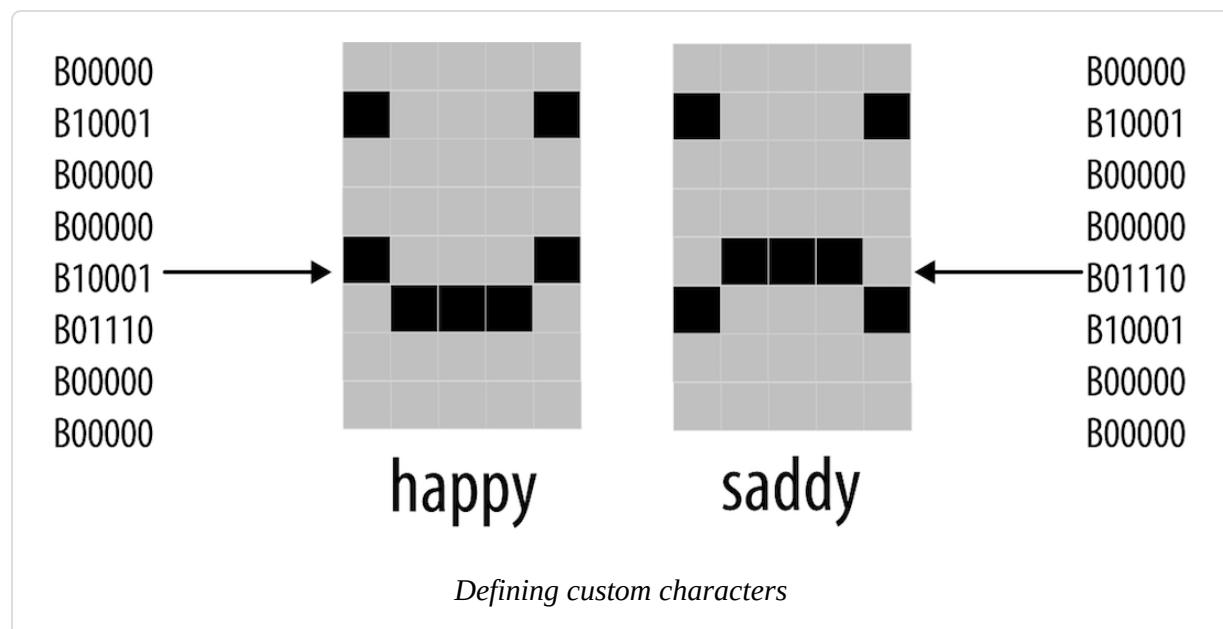
```
void setup() {
  lcd.createChar(0, happy);
  lcd.createChar(1, saddy);
  lcd.begin(16, 2);
}

void loop() {
  for (int i=0; i<2; i++)
  {
    lcd.setCursor(0,0);
    lcd.write(i);
    delay(500);
  }
}
```

## Discussion

The LiquidCrystal library enables you to create up to eight custom characters, which can be printed as character codes 0 through 8. Each character on the screen is drawn on a grid of 5 × 8 pixels. To define a character, you need to create an array of eight bytes. Each byte defines one of the rows in the character. When written as a binary number, the 1 indicates a pixel is on, 0 is off (any values after the fifth bit are ignored). The sketch example creates two characters, named happy and saddy (see Figure 11-3).



*Defining custom characters*

The following line in `setup` creates the character using data defined in the `happy` array that is assigned to character 0:

```
lcd.createChar(0, happy);
```

To print the custom character to the screen you would use this line:

```
lcd.write(0);
```

Code in the `for` loop switches between character 0 and character 1 to produce an animation.

> **NOTE**
>
> Note the difference between writing a character with or without an apostrophe. The following will print a zero, not the happy symbol:
>
> ```
> lcd.write('0'); // this prints a zero
> ```

## 11.7 Displaying Symbols Larger than a Single Character

### Problem

You want to combine two or more custom characters to print symbols larger than a single character; for example, double-height numbers on the screen.

### Solution

The following sketch writes double-height numbers using custom characters:

```
/*
 * customChars
 * This sketch displays double-height digits
```

```
 * the bigDigit arrays were inspired by Arduino forum member dcb
 */

#include <LiquidCrystal.h>

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

byte glyphs[5][8] = {
  { B11111,B11111,B00000,B00000,B00000,B00000,B00000,B00000 },
  { B00000,B00000,B00000,B00000,B00000,B00000,B11111,B11111 },
  { B11111,B11111,B00000,B00000,B00000,B00000,B11111,B11111 },
  { B11111,B11111,B11111,B11111,B11111,B11111,B11111,B11111 } ,
  { B00000,B00000,B00000,B00000,B00000,B01110,B01110,B01110 } };

const int digitWidth = 3; // the width in characters of a big digit
                          // (excludes space between characters)

//arrays  to index into custom characters that will comprise the big numbers
// digits 0 - 4                               0     1      2     3     4
const char bigDigitsTop[10][digitWidth]={ 3,0,3, 0,3,32, 2,2,3, 0,2,3, 3,1,3,
// digits 5-9                               5      6      7      8      9
                                          3,2,2, 3,2,2, 0,0,3,   3,2,3,
3,2,3};

const char bigDigitsBot[10][ digitWidth]={ 3,1,3, 1,3,1,  3,1,1, 1,1,3,
32,32,3,
                                          1,1,3, 3,1,3, 32,32,3, 3,1,3,
1,1,3};

char buffer[12]; // used to convert a number into a string
void setup ()
{
  lcd.begin(20,4);
  // create the custom glyphs
  for(int i=0; i < 5; i++)
    lcd.createChar(i, glyphs[i]); // create the 5 custom glyphs
  // show a countdown timer
  for(int digit = 9; digit >= 0; digit--)
  {
    showDigit(digit, 0);  // show the digit
    delay(1000);
  }
  lcd.clear();
}

void loop ()
{
  // now show the number of seconds since the sketch started
```

```
  int number = millis() / 1000;
  showNumber(number, 0);
  delay(1000);
  Serial.begin(9600);
}

void showDigit(int digit, int position)
{
  lcd.setCursor(position * (digitWidth + 1), 0);
  for(int i=0; i < digitWidth; i++)
    lcd.write(bigDigitsTop[digit][i]);
  lcd.setCursor(position * (digitWidth + 1), 1);
  for(int i=0; i < digitWidth; i++)
    lcd.write(bigDigitsBot[digit][i]);
}

void showNumber(int value, int position)
{
  int index; // index to the digit being printed, 0 is the leftmost digit
  String valStr = String(value);

  // display each digit in sequence
  for(index = 0; index < 5; index++) // display up to five digits
  {
    char c = valStr.charAt(index);
    if(c == 0)  // check for null (not the same as '0')
      return; // the end of string character is a null
    c = c - 48; // convert ascii value to a numeric value (see Chapter 2)
    showDigit(c, position + index);
  }
}
```

## Discussion
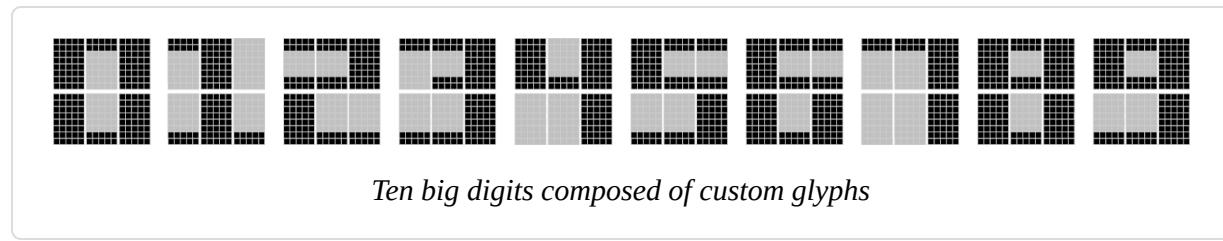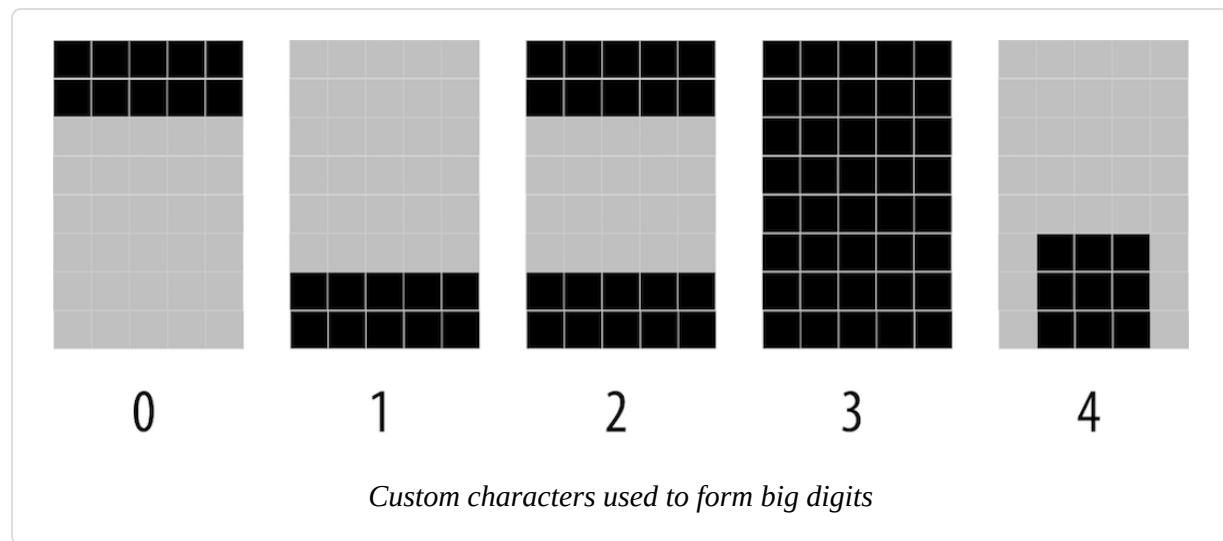
The LCD display has fixed-size characters, but you can create larger symbols by combining characters. This recipe creates five custom characters using the technique described in Recipe 11.6. These symbols (see Figure 11-4) can be combined to create double-sized digits (see Figure 11-5). The sketch displays a countdown from 9 to 0 on the LCD using the big digits. It then displays the number of seconds since the sketch started.

The glyphs array defines pixels for the five custom characters. The array has two dimensions given in the square brackets:

```
byte glyphs[5][8] = {
```

[5] is the number of glyphs and [8] is the number of rows in each glyph.
Each element contains 1s and 0s to indicate whether a pixel is on or off in
that row. If you compare the values in glyph[0] (the first glyph) with
Figure 11-2, you can see that the 1s correspond to dark pixels:

```
{ B11111,B11111,B00000,B00000,B00000,B00000,B00000,B00000  } ,
```



*Custom characters used to form big digits*



*Ten big digits composed of custom glyphs*

Each big number is built from six of these glyphs, three forming the upper
half of the big digit and three forming the lower half. bigDigitsTop and
bigDigitsBot are arrays defining which custom glyph is used for the top
and bottom rows on the LCD screen.

## See Also

See Chapter 7 for information on 7-segment LED displays if you need
really big numerals. Note that 7-segment displays can give you digit sizes

from one-half inch to two inches or more. They can use much more power than LCD displays and don't present letters and symbols very well, but they are a good choice if you need something big.
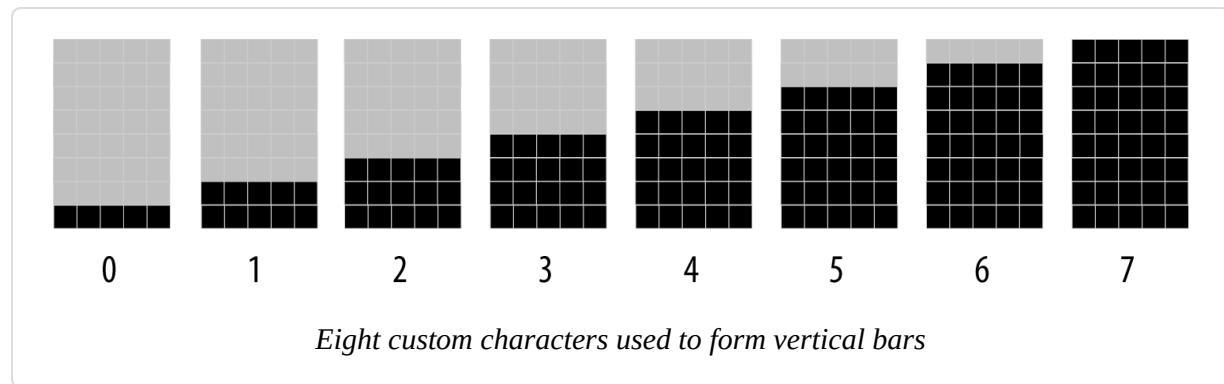
# 11.8 Displaying Pixels Smaller than a Single Character

## Problem

You want to display information with finer resolution than an individual character; for example, to display a bar chart.

## Solution

Recipe 11.7 describes how to build big symbols composed of more than one character. This recipe uses custom characters to do the opposite; it creates eight small symbols, each a single pixel higher than the previous one (see Figure 11-6).



*Eight custom characters used to form vertical bars*

These symbols are used to draw bar charts, as shown in the sketch that follows:

```
/*
 * customCharPixels
 */

#include <LiquidCrystal.h>

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

```
//set constants for number of rows and columns to match your LCD
const int numRows = 2;
const int numCols = 16;

// array of bits defining pixels for 8 custom characters
// ones and zeros indicate if a pixel is on or off

byte glyphs[8][8]  = {
      {B00000,B00000,B00000,B00000,B00000,B00000,B00000,B11111},   // 0
      {B00000,B00000,B00000,B00000,B00000,B00000,B11111,B11111},   // 1
      {B00000,B00000,B00000,B00000,B00000,B11111,B11111,B11111},   // 2
      {B00000,B00000,B00000,B00000,B11111,B11111,B11111,B11111},   // 3
      {B00000,B00000,B00000,B11111,B11111,B11111,B11111,B11111},   // 4
      {B00000,B00000,B11111,B11111,B11111,B11111,B11111,B11111},   // 5
      {B00000,B11111,B11111,B11111,B11111,B11111,B11111,B11111},   // 6
      {B11111,B11111,B11111,B11111,B11111,B11111,B11111,B11111}};   // 7

void setup ()
{
  lcd.begin(numCols, numRows);
  for(int i=0; i < 8; i++)
    lcd.createChar(i, glyphs[i]);     // create the custom glyphs
  lcd.clear();
}

void loop ()
{
  for( byte i=0; i < 8; i++)
    lcd.write(i);   // show all eight single height bars
  delay(2000);
  lcd.clear();
}
```

## Discussion

The sketch creates eight characters, each a single pixel higher than the one
before; see Figure 11-6. These are displayed in sequence on the top row of
the LCD. These "bar chart" characters can be used to display values in your
sketch that can be mapped to a range from 0 to 7. For example, the
following will display a value read from analog input 0:

```
int value = analogRead(A0);
byte glyph = map(value, 0,1023, 0,8); // proportional value from 0 through 7
```

```
  lcd.write(glyph);
  delay(100);
```

You can stack the bars for greater resolution. The doubleHeightBars function shown in the following code displays a value from 0 to 15 with a resolution of 16 pixels, using two lines of the display:

```
void doubleHeightBars(int value, int column)
{
  char upperGlyph;
  char lowerGlyph;

  if(value < 8)
  {
    upperGlyph = ' ';  // no pixels lit
    lowerGlyph = value;
  }
  else
  {
    upperGlyph = value - 8;
    lowerGlyph = 7; // all pixels lit
  }

  lcd.setCursor(column, 0); // do the upper half
  lcd.write(upperGlyph);
  lcd.setCursor(column, 1); // now to the lower half
  lcd.write(lowerGlyph);
}
```

The doubleHeightBars function can be used as follows to display the value of an analog input:

```
for(int i=0; i < 16; i++)
{
  int value = analogRead(A0);
  value = map(value, 0, 1023, 0,16);
  doubleHeightBars(value, i); // show a value from 0 to 15
  delay(1000);   // one second interval between readings
}
```

If you want horizontal bars, you can define five characters, each a single pixel wider than the previous one, and use similar logic to the vertical bars

to calculate the character to show.

# 11.9 Selecting a Graphical LCD Display

**Problem**

You want to show graphics or text on either a color or monochrome display.

## Solution

There are many graphical displays suitable for use with Arduino. Some of the things to consider when choosing a display are: resolution (number of pixels and text lines), display size, color, touch screen capability, onboard SD memory, and price. You will also need a library to interface with your display and you may be spoiled for choice for well-written and documented libraries that you can use. This recipe provides an overview of what is available along with links for more reading to help you select and use the display that is right for your project.

The increasingly rapid introduction of new smartphones and related devices has resulted in reducing costs to a level where monochrome or color displays can be purchased for a fraction of the cost of an Arduino board. When selecting a graphical display, the first thing to consider is color or monochrome. This is partially an aesthetic decision, but it can also be influenced by things like display size (in some cases, larger color panels may be cheaper than large monochrome displays), power consumption (OLED displays described later need less power), touch capability (color panels tend to have better support for touch), and connection type.

Graphical displays that are popular for Arduino projects typically use either liquid crystal display (LCD) or organic light-emitting diode (OLED) technology. OLED is newer technology that consumes less power than a conventional LCD because it does not need a backlight to be visible in low-light conditions. However, OLED panels cost more to manufacture so they tend to be much smaller in size than similarly priced LCD panels. Color

OLEDs are available but are very expensive and those sold for use with Arduino tend to be very small.

## Discussion

Monochrome displays were historically much cheaper than color but the price differential is decreasing. Monochrome displays are a good choice if you want a small display or if low current consumption is important. Also, sketch memory needed for monochrome is generally much less than color so that may be a factor if you have limited available memory.

There are Adafruit libraries for the SSD1306, SSD1325, SSD1331, and SSD1351 OLED displays. A nice feature of these is that the API is similar to the Adafruit color libraries if you want to use both mono and color panels in future projects, or may upgrade the screen in the project you are working on at a later date. Refer to the documentation for the Adafruit displays and libraries.

A highly capable monochrome library supporting over 60 controller variants is the u8g2 library. These include: SSD1305, SSD1306, SSD1309, SSD1322, SSD1325, SSD1327, SSD1606, SH1106, T6963, RA8835, LC7981, PCD8544, PCF8812, UC1604, UC1608, UC1610, UC1611, UC1701, ST7565, ST7567, NT7534, IST3020, ST7920, LD7032, and KS0108.

U8g2 is a good choice if you want the widest range of monochrome options. The full list of controller variants can be found on the u8g2 website.

This library supports common connection types including I2C, SPI, and parallel. If you have a monochrome graphical display, the u8g2 will more than likely work with it. You can find the library source here. The u8g2 wiki has extensive documentation.

Color displays offer the ability to display information in full color. The Adafruit GFX library provides a common set of graphic primitives that support many display-specific libraries such as the Adafruit_ST7735, Adafruit_HX8340B, Adafruit_HX8357D, Adafruit_ILI9340/1, and

Adafruit_PCD8544 libraries. This library is popular because it is well documented with many tutorials. This Adafruit tutorial covers the graphical functions common to all Adafruit libraries. The Adafruit Arcada library combines GFX support with a large collection of functionality useful for creating games or rich graphical user experiences, including various types of user input.

## See Also

The library for four-wire resistive touch screens

The library for touch screens

If you have a low-cost TFT screen with or without touch capability that uses ITDB02 or ILI9341 display controllers, check out the UTFT library.

URTouch is a companion to UTFT supporting touch screens.

---

### Beware Low-Cost Displays with Limited or No Supplier Support

Because of the vast variety of graphical displays, selecting a display can be daunting. If you don't have the experience to understand highly technical display controller datasheets, then do not be tempted by some ultra-cheap item from suppliers that do not provide documentation and support. A product listing may say it comes with software for Arduino, but it is not unusual for that code to not actually work with the supplied display or with the latest Arduino environment. The safer choice is through suppliers like Adafruit and SparkFun that have libraries and tutorials and support forums for their products.

---

# 11.10 Control a Full-Color LCD Display

## Problem

You want to display full-color graphics on a graphical LCD such as the ones based on the ST7735.

## Solution

This recipe uses the Adafruit ST7735 and ST7789 library, which provides support for ST7735- and ST7789-based TFT LCD displays, such as the 1.8a″ breakout (Adafruit product ID 358) or the 2.0″ IPS breakout (product ID 4311). Or if you want a simple and strange all-in-one option, the Adafruit HalloWing M0 combines a SAMD21 dev board, 8 MB of flash, sensors, speaker driver, and a 1.44″ full-color LCD, all on a skull-shaped PCB board. You can install this library using the Arduino Library Manager (see Chapter 16).

The sketch initializes the display on a HalloWing M0, shows three lines of text in varying sizes, and then switches to an animation of a yellow ball moving back and forth:

```
/*
 * Adafruit GFX ST7735 sketch
 * Display text and a moving ball on the display
 */

#include <Adafruit_GFX.h>      // Core graphics library
#include <Adafruit_ST7735.h>   // Hardware-specific library for ST7735
#include <SPI.h>

// Define the connections for your panel. This will vary depending on
// your display and the board you are using
#define TFT_CS    39
#define TFT_RST   37
#define TFT_DC    38
#define TFT_BACKLIGHT   7
Adafruit_ST7735 tft = Adafruit_ST7735(TFT_CS, TFT_DC, TFT_RST);

void setup(void)
{
  tft.initR(INITR_144GREENTAB); // Initialize ST7735, green tab packaging

  pinMode(TFT_BACKLIGHT, OUTPUT); // Backlight pin
  digitalWrite(TFT_BACKLIGHT, HIGH); // Turn on the backlight

  tft.setRotation(2); // This will depend on how you mounted the panel

  tft.fillScreen(ST77XX_BLACK); // Fill the screen with black
```

```
  // Display some text in a variety of fonts
  tft.setCursor(0, 0);
  tft.setTextWrap(false);

  tft.setTextColor(ST77XX_RED);
  tft.setTextSize(1);
  tft.println("Small");

  tft.setTextColor(ST77XX_GREEN);
  tft.setTextSize(2);
  tft.println("Medium");

  tft.setTextColor(ST77XX_BLUE);
  tft.setTextSize(3);
  tft.println("Large");
}

int ballDir = 1;         // Current direction of motion
int ballDiameter = 8;    // Diameter
int ballX = ballDiameter; // Starting X position
void loop()
{
  // If the ball is approaching the edge of the screen, reverse direction
  if (ballX >= tft.width() - ballDiameter || ballX < ballDiameter) {
    ballDir *= -1;
  }

  ballX += ballDir; // Move the ball's X position

  // Calculate the Y position based on where the cursor was
  int ballY = tft.getCursorY() + ballDiameter*2;

  tft.fillCircle(ballX, ballY, ballDiameter/2, 0xffff00); // Yellow ball
  delay(25);
  tft.fillCircle(ballX, ballY, ballDiameter/2, 0x000000); // Erase the ball
}
```

---
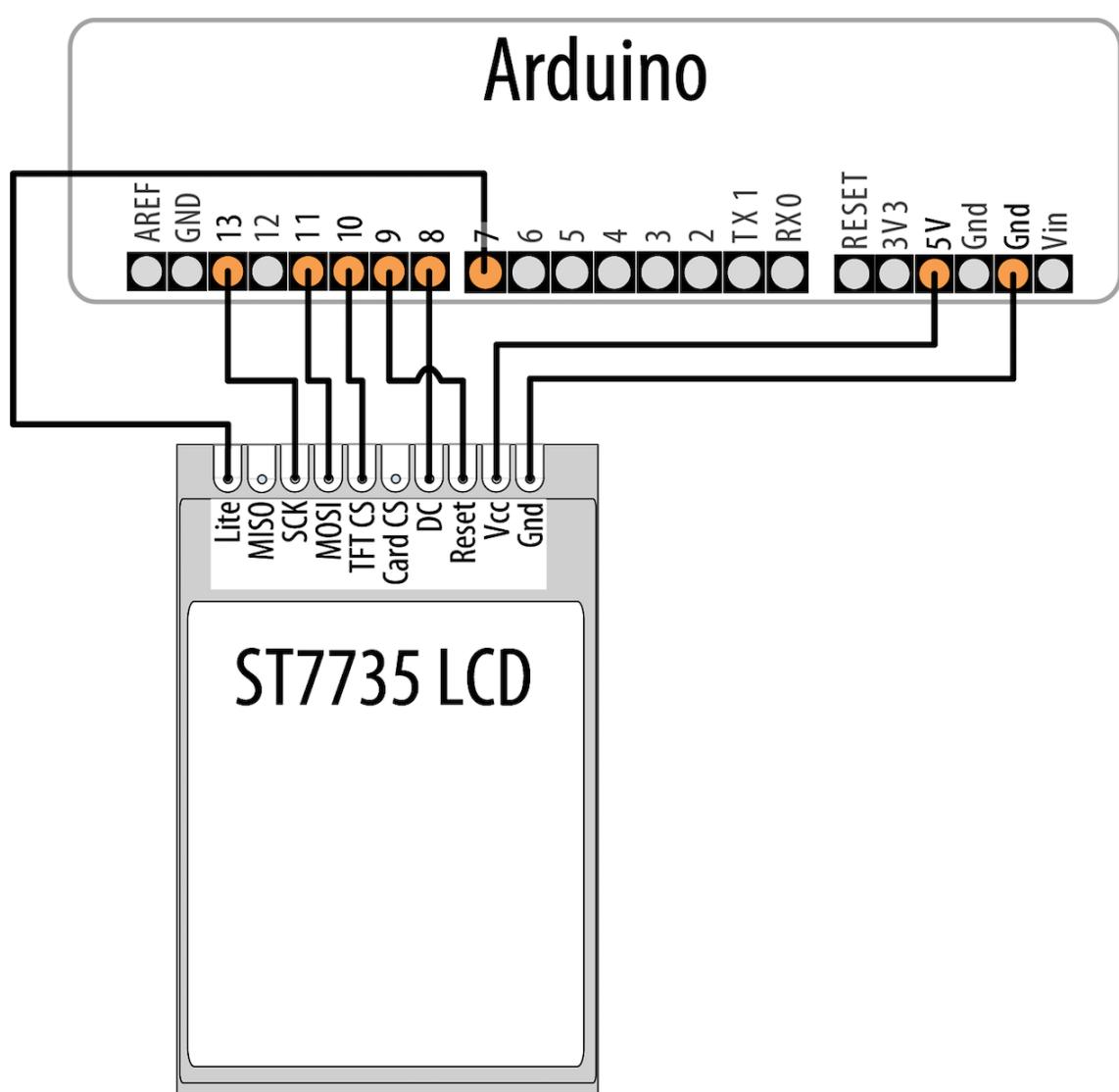
> ## NOTE
>
> If you are using the HalloWing all-in-one board (or any Adafruit board), you'll need to add the Adafruit board manager URL to the Arduino IDE and use Tools→Board→Boards Manager to install support for Adafruit SAMD boards, then select the Adafruit HalloWing M0 from the Tools→Board menu.

## Discussion

If you use this sketch with a standalone Arduino board (or compatible), you will need a graphical LCD supported by the Adafruit ST7735/ST7789 library such as those sold by Adafruit. You'll need to connect the board as shown in Figure 11-7. The pins used for MOSI and SCLK are dependent on your board (the Uno wiring is shown in the figure, but see "SPI" for other boards). For the Uno, you'd need to modify the `#define`s and the initialization as follows:

```
#define TFT_CS    10
#define TFT_RST   9
#define TFT_DC    8
#define TFT_BACKLIGHT  7
Adafruit_ST7735 tft = Adafruit_ST7735(TFT_CS, TFT_DC, TFT_RST);
```

The chip select and data/command pins are defined by `TFT_CS` and `TFT_DC`, and you are free to change these to another pin. The `TFT_RST` pin is used to reset the display, and you can change this as well. If you use a different connection for any of these, be sure to change the code, too.

*Wiring an ST7735 LCD display to Arduino*

The code sets up the `#defines` for the pins that are not defined by your board, and then initializes an instance of Adafruit_ST7735 as the object `tft`. Depending on what board you are using and how you've wired it, this initialization could vary. For example, you can, in fact, use software SPI and choose different pins for MOSI and SCLK, but this will result in slower performance (and if your LCD board has an SD card included, it won't work with software SPI):

```
#define TFT_SCLK 5
#define TFT_MOSI 6
Adafruit_ST7735 tft =
  Adafruit_ST7735(TFT_CS, TFT_DC, TFT_MOSI, TFT_SCLK, TFT_RST);
```

Within `setup()`, the sketch initializes the display with a call to `initR()`.
There are many variants of these displays, so be sure to consult the
documentation for them (the graphics test example program included with
the library has explanatory comments for many of the supported boards).
For example, there is a group of variants that can be identified by the color
of the tab on the tape that's affixed to the screen when you unbox it. The
Adafruit 1.8″ TFT screen uses `INITR_BLACKTAB`. The 1.44″ display on the
HalloWing M0 is the same as the 1.44″ green tab display, except that it's
mounted upside down. You can either use `INITR_144GREENTAB` and
`setRotation(2)` (upside-down portrait mode) as shown in the sketch or
just pass `INITR_HALLOWING` to `initR()`.

After initialization is complete, the sketch draws black over the entire
screen, and uses `setCursor()` to position the cursor on the screen before
displaying three lines of text that get progressively larger. In the `loop()`
function, the sketch moves a ball from one side of the screen to the other. It
uses a `ballDir` variable to determine whether the ball moves to the right (1)
or to the left (–1). When the ball comes within a ball's width of one side or
the other, it reverses direction.

## See Also

The datasheet for the ST7735

# 11.11 Control a Monochrome OLED Display

## Problem

You want to display single-color graphics on a monochrome OLED such as
the ones based on the SSD13xx.

## Solution

This recipe uses the Adafruit 128x32 SPI SSD1306 OLED display and the Adafruit SSD1306 library to display text and graphics on the display. The SSD1306 library works with a variety of OLED displays that use this chipset. You can install this library using the Arduino Library Manager (see Chapter 16).

Wire the display to your Arduino as shown in Figure 11-8, and run this sketch, which displays some scrolling text followed by a moving ball:

```
/*
 * OLED SSD13xx sketch
 * Display text and a moving ball on an OLED display.
 */

#include <SPI.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#define WIDTH 128
#define HEIGHT 32

#define OLED_DC     8
#define OLED_CS    10
#define OLED_RESET  9
Adafruit_SSD1306 display(WIDTH, HEIGHT, &SPI, OLED_DC, OLED_RESET, OLED_CS);

#define MODE SSD1306_SWITCHCAPVCC // get display voltage from 3.3V internally

void setup()
{
  Serial.begin(9600);
  if(!display.begin(MODE))
  {
    Serial.println("Could not initialize display");
    while(1); // halt
  }

  showAndScroll("Small",  1);
  showAndScroll("Medium", 2);
  showAndScroll("Large",  3);
}

// Show text and scroll it briefly
```

```cpp
void showAndScroll(String text, int textSize)
{
  display.setTextColor(SSD1306_WHITE); // Draw white text
  display.setCursor(0, 0); // Move the cursor to 0,0
  display.clearDisplay();  // clear the screen

  display.setTextSize(textSize);
  display.println(text);
  display.display();

  // Scroll the display right for 3 seconds
  display.startscrollright(0x00, 0x0F);
  delay(3000);
  display.stopscroll();
}

int ballDir = 1;            // Current direction of motion
int ballDiameter = 8;       // Diameter
int ballX = ballDiameter;   // Starting X position
int ballY = ballDiameter*2; // Y position
void loop()
{
  display.clearDisplay();

  // If the ball is approaching the edge of the screen, reverse direction
  if (ballX >= WIDTH - ballDiameter || ballX < ballDiameter)
  {
    ballDir *= -1;
  }

  // Move the ball's X position
  ballX += ballDir;

  // Draw a ball
  display.fillCircle(ballX, ballY, ballDiameter/2, SSD1306_INVERSE);
  display.display();
  delay(25);

  // Erase the ball
  display.fillCircle(ballX, ballY, ballDiameter/2, SSD1306_INVERSE);
  display.display();
}
```
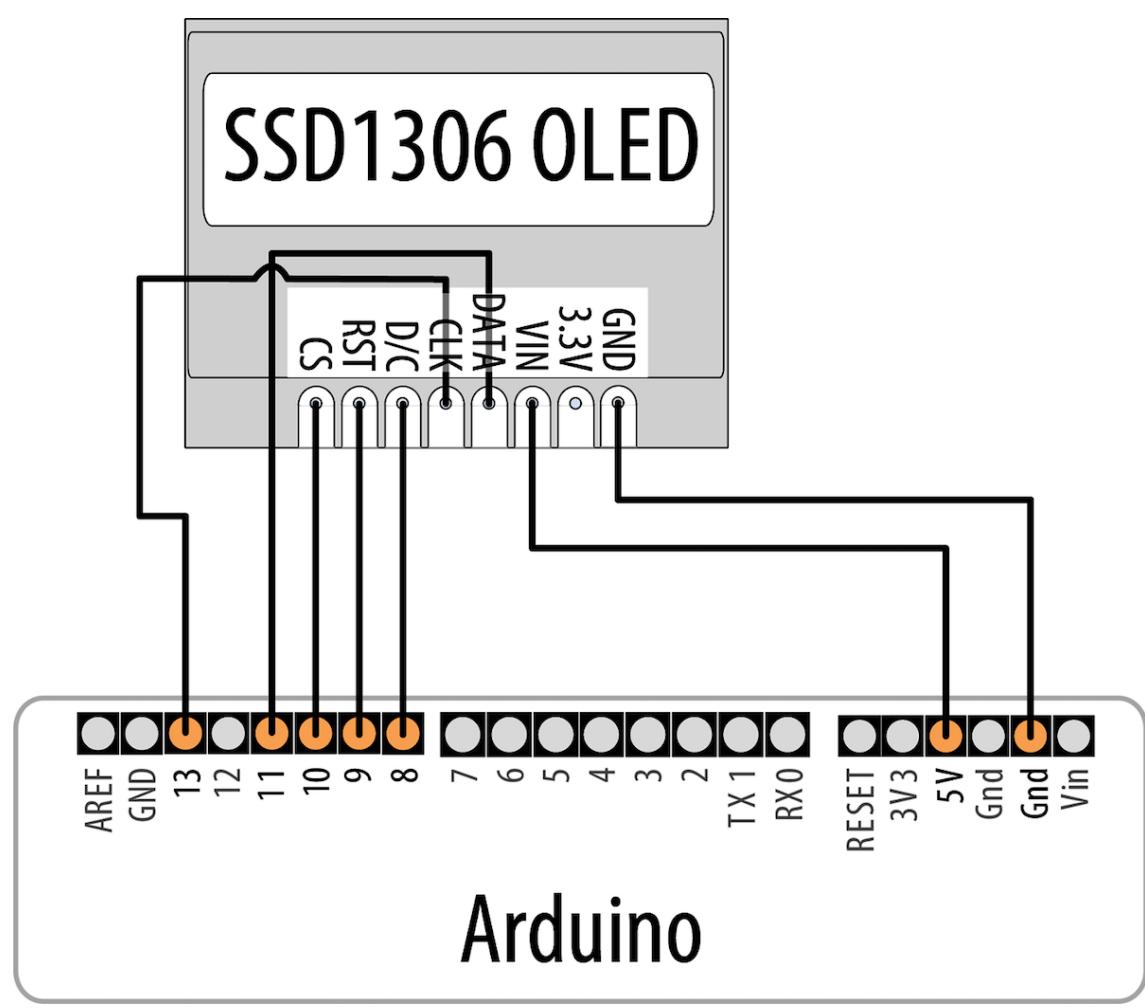
*Wiring an SSD13xx OLED display to Arduino*

## Discussion

The chip select and data/command pins are defined in the sketch by `OLED_CS` and `OLED_DC`. The `OLED_RESET` pin is used to reset the display. You can use different pins for any of those. The pins used for MOSI and SCLK are dependent on your board (the Uno wiring is shown in the figure, but see "SPI" for other boards).

The wiring in the Solution uses hardware SPI. If you want to use software SPI, you will need to add a `#define` for each of those pins, wire them up, and use a different form of the constructor:

```
#define OLED_CLK    5
#define OLED_MOSI   6
Adafruit_SSD1306 display(WIDTH, HEIGHT,
   OLED_MOSI, OLED_CLK, OLED_DC, OLED_RESET, OLED_CS);
```

The Adafruit library supports a variety of boards that use this chipset, and you'll need to change your code to match the display you are using. If your display is a different size but also uses SPI, you can probably just change the #defines for WIDTH and HEIGHT. If your display uses I2C instead of SPI, you can connect with fewer wires. You'll need to connect the reset pin as before, but you'll then only need to connect the SCL and SDA pins between the Arduino and the display. You'll need to modify the sketch to include *Wire.h* and use a different variant of the initialization:

```
#include <Wire.h>
#define WIDTH 128
#define HEIGHT 32
#define OLED_RESET 13
Adafruit_SSD1306 display(WIDTH, HEIGHT, &Wire, OLED_RESET);
```

The initialization of display gives you an Adafruit_SSD1306 object that you can use to control the display. In setup(), the sketch starts the display in the SSD1306_SWITCHCAPVCC mode (to tell the display to get its voltage internally). After that, the sketch uses the showAndScroll function to display and scroll text in three sizes. This function configures the display to draw in white text, clears the screen, and sets the cursor position to the top left. Then it sets the font, draws it on the screen, and calls display.display() to show what you've drawn. This is different from how you worked with the color LCD (Recipe 11.10) where anything you drew was immediately displayed. After the text is displayed, the sketch scrolls the screen for three seconds.

In the loop(), the sketch draws a ball on the screen and moves it back and forth across the screen. It uses the SSD1306_INVERSE color to alternate the color from white to black each time it's drawn. Each time through loop(), the sketch increments the ballX variable until it hits the right side of the

screen, when it starts decrementing it (until it hits the left wall). It then displays the ball and shows it for a fraction of a second, before erasing it.

You can also use the u8g2 library to generate a similar display. The code is slightly different, but mostly the same. The biggest difference is that instead of drawing on the screen and calling `display.display()` to update it, the sketch uses the u8g2 library's page buffer to update the display in stages. To use this, you must first call `u8g2.firstPage()`, then set up a do-while loop that calls `u8g2.nextPage()` at the end. Your drawing commands go inside the loop, and u8g2 goes as many times through the loop as is needed to draw the complete screen. U8g2 supports a wide variety of monochrome displays, and there are variants of each display (for example, hardware SPI, software SPI, I2C, and also permutations of the preceding for a variety of frame buffer/page buffer sizes). See the list of supported devices and their corresponding setup functions.

```
/*
 * u8g2 oled sketch
 * Draw some text, move a ball.
 */
#include <Arduino.h>
#include <U8g2lib.h>
#include <SPI.h>

#define OLED_DC     8
#define OLED_CS     10
#define OLED_RESET  9
U8G2_SSD1306_128X32_UNIVISION_2_4W_HW_SPI u8g2(U8G2_R0, OLED_CS,
  OLED_DC, OLED_RESET);

u8g2_uint_t displayWidth;

void setup(void)
{
  u8g2.begin();
  u8g2.setFontPosTop();
  displayWidth = u8g2.getDisplayWidth();

  showAndScroll("Small",  u8g2_font_6x10_tf);
  showAndScroll("Medium", u8g2_font_9x15_tf);
  showAndScroll("Large",  u8g2_font_10x20_tf);
}
```

```
int ballDir = 1;          // Current direction of motion
int ballRadius = 4;       // Radius
int ballX = ballRadius*2; // Starting X position
int ballY = ballRadius*4; // Y position
void loop(void)
{
  u8g2.firstPage(); // picture loop
  do
  {
    // If the ball is approaching the edge of the screen, reverse direction
    if (ballX >= displayWidth - ballRadius*2 || ballX < ballRadius*2)
    {
      ballDir *= -1;
    }

    ballX += ballDir; // Move the ball's X position
    u8g2.drawDisc(ballX, ballY, ballRadius); // Draw the ball
  } while ( u8g2.nextPage() );
  delay(25);
}

void showAndScroll(String text, uint8_t *font)
{
  for (int i = 0; i < 20; i++)
  {
    u8g2.firstPage(); // picture loop
    do
    {
      u8g2.setFont(font);
      u8g2.drawStr(10 + i, 10, text.c_str());
    } while ( u8g2.nextPage() );
    delay(125);
  }
}
```

## See Also

The documentation for Adafruit OLED displays

The SSD1306 datasheet

# Using Time and Dates

## 12.0 Introduction

Managing time is a fundamental element of interactive computing. This chapter covers built-in Arduino functions and introduces many additional techniques for handling time delays, time measurement, and real-world times and dates. You'll learn about Arduino's built-in function for introducing delays into your sketch, as well as more advanced techniques for intermittently performing operations. Other recipes in this chapter cover how to measure time as it passes, and even how to use an external real-time clock for tracking time and dates.

## 12.1 Using millis to Determine Duration

### Problem

You want to know how much time has elapsed since an event happened; for example, how long a switch has been held down.

### Solution

The following sketch uses the `millis()` function to print how long a button was pressed (see Recipe 5.2 for details on how to connect the switch):

```
/*
  millisDuration sketch
  returns the number of milliseconds that a button has been pressed
 */

const int switchPin = 2;  // the number of the input pin

unsigned long startTime; // the value of millis when the switch is pressed
unsigned long duration;  // variable to store the duration

void setup()
{
```

```
  pinMode(switchPin, INPUT_PULLUP);
  Serial.begin(9600);
}

void loop()
{
  if(digitalRead(switchPin) == LOW)
  {
    // here if the switch is pressed
    startTime = millis();
    while(digitalRead(switchPin) == LOW)
      ; // wait while the switch is still pressed
    unsigned long duration = millis() - startTime;
    Serial.println(duration);
  }
}
```

## Discussion

The `millis` function returns the number of milliseconds since the current sketch started running.

> **WARNING**
>
> The `millis` function will *overflow* (go back to zero) after approximately 50 days. See Recipes 12.4 and 12.5 for information about using the Time library for handling intervals from seconds to years.

By storing the start time for an event, you can determine the duration of the event by subtracting the start time from the current time, as shown here:

```
unsigned long duration = millis() - startTime;
```

## See Also

The Arduino reference for `millis`

See Recipes 12.4 and 12.5 for information about using the Time library to handle intervals from seconds to years.

# 12.2 Creating Pauses in Your Sketch

## Problem

You want your sketch to pause for some period of time. This may be some number of milliseconds, or a time given in seconds, minutes, hours, or days.

## Solution

The Arduino `delay` function is used in many sketches throughout this book. It pauses a sketch for the number of milliseconds specified as a parameter. (There are 1,000 ms in one second.) The sketch that follows shows how you can use `delay` to get almost any interval:

```
/*
 * delay sketch
 */

const unsigned long oneSecond = 1000;  // a second is 1,000 ms
const unsigned long oneMinute = oneSecond * 60;
const unsigned long oneHour   = oneMinute * 60;
const unsigned long oneDay     = oneHour * 24;

void setup()
{
  Serial.begin(9600);
  while(!Serial); // Needed on Leonardo and ARM-based boards
}

void loop()
{
  Serial.println("delay for 1 millisecond");
  delay(1);
  Serial.println("delay for 1 second");
  delay(oneSecond);
  Serial.println("delay for 1 minute");
  delay(oneMinute);
  Serial.println("delay for 1 hour");
  delay(oneHour);
  Serial.println("delay for 1 day");
  delay(oneDay);
  Serial.println("Ready to start over");
}
```

## Discussion

Because it is limited by the maximum value of an integer, the `delay` function has a range from one one-thousandth of a second to around 25 days when you use a long integer. If you used an unsigned long, it will reach just under 50 days; see Chapter 2 for more on variable types).

You can use `delayMicroseconds` to delay short periods. There are 1,000 microseconds in 1 ms, and 1 million ms in 1 second. `delayMicroseconds` will pause from one microsecond to around 16 ms, but for delays longer than a few thousand microseconds you should use `delay` instead:

```
delayMicroseconds(10);  // delay for 10 microseconds
```

> **NOTE**
>
> `delay` and `delayMicroseconds` will delay for at least the amount of time given as the parameter, but they could delay a little longer if interrupts occur within the delay time.

The drawback of using the `delay` function is that your sketch can't do anything else during the delay period. You can find an alternative approach in the BlinkWithoutDelay example (File → Examples → 02. Digital → BlinkWithoutDelay). This approach uses a variable, `previousMillis`, to store the time at which an action was last performed. The sketch then checks the value of `millis()` (which is based on an internal clock that ticks once every millisecond the sketch is running). When the difference between the current value of `millis()` and `previousMillis` reaches or exceeds a given interval, it performs an action, such as blinking an LED. Here is an abbreviated version of that sketch:

```
int ledState = LOW;
unsigned long previousMillis = 0;
const long interval = 1000;

void setup()
{
```

```
    pinMode(LED_BUILTIN, OUTPUT);
}

void loop()
{
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval)
  {
    previousMillis = currentMillis;
    if (ledState == LOW)
    {
      ledState = HIGH;
    }
    else
    {
      ledState = LOW;
    }
    digitalWrite(LED_BUILTIN, ledState);
  }
  // You can perform other actions here.
}
```

Here is a way to package this logic into a function named `myDelay` that will delay the code in `loop` but can perform some action during the delay period. You can customize the functionality for your application, but in this example, an LED is blinked on or off every 250 ms:

```
/*
 * myDelay example sketch to blink an LED for a set amount of time
 */
const int ledPin =  LED_BUILTIN; // the number of the LED pin

int ledState = LOW;                   // ledState used to set the LED
unsigned long previousMillis = 0; // will store last time LED was updated

void setup()
{
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  if (myDelay(blink, 250))
  {
```

```
      Serial.println(millis() / 1000.0); // print elapsed time in seconds
    }
  }

  /*
   * Perform the specified function, return true if it was performed
   */
  bool myDelay(void (*func)(void), long interval)
  {
    unsigned long currentMillis = millis();
    if (currentMillis - previousMillis >= interval)
    {
      // save the last time you blinked the LED
      previousMillis = currentMillis;
      func(); // invoke the function
      return true;
    }
    return false;
  }

  void blink()
  {
    // if the LED is off turn it on and vice versa:
    if (ledState == LOW)
    {
      ledState = HIGH;
    }
    else
    {
      ledState = LOW;
    }
    digitalWrite(ledPin, ledState);
  }
```

The `void (*func)(void)` parameter in the definition of `myDelay` indicates that the `func` argument is a pointer to a `void` function that takes no (`(void)`) arguments. So, every time `func()` is invoked in `myDelay`, it's really calling `blink()`. When the interval is reached, `myDelay` resets `previousMillis`, calls `blink`, and returns true.

Arduino style is to avoid the use of pointers in sketches and to reserve those for use in libraries, which avoids confusing beginners who use your sketch. So, another approach is to use a third-party library available from the Library Manager, such as Tasker. This example blinks LEDs on two pins at

different rates, the built-in LED and an LED connected to pin 10. The sketch avoids needing to store the state of each pin in a separate variable by using `digitalRead` to determine whether the LED is currently on or off:

```
/*
 * Tasker demo sketch
 */
#define TASKER_MAX_TASKS 2 // Set this to the number of tasks you need
#include <Tasker.h>

// Declare the Tasker object
Tasker tasker;

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(10, OUTPUT);

  // Blink the built-in LED every second
  tasker.setInterval(blink, 1000, LED_BUILTIN);

  // Blink the LED on pin 10 twice per second
  tasker.setInterval(blink, 500, 10);
}

void loop()
{
  tasker.loop(); // Run any pending tasks
}

void blink(int pinNumber)
{
  bool ledState = !digitalRead(pinNumber); // Toggle the current pin state
  if (ledState)
  {
    digitalWrite(pinNumber, HIGH);
  }
  else
  {
    digitalWrite(pinNumber, LOW);
  }
}
```

## See Also

# 12.3 More Precisely Measuring the Duration of a Pulse

## Problem

You want to determine the duration of a pulse (when a digital signal transitions from low to high back to low again, or high to low back to high) with microsecond accuracy; for example, to measure the exact duration of `HIGH` or `LOW` pulses on a pin.

## Solution

The `pulseIn` function returns the duration in microseconds for a changing signal on a digital pin. This sketch prints the time in microseconds of the `HIGH` and `LOW` pulses generated by `analogWrite` (see the section on "Analog Output" in Chapter 7). Because the `analogWrite` pulses are generated internally by Arduino, no external wiring is required:

```
/*
  PulseIn sketch
  displays duration of high and low pulses from analogWrite
 */

const int inputPin = 3;    // analog output pin to monitor
unsigned long val;  // this will hold the value from pulseIn

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

void loop()
{
  analogWrite(inputPin, 128);
  Serial.print("Writing 128 to pin ");
  Serial.print(inputPin);
  printPulseWidth(inputPin);
```

```
    analogWrite(inputPin, 254);
    Serial.print("Writing 254 to pin ");
    Serial.print(inputPin);
    printPulseWidth(inputPin);
    delay(3000);
}

void printPulseWidth(int pin)
{
    val = pulseIn(pin, HIGH);
    Serial.print(": High Pulse width = ");
    Serial.print(val);
    val = pulseIn(pin, LOW);
    Serial.print(", Low Pulse width = ");
    Serial.println(val);
}
```

## Discussion

The Serial Monitor will display:

```
Writing 128 to pin 3: High Pulse width = 989, Low Pulse width = 997
Writing 254 to pin 3: High Pulse width = 1977, Low Pulse width = 8
```

pulseIn can measure how long a pulse is either HIGH or LOW:

```
pulseIn(pin, HIGH); // returns microseconds that pulse is HIGH
pulseIn(pin, LOW);  // returns microseconds that pulse is LOW
```

The pulseIn function waits for the pulse to start (or for a timeout if there is no pulse). By default, it will stop waiting after one second, but you can change that by specifying the time to wait in microseconds as a third parameter (note that 1,000 microseconds equals 1 millisecond):

```
pulseIn(pin, HIGH, 5000); // wait 5 ms for the pulse to start
```

> **TIP**
>
> The timeout value only matters if the pulse does not start within the given period. Once the start of a pulse is detected, the function will start timing and will not return until the pulse ends.

`pulseIn` can measure values between around 10 microseconds to three minutes in duration, but the value of long pulses may not be very accurate.

## See Also

The Arduino reference for `pulseIn`

Recipe 6.5 shows `pulseIn` used to measure the pulse width of an ultrasonic distance sensor.

Recipe 18.2 provides more information on using hardware interrupts.

# 12.4 Using Arduino as a Clock

## Problem

You want to use the time of day (hours, minutes, and seconds) in a sketch, and you don't want to connect external hardware.

## Solution

This sketch uses the Time library to display the time of day. The Time library can be installed using the Arduino Library Manager (if you have trouble finding it, try searching the Arduino Library Manager for "timekeeping"):

```
/*
 * Time sketch
 */

#include <TimeLib.h>

void setup()
{
  Serial.begin(9600);
  setTime(12,0,0,1,1,2020); // set time to noon Jan 1 2020
}

void loop()
{
  digitalClockDisplay();
```

```
    delay(1000);
  }

  // Pad digits with a leading 0
  String padDigits(int digit)
  {
    String str = String("0") + digit; // Put a zero in front of the digit
    return str.substring(str.length() - 2); // Remove all but the
                                            // last two characters
  }

  void digitalClockDisplay(){
    String timestr = String(hour()) + ":" + padDigits(minute()) +
                     ":" + padDigits(second());
    Serial.println(timestr);

    String datestr = String(year()) + "-" + padDigits(month()) +
                     "-" + padDigits(day());
    Serial.println(datestr);
  }
```

## Discussion

The Time library enables you to keep track of the date and time. Many Arduino boards use a quartz crystal for timing, and this is accurate to a couple of seconds per day, but it does not have a battery to remember the time when power is switched off. Therefore, time will restart from 0 each time a sketch starts, so you need to set the time using the `setTime` function. The sketch sets the time to noon on January 1, 2020 each time it starts.

> **NOTE**
>
> The Time library uses a standard known as Unix time (also called POSIX time or Epoch time). The values represent the number of elapsed seconds since January 1, 1970. Experienced C programmers may recognize that this is the same as the `time_t` used in the ISO standard C library for storing time values.

Of course, it's more useful to set the time to your current local time instead of a fixed value. The following sketch gets the numerical time value (the number of elapsed seconds since January 1, 1970) from the serial port to set

the time. You can enter a value using the Serial Monitor (the current Unix time can be found on a number of websites, including Epoch Converter):

```
/*
 * SetTimeSerial sketch
 * Set the time from the serial port. Simplified version of TimeSerial example
 * from the Time library.
 *
 * Set the time by sending the letter T followed by 10 digits indicating
 * number of seconds since January 1, 1970, for example T1569888000 would
 * represent 12am on October 1, 2019.
 */

#include <TimeLib.h>

#define TIME_HEADER  'T'   // Header tag for serial time sync message

void setup()  {
  Serial.begin(9600);
  Serial.println("Waiting for time sync message");
}

void loop(){
  if(Serial.available())
  {
    processSyncMessage();
  }
  if(timeStatus() != timeNotSet)
  {
    // Display the time and date
    digitalClockDisplay();
  }
  delay(1000);
}

// Pad digits with a leading 0
String padDigits(int digit)
{
  String str = String("0") + digit; // Put a zero in front of the digit
  return str.substring(str.length() - 2); // Remove all but the last two
characters
}

void digitalClockDisplay(){
  String timestr = String(hour()) + ":" + padDigits(minute()) +
                   ":" + padDigits(second());
```

```
    Serial.println(timestr);

    String datestr = String(year()) + "-" + padDigits(month()) +
                     "-" + padDigits(day());
    Serial.println(datestr);
}

// Parse the time message
void processSyncMessage() {

    time_t pctime = 0;

    if(Serial.find(TIME_HEADER)) {
        pctime = Serial.parseInt();
        setTime(pctime); // Set clock to the time received on serial port
    }

}
```

The code to display the time and date is the same as before, but now the
sketch waits to receive the time from the serial port. See the Discussion in
Recipe 4.3 if you are not familiar with how to receive numeric data using
the serial port.

A processing sketch named SyncArduinoClock is included with the Time
library examples (it's in the *Time/Examples/Processing/SyncArduinoClock*
folder). This Processing sketch will send the current time from your
computer to Arduino at the click of a mouse. Run SyncArduinoClock in
Processing, ensuring that the serial port is the one connected to Arduino
(Chapter 4 describes how to run a Processing sketch that talks to Arduino).
You should see the message `Waiting for time sync message` sent by
Arduino and displayed in the Processing text area (the black area for text
messages at the bottom of the Processing IDE). Click the Processing
application window (it's a 200-pixel gray square) and you should see the
text area display the time as printed by the Arduino sketch.

You can also set the clock from the Serial Monitor if you can get the current
Unix time; Epoch Converter is one of many websites that provide the time
in this format. Make sure the converter you use is configured for
microseconds (a 10-digit value, at least until sometime in 2286); if it is

configured for milliseconds the number will be 1,000 times too large. Copy the 10-digit number indicated as the current Unix time and paste this into the Serial Monitor Send window. Precede the number with the letter *T* and click Send. For example, if you send this:

```
T1282041639
```

Arduino should respond by displaying the time every second:

```
10:40:49 17 8 2019
10:40:50 17 8 2019
10:40:51 17 8 2019
10:40:52 17 8 2019
10:40:53 17 8 2019
10:40:54 17 8 2019
. . .
```

You can also set the time using buttons or other input devices such as tilt sensors, a joystick, or a rotary encoder.

The following sketch uses two buttons to move the clock "hands" forward or backward. Figure 12-1 shows the connections (see Recipe 5.2 if you need help using switches):

```
/*
  AdjustClockTime sketch
  buttons on pins 2 and 3 adjust the time
 */

#include <TimeLib.h>

const int  btnForward = 2;  // button to move time forward
const int  btnBack = 3;     // button to move time back

unsigned long  prevtime;   // when the clock was last displayed

void setup()
{
  pinMode(btnForward, INPUT_PULLUP);  // enable internal pull-up resistors
  pinMode(btnBack, INPUT_PULLUP);
  setTime(12,0,0,1,1,2020); // start with the time set to noon Jan 1 2020
  Serial.begin(9600);
```

```
}

void loop()
{
  prevtime = now();    // note the time
  while( prevtime == now() )    // stay in this loop till the second changes
  {
    // check if the set button pressed while waiting for second to roll over
    if(checkSetTime())
      prevtime = now(); // time changed so reset start time
  }
  digitalClockDisplay();
}

// functions check to see if the time should be adjusted
// return true if time was changed
bool checkSetTime()
{
  int step;    // the number of seconds to move (backward if negative)
  bool isTimeAdjusted = false;  // set to true if the time is adjusted
  step = 1;    // ready to step forward

  while(digitalRead(btnForward)== LOW)
  {
    adjustTime(step);
    isTimeAdjusted = true; // to tell the user that the time has changed
    step = step + 1; // next step will be bigger
    digitalClockDisplay(); // update clock
    delay(100);
  }
  step = -1;    // negative numbers step backward
  while(digitalRead(btnBack)== LOW)
  {
    adjustTime(step);
    isTimeAdjusted = true; // to tell the user that the time has changed
    step = step - 1; // next step will be a bigger negative number
    digitalClockDisplay(); // update clock
    delay(100);
  }
  return isTimeAdjusted;  // tell the user if the time was adjusted
}

// Pad digits with a leading 0
String padDigits(int digit)
{
  String str = String("0") + digit; // Put a zero in front of the digit
  return str.substring(str.length() - 2); // Remove all but the
                                          // last two characters
```

```
  }

void digitalClockDisplay(){
  String timestr = String(hour()) + ":" + padDigits(minute()) +
                 ":" + padDigits(second());
  Serial.println(timestr);

  String datestr = String(year()) + "-" + padDigits(month()) +
                 "-" + padDigits(day());
  Serial.println(datestr);
}
```



*Two buttons used to adjust the time*

The sketch uses the same `digitalClockDisplay` and `printDigits` functions from Recipe 12.3, so copy those prior to running the sketch.

Here is a variation on this sketch that uses the position of a variable resistor to determine the direction and rate of adjustment when a switch is pressed:

```
#include <TimeLib.h>

const int  potPin = A0;    // pot to determine direction and speed
const int  buttonPin = 2;  // button enables time adjustment

unsigned long prevtime;    // when the clock was last displayed
```

```arduino
void setup()
{
  digitalWrite(buttonPin, HIGH);  // enable internal pull-up resistors
  setTime(12,0,0,1,1,2020); // start with the time set to noon Jan 1 2020
  Serial.begin(9600);
}

void loop()
{
  prevtime = now();    // note the time
  while( prevtime == now() )     // stay in this loop till the second changes
  {
    // check if the set button pressed while waiting for second to roll over
    if(checkSetTime())
      prevtime = now(); //  time has changed, so reset start time
  }
  digitalClockDisplay();
}

// functions check to see if the time should be adjusted
// return true if time was changed
bool checkSetTime()
{
  int value;  // a value read from the pot
  int step;   // the number of seconds to move (backward if negative)
  bool isTimeAdjusted = false;  // set to true if the time is adjusted

  while(digitalRead(buttonPin)== LOW)
  {
    // here while button is pressed
    value = analogRead(potPin);  // read the pot value
    step = map(value, 0,1023, 10, -10);  // map value to the desired range
    if( step != 0)
    {
      adjustTime(step);
      isTimeAdjusted = true; // to tell the user that the time has changed
      digitalClockDisplay(); // update clock
      delay(100);
    }
  }
  return isTimeAdjusted;
}
```
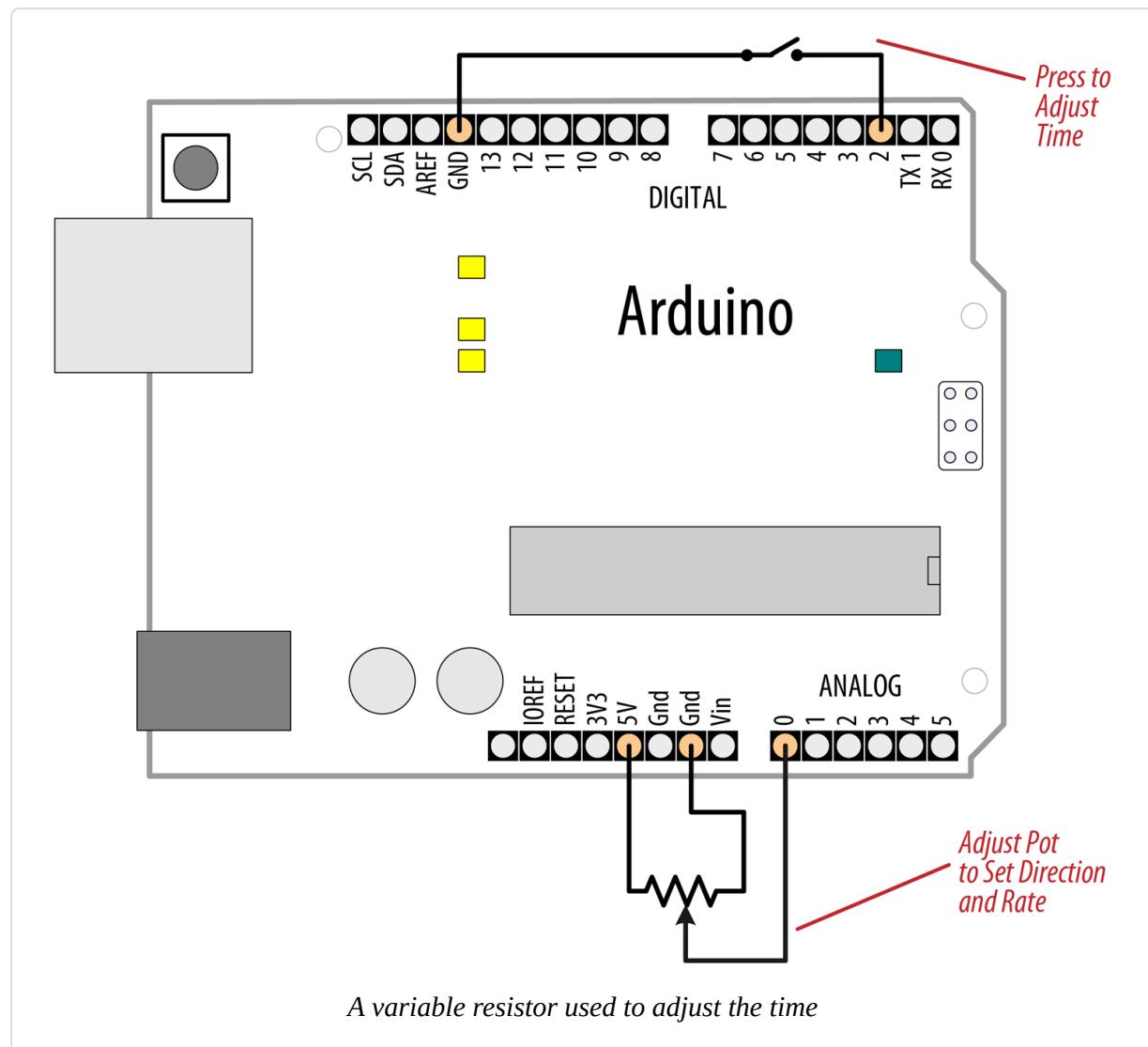
The preceding sketch uses the same `digitalClockDisplay` and `printDigits` functions from Recipe 12.3, so copy those prior to running

the sketch. Figure 12-2 shows how the variable resistor and switch are connected. If you are using a 3.3V board that is not 5-volt tolerant, connect the positive side of the variable resistor to 3.3V instead of 5V.

All these examples print to the serial port, but you can print the output to LEDs or LCDs. The download for the Graphical LCD covered in Recipe 11.9 contains example sketches for displaying and setting time using an analog clock display drawn on the LCD.

The Time library includes convenience functions for converting to and from various time formats. For example, you can find out how much time has elapsed since the start of the day and how much time remains until the day's end.



*A variable resistor used to adjust the time*

You can look in *TimeLib.h* in the *libraries* folder for the complete list. More details are available in Chapter 16:

```
dayOfWeek( now() );           // the day of the week (Sunday is day 1)
elapsedSecsToday( now() );    // returns the number of seconds since the start
                              // of today
nextMidnight( now() );        // how much time to the end of the day
elapsedSecsThisWeek( now() ); // how much time has elapsed since the start of
                              // the week
```

You can also print text strings for the days and months; here is a variation on the digital clock display code that prints the names of the day and month:

```
void digitalClockDisplay(){
  String timestr = String(hour()) + ":" + padDigits(minute()) +
                   ":" + padDigits(second());
  Serial.println(timestr);

  String datestr = String(dayStr(weekday())) + ", " +
                   String(monthShortStr(month())) + " " + String(year());
  Serial.println(datestr);
}
```

## See Also

The Time library reference

This Wikipedia article on Unix time

Epoch Converter and OnlineConversion.com are two popular Unix time-conversion tools.

# 12.5 Creating an Alarm to Periodically Call a Function

## Problem

You want to perform some action on specific days and at specific times of the day.

## Solution

TimeAlarms is a companion to the Time library discussed in Recipe 12.4. Install the TimeAlarms library using the Arduino Library Manager (and install the Time library also if not already installed). TimeAlarms makes it easy to create time and date alarms:

```
/*
 * TimeAlarmsExample sketch
 *
 * This example calls alarm functions at 8:30 am and at 5:45 pm (17:45)
 * and simulates turning lights on at night and off in the morning
 *
 * A timer is called every 15 seconds
 * Another timer is called once only after 10 seconds
 *
 * At startup the time is set to Jan 1 2020  8:29 am
 */

#include <TimeLib.h>
#include <TimeAlarms.h>

void setup()
{
  Serial.begin(9600);
  while(!Serial);
  Serial.println("TimeAlarms Example");
  Serial.println("Alarms are triggered daily at 8:30 am and 17:45 pm");
  Serial.println("One timer is triggered every 15 seconds");
  Serial.println("Another timer is set to trigger only once after 10
seconds");
  Serial.println();

  setTime(8,29,40,1,1,2020); // set time to 8:29:40am Jan 1 2020

  Alarm.alarmRepeat(8,30,0, MorningAlarm);  // 8:30am every day
  Alarm.alarmRepeat(17,45,0,EveningAlarm);  // 5:45pm every day

  Alarm.timerRepeat(15, RepeatTask);              // timer for every 15 seconds
  Alarm.timerOnce(10, OnceOnlyTask);              // called once after 10
seconds
}

void MorningAlarm()
{
  Serial.println("Alarm: - turn lights off");
```

```
}

void EveningAlarm()
{
   Serial.println("Alarm: - turn lights on");
}

void RepeatTask()
{
   Serial.println("15 second timer");
}

void OnceOnlyTask()
{
   Serial.println("This timer only triggers once");
}

void loop()
{
  digitalClockDisplay();
  Alarm.delay(1000); // wait one second between clock display
}

// Pad digits with a leading 0
String padDigits(int digit)
{
  String str = String("0") + digit; // Put a zero in front of the digit
  return str.substring(str.length() - 2); // Remove all but the
                                          // last two characters
}

void digitalClockDisplay(){
  String timestr = String(hour()) + ":" + padDigits(minute()) +
                   ":" + padDigits(second());
  Serial.println(timestr);

  String datestr = String(year()) + "-" + padDigits(month()) +
                   "-" + padDigits(day());
  Serial.println(datestr);
}
```

## Discussion

You can schedule tasks to trigger at a particular time of day (these are called *alarms*) or schedule tasks to occur after an interval of time has elapsed

(called *timers*). Each of these tasks can be created to continuously repeat or to occur only once.

To specify an alarm to trigger a task repeatedly at a particular time of day, use:

```
Alarm.alarmRepeat(8,30,0, MorningAlarm);
```

This calls the function `MorningAlarm` at 8:30 a.m. every day.

If you want the alarm to trigger only once, you can use the `alarmOnce` method:

```
Alarm.alarmOnce(8,30,0, MorningAlarm);
```

This calls the function `MorningAlarm` a single time only (the next time it is 8:30 a.m.) and will not trigger again.

Timers trigger tasks that occur after a specified interval of time has passed rather than at a specific time of day. The timer interval can be specified in any number of seconds, or in hours, minutes, and seconds:

```
Alarm.timerRepeat(15, Repeats);          // timer task every 15 seconds
```

This calls the `Repeats` function in your sketch every 15 seconds.

If you want a timer to trigger once only, use the `timerOnce` method:

```
Alarm.timerOnce(10, OnceOnly);           // called once after 10 seconds
```

This calls the `onceOnly` function in a sketch 10 seconds after the timer is created.

> **NOTE**
>
> Your code needs to call `Alarm.delay` regularly because this function checks the state of all the scheduled events. Failing to regularly call `Alarm.delay` will result in the alarms not being triggered. You can call `Alarm.delay(0)` if you need to service the scheduler without a delay. Always use `Alarm.delay` instead of `delay` when using TimeAlarms in a sketch.

The TimeAlarms library requires the Time library to be installed—see Recipe 12.4. No internal or external hardware is required to use the TimeAlarms library. The scheduler does not use interrupts, so the task-handling function is the same as any other functions you create in your sketch (code in an interrupt handler has restrictions that are discussed in Chapter 18, but these do not apply to TimeAlarms functions).

Timer intervals can range from one second to several years. (If you need timer intervals shorter than one second, the Tasker library may be more suitable.

Tasks are scheduled for specific times designated by the system clock in the Time library (see Recipe 12.4 for more details). If you change the system time (e.g., by calling `setTime`), the trigger times are not adjusted. For example, if you use `setTime` to move one hour ahead, all alarms and timers will occur one hour sooner. In other words, if it's 1:00 and a task is set to trigger in two hours (at 3:00), and then you change the current time to 2:00, the task will trigger in one hour. If the system time is set backward—for example, to 12:00—the task will trigger in three hours (i.e., when the system time indicates 3:00). If the time is reset to earlier than the time at which a task was scheduled, the task will be triggered immediately (actually, on the next call to `Alarm.delay`).

This is the expected behavior for alarms—tasks are scheduled for a specific time of day and will trigger at that time—but the effect on timers may be less clear. If a timer is scheduled to trigger in five minutes' time and then the clock is set back by one hour, that timer will not trigger until one hour and five minutes have elapsed (even if it is a repeating timer—a repeat does not get rescheduled until after it triggers).

Up to six alarms and timers can be scheduled to run at the same time. You can modify the library to enable more tasks to be scheduled; Recipe 16.3 shows you how to do this.

`onceOnly` alarms and timers are freed when they are triggered, and you can reschedule these as often as you want so long as there are no more than six pending at one time. The following code gives one example of how a `timerOnce` task can be rescheduled:

```
Alarm.timerOnce(random(10), randomTimer);  // trigger after random
                                           // number of seconds

void randomTimer(){
  int period = random(2,10);               // get a new random period
  Alarm.timerOnce(period, randomTimer);  // trigger for another random period
}
```

# 12.6 Using a Real-Time Clock

## Problem

You want to use the time of day provided by a real-time clock (RTC) such as the DS1307. External boards usually have battery backup, so the time will be correct even when Arduino is reset or turned off.

## Solution

The simplest way to use an RTC is with a companion library to the Time library, named *DS1307RTC.h*. Install the DS1307RTC library using the Arduino Library Manager (and install the Time library also if not already installed). This recipe is for the widely used DS1307 and DS1337 RTC chips:

```
/*
 * TimeRTC sketch
 * example code illustrating Time library with real-time clock.
 *
 */
```

```
#include <TimeLib.h>
#include <Wire.h>
#include <DS1307RTC.h>  // a basic DS1307 library that returns time as a
time_t

void setup()  {
  Serial.begin(9600);
  while(!Serial); // For Leonardo and 32-bit boards

  setSyncProvider(RTC.get);   // the function to get the time from the RTC
  if(timeStatus()!= timeSet)
     Serial.println("Unable to sync with the RTC");
  else
     Serial.println("RTC has set the system time");
}

void loop()
{
   digitalClockDisplay();
   delay(1000);
}

// Pad digits with a leading 0
String padDigits(int digit)
{
  String str = String("0") + digit; // Put a zero in front of the digit
  return str.substring(str.length() - 2); // Remove all but the
                                          // last two characters
}

void digitalClockDisplay(){
  String timestr = String(hour()) + ":" + padDigits(minute()) +
                   ":" + padDigits(second());
  Serial.println(timestr);

  String datestr = String(year()) + "-" + padDigits(month()) +
                   "-" + padDigits(day());
  Serial.println(datestr);
}
```
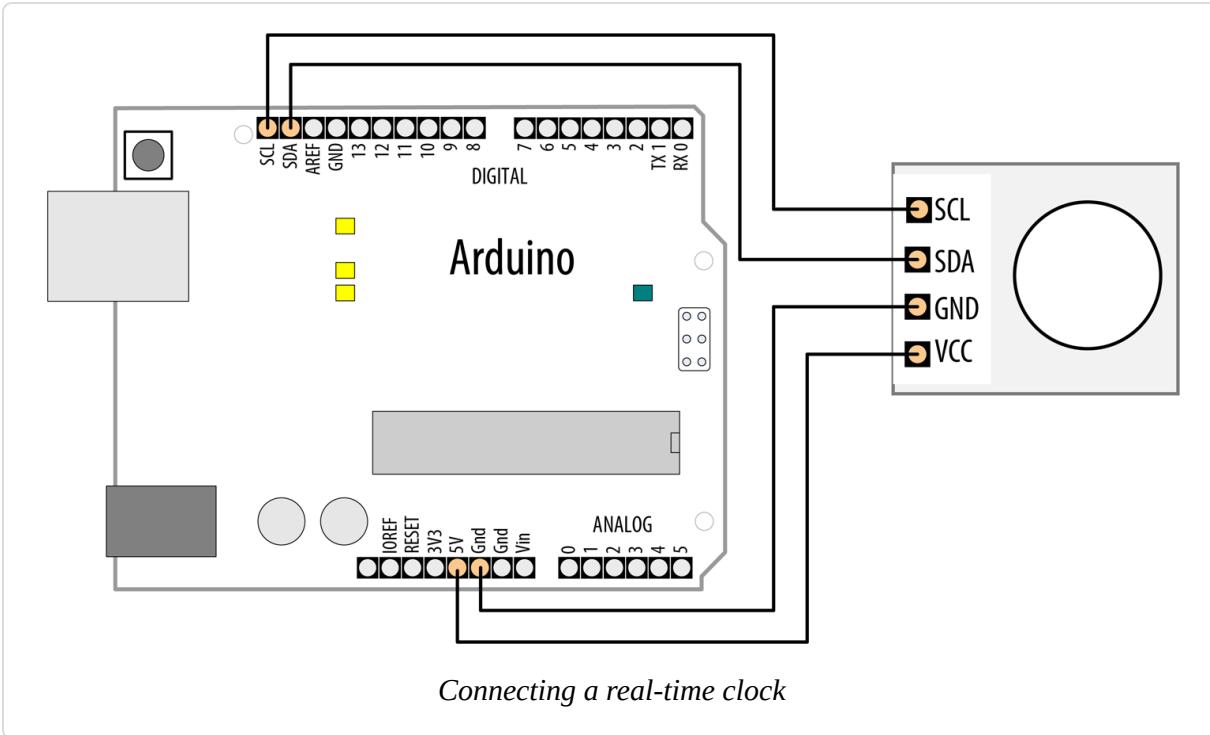
Most RTC boards for Arduino use the I2C protocol for communicating (see Chapter 13 for more on I2C). Connect the line marked SCL (or Clock) to Arduino analog pin 5 and SDA (or Data) to analog pin 4, as shown in Figure 12-3. (Analog pins 4 and 5 are used for I2C; see Chapter 13). Take care to ensure that you connect the +5V power line and GND pins correctly.

*Connecting a real-time clock*

## Discussion

The code is similar to other recipes using the Time library, but it gets its value from the RTC rather than from the serial port or hardcoded value. The only additional line needed is this:

```
setSyncProvider(RTC.get);   // the function to get the time from the RTC
```

The `setSyncProvider` function tells the Time library how it should get information for setting (and updating) the time. `RTC.get` is a method within the RTC library that returns the current time in the format used by the Time library (Unix time).

Each time Arduino starts, the `setup` function will call `RTC.get` to set the time from the RTC hardware.

Before you can get the correct time from the module, you need to set its time. Here is a sketch that enables you to set the time on the RTC hardware —you only need to do this when you first attach the battery to the RTC, when replacing the battery, or if the time needs to be changed:

```
/*
 * Set RTC time sketch
 * Set the RTC from the serial port. Simplified version of TimeSerial example
 * from the Time library.
 *
 * Set the time by sending the letter T followed by 10 digits indicating
 * number of seconds since January 1, 1970, for example T1569888000 would
 * represent 12am on October 1, 2019.
 */

#include <TimeLib.h>
#include <Wire.h>
#include <DS1307RTC.h>  // a basic DS1307 library that returns time as a
time_t

void setup()  {
  Serial.begin(9600);
  setSyncProvider(RTC.get);   // the function to get the time from the RTC
  if(timeStatus()!= timeSet)
    Serial.println("Unable to sync with the RTC");
  else
    Serial.println("RTC has set the system time");
}

void loop()
{
  if(Serial.available())
  {
    processSyncMessage();
  }
   digitalClockDisplay();
    delay(1000);
}

// Pad digits with a leading 0
String padDigits(int digit)
{
  String str = String("0") + digit; // Put a zero in front of the digit
  return str.substring(str.length() - 2); // Remove all but the
                                          // last two characters
}

void digitalClockDisplay(){
  String timestr = String(hour()) + ":" + padDigits(minute()) +
                   ":" + padDigits(second());
  Serial.println(timestr);

  String datestr = String(year()) + "-" + padDigits(month()) +
```

```
                                 "-" + padDigits(day());
   Serial.println(datestr);
}

#define TIME_HEADER  'T'   // Header tag for serial time sync message

// Parse the time message
void processSyncMessage() {

  time_t pctime = 0;

  if(Serial.find(TIME_HEADER)) {
     pctime = Serial.parseInt();
     setTime(pctime); // Set clock to the time received on serial port
     RTC.set(pctime); // Set the RTC too

  }

}
```

This sketch is almost the same as the TimeSerial sketch in Recipe 12.4 for setting the time from the serial port, but here the `RTC.set` function is also called when a time message is received from the computer to set the RTC:

```
setTime(pctime); // Set clock to the time received on serial port
RTC.set(pctime); // Set the RTC too
```

The RTC chip uses I2C to communicate with Arduino. I2C is explained in Chapter 13.

With the Adafruit_RTCLib library, you can set the RTC time using the compilation time of your sketch with
`rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));`. See thisi Adafruit page for more information.

Some of the more recent Arduino boards have RTC capability built in, including Zero, MKRZero, and MKR1000, and just need a battery backup to be connected to maintain the time when the main board is not powered.

## See Also

The SparkFun Real Time Clock Module (BOB-00099)

The Adafruit DS1307 Real Time Clock breakout board (product ID 3296)

# Communicating Using I2C and SPI

## 13.0 Introduction
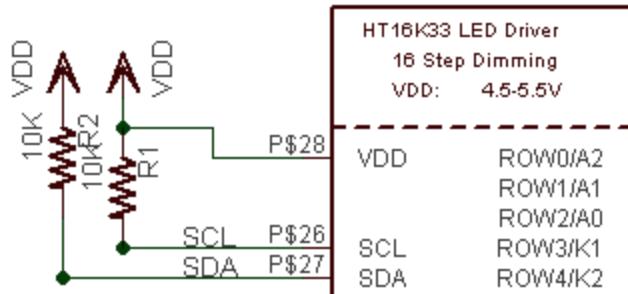
The I2C (Inter-Integrated Circuit) and SPI (Serial Peripheral Interface) standards were created to provide simple ways for digital information to be transferred between sensors and microcontrollers such as Arduino. Arduino libraries for both I2C and SPI make it easy for you to use both of these protocols.

The choice between I2C and SPI is usually determined by the devices (for example, sensors, actuators, other boards) you want to connect. Some devices provide both standards, but usually a device or chip supports one or the other.

I2C has the advantage that it only needs two signal connections (clock and data) to Arduino, while SPI needs four. With I2C, you also get acknowledgment that signals have been correctly received. The disadvantages are that the data rate is slower than SPI and data can only be traveling in one direction at a time, lowering the data rate even more if two-way communication is needed. It is also necessary to connect pull-up resistors to the connections to ensure reliable transmission of signals (see the introduction to Chapter 5 for more on pull-ups). The exact value of an I2C pull-up resistor varies depending on a number of factors, such as the length and type of wire you are using. Generally, you will probably find that 4.7K works best.

If you are connecting to an I2C device that's on a breakout board or shield, it's possible that the manufacturer has included pull-ups. You won't know for sure, so you need to check the datasheet. For example, Figure 13-1 shows a detail from Adafruit's I2C HT16K33 16 x 8 LED driver backpack breakout board (part number 1427) with the pull-ups clearly visible. Adafruit includes 10K pullups on all its boards, as it has found that those

values work well in practice. Now, if you connect two such breakout boards, you will have two 10K resistors in parallel. Applying the parallel resistance formula of `(10K*10K) / (10K+10K)`, you get 5K, which is closer to that 4.7K value. If you connect three devices with 10K resistors, you're going to get `1/(1/10K + 1/10K + 1/10K) = 3.3K`, which is still within a generally acceptable range for I2C pull-ups. For an excellent discussion of I2C pull-up resistor values, see Nick Gammon's article.



*Detail of schematic showing I2C pull-up resistors*

The advantages of SPI are that it runs at a higher data rate, and it has separate input and output connections, so it can send and receive at the same time. It uses one additional line per device to select the active device. It also uses a signal connection for a clock signal, so SPI requires four signal connections in all. This can add up to a tangle of wires if you have many devices to connect.
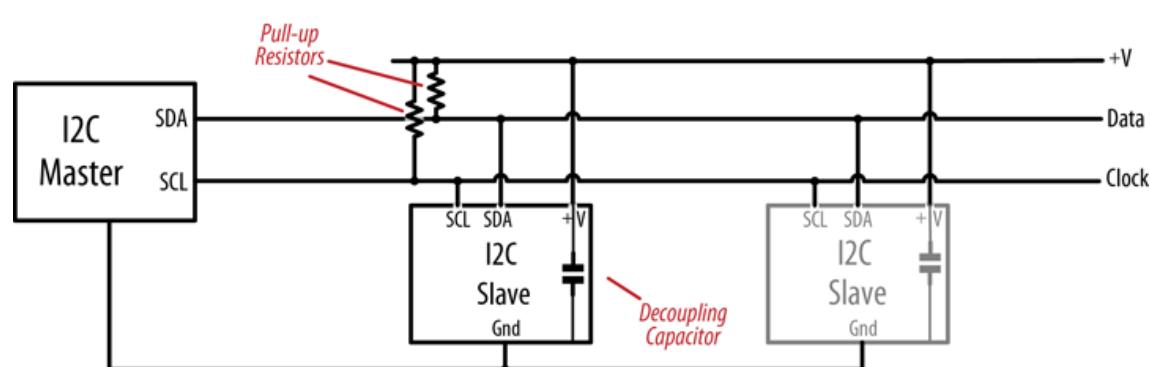
Most Arduino projects use SPI devices for high data rate applications such as Ethernet and memory cards, with just a single device attached. I2C is more typically used with sensors that don't need to send a lot of data.

This chapter shows how to use I2C and SPI to connect to common devices. It also shows how to connect two or more Arduino boards together using I2C for multi-board applications. Before we get into the recipes, let's take a look at some background on I2C and SPI, and examine the issues involved with getting 3.3V devices to work with 5V boards.

## I2C

The two connections for the I2C bus are called *SCL* (clock signal) and *SDA* (data transfer). These are available on the Arduino Uno, Zero, and compatible boards using the SCL and SDA pins (shown back in Recipe 1.2). The Arduino Nano, as well as older Unos, do not have separate pin headers for SCL and SDA pins, so you'll use analog pin 5 for SCL and analog pin 4 for SDA. (On the Mega, use digital pin 20 for SDA and pin 21 for SCL.) If you are using a different board form factor, such as the PJRC Teensy or Adafruit Feather, consult the documentation and/or datasheet for the pin numbers.

One device on the I2C bus is considered the *master* (or *primary*) device. Its job is to coordinate the transfer of information between the other devices (*slaves, secondaries*) that are attached. There must be only one master, and in most cases that is the Arduino, controlling the other chips attached to it. Figure 13-2 depicts an I2C master with multiple secondary I2C devices.



*An I2C master that coordinates one or more I2C devices*

### NOTE

I2C devices need a common ground to communicate. The Arduino GND pin must be connected to ground on each I2C device.

Slave devices are identified by their address number. Each one must have a unique address. Some I2C devices have a fixed address (an example is the nunchuck in Recipe 13.6), while others allow you to configure their address

by setting pins high or low (see Recipe 13.4) or by sending initialization commands.

I2C and SPI only define how communication takes place between devices —the messages that need to be sent depend on each individual device and what it does. You will need to consult the datasheet for your device to determine what commands are required to get it to function, and what data is required, or returned.

The Arduino Wire library hides all the low-level functionality for I2C and enables simple commands to be used to initialize and communicate with devices.

## Migrating Legacy Wire Code to Arduino 1.0 and Later

The Arduino Wire library was changed in release 1.0, and you will need to modify sketches written for previous releases to compile them in 1.0. The send and receive methods have been renamed for consistency with other libraries:

Change `Wire.send()` to `Wire.write()`.

Change `Wire.receive()` to `Wire.read()`.
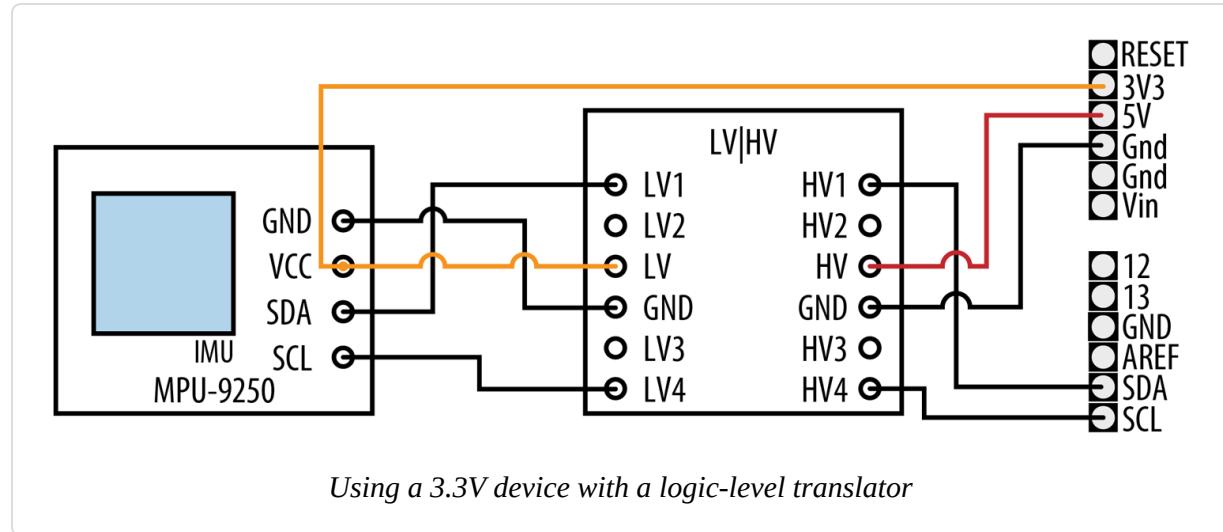
You now need to specify the variable type for literal constant arguments to write. For example:

Change `Wire.write(0x10)` to `Wire.write((byte)0x10)`.

## Using 3.3-Volt Devices with 5-Volt Boards

Many I2C devices are intended for 3.3-volt operation and can be damaged when connected to a 5-volt Arduino board. To connect these devices, you can convert the voltage levels with a bidirectional logic-level translator

such as SparkFun's BOB-12009 breakout board or Adafruit part 757 . See
Figure 13-3. The level converter board has a low-voltage (LV) side for 3.3
volts and a high-voltage (HV) side for 5 volts.



*Using a 3.3V device with a logic-level translator*
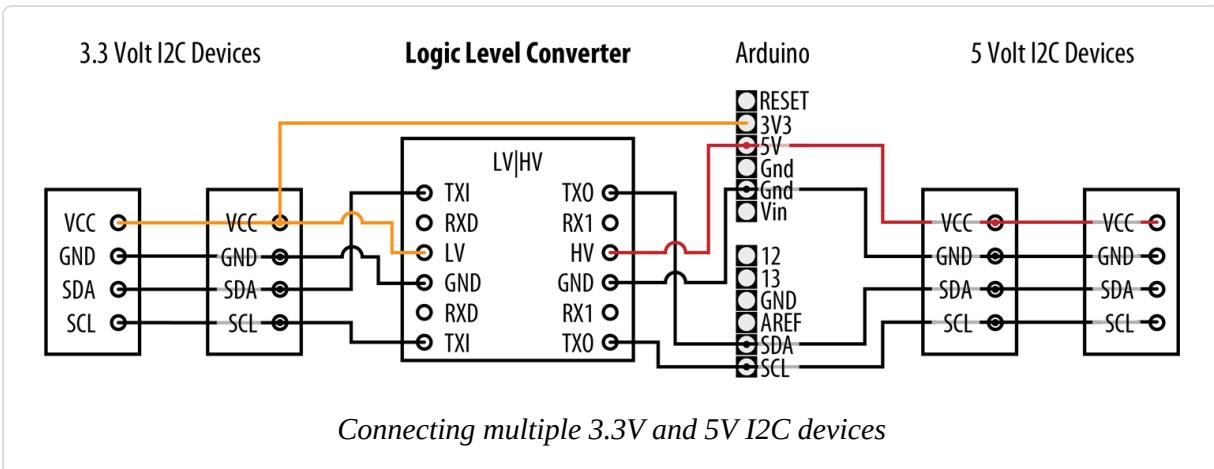
For a 3.3V I2C device, connect the LV side as follows:

- LV1 (A1 on the Adafruit board) pin to I2C SDA pin of the 3.3V device

- LV2 (or A2) pin to I2C SCL pin of the 3.3V device

- LV pin to the 3.3V device's VCC (power in) and a 3.3-volt power source
  such as the 3.3V pin on your Arduino board

- GND pin to 3.3V device's GND

Connect the HV (5V device, such as an Arduino Uno) side as follows:

- HV1 (or B1) pin to I2C SDA pin of the 5V device

- HV2 (or B2) pin to I2C SCL pin of the 5V device

- HV pin to 5V device's power pin, such as the 5V pin on your Arduino

- GND pin GND on the 5V device
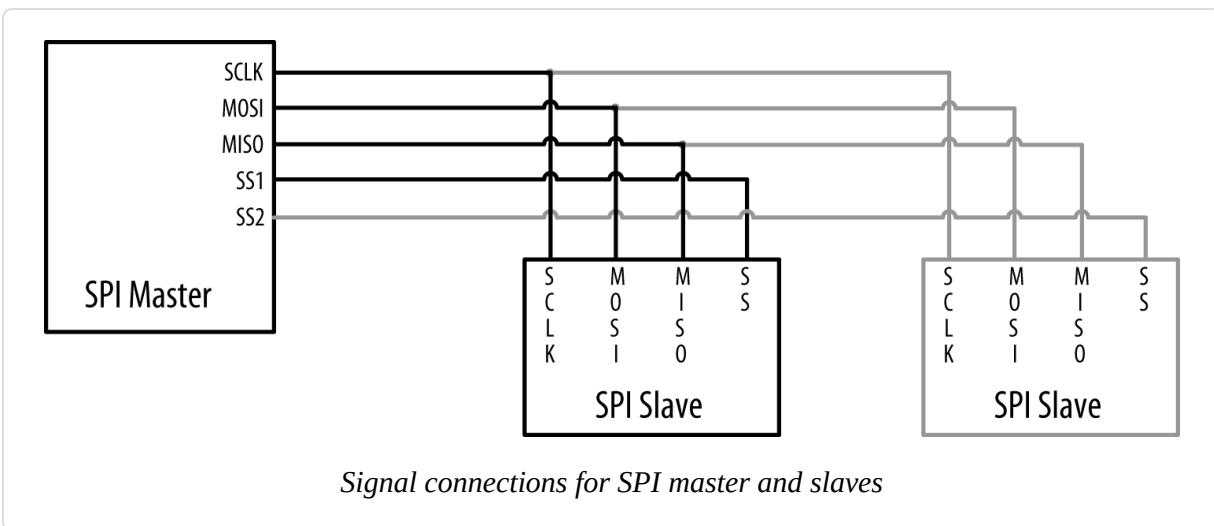
You can connect multiple I2C devices using one logic-level translator, as in
Figure 13-4.

*Connecting multiple 3.3V and 5V I2C devices*

For examples that would require a logic-level translator with 5V boards, see the recipes for the MPU-9250 in Recipes 6.15, 6.16, and 6.17.

## SPI

The Arduino IDE includes a library that allows communication with SPI devices. SPI has separate input (labeled "MOSI") and output (labeled "MISO") lines and a clock line. These three lines are connected to the respective lines on one or more slaves/secondaries, which are identified by signaling with the Slave Select (SS) line, sometimes referred to as Chip Select (CS). Figure 13-5 shows the SPI connections.



*Signal connections for SPI master and slaves*
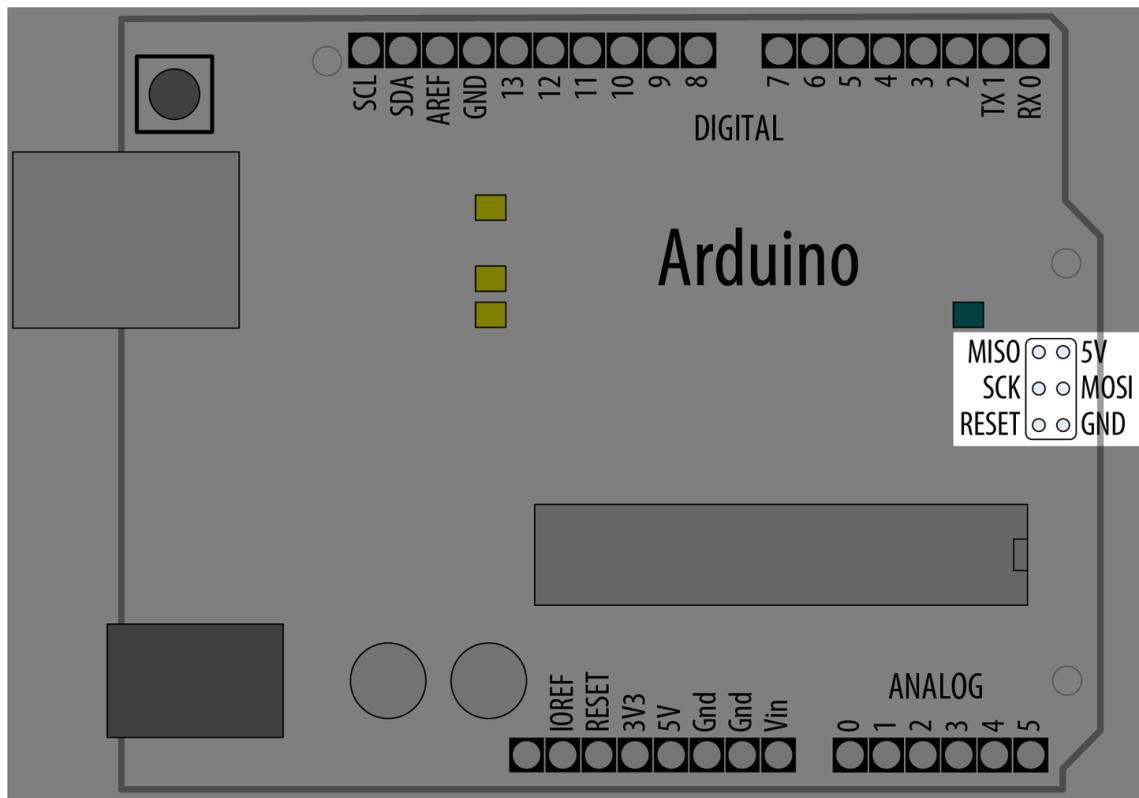
The pins to use for hardware SPI are shown in Table 13-1.

*Arduino digital pins used for SPI*

| SPI signal | Arduino Uno | Arduino Mega |
|---|---|---|
| SCLK (clock) | 13 | 52 |
| MISO (data out) | 12 | 50 |
| MOSI (data in) | 11 | 51 |
| SS/CS (slave/chip select) | 10 | 53 |

Some boards, such as SAMD-based boards like the Arduino Zero, Adafruit M0 Express, and SparkFun RedBoard Turbo, only expose hardware SPI via the ICSP header. Some 8-bit boards, such as the Leonardo, also require you to use the ICSP header. Figure 13-6 shows the connections. On those boards, you can use any digital pin (often pin 10) for SS/CS. But if you are using multiple SPI devices, you'll need to assign a different digital pin for each device's SS/CS signal (SPI devices can share SCL, MISO, and MOSI).

*SPI connections on the ICSP header*

> **NOTE**
>
> You may encounter some libraries that allow you to use a *software SPI,* which is similar in principle to software serial (see "Emulate Serial Hardware with Digital Pins") in that the SPI hardware is not used, and all the SPI operations are done in software. Like software serial, this will result in slower performance and may have some other limitations. But, if you are unable to use the hardware SPI pins for some reason, it can be very useful to have software SPI as an alternative. You can often tell when software SPI is available, because the library will have two forms of the constructor: one in which you pass only the SS/CS pin number (because the other three pins are determined by which Arduino-compatible board you use), and another in which you pass all four pins.
>
> In some cases, such as the ST77xx color LCD (see Recipe 11.10), there is an additional pin, TFT_DC, to toggle between data and command mode (the ST77xx also allows you to specify a reset pin in the constructor, but it is not part of the SPI protocol). So the hardware SPI version of the constructor is:
>
> ```
> Adafruit_ST7735(TFT_CS, TFT_DC, TFT_RST);
> ```
>
> And the software SPI version looks like this:
>
> ```
> Adafruit_ST7735(TFT_CS, TFT_DC, TFT_MOSI, TFT_SCLK,
> TFT_RST);
> ```

## See Also

This applications note comparing I2C to SPI

The Arduino Wire library reference

The Arduino SPI library reference

# 13.1 Connecting Multiple I2C Devices

## Problem

You want to connect more than one I2C device.

## Solution

The following sketch uses an air quality sensor to measure the total volatile organic compound concentration (TVOC) in parts per billion and displays it on a four-digit LED display. You must connect both the air quality sensor and the LED display controller over I2C. Figure 13-7 shows the wiring for those two I2C peripherals, as well as a four-digit LED segment display that's connected to the LED controller:

```
/*
 * Two I2C Device sketch
 * Reads an air quality sensor and displays the VOC
 * concentration on an LED display.
 */

#include <Adafruit_CCS811.h>
#include <Adafruit_GFX.h>
#include <Adafruit_LEDBackpack.h>

// Create objects for the sensor and display.
Adafruit_CCS811 ccs;
Adafruit_7segment matrix = Adafruit_7segment();

void setup()
{
  Serial.begin(9600);
  if(!ccs.begin())
  {
    Serial.println("Could not start sensor.");
    while(1); // halt
  }
  while(!ccs.available()); // Wait until the sensor is ready

  matrix.begin(0x70); // Start the matrix

}

void loop()
{
  if(ccs.available())
  {
    if(!ccs.readData())
    {
      int tvoc = ccs.getTVOC(); // Get the VOC concentration
      matrix.println(tvoc);     // Write the value
      matrix.writeDisplay();    // Update the display
    }
```
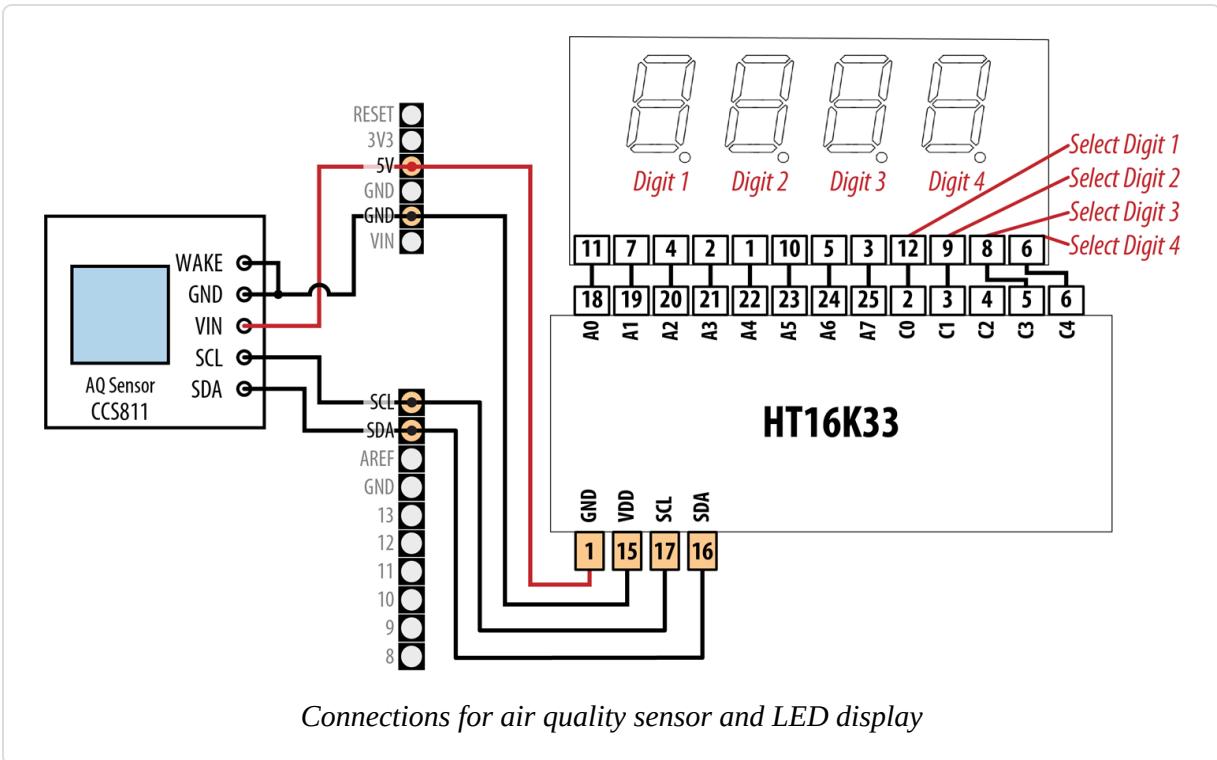
```
    }
    delay(500);
}
```



*Connections for air quality sensor and LED display*

## Discussion

The Solution uses two I2C components: the ams CCS811 air quality sensor, and the Holtek HT16K33 LED controller driver. Both of them are available on breakout boards from a variety of sources. Adafruit offers the air quality sensor as part number 3566 and the LED controller as part number 1427. SparkFun offers the air quality sensor as part number SEN-14193. The Solution uses the Adafruit board design and the Adafruit libraries (Adafruit CCS811 library and Adafruit LED Backpack library). You can install both libraries with the Arduino Library Manager.

When connecting more than one I2C device, you wire all the SDA lines together and all the SCL lines together. Each device connects to power and should have 0.1 uF decoupling capacitors unless they are integrated into the breakout board (check the datasheet or schematic for the breakout board), as is the case with the sensor and LED controller driver. The GND lines

must be connected together, even if the devices use separate power supplies (e.g., batteries).

> **WARNING**
>
> If the breakout boards include the required pull-up resistors on the I2C lines (SCL and SDA), you may not need to include them in your circuit (see Recipe 13.0). The Adafruit boards do include these, but if you are using another brand, you should check the datasheet or schematic.

The sketch initializes both devices in `setup` and repeatedly reads the TVOC concentration inside the `loop`, displaying the value on the LED display each time it gets a reading. Your LED display may have different pins, so be sure to consult the datasheet and adjust the wiring accordingly.

Both boards are 5V tolerant, so you can use them with 5V boards. If you use the CCS811 with a 3.3V board, you must power it with 3.3V instead of 5V, otherwise the voltage on its I2C pins will be too high for your board. If you want to use the HT16K33 with a 3.3V board, it gets a little more complicated because, according to its spec, that board needs at least 4.5V to run. You could use a level converter with it (see "Using 3.3-Volt Devices with 5-Volt Boards") with your 3.3V Arduino-compatible board as the low-voltage (LV) side, but some people have reported that it runs OK when powered at 3.3V, so you could try that first.

The HT16K33 expects a common cathode LED. The pinouts in Figure 13-7 are for a common 4-digit 7-segment display. A0 through A15 on the HT16K33 are used to control the seven segments as well as the decimal point. This sketch only uses A0 through A7. If you were using an LED matrix display, you would use more of the pins. Pins C0 through C7 select which digit to address. The HT16K33 rapidly switches between displaying each digit and relies on persistence of vision to make it appear that all four digits are illuminated at once.

Some of these displays have the ability to display a colon between each pair of digits, which is useful when you want to create a digital clock. The

Adafruit LED Backpack library assumes that this is connected to C4 (pin 4). But this sketch does not use it, so pin C4 is unused.

Under the hood, the Adafruit libraries use the `Wire` library to interact with the devices. For example, you could use `matrix.setBrightness(1);` to set the display to its lowest brightness (15 is the maximum). The library would issue these commands to accomplish that (0x70 is the I2C address of the HT16K33):

```
#define HT16K33_CMD_BRIGHTNESS 0xE0

Wire.beginTransmission(0x70);
Wire.write(HT16K33_CMD_BRIGHTNESS | 1);
Wire.endTransmission();
```

And when the sketch calls `ccs.readData()` followed by `ccs.getTVOC()`, the driver does something like the following (0x5A is the I2C address of the CCS811). The TVOC reading is formed by combining the third and fourth byte (remember, C arrays start at zero) into a word value:

```
uint8_t buf[8];
  Wire.beginTransmission(0x5A);
  Wire.write(0x02); // Write to register 0x02
  Wire.endTransmission();

  Wire.requestFrom(0x5A, 8); // Request 8 bytes from the CCS811
  for(int i=0; i < 8; i++)
  {
    buf[i] = Wire.read();
  }
  int tvoc = word(buf[2], buf[3]);
```

## See Also

The datasheet for the HT16K33

The datasheet for the CCS811

# 13.2 Connecting Multiple SPI Devices

**Problem**

You want to connect more than one SPI device.

## Solution

This following sketch uses an SD card reader to load bitmap images off of an SD card. Those images are displayed on a TFT display. Both of them are SPI devices. Figure 13-8 shows the connections:

```
/*
 * Two SPI Device sketch
 * Loads all the bitmaps on the attached SD card
 * and displays them on a TFT screen.
 */

#include <Adafruit_GFX.h>
#include <Adafruit_ILI9341.h>
#include <SdFat.h>
#include <Adafruit_ImageReader.h>

#define SD_CS    4 // Chip select for SD reader
#define TFT_CS  10 // Chip select for TFT
#define TFT_DC   9 // Data/command pin for TFT
#define TFT_RST  8 // Reset pin for TFT

// Create the objects for each of the SPI devices
SdFat SD;
Adafruit_ILI9341 tft = Adafruit_ILI9341(TFT_CS, TFT_DC, TFT_RST);

SdFile root; // Root directory of SD card
Adafruit_ImageReader reader(SD); // Object to load and display images

void setup(void)
{
  Serial.begin(9600);
  if(!SD.begin(SD_CS, SD_SCK_MHZ(25))) // Start the SD card reader at 25 MHz
  {
    Serial.println("Could not initialize SD card");
    while(1); // halt
  }

  tft.begin(); // Initialize the TFT
```
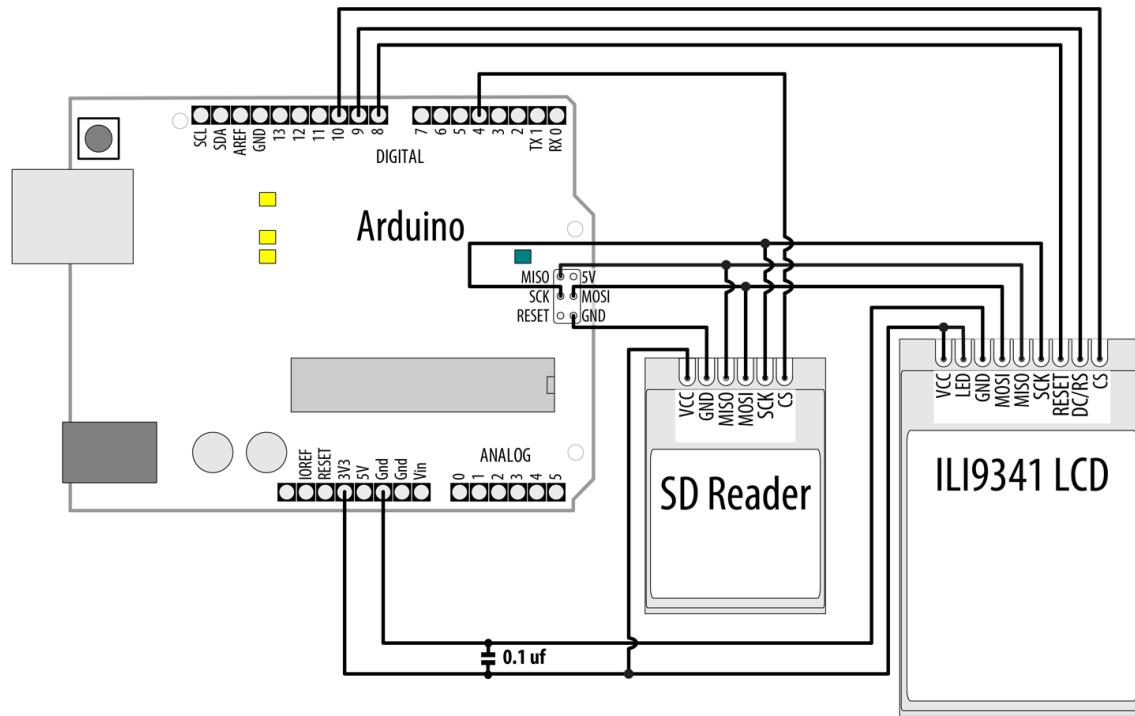
```cpp
  if (!root.open("/"))
  {
    Serial.println("Could not read SD card directory");
    while(1); // halt
  }
}

void loop()
{
  ImageReturnCode rc; // Return code from image operations
  SdFile file; // Current file
  char filename[256]; // Buffer for filename

  while (file.openNext(&root, O_RDONLY)) // Find the next file on the SD card
  {
    file.getName(filename, sizeof(filename)/ sizeof(filename[0]));
    if(isBMP(filename)) // If it's a BMP, display it on the TFT
    {
      tft.fillScreen(0);
      rc = reader.drawBMP(filename, tft, 0, 0);
      delay(2000); // Pause
    }
    file.close();
  }
  root.rewind(); // Go back to the first file in the root directory
}

// Determines whether a file is a bitmap (BMP) file
int isBMP(char fname[])
{
  String fn = String(fname);
  fn.toLowerCase();
  return fn.endsWith("bmp");
}
```

*SPI connections for SD card reader and LCD panel*

## Discussion

The Solution uses an ILI9341-based TFT LCD display and an SD card reader breakout board, both of which are SPI devices. You can find both of these devices from a variety of vendors. In some cases, such as with Adafruit part number 1480, a microSD card reader is included on the TFT display breakout. In that case, you'll have fewer connections to make because the LCD and SD card breakout share the MISO, MOSI, SCK, GND, and VIN pins.

The solution uses several libraries that you'll need to install. Install the Adafruit GFX, Adafruit ILI9341, and Adafruit ImageReader libraries through the Arduino Library Manager. Although the Arduino IDE includes its own SD card library, the Adafruit ImageReader library, which is responsible for loading images from the card, uses a modified version of Bill Greiman's SdFat library, which you can find by searching for "SdFat - Adafruit Fork" in the Library Manager.

> **NOTE**
>
> SD card readers come in a variety of forms. The simplest, such as SparkFun BOB-12941, are an SD card connector soldered to a breakout board. That's possible because SD cards themselves can operate as SPI devices (you could, in a pinch, solder wires directly to the SD card). That type of reader can only operate at 3.3V. Some SD card readers, such as Adafruit part number 254, include a level shifter so you can power them with 5V and connect 5V logic pins to them.

With I2C, each device has a unique address. With SPI, each device has a chip select (CS) line that the library uses to signal that it wants to talk to that device. In the sketch, pin 4 is used as the chip select line for the SD card, and pin 10 is used for the TFT LCD display. There's also a data/command pin that the Adafruit_ILI9341 library uses to talk to the display.

The sketch sets up a number of objects: one to represent the SD card, one for the TFT display, another to represent the root directory of the filesystem on the card, and an object that loads and displays images. In the `setup` function, the sketch initializes the SD card reader and TFT display, then opens the root directory for reading. Within `loop`, the sketch calls `openNext()` to get the next file, and uses the `isBMP()` function to decide whether the file is a bitmap. If it is, the sketch displays the bitmap on the screen before pausing and moving to the next.

The bitmap images must be saved as uncompressed BMP files, 24-bit color, or the sketch will be unable to load the images.

### See Also

# 13.3 Working with an I2C Integrated Circuit
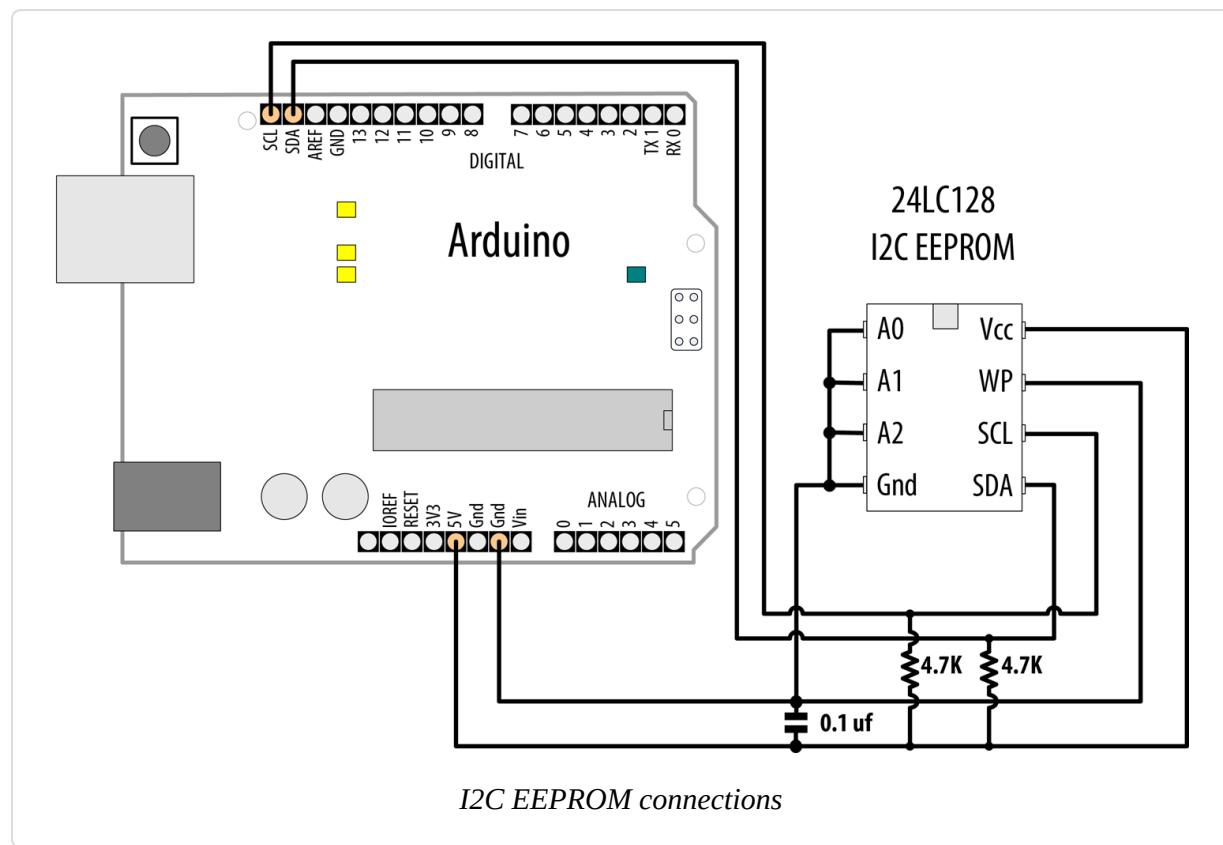
## Problem

You want to use an I2C peripheral that comes in an integrated circuit package, such as a serial EEPROM. You would use such an EEPROM

when you need more permanent data storage than Arduino has onboard, and you want to use an external memory chip to increase the capacity.

## Solution

This recipe uses the 24LC128 I2C-enabled serial EEPROM from Microchip Technology. Figure 13-9 shows the connections. If you are using a 3.3V board, connect Vcc to 3.3V instead of 5V to avoid damaging your board.



*I2C EEPROM connections*

This recipe provides functionality similar to the Arduino EEPROM library (see Recipe 18.1), but it uses an external EEPROM connected using I2C to provide greatly increased storage capacity:

```
/*
 * I2C EEPROM sketch
 * Reads data from and writes data to an 24LC128
 */
#include <Wire.h>
```

```arduino
const byte EEPROM_ID = 0x50; // I2C address for 24LC128 EEPROM

// first human-readable ASCII character '!' is number 33:
int thisByte = 33;

void setup()
{
  Serial.begin(9600);
  while(!Serial); // Required on Leonardo and most ARM-based boards
  Wire.begin();

  Serial.println("Writing 1024 bytes to EEPROM");
  for (int i=0; i < 1024; i++)
  {
    I2CEEPROM_Write(i, thisByte);
    // go on to the next character
    thisByte++;
    if (thisByte == 126)   // you could also use if (thisByte == '~')
      thisByte = 33;     // start over
  }

  Serial.println("Reading 1024 bytes from EEPROM");
  int thisByte = 33;
  for (int i=0; i < 1024; i++)
  {
    char c = I2CEEPROM_Read(i);
    if(c != thisByte)
    {
      Serial.println("read error");
      break;
    }
    else
    {
      Serial.print(c);
    }
    thisByte++;
    if(thisByte == 126)
    {
      Serial.println();
      thisByte = 33;     // start over on a new line
    }
  }
  Serial.println();
  Serial.println("Done.");
}

void loop()
{
```

```
  -

  }

  // This function is similar to Arduino's EEPROM.write()
  void I2CEEPROM_Write(unsigned int address, byte data)
  {
    Wire.beginTransmission(EEPROM_ID);
    Wire.write((int)highByte(address));
    Wire.write((int)lowByte(address));
    Wire.write(data);
    Wire.endTransmission();

    delay(5); // wait for the I2C EEPROM to complete the write cycle
  }

  // This function is similar to EEPROM.read()
  byte I2CEEPROM_Read(unsigned int address )
  {
    byte data;

    Wire.beginTransmission(EEPROM_ID);
    Wire.write((int)highByte(address));
    Wire.write((int)lowByte(address));
    Wire.endTransmission();

    Wire.requestFrom(EEPROM_ID,(byte)1);
    while(Wire.available() == 0) // wait for data
      ;
    data = Wire.read();
    return data;
  }
```

## Discussion

This recipe shows the 24LC128, which has 128K of memory; there are similar chips with higher and lower capacities (the Microchip link in this recipe's "See Also" section has a cross-reference). The chip's address is set using the three pins marked A0 through A2 and is in the range 0x50 to 0x57, as shown in Table 13-2.

*Address values for 24LC128*

| A0 | A1 | A2 | Address |
| --- | --- | --- | --- |

| A0 | A1 | A2 | Address |
| --- | --- | --- | --- |
| GND | GND | GND | 0x50 |
| +5V | GND | GND | 0x51 |
| GND | +5V | GND | 0x52 |
| +5V | +5V | GND | 0x53 |
| GND | GND | +5V | 0x54 |
| +5V | GND | +5V | 0x55 |
| +5V | +5V | GND | 0x56 |
| +5V | +5V | +5V | 0x57 |

Use of the Wire library in this recipe is similar to its use in other recipes in this chapter, so read through those for an explanation of the code that initializes and requests data from an I2C device.

The write and read operations that are specific to the EEPROM are contained in the functions `i2cEEPROM_Write` and `i2cEEPROM_Read`. These operations start with a `Wire.beginTransmission` to the device's I2C address. This is followed by a 2-byte value indicating the memory location for the read or write operation. In the write function, the address is followed by the data to be written—in this example, one byte is written to the memory location.

The read operation sends a memory location to the EEPROM, which is followed by `Wire.requestFrom(EEPROM_ID,(byte)1);`. This returns one byte of data from the memory at the address just set.

If you need to speed up writes, you can replace the 5 ms delay with a status check to determine if the EEPROM is ready to write a new byte. See the "Acknowledge Polling" technique described in Section 7 of the datasheet. You can also write data in pages of 64 bytes rather than individually; details are in Section 6 of the datasheet.

The chip remembers the address it is given and will move to the next sequential address each time a read or write is performed. If you are reading more than a single byte, you can set the start address and then perform multiple requests and receives.

> **TIP**
>
> The Wire library can read or write up to 32 bytes in a single request. Attempting to read or write more than this can result in bytes being discarded.

The pin marked WP is for setting write protection. It is connected to ground in the circuit here to enable the Arduino to write to memory. Connecting it to 5V prevents any writes from taking place. This could be used to write persistent data to memory and then prevent it from being overwritten accidentally.

## See Also

The 24LC128 datasheet

If you need to speed up writes, you can replace the 5 ms delay with a status check to determine if the EEPROM is ready to write a new byte. See the "Acknowledge Polling" technique described in Section 7 of the datasheet.

A cross-reference of similar I2C EEPROMs with a wide range of capacities

A shield is available that combines reading temperature, storing in EEPROM, and 7-segment display.
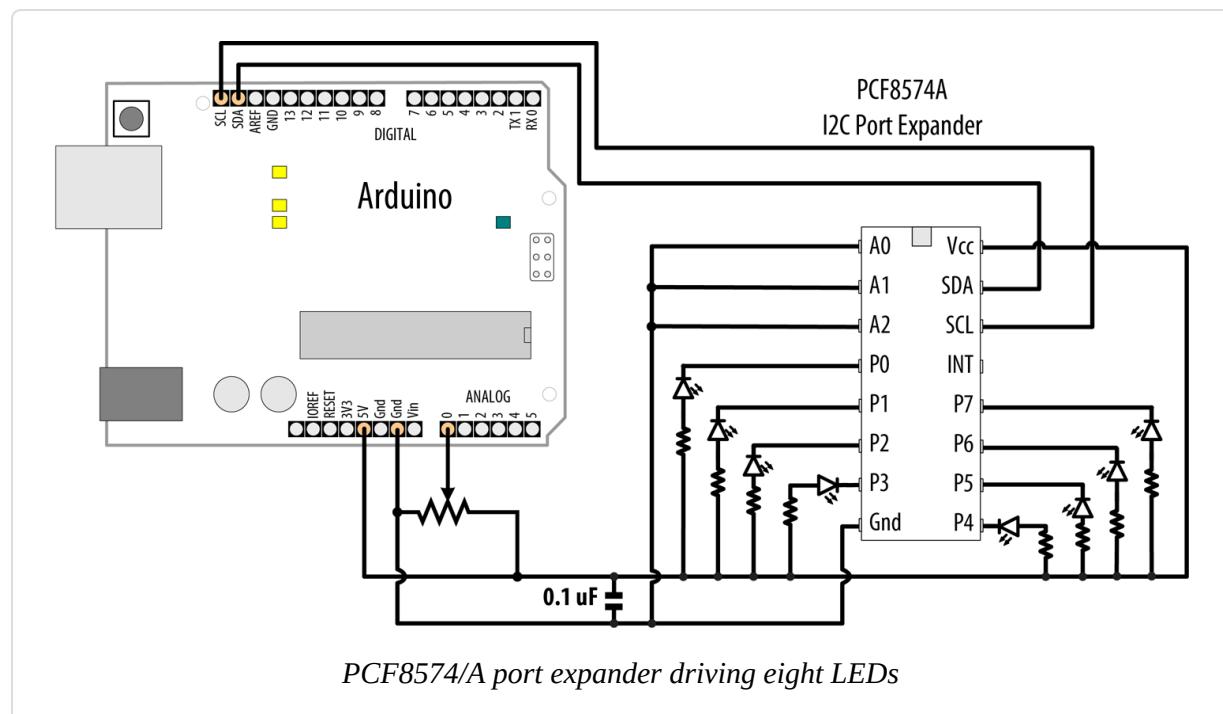
# 13.4 Increase I/O with an I2C Port Expander

## Problem

You want to use more input/output ports than your board provides.

## Solution

You can use an external port expander, such as the PCF8574 or PCF8574A, which have eight input/output pins that can be controlled using I2C. The sketch creates a bar graph with eight LEDs. Figure 13-10 shows the connections.



*PCF8574/A port expander driving eight LEDs*

---

**WARNING**

If you are using a 3.3V board, connect Vcc to 3.3V instead of 5V to avoid damaging your board.

---

The sketch has the same functionality as described in Recipe 7.6, but it uses the I2C port expander to drive the LEDs so that only two pins are required:

```
/*
 * I2C bargraph sketch
 * Uses I2C port to drive a bar graph
 * Turns on a series of LEDs proportional to a value of an analog sensor.
 * see Recipe 7.6
 */

#include <Wire.h>
```

```
const int address = 0x20; // PCF8574 address; use 0x38 for PCF8574/A
const int NbrLEDs = 8;

const int analogInPin = A0; // Analog input pin connected
                            // to a variable resistor

int sensorValue = 0;    // value read from the sensor
int ledLevel = 0;       // sensor value converted into LED 'bars'
int ledBits = 0;        //  bits for each LED will be set to 1 to turn on LED

void setup()
{
  Wire.begin(); // set up Arduino I2C support
}

void loop()
{
  sensorValue = analogRead(analogInPin);              // read the analog value
  ledLevel = map(sensorValue, 0, 1023, 0, NbrLEDs);   // map to number of LEDs
  for (int led = 0; led < NbrLEDs; led++)
  {
    Wire.beginTransmission(address);
    if (led < ledLevel)
    {
      Wire.write(~ (1 << led));
    }
    else
    {
      Wire.write(0xFF); // Turn off all LEDs
    }
    Wire.endTransmission(); // send the value to I2C
  }
}
```

## Discussion

The resistors should be 220 ohms or more (see Chapter 7 for information on selecting resistors).

The sketch reads a value from `analogRead`, then maps it to a value (`ledLevel`) between zero and the number of LEDs. Then the sketch goes into a `for` loop that iterates over each LED. If the LED's number is less than `ledLevel`, then the sketch illuminates that LED. The command to activate a pin on the PCF8574/A is a bitfield: 0b00000001 (1) would take

pin 0 high, and 0b11111111 (255) would take all the pins high. However, you don't want to take a pin high to illuminate an LED with the PCF8574/A!

The PCF8574/A has a lower capacity for driving LEDs than Arduino. Each pin can only provide (*source*) a miniscule amount of current, far less than would be needed to power an LED. However, each pin can receive (*sink*) up to 25 mA. This means that you must use inverted logic, similar to the `INPUT_PULLUP` mode (see Recipe 2.4), with the PCF8574/A. This is why each LED is tied to +5V/+3.3V instead of to GND: when one of the pins goes `LOW`, current is sourced from the positive power supply, and the pin sinks that current. This is why the sketch uses the boolean Not operator, ~, to invert the value (so 0b00000001, or 1 decimal, becomes 0b11111110, or 254 decimal).

Further, there is an additional limit: the PCF8574/A cannot sink more than 80 mA at one time. So if you were to turn all the LEDs on (0b00000000), it would probably work, but you'd exceed the limits of the chip, shortening its life. This is why the sketch only turns on one LED at a time by using boolean shift left to calculate a bitfield where only that pin is enabled, and then inverting it. You would turn on pin 0 with 0b11111110, and pin 3 with 0b11110111. This happens rapidly enough that, thanks to persistence of vision, it appears that multiple lights are illuminated at once. Although it is not strictly necessary to issue the `Wire.write(0xFF);` when encountering an LED that is not illuminated, doing so ensures that the sketch always performs the same number of commands, which keeps the LEDs at a consistent brightness regardless of how many are illuminated.

If you want to minimize flicker while still staying within the limits of the PCF8574/A, you can illuminate four LEDs at a time:

```
int bitField = 0;
  for (int led = 0; led < NbrLEDs; led++)
  {
    if (led < ledLevel)
    {
      bitField |= (1 << led);
    }
```

```
    if ((led + 1) % 4 == 0) // Send a command every four pins
    {
      Wire.beginTransmission(address);
      Wire.write(~bitField);
      Wire.endTransmission(); // send the value to I2C
      bitField = 0; // clear the bitfield
    }
  }
```

You can change the address by changing the connections of the pins marked A0, A1, and A2, as shown in Table 13-3. If you are using a PCF8574/A breakout board, it should have jumpers or solder pads for selecting the address.

*Address values for PCF8574/A*

| A0 | A1 | A2 | PCF8574A address | PCF8574 address |
|------|------|------|------------------|-----------------|
| GND | GND | GND | 0x38 | 0x20 |
| +5V | GND | GND | 0x39 | 0x21 |
| GND | +5V | GND | 0x3A | 0x22 |
| +5V | +5V | GND | 0x3B | 0x23 |
| GND | GND | +5V | 0x3C | 0x24 |
| +5V | GND | +5V | 0x3D | 0x25 |
| +5V | +5V | GND | 0x3E | 0x26 |
| +5V | +5V | +5V | 0x3F | 0x27 |

To use the port expander for input, read a byte from the expander as follows:

```
Wire.requestFrom(address, 1);
  if(Wire.available())
  {
    data = Wire.receive();
    Serial.println(data,BIN);
  }
```

## See Also

The PCF8574/A datasheet

If you need a solution that can handle more current, see Recipe 13.1.
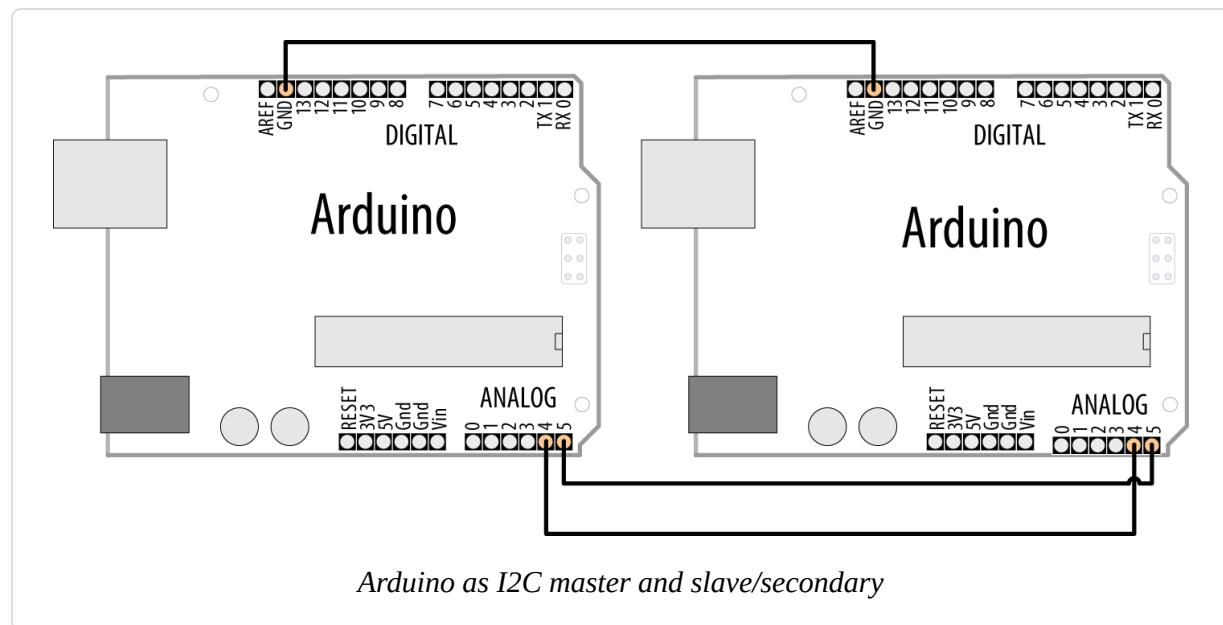
# 13.5 Communicating Between Two or More Arduino Boards

## Problem

You want to have two or more Arduino boards working together. You may want to increase the I/O capability or perform more processing than can be achieved on a single board. You can use I2C to pass data between boards so that they can share the workload.

## Solution

The two sketches in this recipe show how I2C can be used as a communications link between two or more Arduino boards. Figure 13-11 shows the connections.



*Arduino as I2C master and slave/secondary*

The master sends characters received on the serial port to an Arduino secondary using I2C:

```
/*
 * I2C Master sketch
 * Echo Serial data to an I2C secondary
 */

#include <Wire.h>

const int address = 4;  // the address to be used by the communicating devices

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

void loop()
{
  char c;
  if(Serial.available() > 0)
  {
    c = Serial.read();
    // send the data
    Wire.beginTransmission(address); // transmit to device
    Wire.write(c);
    Wire.endTransmission();
  }
}
```

The other Arduino prints characters received over I2C to its serial port:

```
/*
 * I2C Secondary sketch
 * monitors I2C requests and echoes these to the serial port
 */

#include <Wire.h>

const int address = 4;  // the address to be used by the communicating devices

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

```
  Wire.begin(address); // join I2C bus using this address
  Wire.onReceive(receiveEvent); // register event to handle requests
}

void loop()
{
  // nothing here--all the work is done in receiveEvent
}

void receiveEvent(int howMany)
{
  while(Wire.available() > 0)
  {
    char c = Wire.read(); // receive byte as a character
    Serial.write(c); // echo
  }
}
```

## Discussion

This chapter focused on Arduino as the I2C master accessing various I2C secondary devices. Here a second Arduino acts as an I2C secondary that responds to requests from another Arduino. Techniques covered in Chapter 4 for sending bytes of data can be applied here. For example, you can send data using the `print` method.

The following sketch sends its output over I2C using `Wire.println`. Using this with the I2C secondary sketch shown previously enables you to print data from the master without using the serial port (the secondary's serial port is used to display the output):

```
/*
 * I2C Master w/print sketch
 * Sends sensor data to an I2C secondary using print
 */

#include <Wire.h>

const int address = 4;      // the address to be used by the communicating
devices
const int sensorPin = A0;  // select the analog input pin for the sensor
int val;                    // variable to store the sensor value
```

```
void setup()
{
  Wire.begin();
}

void loop()
{
  val = analogRead(sensorPin);        // read the voltage on the pot
                                      // (val ranges from 0 to 1023)
  Wire.beginTransmission(address); // transmit to device
  Wire.println(val);
  Wire.endTransmission();
  delay(1000);
}
```

The next example handles multiple values as described in Recipe 4.5 over I2C instead of serial.

This sketch will send the values of the first three analog pins in a text message of the form H3, v0, v1, v2, where *H* is a character indicating the start of the message followed by the number 3, which indicates that in this example there will be three values in the message. v0, v1, v2 will be the numeric values of the three analog inputs:

```
/*
 * I2C Master multiple sketch
 * Sends multiple sensor data to an I2C secondary using print
 */

#include <Wire.h>

const int address = 4;            // address for the communicating devices
const int firstSensorPin = A0;  // first input pin of sequence
const int nbrSensors = 3;       // three sequential pins will be used
int val;                        // variable to store the sensor value

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

void loop()
{
  Wire.beginTransmission(address); // transmit to device
```

```
    Wire.print('H'); // header indicating start of a message
    Wire.print(nbrSensors);

    for (int i = 0; i < nbrSensors; i++) {
      val = analogRead(firstSensorPin + 1);   // read the sensor
      Wire.print(','); // comma separator
      Wire.print(val);
    }

    Wire.println(); // end of message
    Wire.endTransmission();
    delay(100);
}
```

This sketch handles the messages sent by the previous sketch and prints the values to the Serial Monitor:

```
/*
 * I2C Secondary multiple sketch
 * monitors I2C requests and echoes these to the serial port
 */

#include <Wire.h>

const int address = 4;   //address used by the communicating devices

void setup()
{
  Serial.begin(9600);
  Wire.begin(address); // join I2C bus using this address
  Wire.onReceive(receiveEvent); // register event to handle requests
}

void loop()
{
  // nothing here, all the work is done in receiveEvent
}

void receiveEvent(int howMany)
{
  while(Wire.available() > 0)
  {
    char c = Wire.read(); // receive byte as a character
    if( c == 'H') {
        // here if start of message
        int nbrSensors = Wire.parseInt();
```

```
      if(nbrSensors > 0) {
        for(int i=0; i < nbrSensors; i++ ) {
          int val = Wire.parseInt();
          Serial.print(val); Serial.print(" ");
        }
        Serial.println();
      }
    }
  }
}
```

## See Also

Chapter 4 has more information on using the Arduino print functionality.

# 13.6 Using the Wii Nunchuck Accelerometer
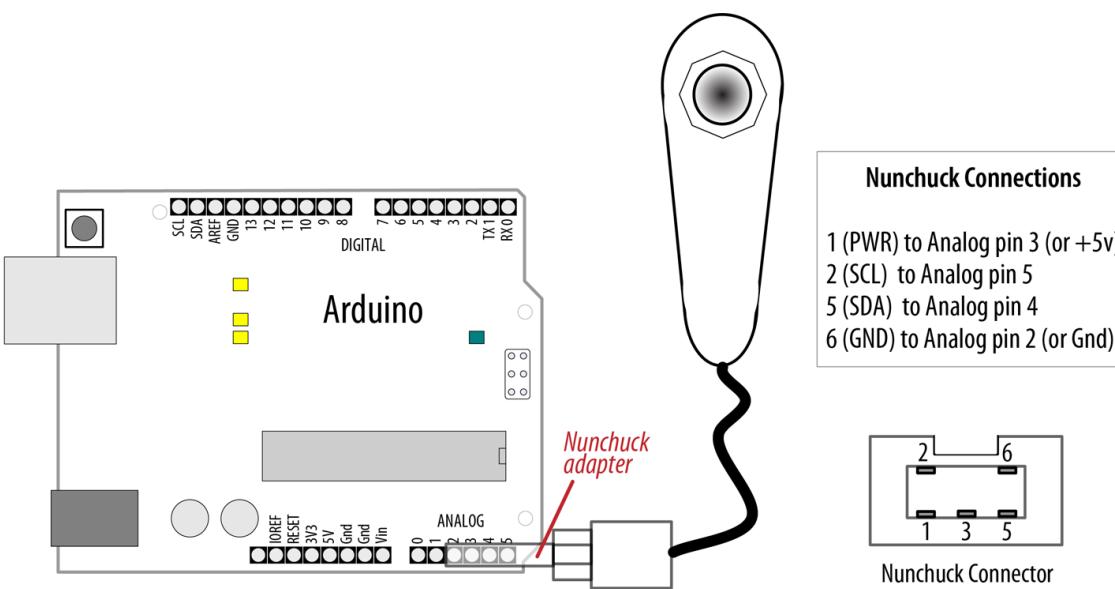
## Problem

You want to connect a Wii nunchuck to your Arduino as a convenient and fun way to use accelerometer input. The nunchuck is a popular low-cost game device that can be used to indicate the orientation of the device by measuring the effects of gravity. You can use either an original Wii nunchuck, or find a third-party clone (which will usually be much cheaper).

## Solution

The nunchuck uses a proprietary plug. If you don't need to use your nunchuck with your Wii again, you can cut the lead to connect it. Alternatively, it is possible to use a small piece of matrix board to make the connections in the plug if you are careful (the pinouts are shown in Figure 13-12) or you can buy an adapter such as the NunChucky Wii Nunchuck I2C Breakout from Solarbotics (part number 31040).

*Connecting a nunchuck to Arduino*

---

## NOTE

Adapters like the NunChucky assume you are using an Arduino board where SDA is available on pin A4 and SCL is available on pin A5. This is true for Arduino Uno and most boards based on the ATmega328 (and earlier chips such as the ATmega168). But it is not true for ARM-based Arduino boards, the Leonardo, and many others. If you are using such a board, or if you are using a board with a layout different than the Uno, you should wire the adapter's SDA and SCL pins to the corresponding pins on your board, and provide power and ground from the board's 3.3V and GND pins. If you do this, you can remove the lines of code that set `gndPin LOW` and `vccPin HIGH`.

---

Here's the Arduino sketch that sends movement data to a Processing sketch:

```
/*
 * nunchuck_lines sketch
 * sends data to Processing to draw line that follows nunchuck movement
 */

#include <Wire.h> // initialize wire
```

```
const int vccPin = A3;    // +v provided by pin 17
const int gndPin = A2;    // gnd provided by pin 16

const int dataLength = 6;          // number of bytes to request
static byte rawData[dataLength];   // array to store nunchuck data

enum nunchuckItems { joyX, joyY, accelX, accelY, accelZ, btnZ, btnC };

void setup() {
  pinMode(gndPin, OUTPUT); // set power pins to the correct state
  pinMode(vccPin, OUTPUT);
  digitalWrite(gndPin, LOW);
  digitalWrite(vccPin, HIGH);
  delay(100);  // wait for things to stabilize

  Serial.begin(9600);
  nunchuckInit();
}

void loop(){
  nunchuckRead();
  int acceleration = getValue(accelX);
  if((acceleration >= 75) && (acceleration <= 185))
  {
    //map returns a value from 0 to 63 for values from 75 to 185
    byte  x = map(acceleration, 75, 185, 0, 63);
    Serial.write(x);
  delay(20); // the time in milliseconds between redraws
  }
}

void nunchuckInit(){
  Wire.begin();                    // join i2c bus as master
  Wire.beginTransmission(0x52);// transmit to device 0x52
  Wire.write((byte)0x40);      // sends memory address
  Wire.write((byte)0x00);      // sends a zero.
  Wire.endTransmission();      // stop transmitting
}

// Send a request for data to the nunchuck
static void nunchuckRequest(){
  Wire.beginTransmission(0x52);// transmit to device 0x52
  Wire.write((byte)0x00);      // sends one byte
  Wire.endTransmission();      // stop transmitting
}

// Receive data back from the nunchuck,
// returns true if read successful, else false
```

```
bool nunchuckRead(){
  int cnt=0;
  Wire.requestFrom (0x52, dataLength); // request data from nunchuck
  while (Wire.available ()) {
    rawData[cnt] = nunchuckDecode(Wire.read());
    cnt++;
  }
  nunchuckRequest();  // send request for next data payload
  if (cnt >= dataLength)
    return true;      // success if all 6 bytes received
  else
    return false;     //failure
}

// Encode data to format that most wiimote drivers accept
static char nunchuckDecode (byte x) {
  return (x ^ 0x17) + 0x17;
}

int getValue(int item){
  if (item <= accelZ)
    return (int)rawData[item];
  else if (item  == btnZ)
    return bitRead(rawData[5], 0) ? 0: 1;
  else if (item  == btnC)
    return bitRead(rawData[5], 1) ? 0: 1;
}
```

## Discussion

I2C is often used in commercial products such as the nunchuck for communication between devices. There are no official datasheets for this device, but the nunchuck signaling was analyzed (reverse engineered) to determine the commands needed to communicate with it.

You can use the following Processing sketch to display a line that follows the nunchuck movement, as shown in Figure 13-13 (see Chapter 4 for more on using Processing to receive Arduino serial data, and also for advice on setting up and using Processing with Arduino):

```
// Processing sketch to draw line that follows nunchuck data

import processing.serial.*;
```
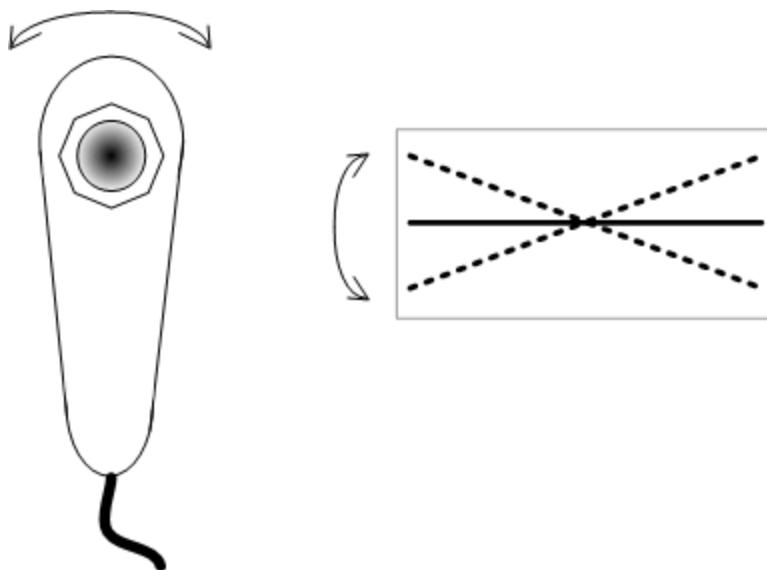
```
Serial myPort;  // Create object from Serial class
public static final short portIndex = 1;

void setup()
{
  size(200, 200);
  // Open whatever port is the one you're using - See Chapter 4
  myPort = new Serial(this,Serial.list()[portIndex], 9600);
}

void draw()
{
  if ( myPort.available() > 0) {  // If data is available,
    int y = myPort.read();        // read it and store it in val
    background(255);              // Set background to white
    line(0,63-y,127,y);          // draw the line
  }
}
```



*Nunchuck movement represented by tilted line in Processing*

The sketch includes the Wire library for I2C communication and defines the pins used to power the nunchuck:

```
#include <Wire.h> // initialize wire

const int vccPin = A3;   // +v (vcc) provided by pin 17
const int gndPin = A2;   // gnd provided by pin 16
```

*Wire.h* is the I2C library that is included with the Arduino release. `A3` is analog pin 3 (digital pin 17), and `A2` is analog pin 2 (digital pin 16); these pins provide power to the nunchuck:

```
enum nunchuckItems { joyX, joyY, accelX, accelY, accelZ, btnZ, btnC };
```

`enum` is the construct to create an enumerated list of constants, in this case a list of the sensor values returned from the nunchuck. These constants are used to identify requests for one of the nunchuck sensor values.

`setup` initializes the pins used to power the nunchuck by setting the `vccPin` `HIGH` and `gndPin` `LOW`. This is only needed if the nunchuck adapter is providing the power source. Using digital pins as a power source is not usually recommended, unless you are certain, as with the nunchuck, that the device being powered will not exceed a pin's maximum current capability (40 mA; see Chapter 5).

The function `nunchuckInit` establishes I2C communication with the nunchuck.

I2C communication starts with `Wire.begin()`. In this example, Arduino as the master is responsible for initializing the desired slave/secondary device, the nunchuck, on address 0x52.

The following line tells the Wire library to prepare to send a message to the device at hexadecimal address 52 (0x52):

```
beginTransmission(0x52);
```

> **TIP**
>
> I2C documentation typically shows addresses with hexadecimal values, so it's convenient to use this notation in your sketch.

`Wire.send` puts the given values into a buffer within the Wire library where data is stored until `Wire.endTransmission` is called to actually do the

sending.

`nunchuckRequest` and `nunchuckRead` are used to request and read data from the nunchuck.

The Wire library `requestFrom` function is used to get six bytes of data from device 0x52 (the nunchuck).

The nunchuck returns its data using six bytes as follows:

| Byte number | Description |
| --- | --- |
| Byte 1 | x-axis analog joystick value |
| Byte 2 | y-axis analog joystick value |
| Byte 3 | x-axis acceleration value |
| Byte 4 | y-axis acceleration value |
| Byte 5 | z-axis acceleration value |
| Byte 6 | Button states and least significant bits of acceleration |

`Wire.available` works like `Serial.available` (see Chapter 4) to indicate how many bytes have been received, but over the I2C interface rather than the serial interface. If data is available, it is read using `Wire.read` and then decoded using `nunchuckDecode`. Decoding is required to convert the values sent into numbers that are usable by your sketch, and these are stored in a buffer (named `rawData`). A request is sent for the next six bytes of data so that it will be ready and waiting for the next call to get data:

```
int acceleration  = getValue(accelX);
```

The function `getValue` is passed one of the constants from the enumerated list of sensors, in this case the item `accelX` for acceleration in the x-axis.

You can send additional fields by separating them using commas (see Recipe 4.4); here is the revised `loop` function to achieve this:

```
void loop(){
  nunchuckRead();
  Serial.print("H,"); // header
  for(int i=0; i < 3; i++)
  {
      Serial.print(getValue(accelX+ i), DEC);
      if( i > 2)
         Serial.write(',');
      else
         Serial.write('\n')  ;
  }
  delay(20); // the time in milliseconds between redraws
}
```

## See Also

See Recipe 16.5 for a library for interfacing with the nunchuck.

See the Discussion of Recipe 4.4 for a Processing sketch that displays a real-time bar chart showing each of the nunchuck values.

# Simple Wireless Communication

## 14.0 Introduction

Arduino's ability to interact with the world is wonderful, but sometimes you might want to communicate with your Arduino from a distance, without wires, and without the overhead of a full TCP/IP network connection. This chapter covers simple wireless modules for applications where low cost is the primary requirement as well as feature-rich options such as the versatile XBee wireless modules and Bluetooth.

Simple packet radio modules like the RFM69HCW allow secure, reliable communication between devices. XBee provides flexible wireless capability to the Arduino, but that very flexibility can be confusing. This chapter provides examples ranging from simple "wireless serial port replacements" to mesh networks connecting multiple boards to multiple sensors.

Bluetooth Classic and Bluetooth Low Energy are popular options for interfacing with computers and mobile phones. Because those devices typically have Bluetooth these days, it offers a convenient way to make a wireless connection without needing any additional special hardware on your phone or computer.

## 14.1 Sending Messages Using Low-Cost Wireless Modules
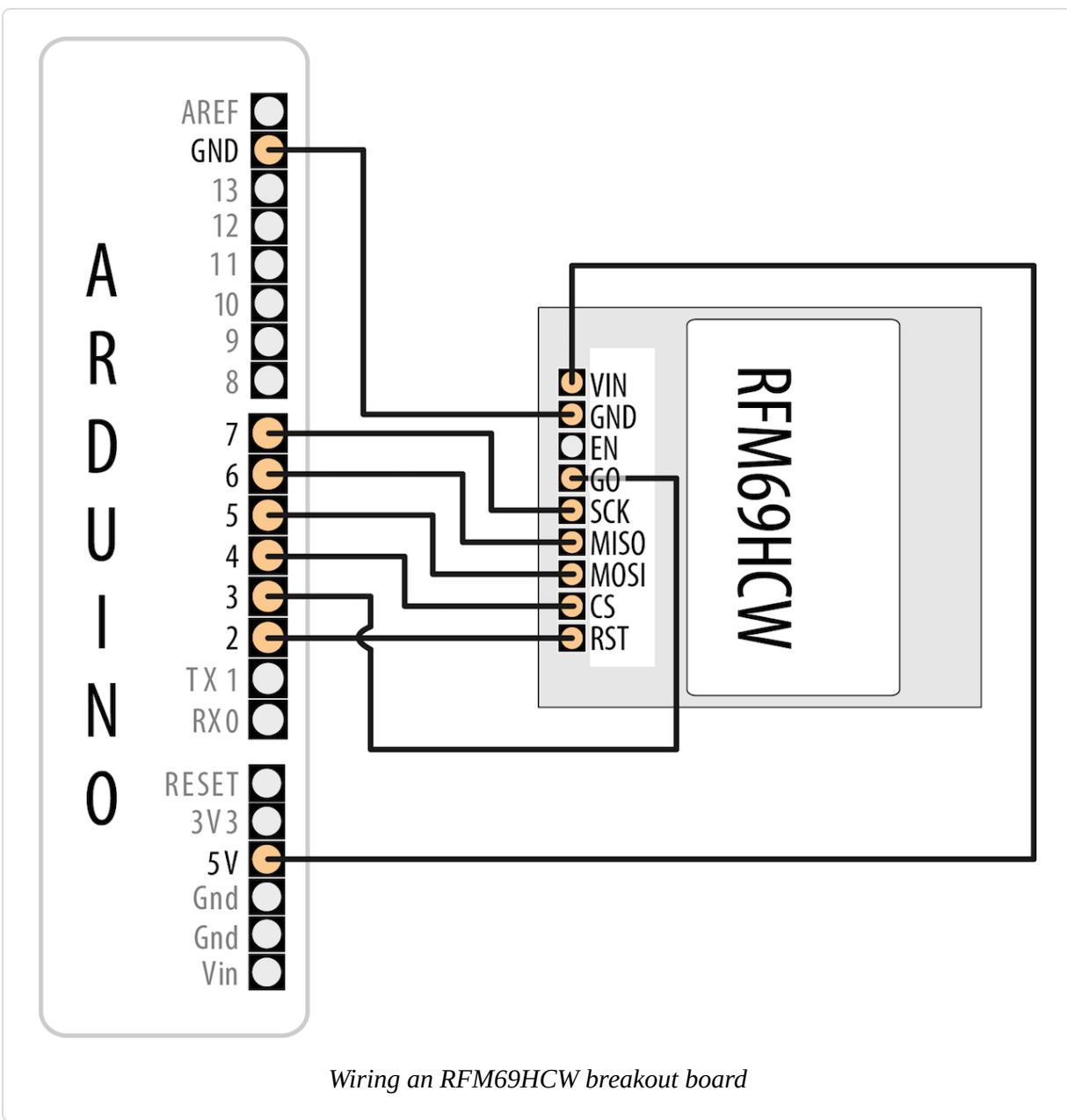
### Problem

You want to transmit data between two Arduino boards using inexpensive hardware.

## Solution

This recipe uses simple transmit and receive modules based on the RFM69HCW module, which transmits and receives in an unlicensed portion of RF spectrum called the ISM (Industrial, Scientific, and Medical) band. Depending on where you plan to use the modules, you must select the appropriate frequency. The 433 MHz modules are available on breakout boards (SparkFun WRL-12823, Adafruit 3071), and are intended for use in Region 1 (Europe, Africa, former Soviet Union, Mongolia, and the portion of the Middle East that lies west of the Persian Gulf). The 915 MHz modules (Adafruit 3070 and SparkFun WRL-12775 breakout boards) are intended for use in Region 2 (the Americas, Greenland, and a portion of the eastern Pacific Islands).

You can also obtain the bare modules, but the breakout boards include circuitry to make it easier to hook up. The Adafruit boards include level shifters so you can use them with either 3.3V or 5V logic (but if you use a 3.3V board, be sure to connect VIN to 3.3V instead of 5V. You must configure each module to use the same frequency. Wire two breakout boards as shown in Figure 14-1.

*Wiring an RFM69HCW breakout board*

The transmit sketch sends a short text message to the receive sketch, which echoes the text to the Serial Monitor and sends a reply.

The transmit and receive sketches use the RadioHead library written by Mike McCauley to provide a generalized interface to a variety of wireless hardware. Although you can download the library, both Adafruit and SparkFun have made their own customized versions available that will support their hardware slightly better. Download the Adafruit library and the SparkFun library (see Recipe 16.2). If you are using a generic module,

or a different radio that's supported by RadioHead, you should use Mike McCauley's original version unless the module vendor offers a customized version:

```
/*
 * RFM69HCW transmit sketch
 * Send a message to another module and look for a reply.
 */
#include <SPI.h>
#include <RH_RF69.h>
#include <RHReliableDatagram.h>

#define MY_ADDR   2 // Address of this node
#define DEST_ADDR 1 // The other node

#define RF69_FREQ 915.0 // Set to a supported frequency

// Define the radio driver
#define RFM69_INT   3
#define RFM69_CS    4
#define RFM69_RST   2
RH_RF69 rf69(RFM69_CS, RFM69_INT);

// This object manages message delivery
RHReliableDatagram rf69_manager(rf69, MY_ADDR);

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

  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(RFM69_RST, OUTPUT);
  digitalWrite(RFM69_RST, LOW);

  Serial.println("Resetting radio");
  digitalWrite(RFM69_RST, HIGH); delay(10);
  digitalWrite(RFM69_RST, LOW); delay(10);

  if (!rf69_manager.init())
  {
    Serial.println("Could not start the radio");
    while (1); // halt
  }

  if (!rf69.setFrequency(RF69_FREQ)) {
    Serial.println("Could not set frequency");
```

```
    while (1); // halt
  }

  // If you are using a high power version of the RF69 (RFM69HW/HCW),
  // the following is required:
  rf69.setTxPower(20, true);  // Power range is from 14-20

  // Each node must use the same key.
  uint8_t key[] = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08};
  rf69.setEncryptionKey(key);

  Serial.print("RFM69 radio running at ");
  Serial.print((int)RF69_FREQ); Serial.println(" MHz");
}

byte response[RH_RF69_MAX_MESSAGE_LEN]; // Contains message from other device
byte message[] = "Hello!";
void loop()
{
  delay(1000); // Wait 1 second

  if (rf69_manager.sendtoWait((byte *)message, strlen(message), DEST_ADDR))
  {
    byte len = sizeof(response);
    byte sender; // Sender ID

    // Wait for a reply
    if (rf69_manager.recvfromAckTimeout(response, &len, 2000, &sender))
    {
      response[len] = 0; // Add a nul (0) to end of response

      Serial.print("Got ["); Serial.print((char *) response);
      Serial.print("] from "); Serial.println(sender);

      // Blink the LED
      digitalWrite(LED_BUILTIN, HIGH); delay(250);
      digitalWrite(LED_BUILTIN, LOW); delay(250);
    }
    else
    {
      Serial.print("Received no reply from "); Serial.println(sender);
    }
  }
  else
  {
    Serial.print("Failed to send message to "); Serial.println(DEST_ADDR);
```

```
    }
  }
```

The receive sketch is identical to the transmit sketch except for two parts. First, you do not need to define `DEST_ADDR`, and you need to change the definition of `MY_ADDR` to 1:

```
/*
 * RFM69HCW transmit sketch
 * Receive a message from another module and send a reply.
 */
#include <SPI.h>
#include <RH_RF69.h>
#include <RHReliableDatagram.h>

#define MY_ADDR 1 // Address of this node
```

Next, you need to replace the `loop` function (and the two definitions that precede it) with this version:

```
byte message[RH_RF69_MAX_MESSAGE_LEN]; // Contains message from other device
byte reply[] = "Goodbye!";
void loop()
{
  if (rf69_manager.available()) // Received a message
  {
    byte len = sizeof(message);
    byte sender; // Sender ID
    if (rf69_manager.recvfromAck(message, &len, &sender)) // Wait for a
message
    {
      message[len] = 0; // Add a nul (0) to end of message

      Serial.print("Got ["); Serial.print((char *) message);
      Serial.print("] from "); Serial.println(sender);

      // Blink the LED
      digitalWrite(LED_BUILTIN, HIGH); delay(250);
      digitalWrite(LED_BUILTIN, LOW); delay(250);

      // Reply to sender
      if (!rf69_manager.sendtoWait(reply, sizeof(reply), sender))
      {
        Serial.print("Failed to send message to "); Serial.println(sender);
```

```
        }
      }
    }
  }
```

## Discussion

The RadioHead library includes a number of drivers that support a wide variety of radios. To use the library, you need to define a radio driver (such as `RH_RF69`). Then, inside the `setup` function, you'll perform a series of initialization steps. The definition and initialization varies from radio to radio, but once the radios are up and running in your sketch, the rest of the code is generally similar from one device to another.

The sketch begins by importing SPI and two RadioHead libraries. The first (`RF_RF69`) is the driver for the RFM69 series of radios, and the second (`RHReliableDatagram`) is an API that allows reliable delivery of messages to a specific module (or node) on the network.

The sketch then defines the node address (`MY_ADDR`) of this module: 1 for the transmit node and 2 for the receive node. The transmit sketch also defines the destination address of the other node (1). After that, it defines the frequency. Change this to a value within the frequency range of your module, typically 915.0 or 433.0 depending on which ISM region you are in. You should under no circumstances use a Region 1 module (433 MHz) in a Region 2 country (such as the United States), because there, the 433 MHz frequency is reserved for licensed amateur radio transmission. Even if a vendor is willing to sell you a radio that operates in the wrong band for your location, it is up to you to understand and comply with local regulations. Be sure to consult with the vendor's documentation and your local laws. This Wikipedia article offers some background on this.

The `setup` function performs a number of initialization tasks, including configuring the pins for the built-in LED and the reset line to the radio. It then resets the radio by pulling the reset line low, then high, then low again. After that, it configures the radio.

Within the loop, the sketch sends a message to the other node with the `sendtoWait` method. It then waits for a reply from that node with the `recvfromAckTimeout` function. It then prints the message it got to the serial port, and blinks the LED.

The sketch for the receiving node is very similar, except in the `loop` function, it waits until it receives a message by repeatedly checking the `available` method. When it receives a message, it displays it to the serial port, blinks the LED, and sends a response to the sender.

If you open the Serial Monitor on the transmitting node while both nodes are up and running, you'll see this output:

```
Resetting radio
RFM69 radio running at 915 MHz
Got [Goodbye!] from 1
Got [Goodbye!] from 1
Got [Goodbye!] from 1
```

And you will see this on the receiving node:

```
Resetting radio
RFM69 radio running at 915 MHz
Got [Hello!] from 2
Got [Hello!] from 2
Got [Hello!] from 2
```

> **NOTE**
>
> If you're running either sketch on a 32-bit board, a Leonardo, or a board that doesn't reset when the serial port is opened, you can add `while(!Serial);` immediately after `Serial.begin(9600);` (see "Serial Hardware Behavior"). You won't want this line of code in a production environment, because it means the sketch would need to wait for a serial connection before it will continue running.

The RadioHead library handles the assembly of multiple bytes into packets, so sending binary data consists of passing the address of the data and the number of bytes to send.

The sketch that follows is similar to the transmit sketch in this recipe's Solution, but instead of sending a string, it fills the message buffer with a struct that contains values from reading three analog input ports using `analogRead`. The struct also includes a header at the beginning of it, which you could use to indicate the type of the message. The struct uses `unsigned short int` to represent the analog values just in case your transmitter and receiver are different architectures. An `int` is two bytes on 8-bit boards and four bytes on 32-bit boards, but an `unsigned short int` is two bytes on both architectures (see Recipe 2.2). However, each architecture has different ways of aligning structs. Without the otherwise unused `padding` struct member, the struct would be seven bytes on 8-bit boards and eight bytes on 32-bit boards, which means the data would appear corrupt on the receiving end (see Recipe 4.6).

> **NOTE**
>
> The data structures that are transmitted over the radio are defined outside the function, which ensures they are not defined in the function's stack. The RadioHead library needs this in order to be able to reliably access the memory where your data structures are located.

Here is the modified `loop` for the transmitter sketch:

```
struct sensor {
  char header = 'H';
  char padding; // ensure same alignment on 8-bit and 32-bit
  unsigned short int pin0;
  unsigned short int pin1;
  unsigned short int pin2;
} sensorStruct;
void loop()
{
  delay(1000); // Wait 1 second

  sensorStruct.pin0 = analogRead(A0);
  sensorStruct.pin1 = analogRead(A1);
  sensorStruct.pin2 = analogRead(A2);

  byte len = sizeof(sensorStruct);
```

```
    memcpy(message, &sensorStruct, len);

    if (!rf69_manager.sendtoWait((byte *)message, len, DEST_ADDR))
    {
      Serial.print("Failed to send message to "); Serial.println(DEST_ADDR);
    }
  }
}
```

And here is the modified version for the receiver sketch:

```
struct sensor {
  char header;
  char padding; // ensure same alignment on 8-bit and 32-bit
  unsigned short int pin0;
  unsigned short int pin1;
  unsigned short int pin2;
} sensorStruct;

// define anything you send over the radio channel outside the
// loop function so it's not on the stack
byte message[sizeof(sensorStruct)];
void loop()
{
  if (rf69_manager.available()) // Received a message
  {
    byte len = sizeof(message);
    byte sender; // Sender ID
    if (rf69_manager.recvfromAck(message, &len, &sender)) // Wait for a
message
    {
      memcpy(&sensorStruct, message, len);
      Serial.print("Header: "); Serial.println(sensorStruct.header);
      Serial.print("Sensor 0: "); Serial.println(sensorStruct.pin0);
      Serial.print("Sensor 1: "); Serial.println(sensorStruct.pin1);
      Serial.print("Sensor 2: "); Serial.println(sensorStruct.pin2);
    }
  }
}
```

The Serial Monitor will display the analog values on the receiver:

```
Header: H
Sensor 0: 289
Sensor 1: 288
Sensor 2: 281
```

```
Header: H
Sensor 0: 287
Sensor 1: 286
Sensor 2: 280
```

Bear in mind that the maximum buffer size for the RH_RF69 is 60 bytes long (the constant `RH_RF69_MAX_MESSAGE_LEN` is defined in the library header file).

Wireless range can be up to 500 meters or so depending on supply voltage and antenna and is reduced if there are obstacles between the transmitter and the receiver.

## See Also

LoRa is a low-power long-range networking technology that RadioHead also supports. Under optimal operating conditions, a similar LoRa module will have quite a bit more range than the radios covered in this Solution. You can find LoRa radio modules based on the RFM95W module from Adafruit (product ID 3072) and SparkFun (WRL-14916).

Download datasheets for the transmitter and receiver modules here or here.

# 14.2 Connecting Arduino over a ZigBee or 802.15.4 Network

## Problem

You'd like your Arduino to communicate over a ZigBee or 802.15.4 network.

*802.15.4* is an IEEE standard for low-power digital radios that are implemented in products such as the inexpensive XBee modules from Digi International. *ZigBee* is an alliance of companies and also the name of a standard maintained by that alliance. ZigBee is based on IEEE 802.15.4 and is a superset of it. ZigBee is implemented in many products, including certain XBee modules from Digi.

## Solution

Obtain two or more XBee modules of the same type (two Xbee 3 modules, two Series 2, two Series 1, etc.), configure them (as described in the Discussion) to communicate with one another (do this before you physically wire them to the Arduino), and hook one up to an Arduino (leave the other connected to your computer as directed in the Discussion). Figure 14-2 shows the connection between an XBee Adapter and Arduino. Notice that the Arduino's RX is connected to the XBee's TX and vice versa.

If you connect the Arduino to the XBee and run this simple sketch, the Arduino will reply to any message it receives by simply echoing what the other XBee sends it. Use the appropriate `MYSERIAL` definition for whichever board you using (see "Serial Hardware" for details):

```
/*
 * XBee Echo sketch
 * Reply with whatever you receive over the serial port
 */
```
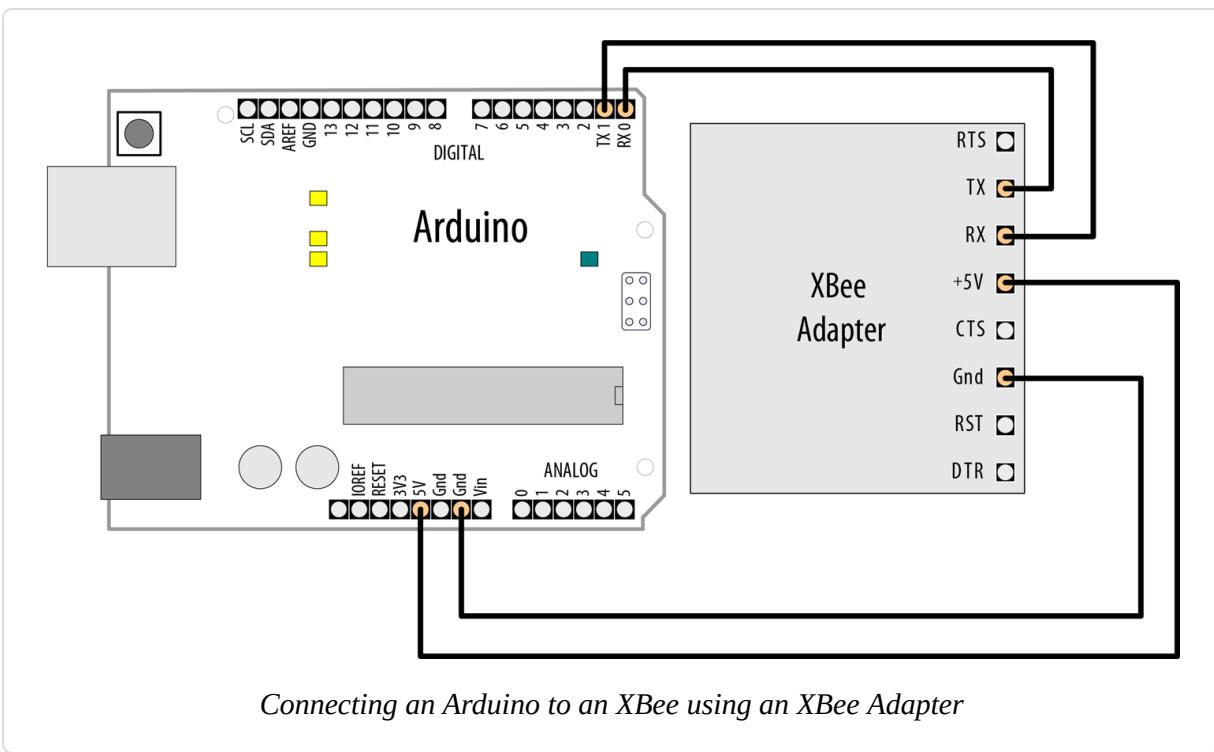
```
// Uncomment only one of the following
#define MYSERIAL Serial // Uno, Nano, and other AVR boards
//#define MYSERIAL Serial1 // Nano Every, Uno WiFi R2, Leonardo, and ARM
boards

void setup()
{
  MYSERIAL.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop()
{
  while (MYSERIAL.available() ) {
    MYSERIAL.write(MYSERIAL.read()); // reply with whatever you receive
    digitalWrite(LED_BUILTIN, HIGH); // flash LED to show activity
    delay(10);
    digitalWrite(LED_BUILTIN, LOW);
    delay(10);
  }
}
```



*Connecting an Arduino to an XBee using an XBee Adapter*

> ## WARNING
>
> If you are using an adapter that does not have an onboard voltage regulator, it will be sending voltage directly into the XBee. If this is the case, you will need to use a voltage regulator that converts 5V to 3.3V, because the Arduino's 3.3V pin cannot supply enough power to run the XBee (see the Discussion for more details).

With the XBees configured and connected to a computer and/or Arduino, you can send messages back and forth.

> ## WARNING
>
> If you are using an Arduino Uno, or any model that shares pins 0 and 1 (RX and TX) with the USB-to-Serial interface, you must disconnect the Arduino from the XBee before you attempt to program the Arduino. Otherwise, the signals will get crossed if the XBee is connected to those pins.

## Discussion

To configure your XBees, plug them into an XBee adapter such as the Parallax USB XBee Adapter (part number 32400) or the SparkFun XBee Explorer USB (WRL-11812) and connect them to a computer running Windows, macOS, or Linux. These boards can do double duty as a USB-to-Serial adapter to connect the XBee to your PC and as a breakout board to connect the XBee to a solderless breadboard (the XBee pins are not the correct size for a breadboard). If you've already wired the XBee to your Arduino, you should disconnect the four Arduino connections (5V, GND, TX, RX) before connecting to a computer via USB (you only need to connect the XBee to a computer for initial configuration).

You can find XBee breakout boards without USB, but you would need to use a separate USB-to-Serial adapter to connect the breakout to your computer. One option is to buy only one of the USB-enabled adapters and as many (cheaper) breakout boards as needed to connect your XBees to Arduino and other devices. The problem there is that cheaper breakout boards without USB often do not include a voltage regulator: you can't

power the XBee from 5V directly, and Arduino's 3.3V supply is generally not enough for the XBee, so you'd need a voltage regulator like the LD1117V33. You'll need two decoupling capacitors across the voltage regulator: 10 uF between 5V and GND, and 1 uF between 3V and GND (see Recipe 15.5 for an example of how to wire this).

If you are using a 5V board with XBee, you may want to add a level shifter between the Arduino's TX/RX pins and the XBee's, but it is generally not necessary. Robert Faludi, former chief innovator at Digi International (maker of the XBee modules), says that XBees can "operate off of a 5V signal with the Arduino," and that while a level-shifting circuit is not always needed, it is recommended in "commercial applications where you will need to keep the module stable across its entire temperature range."

> **TIP**
>
> Purchase at least two adapters, so you can have two XBees connected to your computer at the same time. These same adapters can be used to connect an XBee to an Arduino.

## XBee configuration

For the initial configuration, you will need to plug your XBees into a computer. Plug only one of them into the computer now. You may need to install a driver to use your XBee USB breakout board, so you should check the manufacturer's product web page and look for instructions and/or any driver downloads. Most use the FTDI chipset (see the discussion of FTDI drivers in Recipe 1.1).
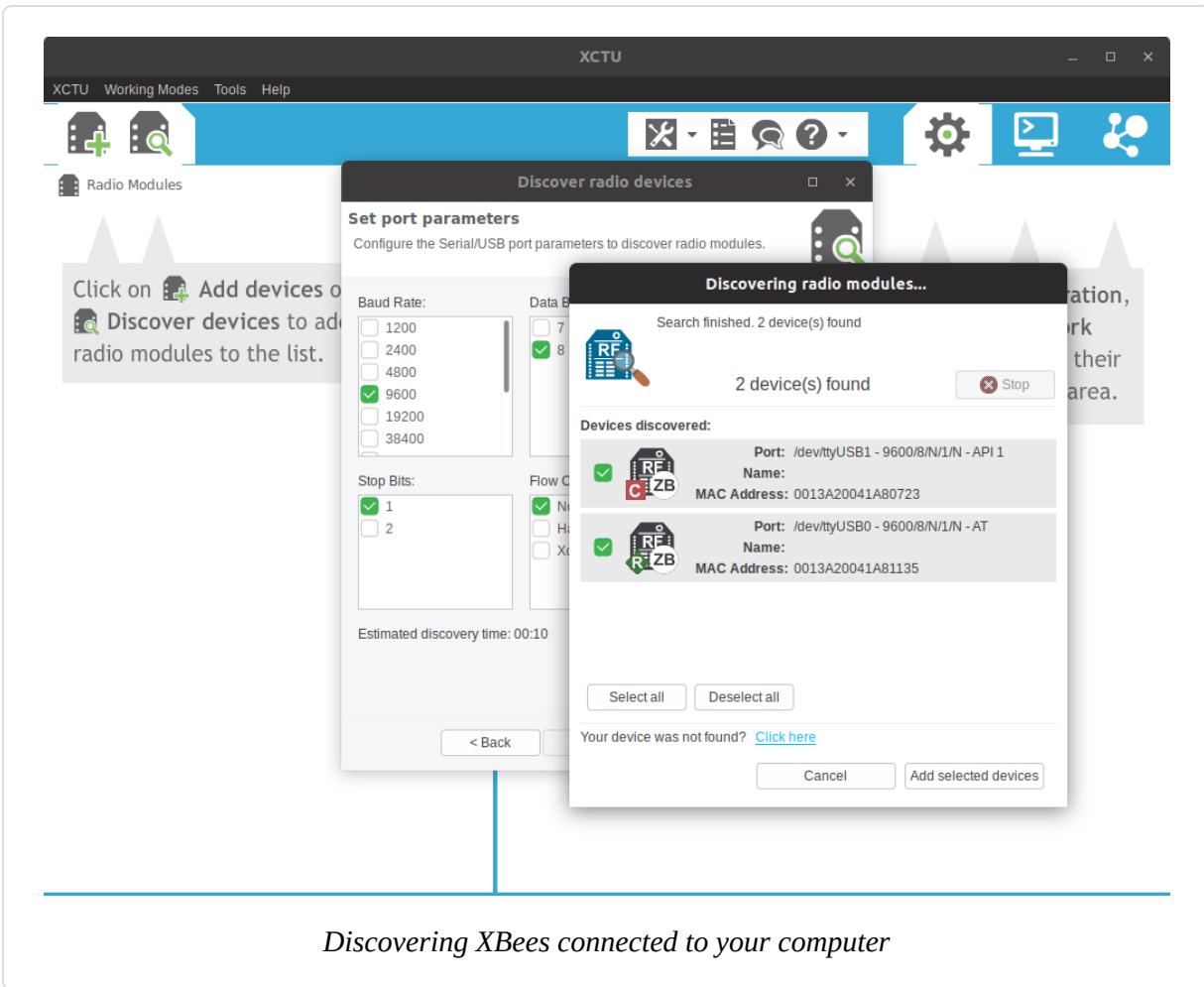
> **TIP**
>
> If you have any trouble with X-CTU, see the support document.

Before you proceed, download and install the X-CTU application, which is available for Windows, macOS, and Linux. Next, perform the following

steps for each XBee:

1. Run the X-CTU application, then click the XCTU menu and choose Discover Radio Modules. X-CTU will show you a list of serial ports on your computer. Select the one that you think your XBee is likely to be connected to (you are probably best off clicking Select All). Click Next.

2. You'll be prompted to set the port parameters. This determines how X-CTU will try to communicate with the connected devices. If you haven't reconfigured your XBee to use a different baud rate, data bits number, parity, stop bits, or flow control, you can leave these at their defaults (otherwise, choose all options that apply). Click Finish and X-CTU will scan for attached XBees.

3. X-CTU scans the serial devices. When it's done, it will show a list of XBee(s) it discovered, and they will be selected by default, as shown in Figure 14-3. Make sure you've selected the one(s) you want to configure, and click Add Selected Devices.

4. If you are connecting an older (Series 1 or 2) XBee for the first time, X-CTU will prompt you to update the firmware library (click Yes, then install the legacy firmware package). X-CTU will read the current settings of the module. For best results, you should do two things now for each of the XBee modules you just connected, and do this every time you go to reconfigure an XBee:

    a. Click to select the device, then click the Update button. This will bring up the firmware updater. Choose the newest version of the available firmware. For XBee 3, choose the Digi XBee 3 Zigbee 3.0 TH firmware. For XBee Series 2, you will use a different firmware for the first (ZigBee Coordinator AT) and for the second (ZigBee Router AT). For XBee Series 1, choose XBEE 802.15.4. Click Update. If it turns out you're already using the latest version of the firmware, you can skip this step.

    b. Click the Default button, then click Write. This will return the device to its default settings.

*Discovering XBees connected to your computer*

Label the first XBee with a piece of tape or a sticker, and number it 1 (or A, or whatever will help you remember it's the first one you configured). Next, connect the second XBee, and repeat the preceding steps (you can keep the first one plugged in). Label the second XBee 2, B, or whatever you prefer to keep track of which is which. (If you're using a Series 2 XBee, you'll install the ZigBee Coordinator AT firmware on #1 and the ZigBee Router AT firmware on #2.) Now you're ready for the final step of the configuration (see Figure 14-4):

1. Click the first XBee in X-CTU. For an XBee 3, configure the following options:

    a. **CE** Device Role: Form Network [1]

    b. **ID** Extended Pan ID: 1234 (or any hexadecimal number you want, as long as you use the same PAN ID for all devices on the

same network)

c. **DH** Destination Address High: 0

d. **DL** Destination Address Low: FFFF (this allows the coordinator XBee to broadcast to the router XBee)

For an XBee Series 2, configure the following options:

a. **ID** Pan ID: 1234

b. **DH** Destination Address High: 0

c. **DL** Destination Address Low: FFFF

For an XBee Series 1, configure these options:

a. **ID** Pan ID: 1234

b. **DH** Destination Address High: 0

c. **DL** Destination Address Low: 2222

d. **MY** 16-Bit Source Address: 1111

The `MY` command sets the identifier for an XBee. `DL` and `DH` set the low byte and the high byte of the destination XBee. `ID` sets the network ID (it needs to be the same for XBee Series 1s to talk to one another).

2. Click the Write button.

3. Click the second XBee in X-CTU. For an XBee 3, configure the following options:

a. **CE** Device Role: Join Network [0]

b. **ID** Extended Pan ID: 1234

c. **JV** Coordinator Verification: Enabled. This ensures the XBee will confirm that it's on the right channel, which makes its connection to the coordinator more reliable.

For an XBee 2, configure the following options:
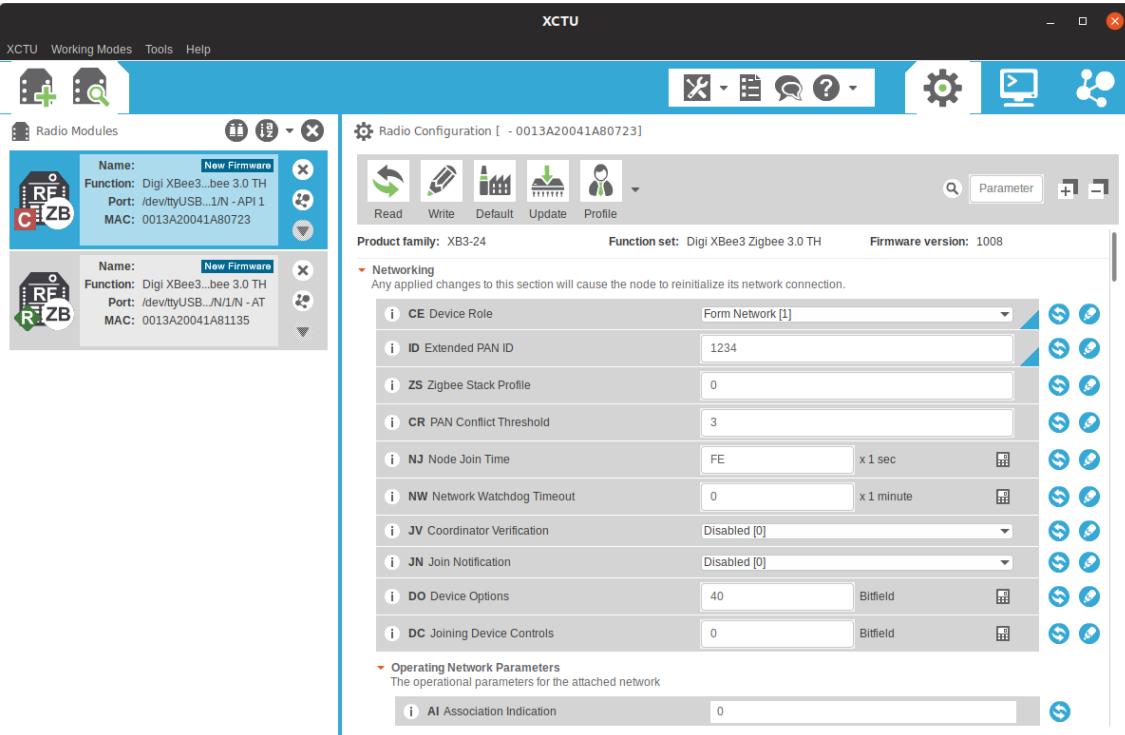
a. **ID** Pan ID: 1234

b. **JV** Channel Verification: Enabled

For an XBee Series 1, configure the following options:

  a. **ID** Pan ID: 1234

  b. **DH** Destination Address High: 0

  c. **DL** Destination Address Low: 1111

  d. **MY** 16-Bit Source Address: 2222

4. Click the Write button.

---

### TIP

If you have two computers available, you can connect each XBee to a separate computer.

---



*Configuring the XBee*

Next, click the Console icon in X-CTU or choose Working Modes→Consoles Working Mode. Click each XBee in the list on the left, and then click the Open icon in the right pane of the console window. After

you've connected both XBees, you can switch between them and type directly into the Console Log. Whatever you type on one XBee will be shown in blue text. If you click the other XBee to view its console, you'll see what you typed on the second XBee appear in red text.

**Talking to the Arduino**

Now that you've got your XBee modules configured, pick one of the XBees and close the serial terminal that was connected to it, and disconnect it from your computer. Next, program your Arduino with the code shown in this recipe's Solution, and connect the XBee to your Arduino as shown in Figure 14-2. When you type characters into the X-CTU console connected to your other XBee, you'll see the characters echoed back (if you type `a`, you'll see `aa`). The built-in LED will blink as the Arduino receives characters.

## See Also

Recipe 14.3; Recipe 14.4; Recipe 14.5

# 14.3 Sending a Message to a Particular XBee

## Problem

You want to configure which node your message goes to from your Arduino sketch.

## Solution

Send the configuration commands directly from your Arduino sketch, prefixed with the letters `AT` (attention), which is a standard way of sending control commands to devices connected via a serial port:

```
/*
 * XBee Message sketch
 * Send a message to an XBee using its address
 */
```

```
// Uncomment only one of the following
#define MYSERIAL Serial // Uno, Nano, and other AVR boards
//#define MYSERIAL Serial1 // Nano Every, Uno WiFi R2, Leonardo, and ARM
boards

bool configured;

bool configureRadio()
{
  // put the radio in command mode:
  MYSERIAL.flush();
  MYSERIAL.print("+++");
  delay(100);

  String ok_response = "OK\r"; // the response we expect.

  // Read the text of the response into the response variable
  String response = String("");
  while (response.length() < ok_response.length())
  {
    if (MYSERIAL.available() > 0)
    {
      response += (char) MYSERIAL.read();
    }
  }

  // If we got the right response, configure the radio and return true.
  if (response.equals(ok_response))
  {
    MYSERIAL.print("ATDH0013A200\r"); // destination high-REPLACE 0013A200
    delay(100);
    MYSERIAL.print("ATDL403B9E1E\r"); // destination low-REPLACE 403B9E1E
    delay(100);
    MYSERIAL.print("ATCN\r");      // back to data mode
    return true;
  }
  else
  {
    return false; // This indicates the response was incorrect.
  }
}

void setup ()
{
  MYSERIAL.begin(9600); // Begin serial
  delay(1000);
  configured = configureRadio();
}
```

```
void loop ()
{
  if (configured)
  {
    MYSERIAL.print("Hello!");
    delay(3000);
  }
  else
  {
    delay(30000);      // Wait 30 seconds
    configured = configureRadio(); // try again
  }
}
```

## Discussion

Although the configurations in Recipe 14.2 work for two XBees, they are not as flexible when used with more than two.

For example, consider a three-node network of Series 2 XBees or XBee 3s, with one XBee configured with the Coordinator AT firmware (or, in the case of the XBee 3, configured to form a network) and the other two with the Router AT firmware (or configured to join a network). Messages you send from the coordinator will be broadcast to the two routers. Messages you send from each router go to the coordinator.

The Series 1 configuration in that recipe is a bit more flexible, in that it specifies explicit destinations: by configuring the devices with AT commands and then writing the configuration, you effectively hardcode the destination addresses in the firmware.
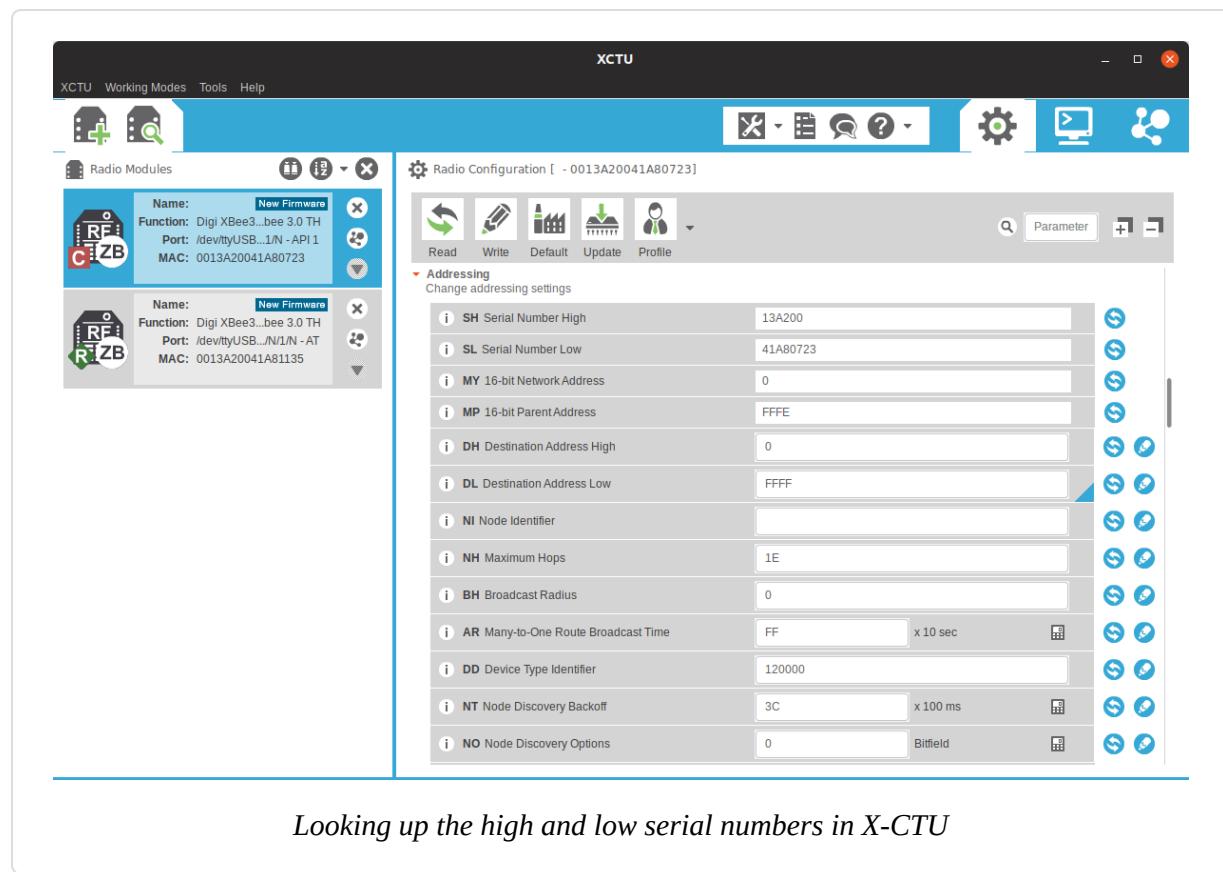
This solution instead lets the Arduino code send the AT commands to configure the XBees on the fly. The heart of the solution is the `configureRadio()` function. It sends the +++ escape sequence to put the XBee in command mode, just as the Series 1 configuration did at the end of Recipe 14.2. After sending this escape sequence, the Arduino sketch waits for the OK response before sending these AT commands:

```
ATDH0013A200
ATDL403B9E1E
```

The first two commands are similar to what is shown in the Series 1 configuration at the end of Recipe 14.2, but the numbers are longer. That's because the example shown in that recipe's Solution uses Series 2–style addresses. As you saw in Recipe 14.2, you can specify the address of a Series 1 XBee with the ATMY command, but in a Series 2 XBee, each module has a unique address that is embedded in each chip.

In your code, you must replace *0013A200* (DH) and *403B9E1E* (DL) with the high and low addresses of the destination radio. You can look up the high (ATDH) and low (ATDL) portions of the serial number using X-CTU, as shown in Figure 14-5. The numbers are also printed on the label underneath the XBee. For a Series 1 XBee use 0 for the DH, and for the DL use the MY address of the XBee you want to talk to.



*Looking up the high and low serial numbers in X-CTU*

This recipe will be most effective with a third XBee, configured as the second XBee as described in Recipe 14.2. If you are using a Series 1 XBee, be sure to give the device a unique MY ID, such as 3333. Leave this third XBee connected to your computer and switch to the Console mode in X-CTU as you did in that recipe, and click Open. When the Arduino sketch starts, you should see the message `Hello!` appear on the console.

The `ATCN` command exits command mode; think of it as the reverse of what the `+++` sequence accomplishes.

## See Also

Recipe 14.2

# 14.4 Sending Sensor Data Between XBees

## Problem

You want to send the status of digital and analog pins or control pins based on commands received from an XBee.

## Solution

Hook one of the XBees (the transmitting XBee) up to an analog sensor and configure it to read the sensor and transmit the value periodically. Connect the Arduino to an XBee (the receiving XBee) configured in API mode and read the value of the API frames that it receives from the other XBee.

## Discussion

XBees have a built-in analog-to-digital converter (ADC) that can be polled on a regular basis. The XBee can be configured to transmit the values (between 0 and 1,023) to other XBees in the network.

### Configuration

Using X-CTU, configure the devices as described in "XBee configuration", but with these changes:

1. For XBee 3:

    a. The second (transmitting) XBee is the one you configured with a device role (CE) of Join Network. In addition to the configuration you did in "XBee configuration" (CE=Join Network [0], ID=1234, JV=Enabled [1]) for the second XBee, set the following options in X-CTU and write them to the module: Under I/O Settings, set **D0** AD0/DIO0 Commissioning Button Configuration to ADC [2] and set **IR** Sampling Rate to 64 (hex for 100 ms). This is the XBee you will connect to the sensor.

2. For XBee Series 2:

    a. Instead of flashing the first XBee with the ZigBee Coordinator AT firmware, flash it with the ZigBee Coordinator API firmware. This will allow it to receive API frames. The rest of the configuration is the same (ID=1234, DL=FFFF). This is the XBee you will connect to the Arduino.

    b. The second (transmitting) XBee is the one you flashed with the ZigBee Router AT firmware (no need to flash it with the API firmware). In addition to the configuration you did in "XBee configuration" (ID=1234, JV=Enabled [1]) for the second XBee, set the following options in X-CTU and write them to the module: Under I/O Settings, set **D0** AD0/DIO0 Configuration to ADC [2] and set **IR** Sampling Rate to 64 (hex for 100 ms). This is the XBee you will connect to the sensor.

3. For Series 1:

    a. The second (transmitting) XBee is the one you configured with a 16-bit Source Address (MY) of 2222. In addition to the configuration you did in "XBee configuration" (ID: 1234, DH: 0, DL: 1111, MY: 2222) for the second XBee, set the following options in X-CTU and write them to the module: Under I/O

Settings, set **D0** DIO0 Configuration to ADC [2] and set **IR** Sampling Rate to 64 (hex for 100 ms). This is the XBee you will connect to the sensor.

---

**NOTE**

For the XBee 3 and XBee Series 2, you don't need to set the Destination Address values (DH, DL) because the router communicates with the coordinator by default. If you configure your network differently, you can configure the transmitting XBee with these additional settings to tell it which XBee to send data to:

- Destination Address High (DH): the high address (SH) of the other XBee, usually 13A200
- Destination Address Low (DL): the low address (SL) of the other XBee

---

Next, wire the receiving XBee to the Arduino as shown in Recipe 14.2. You need to also connect an LED to pin 5 and GND, and use a current-limiting resistor as described in Recipe 7.2. If your board does not support PWM on digital pin 5, change the wiring and sketch accordingly.
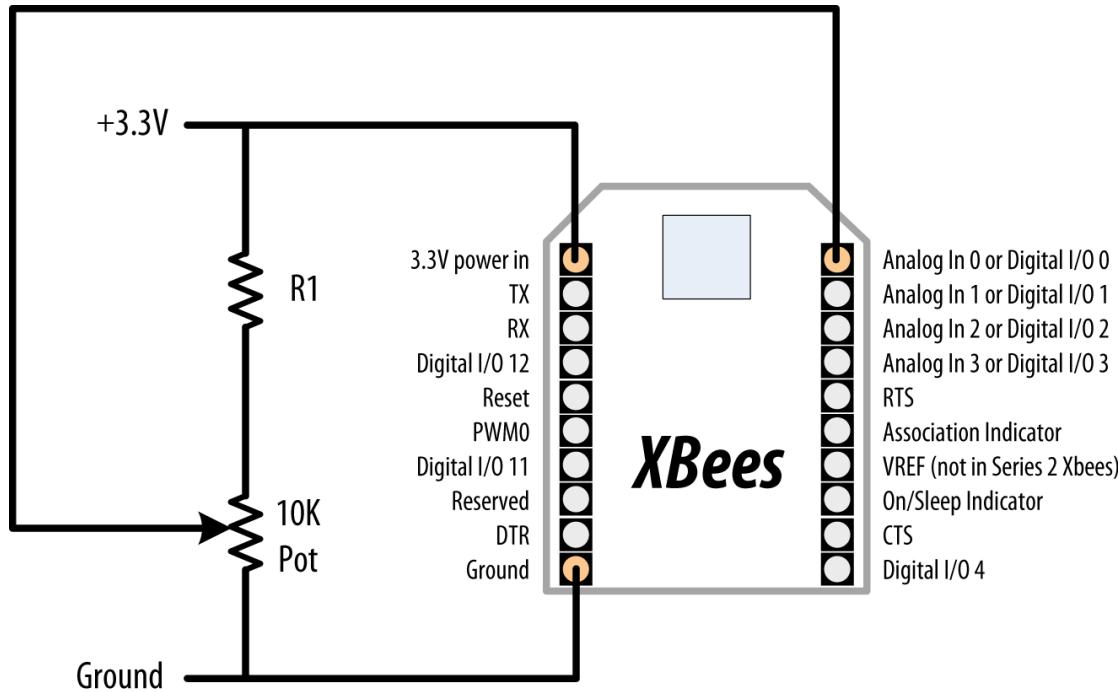
Depending on which XBee you are using, the wiring will be different. XBee Series 2 and XBee 3 use the same wiring. XBee Series 1 uses a different wiring method. The sketch code will also be slightly different. On XBee Series 2, API mode is determined by which firmware you are running. On XBee Series 1 and XBee 3, you can enter API mode with the `ATAP` command.

---

**WARNING**

Check the pinout of your XBee breakout board carefully, as the pins on the breakout board don't always match up exactly to the pins on the XBee itself. For example, on some breakout boards, the upper-left pin is GND, and the pin below it is 3.3V. Similarly, you might find that the VREF pin (labeled RES on the SparkFun XBee Explorer USB) is fifth from the bottom on the right, while it is fourth from the bottom on the XBee itself.

---

For Series 2 or XBee 3, wire up the transmitting XBee to the sensor, as shown in Figure 14-6. The value of R1 should be double whatever your

potentiometer is (if you are using a 10K pot, use a 20K resistor). This is because the Series 2 XBees' analog-to-digital converters read a range of 0 to 1.2 volts, and R1 reduces the 3.3 volts to stay below 1.2 volts.
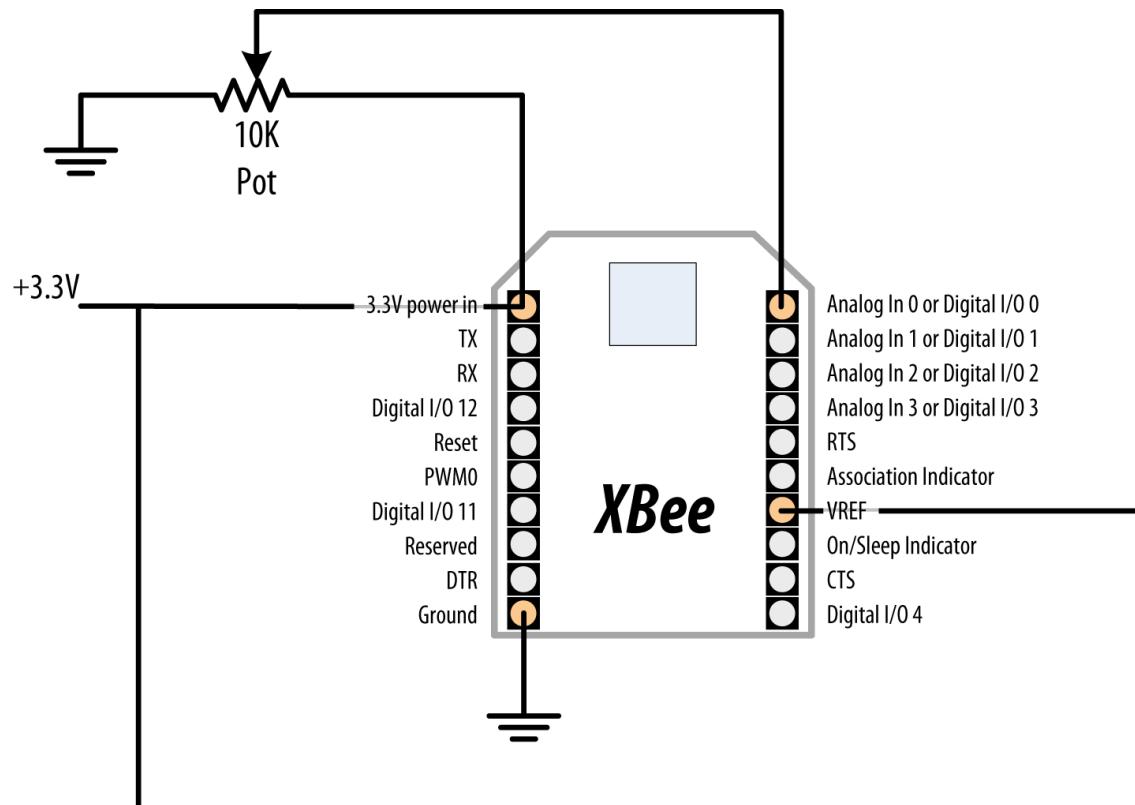


*Connecting the transmitting Series 2 XBee or XBee 3 to an analog sensor*

For a Series 1 XBee, wire up the transmitting XBee to the sensor, as shown in Figure 14-7.

---

### NOTE

Unlike Series 2 or XBee 3, Series 1 XBee uses an external reference connected to 3.3V. Because the voltage on the slider of the pot can never be greater than the reference voltage, the resistor shown in Figure 14-6 is not needed.

*The transmitting Series 1 XBee connected to an analog sensor*

For XBee Series 2, load the following sketch onto the Arduino, and wire the transmitting XBee to the Arduino as shown in Recipe 14.2. If you need to reprogram the Arduino, remember to disconnect it from the XBee first:

```
/*
 * XBeeAnalogReceive Series 2 sketch
 * Read an analog value from an XBee API frame and set the brightness
 * of an LED accordingly.
 */

// Uncomment only one of the following
#define MYSERIAL Serial // Uno, Nano, and other AVR boards
//#define MYSERIAL Serial1 // Nano Every, Uno WiFi R2, Leonardo, and ARM
boards

#define MIN_CHUNK 24
#define OFFSET 18

const int ledPin = A5; // Analog pin 5
```

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

void loop()
{
  if (MYSERIAL.available() >= MIN_CHUNK) // Wait until we have a mouthful of
data
  {
    if (MYSERIAL.read() == 0x7E) // Start delimiter of a frame
     {
       // Skip over the bytes in the API frame we don't care about
       for (int i = 0; i < OFFSET; i++)
       {
         MYSERIAL.read();
       }

       // The next two bytes are the high and low bytes of the sensor reading
       int analogHigh = MYSERIAL.read();
       int analogLow = MYSERIAL.read();
       int analogValue = analogLow + (analogHigh * 256);

       // Scale the brightness to the Arduino PWM range
       int brightness = map(analogValue, 0, 1023, 0, 255);

       // Light the LED
       analogWrite(ledPin, brightness);
    }
  }
}
```

For XBee Series 1 or XBee 3, load the following sketch onto the Arduino. Whenever you need to reprogram the Arduino, disconnect it from the XBee first. If you are using an XBee Series 1, comment out the MIN_CHUNK and OFFSET values for the XBee 3 and use the ones for XBee Series 1:

```
/*
 * XBeeAnalogReceive Series 1 or XBee 3 Sketch
 * Read an analog value from an XBee API frame and set the brightness
 * of an LED accordingly.
 */

// Uncomment only one of the following
```

```
#define MYSERIAL Serial // Uno, Nano, and other AVR boards
//#define MYSERIAL Serial1 // Nano Every, Uno WiFi R2, Leonardo, and ARM
boards

// Use these settings for XBee 3:
#define MIN_CHUNK 21
#define OFFSET 18

// Use these settings for XBee Series 1:
//#define MIN_CHUNK 14
//#define OFFSET 10

const int ledPin = A5;

void setup()
{
  MYSERIAL.begin(9600);
  delay(1000);
  configureRadio(); // check the return value if you need error handling
}

bool configureRadio()
{
  // put the radio in command mode:
  MYSERIAL.flush();
  MYSERIAL.print("+++");
  delay(100);

  String ok_response = "OK\r"; // the response we expect.

  // Read the text of the response into the response variable
  String response = String("");
  while (response.length() < ok_response.length())
  {
    if (MYSERIAL.available() > 0)
    {
      response += (char) MYSERIAL.read();
    }
  }

  // If we got the right response, configure the radio and return true.
  if (response.equals(ok_response))
  {
    MYSERIAL.print("ATAP1\r"); // Enter API mode
    delay(100);
    MYSERIAL.print("ATCN\r");  // back to data mode
    return true;
  }
```

```
    else
    {
      return false; // This indicates the response was incorrect.
    }
  }
}

void loop()
{
  if (MYSERIAL.available() >= MIN_CHUNK) // Wait until we have a mouthful of
data
  {
    if (MYSERIAL.read() == 0x7E) // Start delimiter of a frame
    {
     // Skip over the bytes in the API frame we don't care about
     for (int i = 0; i < OFFSET; i++)
     {
       MYSERIAL.read();
     }
     // The next two bytes are the high and low bytes of the sensor reading
     int analogHigh = MYSERIAL.read();
     int analogLow = MYSERIAL.read();
     int analogValue = analogLow + (analogHigh * 256);

     // Scale the brightness to the Arduino PWM range
     int brightness = map(analogValue, 0, 1023, 0, 255);

     // Light the LED
     analogWrite(ledPin, brightness);
    }
  }
}
```

---

**NOTE**

On the Series 1 XBees and XBee 3, the Arduino code needs to configure the radio for API mode
with an AT command (ATAP1). On Series 2 XBees, this is accomplished by flashing the XBee with
a different firmware version. The reason for the return to data mode (ATCN) is because command
mode was entered earlier with +++ and a return to data mode to receive data is required.

---

# See Also

Recipe 14.2

# 14.5 Activating an Actuator Connected to an XBee

## Problem

You want to tell an XBee to activate a pin, which could be used to turn on an actuator connected to it, such as a relay or LED.

## Solution

Configure the XBee connected to the actuator so that it will accept instructions from another XBee. Connect the other XBee to an Arduino to send the commands needed to activate the digital I/O pin that the actuator is connected to.

## Discussion

The XBee digital/analog I/O pins can be configured for digital output. Additionally, XBees can be configured to accept instructions from other XBees to take those pins high or low. In Series 2 XBees, you'll be using the Remote AT Command feature. In Series 1 XBees, you can use the direct I/O, which creates a virtual wire between XBees.

### Series 2 and Series 3 XBees
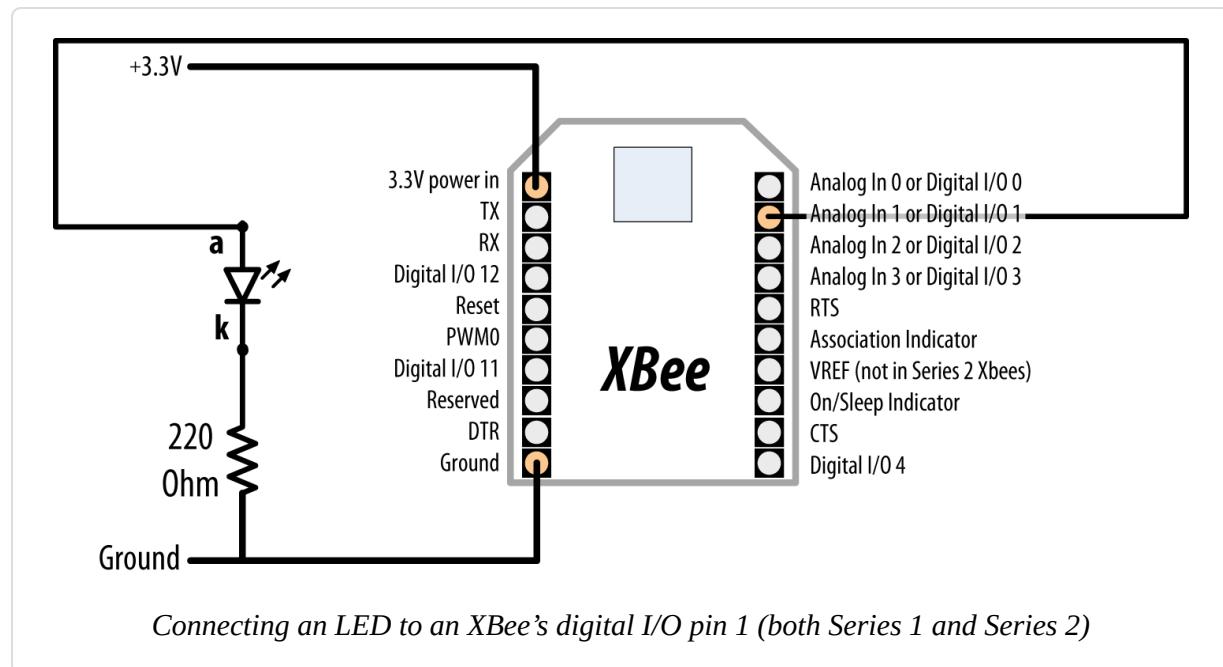
Using X-CTU (see "XBee configuration"), configure the *receiving* XBee (this is the XBee you'll connect to the LED). For XBee Series 2, flash it with the ZigBee Router AT (*not* API) function set. Next, apply the following settings:

- (XBee 3 only) **CE** Device Role: Join Network [0]

- **ID** Extended PAN ID: 1234 (or a number you pick, as long as you use the same one for both XBees)

- **JV** Channel Verification: Enabled [1]

- Under I/O Settings, **D1** DIO1/AD1/SPI_nATTN Configuration: Digital Out, Low [4]

Next, configure the *transmitting* XBee (the one you'll connect to Arduino). For XBee Series 2, make sure you flash it with the ZigBee Coordinator API (*not* AT) firmware. Next, set the following settings:

- (XBee 3 only) **CE** Device Role: Form Network [1]

- (XBee 3 only) **AP** API Enable: API Mode Without Escapes [1]

- **ID** Extended PAN ID: 1234 (or a number you pick, as long as you use the same one for both XBees)

- **DH** Destination Address High: 0

- **DL** Destination Address Low: FFFF

Wire up the receiving XBee to an LED, as shown in Figure 14-8.



*Connecting an LED to an XBee's digital I/O pin 1 (both Series 1 and Series 2)*

Next, load the following sketch onto the Arduino, and wire the transmitting XBee to the Arduino as shown in Recipe 14.2. If you need to reprogram the Arduino, remember to disconnect it from the XBee first. This sketch sends a Remote AT command (`ATD14` or `ATD15`) that sets the state of pin 1 (`ATD1`) alternatingly on (digital out high, 5) and off (digital out low, 4):

```
/*
  XBeeActuate sketch
```

```
   Send a Remote AT command to activate a digital pin on another XBee.
 */

// Uncomment only one of the following
#define MYSERIAL Serial // Uno, Nano, and other AVR boards
//#define MYSERIAL Serial1 // Nano Every, Uno WiFi R2, Leonardo, and ARM
boards

const byte frameStartByte = 0x7E;
const byte frameTypeRemoteAT  = 0x17;
const byte remoteATOptionApplyChanges = 0x02;

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

void loop()
{
  toggleRemotePin(1);
  delay(2000);
  toggleRemotePin(0);
  delay(2000);
}

byte sendByte(byte value)
{
  MYSERIAL.write(value);
  return value;
}

void toggleRemotePin(int value) // 0 = off, nonzero = on
{
  byte pin_state;
  if (value)
  {
    pin_state = 0x5;
  }
  else
  {
    pin_state = 0x4;
  }

  sendByte(frameStartByte); // Begin the API frame

  // High and low parts of the frame length (not counting checksum)
  sendByte(0x0);
  sendByte(0x10);
```

```
    long sum = 0; // Accumulate the checksum

    sum += sendByte(frameTypeRemoteAT); // Indicate this frame contains a
                                        // Remote AT command

    sum += sendByte(0x0);   // frame ID set to zero for no reply

    // The following 8 bytes indicate the ID of the recipient.
    // Use 0xFFFF to broadcast to all nodes.
    sum += sendByte(0x0);
    sum += sendByte(0x0);
    sum += sendByte(0x0);
    sum += sendByte(0x0);
    sum += sendByte(0x0);
    sum += sendByte(0x0);
    sum += sendByte(0xFF);
    sum += sendByte(0xFF);

    // The following 2 bytes indicate the 16-bit address of the recipient.
    // Use 0xFFFE to broadcast to all nodes.
    sum += sendByte(0xFF);
    sum += sendByte(0xFF);

    sum += sendByte(remoteATOptionApplyChanges); // send Remote AT options

    // The text of the AT command
    sum += sendByte('D');
    sum += sendByte('1');

    // The value (0x4 for off, 0x5 for on)
    sum += sendByte(pin_state);

    // Send the checksum
    sendByte( 0xFF - ( sum & 0xFF));

    delay(10); // Pause to let the microcontroller settle down if needed
}
```

## Series 1 XBees

Using X-CTU, configure the *transmitting* XBee (the one you'll connect to the Arduino) as follows:
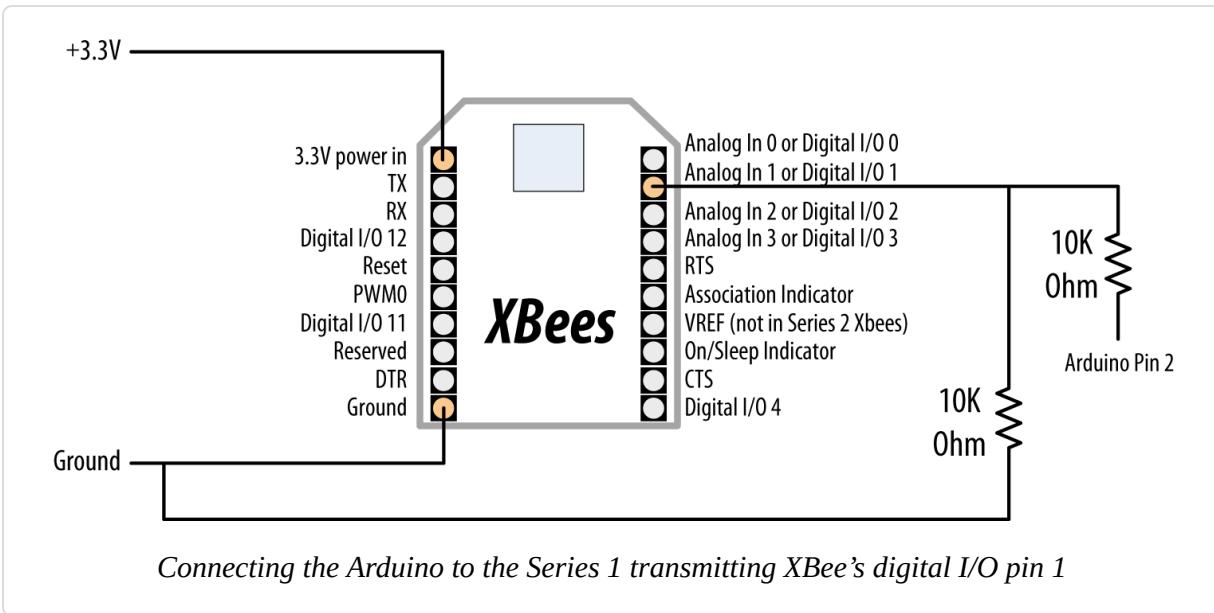
- **ID** Pan ID: 1234

- **DH** Destination Address High: 0

- **DL** Destination Address Low: 2222

- **MY** 16-Bit Source Address: 1111

- **D1** DIO1 Configuration: DI[3]. This configures the XBee's analog/digital input 1 to be in digital input mode. The state of this pin will be relayed from the transmitting XBee to the receiving XBee.

- **IC** DIO Change Detect: FF. This tells the XBee to check every digital input pin and send their values to the XBee specified by `ATDL` and `ATDH`.

Next, set the following configuration on the *receiving* XBee:

- **ID** Pan ID: 1234

- **DH** Destination Address High: 0

- **DL** Destination Address Low: 1111

- **MY** 16-Bit Source Address: 2222

- **D1** DIO1 Configuration: DO Low [4]. This configures pin 19 (analog or digital input 1) to be in low digital output mode (off by default).

- **IU** I/O Output Enable: Disabled [0]. This tells the XBee to not send the frames it receives to the serial port.

- **IA** I/O Input Address: 1111. Configures the XBee to accept commands from the other XBee (whose MY address is 1111).

Wire up the transmitting XBee to the Arduino, as shown in Figure 14-9.

*Connecting the Arduino to the Series 1 transmitting XBee's digital I/O pin 1*

Next, wire the receiving XBee to an LED, as shown in Figure 14-8. Note that instead of sending AT commands over the serial port, we're using an electrical connection to take the XBee's pin high. The two 10K resistors form a voltage divider that drops the Arduino's 5V logic to about 2.5 volts (high enough for the XBee to recognize, but low enough to avoid damaging the XBee's 3.3V logic pins).

Next, load the following sketch onto the transmitting Arduino. This sketch takes the XBee's digital I/O pin 1 alternatingly on (digital out high, 5) and off (digital out low, 4). Because the transmitting XBee is configured to relay its pin states to the receiving XBee, when its pin 1 changes state the receiving XBee's pin 1 changes as well:

```
/*
  XBeeActuateSeries1
  Activate a digital pin on another XBee.
*/

const int xbeePin = 2;

void setup() {
  pinMode(xbeePin, OUTPUT);
}

void loop()
{
```

```
    digitalWrite(xbeePin, HIGH);
    delay(2000);
    digitalWrite(xbeePin, LOW);
    delay(2000);
  }
```

## See Also

Recipe 14.2

# 14.6 Communicating with Classic Bluetooth Devices

## Problem
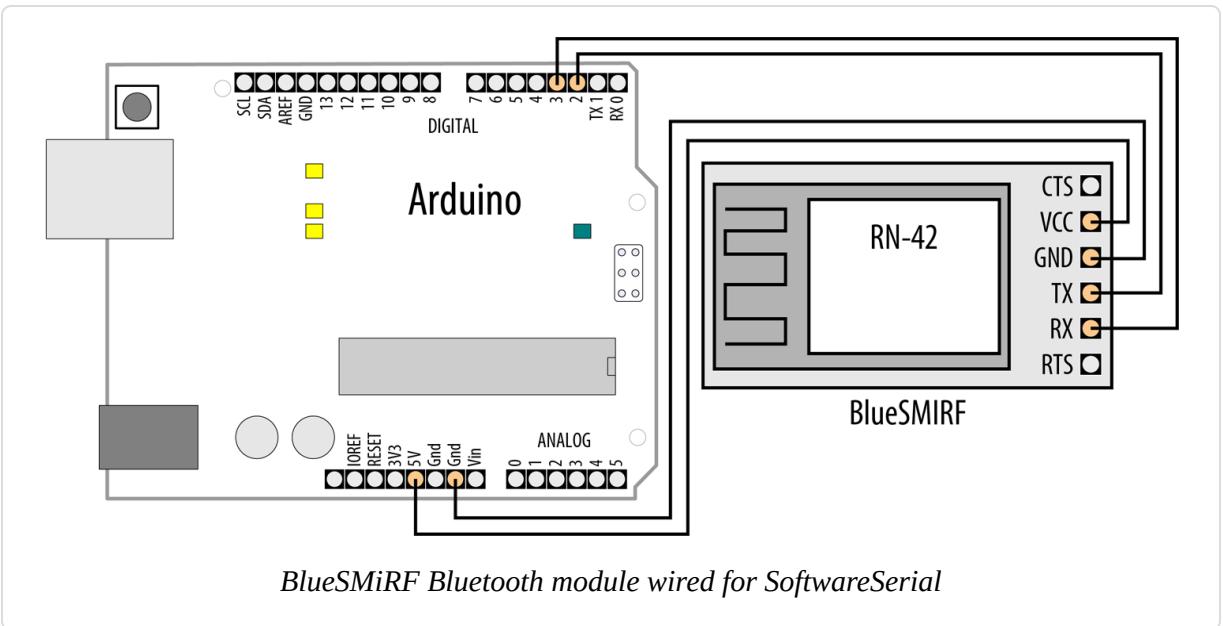
You want to send and receive information with another device using Bluetooth; for example, a laptop or cellphone.

## Solution

Connect Arduino to a Bluetooth module such as the BlueSMiRF, Bluetooth Mate, or Bluetooth Bee.

The following sketch is similar to the one in Recipe 4.11; it monitors characters received on the hardware serial port and either a software serial port or `Serial1` (wired to the Bluetooth module's TX/RX pins), so anything received on one is sent to the other.

Connect the module as shown in Figure 14-10. If you are using a board such as the Leonardo, or 32-bit boards, uncomment `#define USESERIAL1`, and connect the Bluetooth module to the appropriate RX/TX pins (pins 0 and 1 on boards with the Uno form factor):

*BlueSMiRF Bluetooth module wired for SoftwareSerial*

```
/*
 * Classic Bluetooth sketch
 * Use SoftwareSerial or Serial1 to talk to BlueSMiRF module
 * Pairing code is usually 1234
 */

//#define USESERIAL1 // Uncomment this if you're on an ARM or Leonardo
#ifdef USESERIAL1
  #define BTSERIAL Serial1
#else
  #include <SoftwareSerial.h>
  const int rxpin = 2;  // pin used to receive
  const int txpin = 3;  // pin used to send to
  SoftwareSerial mySerial(rxpin, txpin); // new serial port on given pins
  #define BTSERIAL mySerial // software serial
#endif

void setup()
{
  Serial.begin(9600);
  BTSERIAL.begin(9600); // initialize the software serial port
  Serial.println("Serial ready");
  BTSERIAL.println("Bluetooth ready");
}

void loop()
{
  if (BTSERIAL.available())
  {
```

```
      char c = (char)BTSERIAL.read();
      Serial.write(c);
    }
    if (Serial.available())
    {
      char c = (char)Serial.read();
      BTSERIAL.write(c);
    }
  }
```

## Discussion

You will need Bluetooth capability on your computer to communicate with this sketch. Both sides participating in a Bluetooth conversation need to be paired—the ID of the module connected to Arduino needs to be known to the other end. The default pin code for the BlueSMiRF is 1234. See the documentation for your computer Bluetooth to set the pairing ID and accept the connection.

After you've paired your computer with the Bluetooth module, you'll find that there are some new serial ports on your computer. In Windows, check the Ports (COM & LPT) section of Device Manager and look for Serial over Bluetooth Link ports (if more than one appears, try them both). On macOS, you can open the Terminal application, and run this command to get a list of serial ports: `ls /dev/{cu,tty}.*`. The port you want to use will look something like `/dev/cu.RN42-22AC-SPP`.

> **NOTE**
>
> All the common Bluetooth modules used with Arduino implement the Bluetooth Serial Port Profile (SPP). Once the devices are paired, the computer or phone will see the module as a serial port. These modules are not capable of appearing as other types of Bluetooth service, such as a Bluetooth mouse or keyboard.

To connect to the Bluetooth serial port on your computer, you can use PuTTY on Windows, or the screen command on Linux or macOS. To connect to a serial port at 9,600 baud with screen, open the Terminal and

type this command (replace `tty.RN42-22AC-SPP` with the name of the serial port):

```
screen /dev/cu.RN42-22AC-SPP 9600
```

Bluetooth range is between five and 100 meters, depending on whether you have class 3, 2, or 1 devices.

## See Also

This SparkFun tutorial covers the installation and use of Bluetooth.

Bluetooth Bee is a Bluetooth module that plugs into an XBee socket so you can use shields and adapters designed for XBee.

# 14.7 Communicating with Bluetooth Low Energy Devices

## Problem

You want to send and receive information with another device using Bluetooth Low Energy, a more advanced, flexible, and modern alternative to Bluetooth Classic. Bluetooth Low Energy (BLE) behaves very differently than Bluetooth Classic and solves a different set of problems. While Bluetooth Classic is good for sending relatively large amounts of data, BLE is designed for devices that send less frequent, short messages (such as sensors).

## Solution

Use one of the many Arduino boards that have built-in BLE at Bluetooth 4.0 or higher: the Nano 33 BLE, Nano 33 IoT, Uno WiFi Rev 2, and MKR WiFi 1010. Any of these will support the ArduinoBLE library, which makes it easy to incorporate Bluetooth Low Energy in your projects.

The sketch is a simplified version of one of the examples included with the ArduinoBLE library. Install the library using the Library Manager, and run

the following sketch:

```
/*
 * ArduinoBLE sketch
 * Allows control of the onboard LED over Bluetooth Low Energy
 */

#include <ArduinoBLE.h>

#define SERVICE_ID "19B10010-E8F2-537E-4F6C-D104768A1214"
#define CHAR_ID    "19B10011-E8F2-537E-4F6C-D104768A1214"

// Create the service ID and the characteristic (read-write)
BLEService ledService(SERVICE_ID);
BLEByteCharacteristic ledCharacteristic(CHAR_ID, BLERead | BLEWrite);
BLEDescriptor ledDescriptor("2901", "LED state");

void setup()
{
  Serial.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT);

  if (!BLE.begin())
  {
    Serial.println("Failed to start BLE");
    while (1); // halt
  }

  // Set the name and add the ledService as an advertised service
  BLE.setLocalName("RemoteLED");
  BLE.setAdvertisedService(ledService);

  // Add the descriptor to the characteristic
  ledCharacteristic.addDescriptor(ledDescriptor);

  // Add the characteristic to the service
  ledService.addCharacteristic(ledCharacteristic);
  BLE.addService(ledService); // Add the service to the BLE system

  ledCharacteristic.writeValue(0); // Init to 0

  BLE.advertise();
}

void loop()
{
  BLE.poll();
```

```
  if (ledCharacteristic.written())
  {
    if (ledCharacteristic.value())
    {
      digitalWrite(LED_BUILTIN, HIGH);
    }
    else
    {
      digitalWrite(LED_BUILTIN, LOW);
    }
  }
}
```

## Discussion

The sketch creates a service (`ledService`), which you can think of as representing the Arduino itself. Next, it creates a read/write characteristic (`ledCharacteristic`) to represent the onboard LED and the actions you can take with it (turning it on and off). The long service and characteristic IDs are taken from the ButtonLED example program, which you can find under File→Examples (Examples from Custom Libraries)→ArduinoBLE→Peripheral→ButtonLED. The sketch also creates a descriptor for the characteristic, which gives it a friendly name.

Inside the `setup` function, the sketch opens the serial port, configures the LED pin, and tries to start the BLE system. Next, it sets the name of the device (RemoteLED) and configures the `ledService` as an advertised service. It then adds the characteristic to the service, and registers the service itself with the BLE system. Finally, it sets the `ledCharacteristic` value to 0 and starts advertising the service.

Inside `loop`, the sketch repeatedly polls the BLE system. If the characteristic has been written to remotely, it will turn the LED on or off, depending on what its state has been changed to. To control it remotely, you can install the LightBlue-Bluetooth Low Energy app on a mobile phone (Android or iOS), and wait until the RemoteLED device appears in the list as shown in Figure 14-11.

# LightBlue®

**[TV] Samsung 6 Series (40)**
-69dBm  *28:39:5E:5A:21:1D*

**Unnamed**
-61dBm  *19:04:84:61:CE:E1*

**Unnamed**
-78dBm  *4E:B3:61:DF:95:0E*

**RemoteLED**
-88dBm  *C5:53:A2:0C:46:05*

If you have a lot of previously paired devices, you can click the three-dot icon in the upper right and deselect Show Paired Devices. Click RemoteLED to connect to it, and then scroll down to Generic Attribute, and look for the attribute label ending with a1214 that is tagged as Readable, Writable. Click that line, then scroll down to Written Values, set the value to 1, and click Write. You should see the LED turn on. You can set it to 0 to turn it off.

The ArduinoBLE library allows you to create devices that express many of BLE's capabilities: you could create a temperature sensor, heart rate monitor, magnetometer, and more. It also offers the ability to interact with various Bluetooth Low Energy devices, so you can connect to BLE peripherals and exchange data with them as well.

## See Also

*Getting Started with Bluetooth Low Energy* by Kevin Townsend, Carles Cufí, Akiba and Robert Davidson (O'Reilly)

*Make: Bluetooth* by Alasdair Allan, Don Coleman, and Sandeep Mistry (Make Community)

GATT specifications for BLE (services, characteristics, descriptors)

# WiFi and Ethernet

## 15.0 Introduction

Want to share your sensor data? Let other people take control of your Arduino's actions? Your Arduino can communicate with a broader world over Ethernet and WiFi networks. This chapter describes the many ways you can use Arduino with the internet. It has examples that demonstrate how to build and use web clients and servers, and it shows how to use the most common internet communication protocols with Arduino.

The internet allows a client (e.g., a web browser) to request information from a server (a web server or other internet service provider). This chapter contains recipes showing how to make an internet client that retrieves information from a web service. Other recipes in this chapter show how Arduino can be an internet server that provides information to clients using internet protocols and can even act as a web server that creates pages for viewing in web browsers.

The Arduino Ethernet and WiFi libraries support a range of methods (protocols) that enable your sketches to be an internet client or a server. The libraries use a suite of standard internet protocols, and most of the low-level plumbing is hidden. Getting your clients or servers up and running and doing useful tasks will require understanding of the basics of network addressing and protocols, and you may want to consult an online reference or one of these introductory books:

- *Head First Networking* by Al Anderson and Ryan Benedetti (O'Reilly)
- *Network Know-How: An Essential Guide for the Accidental Admin* by John Ross (No Starch Press)
- *Making Things Talk* by Tom Igoe (Make Community)

Here are some of the key concepts in this chapter. You may want to explore them in more depth than is possible here:

Ethernet

> This is the low-level signaling layer providing basic physical message-passing capability. Source and destination addresses for these messages are identified by a Media Access Control (MAC) address. Your Arduino sketch defines a MAC address value that must be unique on your network.

WiFi

> In many respects, WiFi is a functional replacement for Ethernet. Like Ethernet, WiFi provides also is a low-level signaling layer, and it also uses MAC addresses to identify devices uniquely on the network. You won't need to hardcode a MAC address into your sketch because it is embedded in the radio. In terms of where WiFi and Ethernet live in the various layers that make up a networking stack, they are at the bottom, so to all intents and purposes, they are interchangeable with each other, at least from the Arduino programmer's viewpoint. The setup and initialization code is slightly different between WiFi and Ethernet, but once the connection is up and running, the rest of the code can be identical.

TCP and IP

> Transmission Control Protocol (TCP) and Internet Protocol (IP) are core internet protocols built above Ethernet or WiFi. They provide a message-passing capability that operates over the global internet. TCP/IP messages are delivered through unique IP addresses for the sender and receiver. A server on the internet uses a numeric label (address) that no other server will have so that it can be uniquely identified. This address consists of four bytes, usually represented with dots separating the bytes (e.g., 207.241.224.2 was, at the time of this writing, an IP address used by the Internet Archive). The internet uses

the Domain Name System (DNS) service to translate the host name (`google.com`) to the numeric IP address.

Local IP addresses

> If you have more than one computer connected to the internet on your home network using a broadband router or gateway, each computer probably uses a local IP address that is provided by your router. The local address is created using a Dynamic Host Configuration Protocol (DHCP) service in your router, which the Arduino Ethernet and WiFi libraries can use to obtain IP addresses from the router.

Web requests from a web browser and the resultant responses use Hypertext Transfer Protocol (HTTP) messages. For a web client or server to respond correctly, it must understand and respond to HTTP requests and responses. Many of the recipes in this chapter use this protocol, and referring to one of the references listed earlier for more details will help with understanding how these recipes work in detail.

Web pages are usually formatted using Hypertext Markup Language (HTML). Although it's not essential to use HTML if you are making an Arduino web server, as Recipe 15.11 illustrates, the web pages you serve can use this capability.

Extracting data from a web server page intended to be viewed by people using a web browser can be a little like finding a needle in a haystack because of all the extraneous text, images, and formatting tags used on a typical page. This task can be simplified by using the Stream parsing functionality in Arduino to find particular sequences of characters and to get strings and numeric values from a stream of data. In fact, it is unwise and potentially dangerous to create any automated system that makes requests to web servers that were intended to be used by humans. For example, if you were to accidentally (or intentionally) create code that performed a Google search every 5 seconds, your IP address may be blocked from accessing Google services until you stop. If you are on an office or school network where all the devices on your network are behind a gateway, that gateway's IP address may be blocked, which would be

extremely inconvenient for others. For this reason, it is best to use a documented Web API, which you'll see done throughout this chapter. An API allows you to receive web responses in a leaner format than HTML, such as JSON, XML, or CSV. This lets you limit the amount of data you are requesting, and using API arguments, you can narrow those requests down to just the data you need. Most important, using an API and abiding by its rules allows you to work within agreed-upon parameters that the web server's operator has established.

# 15.1 Connecting to an Ethernet Network

## Problem

You want to connect Arduino to an Ethernet network using an Ethernet module such as the Arduino Ethernet shield or the Adafruit Ethernet FeatherWing (which connects to boards in the Adafruit Feather form factor).

## Solution

This sketch uses the Ethernet library that is included with the Arduino IDE to request some information from the Internet Archive. The library supports a number of Ethernet modules. In order for this sketch to work correctly, you will need to know several things about your network: the DNS server IP address, the default gateway IP address, and one available static IP address (an address that is outside the pool of automatically assigned addresses). This information can be found in your network router's configuration utility, which is typically accessed via a web browser:

```
/*
 * Ethernet Web Client sketch
 * Connects to the network without DHCP, using
 * hardcoded IP addresses for device.
 */


#include <SPI.h>
#include <Ethernet.h>
```

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED }; // Must be unique
IPAddress ip(192, 168, 1, 177); // Must be a valid address for your network
char serverName[] = "archive.org";


EthernetClient client;


String request = "GET /advancedsearch.php?q=arduino&fl%5B%5D=description"
                 "&rows=1&sort%5B%5D=downloads+desc&output=csv#raw HTTP/1.0";


void setup()
{
  Serial.begin(9600);
  while(!Serial); // for Leonardo and 32-bit boards


  Ethernet.begin(mac, ip);
  delay(1000); // give the Ethernet hardware a second to initialize

  Serial.println("Connecting to server...");
  int ret = client.connect(serverName, 80);


  if (ret == 1)


  {
    Serial.println("Connected");
    client.println(request);
    client.print("Host: "); client.println(serverName);
    client.println("Connection: close");
    client.println(); // Send the terminating blank line that HTTP requires
  }

  else


  {
    Serial.println("Connection failed, error was: ");
    Serial.print(ret, DEC);
  }
}


void loop()
{
  if (client.available())
```

```
      {
        char c = client.read();
        Serial.print(c);   // echo all data received to the Serial Monitor
      }
      if (!client.connected())


      {
        Serial.println();
        Serial.println("Disconnecting.");
        client.stop();
        while(1); // halt
      }
    }
```

## Discussion

This sketch provides some simple code to help you confirm that your Ethernet board is connected and configured correctly, and that it can reach remote servers. It uses the Internet Archive API to search for Arduino by using the parameter `q=arduino` in the request. That `request` variable contains the request method (`GET`), the path of the request (`/advancedsearch.php`), and the query string, which includes everything from ? to the space before the HTTP protocol (`HTTP/1.0`). After the search term, the query string specifies just one field in the response (`fl%5B%5D=description`, or `fl[]=description` unescaped), and only one result (`rows=1`). Because it sorts by number of downloads in descending order (`sort%5B%5D=downloads+desc`), that one result is the #1 downloaded Arduino resource on Archive.org. The sketch uses the HTTP 1.0 protocol rather than the HTTP 1.1 protocol because HTTP 1.1 servers may use features that make your sketch's life more complicated. For example, HTTP 1.1 clients must support chunked responses, which causes the server to split the responses into one or more chunks separated by a delimiter that represents the length of each chunk. It's up to the server as to whether it sends a chunked response, but if the client specifies HTTP/1.0 in the request, the server knows to not use HTTP/1.1 features.

> **NOTE**
>
> Because Arduino uses SPI to communicate with the Ethernet hardware, the line at the top of the sketch that includes `<SPI.h>` is required for the Ethernet library to function properly.

There are several addresses that you may need to configure for the sketch to successfully connect and display the results of the search on the Serial Monitor:

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
```

The MAC address uniquely identifies your Ethernet shield. Every network device must have a different MAC address, and if you use more than one Arduino shield on your network, each must use a different address. Current Ethernet shields have a MAC address printed on a sticker on the underside of the board. If you have a single Ethernet shield, you don't need to change the MAC address, unless by some strange coincidence, you happen to have a device on your network that uses the MAC address used in this example. You cannot make up any MAC address you want, because most MAC addresses are assigned by a central authority. However, there is a defined set of MAC addresses that you can use for local addressing, and will work fine as long as you don't assign the same address to multiple devices.

MAC addresses consist of a series of bytes (called *octets*), and in the sketch, they are expressed as a pair of nybbles (four bits, a half-byte that can be represented by a single hexadecimal character). For the first octet (0xDE), the high nybble is D and the low is E. If the second nybble of the first octet is 2, 6, A, or E, you can use it as a local MAC address. But, for example, `{ 0xAD, 0xDE, 0xBE, 0xEF, 0xFE, 0xED }` would not be a valid local MAC address because the second nybble of the first octet (D) disqualifies it. So if you are putting multiple devices on the same network and need to create MAC addresses, be sure to follow that rule to avoid surprises.

```
IPAddress ip(192, 168, 1, 177);
```

The IP address is used to identify something that is communicating on the internet and must also be unique on your network. The address consists of four bytes, and the range of valid values for each byte depends on how your network is configured. IP addresses are usually expressed with dots separating the bytes—for example, 192.168.1.177. In all the Arduino sketches, commas are used instead of dots because the `IPAddress` class represents an IP address internally as an array of bytes (see Recipe 2.4).

If your network is connected to the internet using a router or gateway, you may need to provide the IP address of the gateway when you call the `Ethernet.begin` function. If you don't, the Ethernet library replaces the last digit (177 in this example) with 1 to determine the gateway and DNS addresses, which is a good guess most of the time. You can find the address of the gateway and DNS server in the configuration utility for your router, which is often web-based. Add two lines after the IP and server addresses at the top of the sketch with the address of your DNS server and gateway:

```
// add if needed by your router or gateway
IPAddress dns_server(192, 168, 1, 2); // The address of your DNS server
IPAddress gateway(192, 168, 1, 254); // your gateway address
```

And change the first line in `setup` to include the gateway address in the startup values for Ethernet:

```
Ethernet.begin(mac, ip, dns_server, gateway);
```

The default gateway's job is to route network packets to and from the world outside your network, and the DNS server's job is to convert a server name like `archive.org` into an IP address like `207.241.224.2` so the Ethernet library knows the address of the server you are trying to reach. Behind the scenes, the Ethernet library will pass that IP address to your default gateway, which acts like the local post office: it will put

your message on the right "truck" needed to get to the next post office between you and your destination. Each post office sends your message along to the next until it reaches `archive.org`.

The `client.connect` function will return 1 if the hostname can be resolved to an IP address by the DNS server and the client can connect successfully. Here are the values that can be returned from `client.connect`:

```
 1 = success
 0 = connection failed
-1 = no DNS server given
-2 = No DNS  records found
-3 = timeout
```

If the error is –1, you will need to manually configure the DNS server as described earlier in this recipe.

Most Ethernet add-on modules will work without additional configuration. However, in some cases, you need to specify the chip select pin to get the Ethernet module to work correctly. Here is an excerpt from an Ethernet library example sketch that shows some of the possibilities:

```
//Ethernet.init(10);  // Most Arduino shields
  //Ethernet.init(5);   // MKR ETH shield
  //Ethernet.init(0);   // Teensy 2.0
  //Ethernet.init(20);  // Teensy++ 2.0
  //Ethernet.init(15);  // ESP8266 with Adafruit Featherwing Ethernet
  //Ethernet.init(33);  // ESP32 with Adafruit Featherwing
```

If you need to use one of these, you can uncomment it and add it to your sketch. You need to call `Ethernet.init` before `Ethernet.begin`. See the documentation for your Ethernet module for more details. When the sketch is running correctly, you'll see the following output, which displays the HTTP headers followed by a blank line, which is followed by the body of the response. You'll also see some diagnostic info indicating when the connection is initiated and when the client disconnects from the server:

```
Connecting to server...
Connected
```

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Sun, 24 Nov 2019 03:36:50 GMT
Content-Type: text/csv;charset=UTF-8
Connection: close
Content-disposition: attachment; filename=search.csv
Strict-Transport-Security: max-age=15724800


"description"
"Arduino The Documentary 2010"


Disconnecting.
```

## See Also

The web reference for the Arduino Ethernet library

# 15.2 Obtaining Your IP Address Automatically

## Problem

The IP address you use for the Ethernet shield must be unique on your network and you would like this to be allocated automatically. You want the Ethernet shield to obtain an IP address from a DHCP server.

## Solution

This sketch uses a similar configuration process to the one from Recipe 15.1 but it does not pass an IP address to the `Ethernet.begin` method:

```
/*
 * DHCP sketch
 * Obtain an IP address from the DHCP server and display it
 */


#include <SPI.h>
#include <Ethernet.h>
```

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED }; // Must be unique

EthernetClient client;

void setup()
{
  Serial.begin(9600);
  while(!Serial); // for Leonardo and 32-bit boards


  if (Ethernet.begin(mac) == 0)
  {
    Serial.println("Failed to configure Ethernet using DHCP");
    while(1); // halt
  }
  delay(1000); // give the Ethernet hardware a second to initialize

}

#define MAINTAIN_DELAY 750 // Maintain DHCP lease every .75 seconds
void loop()
{
  static unsigned long nextMaintain = millis() + MAINTAIN_DELAY;
  if (millis() > nextMaintain)
  {
    nextMaintain = millis() + MAINTAIN_DELAY;
    int ret = Ethernet.maintain();
    if (ret == 1 || ret == 3)
    {
      Serial.print("Failed to maintain DHCP lease. Error: ");
      Serial.println(ret);
    }
    Serial.print("Current IP address: ");
    IPAddress myIPAddress = Ethernet.localIP();

    Serial.println(myIPAddress);

  }
}
```

## Discussion

The major difference from the sketch in Recipe 15.1 is that there is no IP (or gateway or DNS server) address variable. These values are acquired from your DHCP server when the sketch starts. Also, there is a check to confirm that the `Ethernet.begin` statement was successful. This is needed to ensure that a valid IP address has been provided by the DHCP server (network access is not possible without a valid IP address).

When a DHCP server assigns an IP address, your Arduino is given a *lease*. When the lease expires, the DHCP server may give you the same IP address or it may give you a new one. You must periodically call `Ethernet.maintain()` to let the DHCP server know you're still active. If you don't call it at least once per second, you could lose out on the renewal when the time comes. DHCP lease behavior (length of lease, what the DHCP server does when the lease is renewed) depends on the configuration of your network router. Each time through, this code prints the IP address to the Serial Monitor.

---

### WARNING

Using DHCP features will increase your sketch size by a couple of kilobytes of program storage space. If you are low on storage space, you can use a fixed IP address (see Recipe 15.1).

---

# 15.3 Sending and Receiving Simple Messages (UDP)

## Problem

You want to send and receive simple messages over the internet.

## Solution

This sketch uses the Arduino UDP (User Datagram Protocol) library to send and receive strings. UDP is a simpler, but slightly messier, message protocol compared to TCP. While sending a TCP message will result in an error if a message can't reach its destination intact, UDP messages may

arrive out of order, or not at all, and your sketch won't receive an error when something goes wrong with delivery. But UDP has less overhead than TCP, and is a good choice when you need to trade speed for reliability. In this simple example, Arduino prints the received string to the Serial Monitor and a string is sent back to the sender saying "acknowledged":

```
/*
 * UDPSendReceiveStrings
 * This sketch receives UDP message strings, prints them to the serial port
 * and sends an "acknowledge" string back to the sender
 */


#include <SPI.h>
#include <Ethernet.h>
#include <EthernetUdp.h>


byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED }; // MAC address to use


unsigned int localPort = 8888; // local port to listen on


// buffers for receiving and sending data
char packetBuffer[UDP_TX_PACKET_MAX_SIZE]; // buffer to hold incoming packet,
char replyBuffer[] = "acknowledged";       // a string to send back


// A UDP instance to let us send and receive packets over UDP
EthernetUDP Udp;


void setup()


{
  Serial.begin(9600);


  // start Ethernet and UDP
  Ethernet.begin(mac);
  Udp.begin(localPort);
}


void loop()
```

```arduino
{
  // if there's data available, read a packet
  int packetSize =  Udp.parsePacket();

  if(packetSize)
  {
    Serial.print("Received packet of size ");
    Serial.println(packetSize);

    // read packet into packetBuffer and get sender's IP addr and port number
    Udp.read(packetBuffer,UDP_TX_PACKET_MAX_SIZE);
    Serial.println("Contents:");
    Serial.println(packetBuffer);

    // send a string back to the sender
    Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
    Udp.write(replyBuffer);
    Udp.endPacket();
  }
  maintainLease(); // Keep our DHCP connection
  delay(10);
}

#define MAINTAIN_DELAY 750 // Maintain DHCP lease every .75 seconds
void maintainLease()
{
  static unsigned long nextMaintain = millis() + MAINTAIN_DELAY;
  if (millis() > nextMaintain)
  {
    nextMaintain = millis() + MAINTAIN_DELAY;
    int ret = Ethernet.maintain();
    if (ret == 1 || ret == 3)
    {
      Serial.print("Failed to maintain DHCP lease. Error: ");
      Serial.println(ret);
    }
    Serial.print("Current IP address: ");
    IPAddress myIPAddress = Ethernet.localIP();

    Serial.println(myIPAddress);

  }
}
```

You can test this by running the following Processing sketch on your computer (see "The Processing Development Environment"). The Processing sketch uses the UDP library that you need to install by clicking Sketch→Import Library→ Add Library and then find and select the UDP library by Stephane Cousot. When you run the Arduino sketch, it will display its current IP address. You will need to change the IP address in the Processing sketch on the line `String ip = "192.168.1.177";` to match the Arduino's IP address:

```
// Processing UDP example to send and receive string data from Arduino
// press any key to send the "Hello Arduino" message

import hypermedia.net.*; // the Processing UDP library by Stephane Cousot

UDP udp;  // define the UDP object

void setup() {
  udp = new UDP( this, 6000 );  // create datagram connection on port 6000
  //udp.log( true );            // <-- print out the connection activity
  udp.listen( true );           // and wait for incoming message
}

void draw()
{
}

void keyPressed() {
  String ip = "192.168.1.177"; // the remote IP address
  int port = 8888;             // the destination port

  udp.send("Hello World", ip, port );    // the message to send
}

void receive( byte[] data )


{
 for(int i=0; i < data.length; i++)
     print(char(data[i]));
 println();
}
```

## Discussion

Plug the Ethernet shield into Arduino and connect the Ethernet cable to your computer. Upload the Arduino sketch and run the Processing sketch on your computer. Hit any key to send the "hello Arduino" message. Arduino sends back "acknowledged," which is displayed in the Processing text window. String length is limited by a constant set in the *EthernetUdp.h* library file; the default value is 24 bytes, but you can increase this by editing the following line in *Udp.h* if you want to send longer strings:

```
#define UDP_TX_PACKET_MAX_SIZE 24
```

UDP is a simple and fast way to send and receive messages over Ethernet. But it does have limitations—the messages are not guaranteed to be delivered, and on a very busy network some messages could get lost or get delivered in a different order than that in which they were sent. But UDP works well for things such as displaying the status of Arduino sensors— each message contains the current sensor value to display, and any lost messages get replaced by messages that follow.

This sketch demonstrates sending and receiving sensor messages. It receives messages containing values to be written to the analog output ports and replies back to the sender with the values on the analog input pins:

```
/*
 * UDPSendReceive sketch:
 */

#include <SPI.h>
#include <Ethernet.h>
#include <EthernetUDP.h>

byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED }; // MAC address to use

unsigned int localPort = 8888;      // local port to listen on

char packetBuffer[UDP_TX_PACKET_MAX_SIZE]; // buffer to hold incoming packet,
int packetSize; // holds received packet size
```

```cpp
const int analogOutPins[] = { 3,5,6,9 };



// A UDP instance to let us send and receive packets over UDP
EthernetUDP Udp;

void setup() {
  Ethernet.begin(mac,ip);
  Udp.begin(localPort);



  Serial.begin(9600);
  Serial.println("Ready");
}

void loop() {
  // if there's data available, read a packet
  packetSize = Udp.parsePacket();

  if(packetSize > 0)
  {
    Serial.print("Received packet of size ");
    Serial.print(packetSize);
    Serial.println(" with contents:");
    // read packet into packetBuffer and get sender's IP addr and port number
    packetSize = min(packetSize,UDP_TX_PACKET_MAX_SIZE);
    Udp.read(packetBuffer,UDP_TX_PACKET_MAX_SIZE);

    for( int i=0; i < packetSize; i++)
    {
        byte value = packetBuffer[i];
        if( i < 4)
        {
            // only write to the first four analog out pins
            analogWrite(analogOutPins[i], value);
        }
        Serial.println(value, DEC);
    }
    Serial.println();
    // tell the sender the values of our analog ports
    sendAnalogValues(Udp.remoteIP(), Udp.remotePort());
  }
```

```
    //wait a bit
    delay(10);
}


void sendAnalogValues( IPAddress targetIp, unsigned int targetPort )
{
    int index = 0;
    for(int i=0; i < 6; i++)
    {
        int value = analogRead(i);


        packetBuffer[index++] = lowByte(value);  // the low byte);
        packetBuffer[index++] = highByte(value); // the high byte);   }
    }
    //send a packet back to the sender
    Udp.beginPacket(targetIp, targetPort);
    Udp.write(packetBuffer);
    Udp.endPacket();
}
```

The sketch sends and receives the values on analog ports 0 through 5 using binary data. If you are not familiar with messages containing binary data, see the introduction to Chapter 4, as well as Recipes 4.6 and 4.7, for a detailed discussion on how this is done on Arduino.

The difference here is that the data is sent using `Udp.write` instead of `Serial.write`.

Here is a Processing sketch you can use with the preceding sketch. It has six scroll bars that can be dragged with a mouse to set the six `analogWrite` levels; it prints the received sensor data to the Processing text window. After you set a slider, press any key to send the values to the Arduino:

```
// Processing UDPTest
// Demo sketch sends & receives data to Arduino using UDP


import hypermedia.net.*;


UDP udp;  // define the UDP object


HScrollbar[] scroll = new HScrollbar[6];  //see: topics/gui/scrollbar
```

```
void setup() {
  size(256, 200);
  noStroke();
  for (int i=0; i < 6; i++) // create the scroll bars
    scroll[i] = new HScrollbar(0, 10 + (height / 6) * i, width, 10, 3*5+1);


  udp = new UDP( this, 6000 );  // create datagram connection on port 6000
  udp.listen( true );           // and wait for incoming message
}


void draw()
{
  background(255);
  fill(255);
  for (int i=0; i < 6; i++) {
    scroll[i].update();
    scroll[i].display();
  }
}


void keyPressed()


{
  String ip = "192.168.137.64"; // the remote IP address (CHANGE THIS!)
  int port = 8888;             // the destination port
  byte[] message = new byte[6] ;


  for (int i=0; i < 6; i++) {
    message[i] = byte(scroll[i].getPos());
    println(int(message[i]));
  }
  println();
  udp.send( message, ip, port );
}


void receive( byte[] data )


{
  println("incoming data is:");
  for (int i=0; i < min(6, data.length); i++)
  {
    scroll[i].setPos(data[i]);
    print(i);
```

```
      print(":");
      println((int)data[i]);
    }
}


class HScrollbar
{
  int swidth, sheight;      // width and height of bar
  int xpos, ypos;           // x and y position of bar
  float spos, newspos;      // x position of slider
  int sposMin, sposMax;     // max and min values of slider
  int loose;                // how loose/heavy
  Boolean over;             // is the mouse over the slider?
  Boolean locked;
  float ratio;

  HScrollbar (int xp, int yp, int sw, int sh, int l)

  {
    swidth = sw;
    sheight = sh;
    int widthtoheight = sw - sh;
    ratio = (float)sw / (float) widthtoheight;
    xpos = xp;
    ypos = yp-sheight/2;
    spos = xpos + swidth/2 - sheight/2;
    newspos = spos;
    sposMin = xpos;
    sposMax = xpos + swidth - sheight;
    loose = l;
  }


  void update()

  {
    if (over())

    {
      over = true;
    }

    else
```

```
  {
    over = false;
  }
  if (mousePressed && over)


  {
    locked = true;
  }
  if (!mousePressed)


  {
    locked = false;
  }
  if (locked)


  {
    newspos = constrain(mouseX-sheight/2, sposMin, sposMax);
  }
  if (abs(newspos - spos) > 1)


  {
    spos = spos + (newspos-spos)/loose;
  }
}


int constrain(int val, int minv, int maxv)


{
  return min(max(val, minv), maxv);
}


Boolean over()


{
  if (mouseX > xpos && mouseX < xpos+swidth &&
    mouseY > ypos && mouseY < ypos+sheight)


  mouseY > ypos && mouseY < ypos+sheight)


    mouseY > ypos && mouseY < ypos+sheight)


  {
    return true;
```

```
  }

  else

  {
    return false;
  }
}

void display()

{
  fill(255);
  rect(xpos, ypos, swidth, sheight);
  if (over || locked)

  {
    fill(153, 102, 0);
  }

  else

  {
    fill(102, 102, 102);
  }
  rect(spos, ypos, sheight, sheight);
}

float getPos()

{
  return spos * ratio;
}

void setPos(int value)

{
  spos = value / ratio;
}
}
```

# 15.4 Use an Arduino with Built-in WiFi

**Problem**

You want to build wireless networking projects with an Arduino board that has a built-in WiFi coprocessor.

## Solution

A select number of Arduino boards combine an ARM or AVR processor with a WiFi coprocessor in a small form factor. The most current boards are based on the NINA-W102 modules from u-blox, which are powered by an Espressif ESP32 module.

This sketch uses the WiFiNINA library that is available from the Library Manager. It supports the WiFi module that's built into the Arduino Uno WiFi Rev 2, Nano 33 IoT, MKR 1010, and MKR VIDOR 4000. The Adafruit Airlift modules, such as the breakout board (4201) and shield (4285), are compatible with this recipe, but Adafruit recommends that you use their customized WiFiNINA library.

To connect to your WiFi network, add *YOUR_SSID* and password to the sketch where indicated:

```
/*

 * WiFiNINA Web Client sketch

 * Requests some data from the Internet Archive

 */
#include <SPI.h>
#include <WiFiNINA.h>

const char ssid[] = "YOUR_SSID";
const char password[] = "YOUR_PASSWORD";

WiFiClient client; // WiFi client
```

```cpp
char serverName[] = "archive.org";
String request = "GET /advancedsearch.php?q=arduino&fl%5B%5D=description"
                 "&rows=1&sort%5B%5D=downloads+desc&output=csv#raw HTTP/1.0";

bool configureNetwork()
{
  int status = WL_IDLE_STATUS; // WiFistatus


  if (WiFi.status() == WL_NO_MODULE)
  {
    Serial.println("Couldn't find WiFi hardware.");
    return false;
  }
  String fv = WiFi.firmwareVersion();
  if (fv < WIFI_FIRMWARE_LATEST_VERSION)
  {
    Serial.println("Please upgrade the WiFi firmware");
  }
  while (status != WL_CONNECTED)
  {
    Serial.print("Attempting WiFi connection to "); Serial.println(ssid);
    status = WiFi.begin(ssid, password); // Attempt connection until
successful

    delay(1000); // Wait 1 second


  }
  return true;
}

void setup()
{
  Serial.begin(9600);
  if (!configureNetwork())
  {
    Serial.println("Stopping.");
    while(1); // halt


  }


  Serial.println("Connecting to server...");
  int ret = client.connect(serverName, 80);
```

```
  if (ret == 1)

  {
    Serial.println("Connected");
    client.println(request);
    client.print("Host: "); client.println(serverName);
    client.println("Connection: close");
    client.println();
  }

  else

  {
    Serial.println("Connection failed, error was: ");
    Serial.print(ret, DEC);
  }
}

void loop()
{
  if (client.available())

  {
    char c = client.read();
    Serial.print(c);  // echo all data received to the Serial Monitor

  }
  if (!client.connected())

  {
    Serial.println();
    Serial.println("Disconnecting.");
    client.stop();
    while(1); // halt

  }
}
```

## Discussion

This sketch is very similar to the sketch from Recipe 15.1, with a few notable changes. See that recipe's Discussion for details on the structure of the request and HTTP protocol. Most important and obvious is that it uses WiFi instead of Ethernet to connect. Because the code for the configuration of the WiFi module is a little more complex than Ethernet, it's in a separate function called `configureNetwork`. Aside from that, the rest of the code in `loop` and `setup` are the same as the Ethernet sketch.

This sketch doesn't use a hardcoded IP address, but instead gets its address from DHCP. With the Ethernet shield, using DHCP increases the size of your sketch significantly. This is not the case with the WiFiNINA library because so much of the work is handled by the WiFi coprocessor module. If you did want to use a fixed IP address with the WiFiNINA library, you could declare an IP address (for example, `IPAddress ip(192, 168, 0, 177);`), and then call `WiFi.config(ip);` before your call to `WiFi.begin()`. All of the WiFi-enabled Arduino boards have MAC addresses defined in the WiFi module, so you do not need to define a MAC address in your sketch.

To run this sketch, you need to make sure that you've installed the correct board support in the Arduino IDE, and you'll need to install the WiFiNINA library as well. Select Tools→Board→Boards Manager. For the Uno WiFi Rev 2, install support for Arduino megaAVR Boards. For the Nano 33 IoT, MKR WiFi 1010, or MKR Vidor 4000, install support for Arduino SAMD Boards. For all boards, use the Library Manager to install the WiFiNINA library. After you've installed the support for your board and the WiFiNINA library, you can connect your board and choose your board and the port with Tools→Board and Tools→Port, then upload the sketch.

Open the Serial Monitor, and if all goes well, you'll make a connection to the WiFi network and you'll see the response from the Internet Archive appear:

```
Please upgrade the WiFi firmware
Attempting WiFi connection to YOUR_SSID
Connecting to server...
Connected
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Sun, 24 Nov 2019 02:46:19 GMT
Content-Type: text/csv;charset=UTF-8
Connection: close
Content-disposition: attachment; filename=search.csv
Strict-Transport-Security: max-age=15724800


"description"
"Arduino The Documentary 2010"


Disconnecting.
```

If you see the message shown at the top of the preceding output (`Please upgrade the WiFi firmware`) it means that the WiFi module's firmware may be out of date. To update it, open Tools→WiFi101/WiFiNINA Firmware Updater. Using the dialog that appears, first select your board, then open the updater sketch and flash it to your board. After it's flashed, select the latest firmware for your board, then click Update Firmware.

# 15.5 Connect to WiFi with Low-Cost Modules

## Problem

You want to build low-cost embedded WiFi-enabled projects using the Arduino environment.

## Solution

The Arduino environment supports network-enabled projects such as web servers and web clients. With a WiFi-enabled board, you can build wireless networking projects. There are many Arduino and Arduino-compatible boards that support WiFi, but there is only one such board that you can purchase for under US$2 (in quantity): the ESP-01, which is powered by the ESP8266 from Espressif Systems. Unlike most other boards, the ESP8266 does not have built-in USB. You can either use a USB-to-Serial adapter (see "Serial Hardware") or use an Arduino board as a USB-to-Serial adapter, as shown here. Upload an empty sketch (File → Examples → 01.Basics → Bare Minimum), then hook up the ESP8266 and the Arduino as shown in Figure 15-1. The ESP8266 will need a 3.3V voltage regulator such as an LD1117V33 because the Arduino does not supply enough power on the 3.3V pin to drive the ESP8266.

> ### NOTE
>
> You can also use a board that has built-in USB support, such as the Adafruit Feather HUZZAH with ESP8266 (2821) or the SparkFun ESP8266 Thing Dev Board (WRL-13711). You will still need to install the ESP8266 board support package, but you won't need to wire up the module as shown here, nor will you need to use an Arduino as a USB host. These boards aren't as inexpensive as the bare modules, but they are incredibly convenient. You will not need to wire or press the PROG or RESET button as you do with the ESP-01 modules because the bootloader on these boards handles the reset sequence for you.

Before you can program the ESP8266, you'll need to install ESP8266 support in the Arduino IDE. Open the Preferences dialog (File → Preferences) and click the icon to the right of the "Additional Boards Manager URLs" field. Add `http://arduino.esp8266.com/stable/package_esp8266com_index.js on` on a line by itself and click OK, then click OK to dismiss the preferences dialog. Next, open the Boards Manager (Tools → Board → Boards Manager), and search for ESP8266. Install the "esp8266 by ESP8266 Community" board package.

Use Tools→Board→Generic ESP8266 Module to select the ESP8266 board, and then use the Tools→Port menu to specify the port of the Arduino that's connected to the ESP8266 module. Select Tools→Builtin Led and set it to 2, because the external LED is connected to GPIO 2 (there is an onboard LED, but using it can interfere with Serial output). Next, edit the sketch and set `ssid` to the name of a 2.4 GHz WiFi network, and `password` to the password.

Hold down the PROG button, and upload the sketch. If you see the word "Connecting" appear in the IDE output for more than a few seconds, press and release the RESET button (keep holding the PROG button) and wait a couple seconds. You may need to do this more than once, and it may take a few tries to get it right. When you see a message like `Writing at 0x00000000... (7 %)`, you're on your way. The rest of the output may not autoscroll, but keep holding the PROG button until you see `Done uploading`. Next, press the RESET button to reboot the module. Open the Serial Monitor, and then use a web browser to visit the URL displayed in the Serial Monitor. Each time you click the button, the LED will blink. After you have programmed the ESP-01, you can disconnect it from the Arduino or USB-to-Serial adapter and power it from a 3V source, and it will continue to run as long as it has power:

```
/*

 * ESP-01 sketch

 * Control an LED from a web page

 */
#include <ESP8266WiFi.h>
#include <ESP8266WebServer.h>

const char* ssid = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";

ESP8266WebServer server(80);
```

```cpp
const int led = LED_BUILTIN;
int ledState = LOW;

// An HTML form with a button

static const char formText[] PROGMEM =
  "<form action=\"/\">\n"
  "<input type=\"hidden\" name=\"toggle\"/>\n"
  "<button type=\"submit\">Toggle LED</button>\n"
  "</form>\n";

// Handle requests for the root document (/)

void handleRoot()

{
  // If the server got the "toggle" argument, toggle the LED.

  if (server.hasArg("toggle"))
  {
    ledState = !ledState;
    digitalWrite(led, !ledState);
  }

  // Display the form

  server.send(200, "text/html", FPSTR(formText));
}

// Error message for unrecognized file requests

void handleNotFound() {
  server.send(404, "text/plain", "File not found\n\n");
}

void setup()

{
  Serial.begin(9600);

  pinMode(led, OUTPUT);
  digitalWrite(led, !ledState);

  // Initialize WiFi
```

```cpp
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);

  // Wait for connection

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println();

  // Set up handlers for the root page (/) and everything else

  server.on("/", handleRoot);
  server.onNotFound(handleNotFound);

  server.begin(); // Start the server

}

#define MSG_DELAY 10000
void loop()

{
  static unsigned long nextMsgTime = 0;


  server.handleClient(); // Process requests from HTTP clients


  if (millis() > nextMsgTime)
  {
    // Show the URL in the serial port

    Serial.print("Visit me at http://");
    Serial.println(WiFi.localIP());
    nextMsgTime = millis() + MSG_DELAY;
  }
}
```
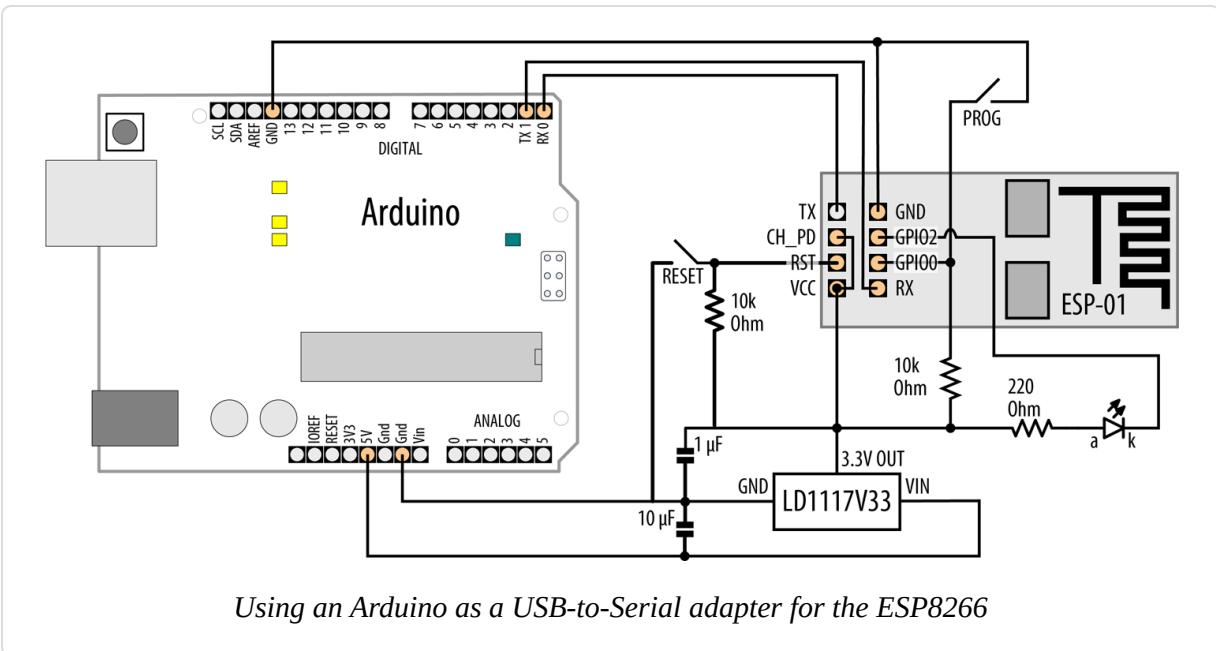
*Using an Arduino as a USB-to-Serial adapter for the ESP8266*

## Discussion

This sketch includes the ESP8266 headers needed to connect to WiFi and construct a web server. It defines variables for your WiFi network's SSID and password (which you will need to set before uploading the sketch) as well as an object for the web server. It also creates variables for the LED (`LED_BUILTIN`) that you'll blink, the LED state, and a variable (`formText`) that holds an HTML fragment that displays a button. That variable is stored in flash memory rather than dynamic memory (see Recipe 17.3). With the ESP-01 board, connecting GPIO2 to GND at boot time prevents the board from booting properly, so we're using inverted logic. Because the LED is tied to 3.3V, driving GPIO2 `LOW` will turn on the LED.

The example shown in the Solution is a bit more involved than the examples you saw for Ethernet or boards with built-in WiFi. That's because the ESP8266WebServer is a bit more capable than the built-in `Server` class. In particular, it includes a number of features to make it easier to serve up web documents. Other boards can accomplish the same tasks, but they are much easier to develop thanks to the libraries that are included with the ESP8266 board package.

The following two functions are *handlers*. A handler's job is to handle requests for a resource such as a web page. Every web server has a root document. When you go to *http://oreilly.com*, with or without a trailing `/`, you are making a request for the root (`/`) of the website. The root handler, `handleRoot`, first checks for the presence of a parameter named `toggle` in the request. If so, it toggles the state of the LED (`HIGH` to `LOW` or vice versa). Then, it displays the HTML fragment contained within `formText`.

The HTML fragment includes a form, which has an associated action (`/`, the web server root). When the form is submitted, it transmits any `input` elements as arguments to the handler. There is a hidden element named `toggle`, and a button that's designed as the submission trigger (`type="submit"`). When you click the Submit button, it sends along `toggle` as an argument, which causes the LED to switch on or off.

The next handler's job is to send a 404 error code for any other resource that you request from the server. After that comes `setup`, which initializes Serial, configures the LED, initializes WiFi, and connects to your network. After that, it configures the two handlers, and starts the server.

Inside the loop, the sketch calls `server.handleClient()` to process any requests. If more than 10 seconds (`MSG_DELAY`) has passed, it prints the URL to the serial port. That way, you can open the Serial Monitor at any time to be reminded of the URL. Open this from a browser that's on the same network as the ESP-01, and you can try out the button.

At the time of writing, there are over 16 different kinds of ESP8266 modules from the manufacturer of the chip, Espressif Systems. These vary by memory capacity, number of pins, and module size. A good overview of the modules can be found on the ESP8266 community wiki.

To enable these modules to be easily used in IoT projects, a number of suppliers to the maker community have created boards that add USB connectivity and other features that provide battery charging and simple programming when connected to the Arduino IDE. The easiest way to get started is to pick one with USB, such as Adafruit's Feather HUZZAH (part

number 2821) or the SparkFun ESP8266 Thing (WRL-13231). Both are relatively inexpensive, but not as cheap as a bare ESP8266 board like the ESP-01. The See Also section has links to step-by-step tutorials for getting started with both of these boards.

All of these have more than enough memory and computing power to support most projects. The ESP8266 has a 32-bit microprocessor core that runs at 80 MHz by default. It has 80K of RAM, and depending on which board you get, anywhere between 512K and 16 MB of flash storage.

## See Also

Sparkfun esp8266-thing tutorial

Adafruit Feather Huzzah esp8266 tutorial

*Make Magazine* article on connecting the ESP8266

The ESP32 is a substantial upgrade to the ESP8266. It has more memory, runs faster, and can also support Bluetooth Low Energy. You can use it as a standalone microcontroller board, but you will also find it as a WiFi coprocessor in boards like the Arduino MKR WiFi 1010 and Arduino Uno WiFi Rev2 (Recipe 15.4).

# 15.6 Extracting Data from a Web Response

## Problem

You want Arduino to get data from a web server. For example, you want to parse string, floating-point, or integer data from a web server's response.

## Solution

This sketch uses the Open Notify web service to determine the position of the International Space Station. It parses out the time of the response as well as the ISS's position in latitude and longitude, and prints the result to the Serial Monitor. This sketch has been designed to work with either the Ethernet library, the WiFiNINA library, or an ESP8266 board. You will

need to uncomment the appropriate `#include` at the top of the sketch. This sketch consists of four files in all. The main sketch is shown first, followed by three header files. You'll need to install the Time library before you compile this sketch (see Recipe 12.4):

```
/*
 * Client-agnostic web data extraction sketch
 * A sketch that can work with ESP8266, WiFiNINA, and Ethernet boards
 */


// Uncomment only one of the following
//#include "USE_NINA.h"      // WiFiNINA boards
//#include "USE_Ethernet.h" // Ethernet
//#include "USE_ESP8266.h"  // ESP8266 boards


#include <TimeLib.h>


char server[] = "api.open-notify.org";
void setup()
{
  Serial.begin(9600);
  if (!configureNetwork()) // Start the network
  {
    Serial.println("Failed to configure the network");
    while(1)


    {
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
    }
  }


  int ret = client.connect(server, 80);


  if (ret == 1)


  {
    Serial.println("Connected");
    client.println("GET /iss-now.json HTTP/1.0"); // the HTTP request
    client.print("Host: "); client.println(server);
    client.println("Connection: close");
    client.println();
  }
```

```arduino
  else

  {
    Serial.println("Connection failed, error was: ");
    Serial.print(ret, DEC);
    while(1)
    {
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
    }
  }
}

char timestampMarker[] = "\"timestamp\":";
char posMarker[] = "\"iss_position\":";
void loop()
{
  if (client.available()) {
    if (client.find('"')) // Start of a string identifier
    {
      String id = client.readStringUntil('"');
      if (id.equals("timestamp")) // Start of timestamp
      {
        if (client.find(':')) // A ":" follows each identifier
        {
          unsigned long timestamp = client.parseInt();
          setTime(timestamp); // Set clock to the time of the response
          digitalClockDisplay();

        }
        else
        {
          Serial.println("Failed to parse timestamp.");
        }
      }

      if (id.equals("iss_position")) // Start of position data
      {
        if (client.find(':')) // A ":" follows each identifier
        {
          // Labels start with a " and position data ends with a }
          while (client.peek() != '}' && client.find('"'))

          {
            String id = client.readStringUntil('"'); // Read the label
```

```
          float val = client.parseFloat(); // Read the value
          client.find('"'); // Consume the trailing " after the float
          Serial.print(id + ": "); Serial.println(val, 4);


        }


      }
      else
      {
        Serial.println("Failed to parse position data.");
      }
    }
  }
}


  if (!client.connected())


  {
    Serial.println();
    Serial.println("disconnecting.");
    client.stop();
    while(1)
    {
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
    }
  }
}


String padDigits(int digit)
{
  String str = String("0") + digit; // Put a zero in front of the digit
  return str.substring(str.length() - 2); // Remove all but last two
characters
}


void digitalClockDisplay()
{
  String datestr = String(year()) + "-" + padDigits(month()) +


                  "-" + padDigits(day());


  String timestr = String(hour()) + ":" + padDigits(minute()) +
                  ":" + padDigits(second());
```

```
    Serial.println(datestr + " " + timestr);
  }
```

Following is the source code for the ESP8266 header file. While it doesn't matter what you name the main sketch, you must create this by clicking the down-pointing arrow icon at the right of the Arduino IDE (just below the Serial Monitor icon), and by choosing New Tab. When Arduino prompts you for the new filename, you must name this *USE_ESP8266.h*. If you use this header, be sure to replace *YOUR_SSID* and *YOUR_PASSWORD*:

```
#include <SPI.h>
#include <ESP8266WiFi.h>
const char ssid[] = "YOUR_SSID";
const char password[] = "YOUR_PASSWORD";
WiFiClient client;
bool configureNetwork()
{
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) // Wait for connection


  {
    delay(1000);
    Serial.print("Waiting for connection to "); Serial.println(ssid);
  }
  return true;
}
```

Here is the source code for the Ethernet header file. You must create this the same way you created the ESP8266 header file, but name this one *USE_Ethernet.h*. If you would prefer to use a hardcoded IP address, see Recipe 15.1 and modify this code accordingly:

```
#include <SPI.h>
#include <Ethernet.h>
byte mac[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
EthernetClient client;
bool configureNetwork()
{
  if (Ethernet.begin(mac))
  {
```

```
      delay(1000); // give the Ethernet module a second to initialize
      return true;
    }
    else
    {
      return false;
    }
  }
```

> **NOTE**
>
> Unlike Recipe 15.2, this recipe doesn't call `Ethernet.maintain()` to maintain the DHCP lease. Neither the WiFiNINA nor the ESP8266 require you to call `maintain()` periodically, but if you plan to create an Ethernet project that runs for a long time, you will need it. For an example of how you could add `maintain()` to a sketch, see Recipe 15.8.

Here is the source code for the WiFiNINA header file. You must create this the same way you created the ESP8266 header file, but name this one *USE_NINA.h*. If you use this header, be sure to replace *YOUR_SSID* and *YOUR_PASSWORD*:

```
#include <SPI.h>
#include <WiFiNINA.h>
const char ssid[] = "YOUR_SSID";
const char password[] = "YOUR_PASSWORD";
WiFiClient client;

bool configureNetwork()
{
  int status = WL_IDLE_STATUS; // WiFistatus


  if (WiFi.status() == WL_NO_MODULE)
  {
    Serial.println("Couldn't find WiFi hardware.");
    return false;
  }
  String fv = WiFi.firmwareVersion();
  if (fv < WIFI_FIRMWARE_LATEST_VERSION)
  {
    Serial.println("Please upgrade the WiFi firmware");
```

```
  }
  while (status != WL_CONNECTED)
  {
    Serial.print("Attempting WiFi connection to "); Serial.println(ssid);
    status = WiFi.begin(ssid, password); // Attempt connection until
successful

    delay(1000); // Wait 1 second

  }
  return true;
}
```

## Discussion

The ISS web service API from Open Notify returns results in the JSON (JavaScript Object Notation) format, which consists of attribute/value pairs of the form `"attribute": value`. The sketch makes a request to the web server using the same technique shown in Recipes 15.1 and 15.4.

The sketch searches the JSON response for values by using the Stream parsing functionality described in Recipe 4.5. In the loop, it looks for a double-quote character ("), which signifies the start of a label such as "timestamp." After the sketch finds the timestamp attribute label, it retrieves the first integer that follows, which is a number of seconds since the beginning of the Unix epoch. Conveniently, these are compatible with the functions from the Time library you saw in Recipe 12.4, so the sketch can use those functions to set the current time. It then prints the time using a `digitalClockDisplay` function similar to the one from that recipe.

If the sketch finds the `"iss_position"` identifier, it then looks for two more labels, which will be latitude and longitude, parses the float values associated with them, and displays each. When it either can't find any more (there should only be those two), or it runs into a } character (the end of the `"iss_position"` identifier), it will finish. Here is sample output from the web service, with attribute values highlighted in bold:

```
{"message": "success", "timestamp": 1574635904, "iss_position":
```

```
{"latitude": "-37.7549", "longitude": "95.5304"}}
```

And here is the output that the sketch will display to the Serial Monitor:

```
Connected


2019-11-24 22:51:44
latitude: -37.7549
longitude: 95.5304

disconnecting.
```

## See Also

Open Notify API documentation

A list of publicly accessible Web APIs

# 15.7 Requesting Data from a Web Server Using XML

### Problem

You want to retrieve data from a site that publishes information in XML format. For example, you want to use values from specific fields in weather providers offering XML API services.

### Solution

This sketch retrieves the weather in London from the Open Weather service. You must set up the three header files as described in Recipe 15.6 and uncomment one of the `#include` lines to choose which kind of network connection to use. If you use WiFiNINA or ESP8266, you'll need to change your SSID and password in the corresponding header file:

```
/*

 * Simple Weather Client

 * gets xml data from http://openweathermap.org/

 * reads temperature from field:  <temperature value="44.89"

 * writes temperature to analog output port.

*/

// Uncomment only one of the following

//#include "USE_NINA.h"     // WiFiNINA boards

//#include "USE_Ethernet.h" // Ethernet

//#include "USE_ESP8266.h"  // ESP8266 boards


char serverName[] = "api.openweathermap.org";
String request =

  "GET /data/2.5/weather?q=London,UK&units=imperial&mode=xml&APPID=";
String APIkey = "YOUR_KEY_HERE"; // see text


void setup()
{
  Serial.begin(9600);
  if (!configureNetwork()) // Start the network

  {
    Serial.println("Failed to configure the network");
    while(1)
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay

  }
}
```

```cpp
void loop()
{
  if (client.connect(serverName, 80) > 0)
  {
    Serial.println("Connected");
    // get weather

    client.println(request + APIkey + " HTTP/1.0");
    client.print("Host: "); client.println(serverName);
    client.println("Connection: close");
    client.println();
  }
  else
  {
    Serial.println(" connection failed");
  }


  if (client.connected())
  {
    if (client.find("<temperature value=") )
    {
      int temperature = client.parseInt();
      Serial.print("Temperature: "); Serial.println(temperature);
    }
    else
      Serial.print("Could not find temperature field");

    if (client.find("<humidity value="))
    {
      int humidity = client.parseInt();
      Serial.print("Humidity: "); Serial.println(humidity);
    }
    else
      Serial.print("Could not find humidity field");
  }
  else


  {
    Serial.println("Disconnected");
  }


  client.stop();
  client.flush();
  delay(60000); // wait a minute before next update
```

```
    }
```

## Discussion

*Open Weather* provides weather data for over 200,000 cities worldwide. It is a free service for casual use but you will need to register to get an API key. Read here for information on how to get a key and the terms of use.

The sketch connects to api.openweathermap.org and then sends the following request to the web service at http://api.openweathermap.org/data/2.5/weather:

```
?q=London,UK&units=imperial&mode=xml&APPID=YOUR_KEY_HERE
```

The string following `q=` specifies the city and country (see this complete list of cities). `units=imperial` will return temperature in Fahrenheit, and `mode=xml` causes the API to return results in XML format. You will need to change this line of code to put your Open Weather Map API key in: `String APIkey = "YOUR_KEY_HERE";`.

The XML data returned looks like this:

```xml
<current>
  <city id="2643743" name="London">
    <coord lon="-0.13" lat="51.51"/>
    <country>GB</country>
    <timezone>0</timezone>
    <sun rise="2019-11-25T07:34:34" set="2019-11-25T16:00:36"/>
  </city>
  <temperature value="48" min="45" max="51.01" unit="fahrenheit"/>
  <humidity value="93" unit="%"/>
  <pressure value="1004" unit="hPa"/>
  <wind>
    <speed value="6.93" unit="mph" name="Light breeze"/>
    <gusts/>
    <direction value="100" code="E" name="East"/>
  </wind>
  <clouds value="75" name="broken clouds"/>
  <visibility value="10000"/>
  <precipitation mode="no"/>
```

```
      <weather number="803" value="broken clouds" icon="04n"/>
      <lastupdate value="2019-11-25T02:02:46"/>
   </current>
```

The sketch parses the data by using `client.find()` to locate the
temperature and humidity tags, and then uses `client.parseInt()` to
retrieve the values for each of those. The sketch will retrieve the weather
data every 60 seconds. This is a relatively small XML message. If you are
processing a very large XML message, you could end up using too much of
Arduino's resources (CPU and RAM). JSON can often be a more compact
notation (see Recipe 15.6).

# 15.8 Setting Up an Arduino to Be a Web Server

## Problem

You want Arduino to serve web pages. For example, you want to use your
web browser to view the values of sensors connected to Arduino analog
pins.

## Solution

This is based on the standard Arduino Web Server Ethernet example sketch
distributed with Arduino that shows the value of the analog input pins. It
has been modified to be adaptable to boards with WiFiNINA modules
(Recipe 15.4) as well as the ESP8266 (Recipe 15.5). You will need to
uncomment the appropriate `#include` at the top of the sketch. This sketch
consists of four files in all. The main sketch is shown first, followed by
three header files:

```
/*
 * Web Server sketch
 */
```

```cpp
// Uncomment only one of the following
//#include "USE_NINA.h"      // WiFiNINA boards
//#include "USE_Ethernet.h" // Ethernet
//#include "USE_ESP8266.h"   // ESP8266 boards

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


  if (!configureNetwork()) // Start the network
  {
    Serial.println("Failed to configure the network");
    while(1)
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
  }
  server.begin();
}

#define MSG_DELAY 10000
void loop() {
  static unsigned long nextMsgTime = 0;
  if (millis() > nextMsgTime)
  {
    Serial.print("Visit me at http://");
    Serial.println(getIP());
    nextMsgTime = millis() + MSG_DELAY;
  }

  maintain(); // Maintain the DHCP lease manually if needed

  client = server.available(); // Listen for connections
  if (client) {
    Serial.println("New client connection");


    // an http request ends with a blank line
    boolean currentLineIsBlank = true;
    while (client.connected())


    {
      if (client.available())


      {
        char c = client.read();
```

```
            Serial.write(c);
            // If you've reached the end of a blank line and found another \n,
            // then you've reached the end of the headers.
            if (c == '\n' && currentLineIsBlank)


            {
              // Send a standard http response header
              client.println("HTTP/1.1 200 OK");
              client.println("Content-Type: text/html");
              client.println("Connection: close"); // Close connection after
response
              client.println("Refresh: 5");   // Refresh every 5 sec
              client.println(); // End of headers

              client.println("<!DOCTYPE HTML>");
              client.println("<HTML>");


              // Display the value of each analog input pin
              for (int analogChannel = 0; analogChannel < 6; analogChannel++)


              {
                int sensorReading = analogRead(analogChannel);
                client.print("A");    client.print(analogChannel);
                client.print(" = "); client.print(sensorReading);
                client.println("<BR />");
              }
              client.println("</HTML>");
              break; // Break out of the while loop
            }
            if (c == '\n')


            {
              // you're starting a new line
              currentLineIsBlank = true;
            }


            else if (c != '\r')


            {
              // you've gotten a character on the current line
              currentLineIsBlank = false;
            }
          }
```

```
    }
    // Give the web browser time to receive the data
    delay(100);

    // close the connection:
    client.stop();
    Serial.println("Client disonnected");
  }
}
```

Here is the source code for the Ethernet header file. You must create this the same way you created the ESP8266 header file, but name this one *USE_Ethernet.h*. Because this sketch is a long-running server, the `maintain()` function is included, which is called from within the sketch's `loop` function. This will keep the DHCP lease active (see Recipe 15.2). If you would prefer to use a hardcoded IP address, see Recipe 15.1 and modify this code accordingly:

```
#include <SPI.h>
#include <Ethernet.h>
byte mac[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
EthernetClient client;
EthernetServer server(80);


bool configureNetwork()
{
  if (Ethernet.begin(mac))
  {
    delay(1000); // give the Ethernet module a second to initialize
    return true;
  }
  else
  {
    return false;
  }
}


IPAddress getIP()
{
  return Ethernet.localIP();
}
```

```
#define MAINTAIN_DELAY 750 // Maintain DHCP lease every .75 seconds
void maintain()
{
  static unsigned long nextMaintain = millis() + MAINTAIN_DELAY;
  if (millis() > nextMaintain)
  {
    nextMaintain = millis() + MAINTAIN_DELAY;
    int ret = Ethernet.maintain();
    if (ret == 1 || ret == 3)
    {
      Serial.print("Failed to maintain DHCP lease. Error: ");
      Serial.println(ret);
    }
  }
}
```

Here is the source code for the ESP8266 header file. While it doesn't matter what you name the main sketch, you must create this by clicking the down-pointing arrow icon at the right of the Arduino IDE (just below the Serial Monitor icon), and choose New Tab. When Arduino prompts you for the new filename, you must name this *USE_ESP8266.h*. If you use this header, be sure to replace *YOUR_SSID* and *YOUR_PASSWORD*. The `maintain` function is empty because you do not need to manually keep the lease active with the ESP8266 (or WiFiNINA):

```
#include <SPI.h>
#include <ESP8266WiFi.h>
const char ssid[] = "YOUR_SSID";
const char password[] = "YOUR_PASSWORD";
WiFiClient client;
WiFiServer server(80);

bool configureNetwork()
{
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) // Wait for connection


  {
    delay(1000);
    Serial.print("Waiting for connection to "); Serial.println(ssid);
  }
  return true;
```

```
}

IPAddress getIP()
{
  return WiFi.localIP();
}

void maintain()
{
  // Do nothing


}
```

Here is the source code for the WiFiNINA header file. You must create this the same way you created the ESP8266 header file, but name this one *USE_NINA.h*. If you use this header, be sure to replace YOUR_SSID and YOUR_PASSWORD:

```
#include <SPI.h>
#include <WiFiNINA.h>
const char ssid[] = "YOUR_SSID";
const char password[] = "YOUR_PASSWORD";
WiFiClient client;
WiFiServer server(80);

bool configureNetwork()
{
  int status = WL_IDLE_STATUS; // WiFistatus



  if (WiFi.status() == WL_NO_MODULE)
  {
    Serial.println("Couldn't find WiFi hardware.");
    return false;
  }
  String fv = WiFi.firmwareVersion();
  if (fv < WIFI_FIRMWARE_LATEST_VERSION)
  {
    Serial.println("Please upgrade the WiFi firmware");
  }
  while (status != WL_CONNECTED)
  {
    Serial.print("Attempting WiFi connection to "); Serial.println(ssid);
```

```
    status = WiFi.begin(ssid, password); // Attempt connection until
successful

    delay(1000); // Wait 1 second


  }
  return true;
}

IPAddress getIP()
{
  return WiFi.localIP();
}

void maintain()
{
  // Do nothing


}
```

## Discussion

Similar to Recipe 15.6, this sketch uses one of three header files to determine how to connect to the network: if you include the *USE_Ethernet.h* header, this sketch will connect to Ethernet using DHCP. Because DHCP with Ethernet requires you to manually maintain the DHCP lease, there's a `maintain` function that you must call from within `loop`. This code wasn't included in Recipe 15.6 because that sketch would run once and stop. If you use an ESP8266 board, you'll need to include the *USE_ESP8266.h* header file, and if you use a WiFiNINA board, you'll need to include *USE_NINA.h*. Each header file also defines a `client` and `server` variable, and exposes methods to configure the network (`configureNetwork`) and to get the assigned IP address (`getIP`).

The sketch begins by starting the serial port, and then it configures the network using the hardware-specific function, and starts the server. Within the loop, it displays a message every 10 seconds showing the URL you need to connect to the server. When you open this URL in a web browser,

you should see a page showing the values on analog input pins 0 through 6 (see Chapter 5 for more on the analog ports).

The two lines in `setup` initialize the Ethernet library and configure your web server to the IP address you provide. The `loop` waits for and then processes each request received by the web server:

```
client = server.available();
```

The `client` object here represents the client connected to the web server. Depending on which header you included, this will either be an `EthernetClient` or `WiFiClient` object.

`if (client)` tests that the client has been successfully connected.

`while (client.connected())` tests if the web server is connected to a client making a request.

`client.available()` and `client.read()` check if data is available from the client, and read a byte if it is. This is similar to `Serial.available()`, discussed in Chapter 4, except the data is coming from the internet rather than the serial port. The code reads data until it finds the first line with no data, signifying the end of a request. An HTTP header is sent using the `client.println` commands followed by the printing of the values of the analog ports.

> **TIP**
>
> The ESP8266 board package includes a rich set of libraries for creating web servers. Unless you are writing code that should support a variety of networking hardware, you may want to use those features. See Recipe 15.5 for more details.

# 15.9 Handling Incoming Web Requests

**Problem**

You want to control digital and analog outputs with Arduino acting as a web server. For example, you want to control the values of specific pins through parameters sent from your web browser.

## Solution

This sketch reads requests sent from a browser and changes the values of digital and analog output ports as requested. You will need to open the Serial Monitor to see what URL to use to connect to the sketch.
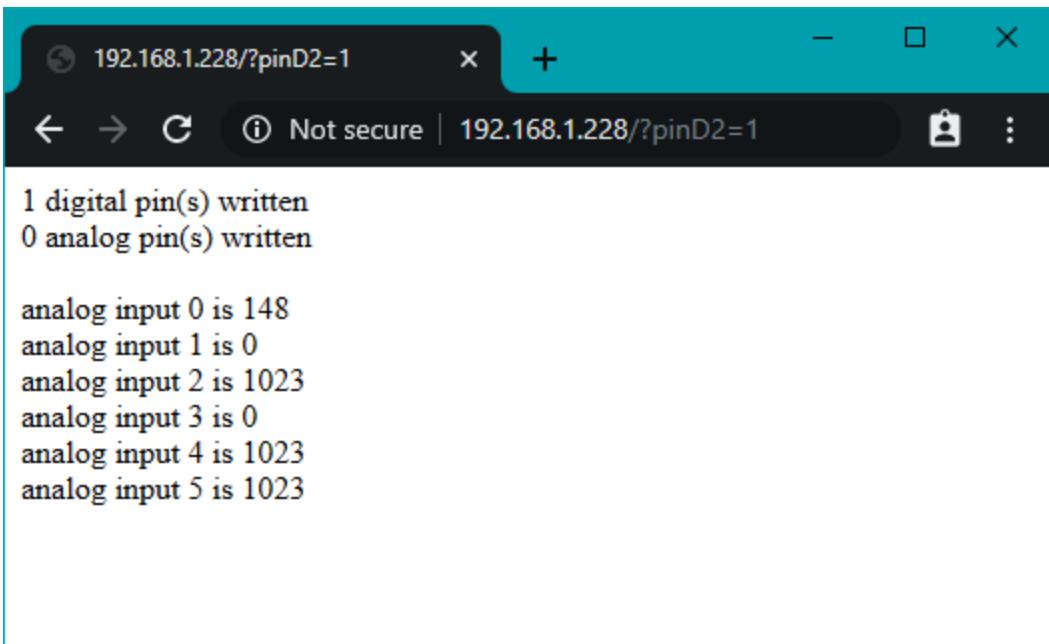
The URL (text received from a browser request) contains one or more fields starting with the word *pin* followed by a *D* for digital or *A* for analog and the pin number. The value for the pin follows an equals sign.

For example, sending *http://IP_ADDRESS/?pinD2=1* from your browser's address bar turns digital pin 2 on; *http://IP_ADDRESS/?pinD2=0* turns pin 2 off. (See Chapter 7 if you need information on connecting LEDs to Arduino pins.) You would need to replace *IP_ADDRESS* with the IP address shown in the Serial Monitor.

> **NOTE**
>
> You must set up the three header files as described in Recipe 15.8 and uncomment one of the `#include` lines to choose which kind of network connection to use. If you use WiFiNINA or ESP8266, you'll need to change your SSID and password in the corresponding header file. The ESP8266 has a limited number of pins available for output, so you will need to consult the documentation for your ESP8266 board to find a pin that can be used. Writing to some pins may cause the board to behave erratically.

Figure 15-2 shows what you will see on your web browser when connected to the web server code that follows.

*Browser page displaying output created by this recipe's Solution*

```
/*
 * Incoming request sketch
 * Respond to requests in the URL to change digital and analog output ports
 * show the number of ports changed and the value of the analog input pins.
 * for example:
 *   sending http://IP_ADDRESS/?pinD2=1 turns digital pin 2 on
 *   sending http://IP_ADDRESS/?pinD2=0 turns pin 2 off.
 * This sketch demonstrates text parsing using Arduino Stream class.
 */


// Uncomment only one of the following
//#include "USE_NINA.h"     // WiFiNINA boards
//#include "USE_Ethernet.h" // Ethernet
//#include "USE_ESP8266.h"  // ESP8266 boards

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


  if (!configureNetwork()) // Start the network
  {
    Serial.println("Failed to configure the network");
    while(1)
```

```
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
  }
  server.begin();
}

#define MSG_DELAY 10000
void loop() {
  static unsigned long nextMsgTime = 0;
  if (millis() > nextMsgTime)
  {
    Serial.print("Try http://");
    Serial.print(getIP()); Serial.println("?pinD2=1");
    nextMsgTime = millis() + MSG_DELAY;
  }

  maintain(); // Maintain the DHCP lease manually if needed

  client = server.available();
  if (client)

  {
    while (client.connected())

    {
      if (client.available())

      {
        // counters to show the number of pin change requests
        int digitalRequests = 0;

        int analogRequests = 0;
        if( client.find("GET /") ) // search for 'GET'
        {

          // find tokens starting with "pin" and stop at the end of the line
          while(client.findUntil("pin", "\r\n"))
          {

            char type = client.read(); // D or A
            // the next ascii integer value in the stream is the pin
            int pin = client.parseInt();
```

```arduino
            int val = client.parseInt(); // the integer after that is the
value

            if( type == 'D')

            {
               Serial.print("Digital pin ");
               pinMode(pin, OUTPUT);
               digitalWrite(pin, val);
               digitalRequests++;
            }
            else if( type == 'A')
            {
               Serial.print("Analog pin ");
               analogWrite(pin, val);
               analogRequests++;
            }
            else

            {
               Serial.print("Unexpected type ");
               Serial.print(type);
            }
            Serial.print(pin);
            Serial.print("=");
            Serial.println(val);
         }
      }
      Serial.println();

      // the findUntil has detected the blank line (a lf followed by cr)
      // so the http request has ended and we can send a reply
      // send a standard http response header
      client.println("HTTP/1.1 200 OK");
      client.println("Content-Type: text/html");
      client.println();

      // output the number of pins handled by the request
      client.print(digitalRequests);
      client.print(" digital pin(s) written");
      client.println("<br />");
      client.print(analogRequests);
      client.print(" analog pin(s) written");
      client.println("<br />");
      client.println("<br />");
```

```
    // output the value of each analog input pin
    for (int i = 0; i < 6; i++) {
      client.print("analog input ");
      client.print(i);
      client.print(" is ");
      client.print(analogRead(i));
      client.println("<br />");
    }
    break; // Exit the while() loop
  }
}
// give the web browser time to receive the data
delay(100);
client.stop();
  }
}
```

## Discussion

If you were to send the command (replacing *IP_ADDRESS* with the IP address displayed in the Serial Monitor) *http://IP_ADDRESS/?pinD2=1*, the sketch would take pin 2 high. Here is how the instructions in the URL are broken down: Everything before the question mark is treated as the address of the web server (for example, 192.168.1.177). The remaining data is a list of fields, each beginning with the word *pin* followed by a *D* indicating a digital pin or *A* indicating an analog pin. The numeric value following the *D* or *A* is the pin number. This is followed by an equals sign and finally the value you want to set the pin to. `pinD2=1` sets digital pin 2 `HIGH`. There is one field per pin, and subsequent fields are separated by an ampersand. You can have as many fields as there are Arduino pins you want to change.

The request can be extended to handle multiple parameters by using ampersands to separate multiple fields. For example:

- *http://IP_ADDRESS/?pinD2=1&pinD3=0&pinA9=128&pinA11=255*

Each field within the ampersand is handled as described earlier. You can have as many fields as there are Arduino pins you want to change.

# 15.10 Handling Incoming Requests for Specific Pages

## Problem

You want to have more than one page on your web server; for example, to show the status of different sensors on different pages.

## Solution

This sketch looks for requests for pages named "analog" or "digital" and displays the pin values accordingly:

---

**TIP**

You must set up the three header files as described in Recipe 15.8 and uncomment one of the `#include` lines to choose which kind of network connection to use. If you use WiFiNINA or ESP8266, you'll need to change your SSID and password in the corresponding header file.

---

```
/*
 * WebServerMultiPage sketch
 * Respond to requests in the URL to view digital and analog output ports
 * http://IP_ADDRESS/analog/   displays analog pin data
 * http://IP_ADDRESS/digital/  displays digital pin data
 */


// Uncomment only one of the following
//#include "USE_NINA.h"      // WiFiNINA boards
//#include "USE_Ethernet.h" // Ethernet
//#include "USE_ESP8266.h"  // ESP8266 boards

const int MAX_PAGE_NAME_LEN = 8;  // max characters in a page name
char buffer[MAX_PAGE_NAME_LEN+1];  // page name + terminating null

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

```cpp
  if (!configureNetwork()) // Start the network
  {
    Serial.println("Failed to configure the network");
    while(1)
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
  }
  server.begin();
}

#define MSG_DELAY 10000
void loop() {
  static unsigned long nextMsgTime = 0;
  if (millis() > nextMsgTime)
  {
    Serial.print("Try http://");
    Serial.print(getIP()); Serial.println("/analog/");
    nextMsgTime = millis() + MSG_DELAY;
  }

  maintain(); // Maintain the DHCP lease manually if needed

  client = server.available();
  if (client)

  {
    while (client.connected())

    {
      if (client.available())

      {
        if( client.find("GET ") )

        {
          // look for the page name
          memset(buffer,0, sizeof(buffer)); // clear the buffer
          if(client.find( "/"))

            if(client.readBytesUntil('/', buffer, MAX_PAGE_NAME_LEN ))

            {
```

```
            if(strcmp(buffer, "analog") == 0)
              showAnalog();
            else if(strcmp(buffer, "digital") == 0)
              showDigital();
            else
              unknownPage(buffer);
          }


        }
        Serial.println();
        break; // Exit the while() loop
      }
    }
    // give the web browser time to receive the data
    delay(100);
    client.stop();
  }
}

void showAnalog()
{
  Serial.println("analog");
  sendHeader();
  client.println("<h1>Analog Pins</h1>");
  // output the value of each analog input pin
  for (int i = 0; i < 6; i++)

  {
    client.print("analog pin ");
    client.print(i);
    client.print(" = ");
    client.print(analogRead(i));
    client.println("<br />");
  }
}

void showDigital()
{
  Serial.println("digital");
  sendHeader();
  client.println("<h1>Digital Pins</h1>");
  // show the value of each digital pin
  for (int i = 2; i < 8; i++)
```

```
  {
    pinMode(i, INPUT_PULLUP);
    client.print("digital pin ");
    client.print(i);
    client.print(" is ");
    if(digitalRead(i) == LOW)
      client.print("HIGH");
    else
      client.print("LOW");
    client.println("<br />");
  }
  client.println("</body></html>");
}

void unknownPage(char *page)
{
  sendHeader();
  client.println("<h1>Unknown Page</h1>");
  client.print(page);
  client.println("<br />");
  client.println("Recognized pages are:<br />");
  client.println("/analog/<br />");
  client.println("/digital/<br />");
  client.println("</body></html>");
}

void sendHeader()
{
  // send a standard http response header
  client.println("HTTP/1.1 200 OK");
  client.println("Content-Type: text/html");
  client.println();
  client.println("<html><head><title>Web server multi-page Example</title>");
  client.println("<body>");
}
```
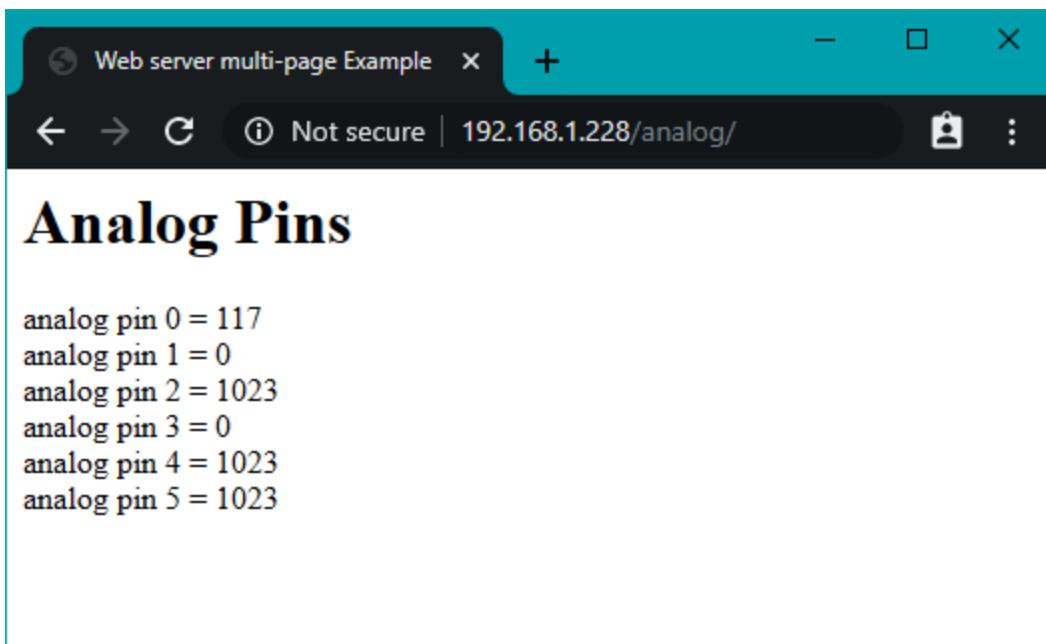
## Discussion

You can test this from your web browser by typing
*http://IP_ADDRESS/analog/* or *http://IP_ADDRESS/digital/* (replace
*IP_ADDRESS* with the IP address displayed in the Serial Monitor).

Figure 15-3 shows the expected output. To test it, you could wire one or
more buttons to the digital pins and one or more potentiometers to the

analog pins. Because the sketch uses the built-in pull-up resistors (see Recipe 2.4), the logic is inverted (LOW means a button is pressed and HIGH means it is not). See Recipe 5.6 for instructions on working with potentiometers.



*Browser output showing analog pin values*

The sketch looks for the / character to determine the end of the page name. The server will report an unknown page if the / character does not terminate the page name.

You can easily enhance this with some code from Recipe 15.9 to allow control of Arduino pins from another page named update. Here is the section of loop that you need to change (added lines shown in **bold**):

```
if(client.readBytesUntil('/', buffer, MAX_PAGE_NAME_LEN ))


{
  if(strcmp(buffer, "analog") == 0)
    showAnalog();
```

```
                else if(strcmp(buffer, "digital") == 0)
                    showDigital();
                // add this code for new page named: update
                else if(strcmp(buffer, "update") == 0)
                    doUpdate();
                else
                    unknownPage(buffer);
            }
```

Here is the doUpdate function. The ESP8266 has a limited number of pins available for output, so you will need to consult the documentation for your ESP8266 board to find a pin that can be used. Writing to some pins may cause the board to behave erratically:

```
void doUpdate()
{
  Serial.println("update");
  sendHeader();
  // find tokens starting with "pin" and stop on the first blank line
  while (client.findUntil("pin", "\n\r"))


  {
    char type = client.read(); // D or A
    int pin = client.parseInt();
    int val = client.parseInt();
    if ( type == 'D')


    {
      client.print("Digital pin ");
      pinMode(pin, OUTPUT);
      digitalWrite(pin, val);
    }
    else if ( type == 'A')


    {
      client.print("Analog pin ");
      analogWrite(pin, val);
    }
    else


    {
      client.print("Unexpected type ");
      Serial.print(type);
```

```
    }
    client.print(pin);
    client.print("=");
    client.println(val);
  }
  client.println("</body></html>");
}
```

Sending *http://IP_ADDRESS/update/?pinA5=128* from your browser's address bar writes the value 128 to analog output pin 5.

---

**NOTE**

The ESP8266 board package includes a rich set of libraries for creating web servers. Unless you are writing code that should support a variety of networking hardware, you may want to use those features. See Recipe 15.5 for more details.

---
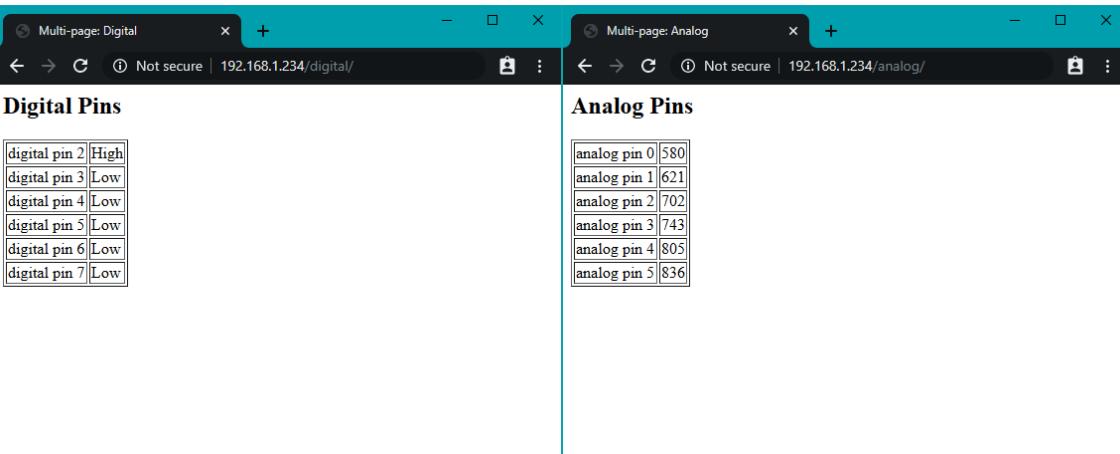
# 15.11 Using HTML to Format Web Server Responses

## Problem

You want to use HTML elements such as tables and images to improve the look of web pages served by Arduino. For example, you want the output from Recipe 15.10 to be rendered in an HTML table.

## Solution

Figure 15-4 shows how the web server in this recipe's Solution formats the browser page to display pin values. (You can compare this to the unformatted values shown in Figure 15-3.)

*Browser pages using HTML formatting*

This sketch shows the functionality from Recipe 15.10 with output formatted using HTML:

---

**NOTE**

You must set up the three header files as described in Recipe 15.8 and uncomment one of the `#include` lines to choose which kind of network connection to use. If you use WiFiNINA or ESP8266, you'll need to change your SSID and password in the corresponding header file. The ESP8266 has a limited number of pins available for output, so you will need to consult the documentation for your ESP8266 board to find a pin that can be used. Writing to some pins may cause the board to behave erratically.

---

```
/*
 * WebServerMultiPageHTML sketch
 * Display analog and digital pin values using HTML formatting
 */


// Uncomment only one of the following
//#include "USE_NINA.h"      // WiFiNINA boards
//#include "USE_Ethernet.h" // Ethernet
//#include "USE_ESP8266.h"  // ESP8266 boards

const int MAX_PAGE_NAME_LEN = 8;  // max characters in a page name
char buffer[MAX_PAGE_NAME_LEN+1];  // page name + terminating null
```

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


  if (!configureNetwork()) // Start the network
  {
    Serial.println("Failed to configure the network");
    while(1)
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
  }
  server.begin();
  pinMode(LED_BUILTIN, OUTPUT);
  for(int i=0; i < 3; i++)
  {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(500);
    digitalWrite(LED_BUILTIN, LOW);
    delay(500);
  }
}

#define MSG_DELAY 10000
void loop() {
  static unsigned long nextMsgTime = 0;
  if (millis() > nextMsgTime)
  {
    Serial.print("Try http://");
    Serial.print(getIP()); Serial.println("/analog/");
    nextMsgTime = millis() + MSG_DELAY;
  }

  maintain(); // Maintain the DHCP lease manually if needed

  client = server.available();
  if (client)

  {
    while (client.connected())

    {
      if (client.available())

      {
        if( client.find("GET ") )
```

```
      {
        // look for the page name
        memset(buffer,0, sizeof(buffer)); // clear the buffer
        if(client.find( "/"))
          if(client.readBytesUntil('/', buffer, MAX_PAGE_NAME_LEN ))


          {
            if(strcasecmp(buffer, "analog") == 0)
              showAnalog();
            else if(strcasecmp(buffer, "digital") == 0)
              showDigital();
            else
              unknownPage(buffer);
          }
      }
      break;
    }
  }
  // give the web browser time to receive the data
  delay(100);
  client.stop();
  }
}


void showAnalog()
{
  sendHeader("Multi-page: Analog");
  client.println("<h2>Analog Pins</h2>");
  client.println("<table border='1' >");
  for (int i = 0; i < 6; i++)


  {
    // output the value of each analog input pin
    client.print("<tr><td>analog pin ");
    client.print(i);
    client.print(" </td><td>");
    client.print(analogRead(i));
    client.println("</td></tr>");
  }
  client.println("</table>");
  client.println("</body></html>");
}
```

```cpp
void showDigital()
{
  sendHeader("Multi-page: Digital");
  client.println("<h2>Digital Pins</h2>");
  client.println("<table border='1'>");
  for (int i = 2; i < 8; i++)

  {
    // show the value of digital pins
    pinMode(i, INPUT_PULLUP);
    digitalWrite(i, HIGH); // turn on pull-ups
    client.print("<tr><td>digital pin ");
    client.print(i);
    client.print(" </td><td>");
    if(digitalRead(i) == LOW)
      client.print("High");
    else
      client.print("Low");
    client.println("</td></tr>");
  }
  client.println("</table>");
  client.println("</body></html>");
}


void unknownPage(char *page)
{
  sendHeader("Unknown Page");
  client.println("<h1>Unknown Page</h1>");
  client.print(page);
  client.println("<br />");
  client.println("Recognized pages are:<br />");
  client.println("/analog/<br />");
  client.println("/digital/<br />");

  client.println("</body></html>");
}


void sendHeader(char *title)
{
  // send a standard http response header
  client.println("HTTP/1.1 200 OK");
  client.println("Content-Type: text/html");
  client.println();
  client.print("<html><head><title>");
  client.println(title);
```

```
    client.println("</title><body>");
  }
```

## Discussion

The same information is provided as in Recipe 15.10, but here the data is formatted using an HTML table. The following code indicates that the web browser should create a table with a border width of 1:

```
  client.println("<table border='1' >");
```

The `for` loop defines the table data cells with the `<td>` tag and the row entries with the `<tr>` tag. The following code places the string `"analog pin"` in a cell starting on a new row:

```
  client.print("<tr><td>analog pin ");
```

This is followed by the value of the variable `i`:

```
  client.print(i);
```

The next line contains the tag that closes the cell and begins a new cell:

```
  client.print(" </td><td>");
```

This writes the value returned from `analogRead` into the cell:

```
  client.print(analogRead(i));
```

The tags to end the cell and end the row are written as follows:

```
  client.println("</td></tr>");
```

The `for` loop repeats this until all six analog values are written.

## See Also

*Learning Web Design* by Jennifer Robbins (O'Reilly)

*Web Design in a Nutshell* by Jennifer Niederst Robbins (O'Reilly)

*HTML & XHTML: The Definitive Guide* by Chuck Musciano and Bill Kennedy (O'Reilly)

# 15.12 Requesting Web Data Using Forms (POST)

## Problem

You want to create web pages with forms that allow users to select an action to be performed by Arduino. Figure 15-5 shows the web page created by this recipe's Solution.

*Web form with buttons*

## Solution

This sketch creates a web page that has a form with buttons. Users navigating to this page will see the buttons in the web browser and the Arduino web server will respond to the button clicks. In this example, the sketch turns a pin on and off depending on which button is pressed:

```
/*
 * WebServerPost sketch
 * Turns a pin on and off using HTML form
 */


// Uncomment only one of the following
//#include "USE_NINA.h"      // WiFiNINA boards
```

```
//#include "USE_Ethernet.h" // Ethernet
//#include "USE_ESP8266.h"  // ESP8266 boards

const int MAX_PAGE_NAME_LEN = 8;   // max characters in a page name
char buffer[MAX_PAGE_NAME_LEN+1];  // page name + terminating null

void setup()


{
  Serial.begin(9600);


  if (!configureNetwork()) // Start the network
  {
    Serial.println("Failed to configure the network");
    while(1)
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
  }
  server.begin();
}


#define MSG_DELAY 10000
void loop() {
  static unsigned long nextMsgTime = 0;
  if (millis() > nextMsgTime)
  {
    Serial.print("Try http://");
    Serial.println(getIP());
    nextMsgTime = millis() + MSG_DELAY;
  }

  maintain(); // Maintain the DHCP lease manually if needed

  client = server.available();
  if (client)

  {
    int type = 0;
    while (client.connected())


    {
      if (client.available())
```

```cpp
{
    // GET, POST, or HEAD
    memset(buffer,0, sizeof(buffer)); // clear the buffer

    if(client.readBytesUntil('/', buffer,sizeof(buffer)))
    {

        Serial.println(buffer);
        if(strcmp(buffer,"POST ") == 0)
        {
            client.find("\r\n\r\n"); // skip to the body

            // find string starting with "pin", stop on first end of line
            // the POST parameters expected in the form pinDx=Y
            // where x is the pin number and Y is 0 for LOW and 1 for HIGH
            while(client.findUntil("pinD", "\r\n"))
            {
                int pin = client.parseInt();        // the pin number
                int val = client.parseInt();        // 0 or 1
                pinMode(pin, OUTPUT);
                digitalWrite(pin, val);
            }
        }
        else // probably a GET
        {

            if (client.find("favicon.ico")) // Send 404 for favicons
                sendHeader("404 Not Found", "Not found");
        }
        sendHeader("200 OK", "Post example");

        //create HTML button to turn off pin 8
        client.println("<h2>Click buttons to turn pin 8 on or off</h2>");
        client.print(
        "<form action='/' method='POST'><p><input type='hidden'
name='pinD8'");
        client.println(" value='0'><input type='submit' value='Off'/>
</form>");

        //create HTML button to turn on pin 8
        client.print(
        "<form action='/' method='POST'><p><input type='hidden'
name='pinD8'");
        client.print(" value='1'><input type='submit' value='On'/></form>");
        client.println("</body></html>");
```

```
        client.stop();
      }
        break; // exit the while loop
    }
  }
  // give the web browser time to receive the data
  delay(100);
  client.stop();
  }
}
void sendHeader(char *code, char *title)
{
  // send a standard http response header
  client.print("HTTP/1.1 "); client.println(code);
  client.println("Content-Type: text/html");
  client.println();
  client.print("<html><head><title>");
  client.print(title);
  client.println("</title><body>");
}
```

## Discussion

A web page with a user interface form consists of HTML tags that identify the controls (buttons, checkboxes, labels, etc.) that comprise the user interface. This recipe uses buttons for user interaction.

These lines create a form with a button named `pinD8` that is labeled "OFF," which will send back a value of `0` (zero) when clicked:

```
client.print(
  "<form action='/' method='POST'><p><input type='hidden' name='pinD8'");
client.println(" value='0'><input type='submit' value='Off'/></form>");
```

When the server receives a request from a browser, it looks for the `"POST "` string to identify the start of the posted form:

```
if (strcmp(buffer,"POST ") == 0) // find the start of the posted form


  client.find("\r\n\r\n"); // skip to the body
  // find parameters starting with "pin" and stop on the first blank line
```

```
// the POST parameters expected in the form pinDx=Y
// where x is the pin number and Y is 0 for LOW and 1 for HIGH
```

If the OFF button was pressed, the received page will contain the string pinD8=0, or pinD8=1 for the ON button.

The sketch searches until it finds the button name (pinD):

```
while(client.findUntil("pinD", "\r\n"))
```

The findUntil method in the preceding code will search for "pinD" and stop searching at the end of a line (\r\n is the newline carriage return sent by the web browser at the end of a form).

The number following the name pinD is the pin number:

```
int pin = client.parseInt();        // the pin number
```

And the value following the pin number will be 0 if button OFF was pressed or 1 if button ON was pressed:

```
int val = client.parseInt();        // 0 or 1
```

The value received is written to the pin after setting the pin mode to output:

```
pinMode(pin, OUTPUT);
digitalWrite(pin, val);
```

More buttons can be added by inserting tags for the additional controls. The following lines add another button to turn on digital pin 9:

```
//create HTML button to turn on pin 9
client.print(
  "<form action='/' method='POST'><p><input type='hidden' name='pinD9'");
client.print(" value='1'><input type='submit' value='On'/></form>");
```

# 15.13 Serving Web Pages Containing Large Amounts of Data

## Problem

Your web pages require more memory than you have available, so you want to use program memory (also known as *progmem* or *flash memory*) to store data (see Recipe 17.4).

## Solution

The following sketch combines the POST code from Recipe 15.12 with the HTML code from Recipe 15.11 and adds new code to access text stored in progmem. As in Recipe 15.11, the server can display analog and digital pin status and turn digital pins on and off (see Figure 15-6).

You must set up the three header files as described in Recipe 15.8 and uncomment one of the #include lines to choose which kind of network connection to use. If you use WiFiNINA or ESP8266, you'll need to change your SSID and password in the corresponding header file. The ESP8266 has a limited number of pins available for output, so you will need to consult the documentation for your ESP8266 board to find a pin that can be used. Writing to some pins may cause the board to behave erratically.

*Web page with LED images*

```
/*
 * WebServerMultiPageHTMLProgmem sketch
 *
 * Respond to requests in the URL to change digital and analog output ports
 * show the number of ports changed and the value of the analog input pins.
 *
 * http://192.168.1.177/analog/   displays analog pin data
 * http://192.168.1.177/digital/  displays digital pin data
 * http://192.168.1.177/change/   allows changing digital pin data
 *
 */


// Uncomment only one of the following
//#include "USE_NINA.h"      // WiFiNINA boards
//#include "USE_Ethernet.h" // Ethernet
//#include "USE_ESP8266.h"  // ESP8266 boards
```

```cpp
#include <avr/pgmspace.h> // for progmem
#define P(name) static const char name[] PROGMEM  // declare a static string

const int MAX_PAGENAME_LEN = 8;  // max characters in page name
char buffer[MAX_PAGENAME_LEN+1]; // additional character for terminating null

void setup()

{
  Serial.begin(9600);


  if (!configureNetwork()) // Start the network
  {
    Serial.println("Failed to configure the network");
    while(1)
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
  }
  server.begin();
  Serial.println(F("Ready"));
}

#define MSG_DELAY 10000
void loop() {
  static unsigned long nextMsgTime = 0;
  if (millis() > nextMsgTime)
  {
    Serial.print("Try http://");
    Serial.print(getIP()); Serial.println("/change/");
    nextMsgTime = millis() + MSG_DELAY;
  }

  maintain(); // Maintain the DHCP lease manually if needed

  client = server.available();
  if (client)

  {
    int type = 0;
    while (client.connected())
```

```cpp
  {
    if (client.available())


    {
      // GET, POST, or HEAD
      memset(buffer,0, sizeof(buffer)); // clear the buffer
      if(client.readBytesUntil('/', buffer,MAX_PAGENAME_LEN))
      {

        if(strcmp(buffer, "GET ") == 0 )
          type = 1;
        else if(strcmp(buffer,"POST ") == 0)
          type = 2;
        // look for the page name
        memset(buffer,0, sizeof(buffer)); // clear the buffer
        if(client.readBytesUntil( '/', buffer,MAX_PAGENAME_LEN ))


        {
          if(strcasecmp(buffer, "analog") == 0)
            showAnalog();
          else if(strcasecmp(buffer, "digital") == 0)
            showDigital();
          else if(strcmp(buffer, "change")== 0)
            showChange(type == 2);
          else
            unknownPage(buffer);
        }
      }
      break;
    }
  }
  // give the web browser time to receive the data
  delay(100);
  client.stop();
  }
}


void showAnalog()
{
  Serial.println(F("analog"));
  sendHeader("Multi-page example-Analog");
  client.println("<h1>Analog Pins</h1>");
  // output the value of each analog input pin
  client.println(F("<table border='1' >"));
```

```
    for (int i = 0; i < 6; i++)


    {
      client.print(F("<tr><td>analog pin "));
      client.print(i);
      client.print(F(" </td><td>"));
      client.print(analogRead(i));
      client.println(F("</td></tr>"));
    }
    client.println(F("</table>"));
    client.println(F("</body></html>"));
}


// mime encoded data for the led on and off images:
// see: http://www.motobit.com/util/base64-decoder-encoder.asp
P(led_on) =  "<img src=\"data:image/jpg;base64,"
```

```
"/9j/4AAQSkZJRgABAgAAZABkAAD/7AARRHVja3kAAQAEAAAAHgAA/+4ADkFkb2JlAGTAAAAAAf/b"
"AIQAEAsLCwwLEAwMEBcPDQ8XGxQQEBQbHxcXFxcXHx4XGhoaGhceHiMlJyUjHi8vMzMvL0BAQEBA"
"QEBAQEBAQEBAQAERDw8RExEVEhIVFBEUERQaFBYWFBomGhocGhomMCMeHh4eIzArLicnJy4rNTUw"
"MDU1QEA/QEBAQEBAQEBAQEBA/8AAEQgAGwAZAwEiAAIRAQMRAf/EAIIAAAICAwAAAAAAAAAAAAAA"
"AAUGAAcCAwQBAAMBAAMBAAAAAAAAAAAAAAAAAAAACBAUQAAECBAQBCgcAAAAAAAAAAECAwARMRIhQQQF"
"UWFxkaHRMoITUwYiQnKSIxQ1EQAAwYEBwAAAAAAAAAAAAARECEgMTBBQhQWEiMVGBMkJiJP/a"
"AAwDAQACEQMRAD8AcNz3BGibKie0nhC0v3A+teKJt8JmZEdHuZalOitgUoHnEpQEWtSyLqgACWFI"
"nixWiaQhsUFFBiQSbiMvvrmeCBp27eLnG7lFTDxs+Kra8oOyium3ltJUAcDIy4EUMN/7Dnq9cPMO"
"W90E9kxeyF2d3HFOQ175olKudUm7TqlfKqDQEDOFR1sNqtC7k5ERYjndNPFSArtvnI/nV+ed9coI"
"ktd2BgozrSZO3J5jVEXRcwD2bbXNdq0zT+BohTyjgPp5SYdPJZ9NP2jsiIz7vhjLohtjnqJ/ouPK"
"co//2Q=="
```
```
"\"/>";


P(led_off) = "<img src=\"data:image/jpg;base64,"
```

```
"/9j/4AAQSkZJRgABAgAAZABkAAD/7AARRHVja3kAAQAEAAAAHgAA/+4ADkFkb2JlAGTAAAAAAf/b"
"AIQAEAsLCwwLEAwMEBcPDQ8XGxQQEBQbHxcXFxcXHx4XGhoaGhceHiMlJyUjHi8vMzMvL0BAQEBA"
"QEBAQEBAQAERDw8RExEVEhIVFBEUERQaFBYWFBomGhocGhomMCMeHh4eIzArLicnJy4rNTUw"
"MDU1QEA/QEBAQEBAQEBA/8AAEQgAHAAZAwEiAAIRAQMRAf/EAHgAAQEAAwAAAAAAAAAAAAAAAAAA"
"AAYFAgQHAQEBAQAAAAAAAAAAAAAACAQQQAAECBQAHBQkAAAAAAAAAAAAABABECIWESA1ETIyIE/9oADAMBAAIRAxEA"
"PwBvl5SWEkkylpJMGsj1XjXSE1kCQuJ8Iy9W5DoxradFa6VDf8IJZAQ6loNtBooTJaqp3DP5oBlV"
"nWrTpEouQS/Cf4PO0uKbqWHGXTSlztSvuVFiZjmfLH3GUuMkzSoTMu8aiNsXet5/17hFyo6PR64V"
"ZnuqfqDDDySFpNpYH3E6aFjzGBr2DkMuFBSFDsWkilUdLftW13pWpcdWqnbBzI/l6hVXKZlROUSe"
"L1KX5zvAPXESjdHsTFWpxLKOJ54hIA1DZCj+Vx/3r96fCNrkvRaT0+V3zV/llplr9sVeHZui/ONk"
"H3dzt6cL/9k="
```
```
"\"/>";


void showDigital()
{
  Serial.println(F("digital"));
```

```cpp
  sendHeader("Multi-page example-Digital");
  client.println(F("<h2>Digital Pins</h2>"));
  // show the value of digital pins
  client.println(F("<table border='1'>"));
  for (int i = 2; i < 8; i++)

  {
    pinMode(i, INPUT_PULLUP);
    client.print(F("<tr><td>digital pin "));
    client.print(i);
    client.print(F(" </td><td>"));
    if(digitalRead(i) == HIGH)
      printP(led_off);
    else
      printP(led_on);
    client.println(F("</td></tr>"));
  }
  client.println(F("</table>"));

  client.println(F("</body></html>"));
}


void showChange(bool isPost)
{
  Serial.println(F("change"));
  if(isPost)
  {
    Serial.println("isPost");
    client.find("\r\n\r\n"); // skip to the body
    // find parameters starting with "pin" and stop on the first blank line
    Serial.println(F("searching for parms"));
    while(client.findUntil("pinD", "\r\n"))
    {
      int pin = client.parseInt();        // the pin number
      int val = client.parseInt();        // 0 or 1
      Serial.print(pin);
      Serial.print("=");
      Serial.println(val);
      pinMode(pin, OUTPUT);
      digitalWrite(pin, val);
    }
  }
  sendHeader("Multi-page example-change");

  // table with buttons from 2 through 9
  // 2 to 5 are inputs, the other buttons are outputs
```

```
client.println(F("<table border='1'>"));

// show the input pins
for (int i = 2; i < 6; i++) // pins 2-5 are inputs
{

  pinMode(i, INPUT_PULLUP);
  client.print(F("<tr><td>digital input "));
  client.print(i);
  client.print(F(" </td><td>"));

  client.print(F("&nbsp </td><td>"));
  client.print(F(" </td><td>"));
  client.print(F("&nbsp </td><td>"));

  if(digitalRead(i) == HIGH)
    printP(led_off);
  else
    printP(led_on);
  client.println("</td></tr>");
}

// show output pins 6-9
// note pins 10-13 are used by the ethernet shield
for (int i = 6; i < 10; i++)

{
  client.print(F("<tr><td>digital output "));
  client.print(i);
  client.print(F(" </td><td>"));
  htmlButton( "On", "pinD", i, "1");
  client.print(F(" </td><td>"));
  client.print(F(" </td><td>"));
  htmlButton("Off", "pinD", i, "0");
  client.print(F(" </td><td>"));

  if(digitalRead(i) == LOW)
    printP(led_off);
  else
    printP(led_on);
  client.println(F("</td></tr>"));
}
  client.println(F("</table>"));
}
```

```cpp
// create an HTML button
void htmlButton( char * label, char *name, int nameId, char *value)
{
  client.print(F(
    "<form action='/change/' method='POST'><p><input type='hidden' name='"));
  client.print(name);
  client.print(nameId);
  client.print(F("' value='"));
  client.print(value);
  client.print(F("'><input type='submit' value='"));
  client.print(label);
  client.print(F("'/></form>"));
}


void unknownPage(char *page)
{
  Serial.print(F("Unknown : "));
  Serial.println(F("page"));

  sendHeader("Unknown Page");
  client.println(F("<h1>Unknown Page</h1>"));
  client.println(page);
  client.println(F("</body></html>"));
}


void sendHeader(char *title)
{
  // send a standard http response header
  client.println(F("HTTP/1.1 200 OK"));
  client.println(F("Content-Type: text/html"));
  client.println();
  client.print(F("<html><head><title>"));
  client.println(title);
  client.println(F("</title><body>"));
}

void printP(const char *str)
{

  // copy data out of program memory into local storage, write out in
  // chunks of 32 bytes to avoid extra short TCP/IP packets
  // from webduino library Copyright 2009 Ben Combee, Ran Talbott
  uint8_t buffer[32];
  size_t bufferEnd = 0;
```

```
    while (buffer[bufferEnd++] = pgm_read_byte(str++))
    {
      if (bufferEnd == 32)
      {
        client.write(buffer, 32);
        bufferEnd = 0;
      }
    }


    // write out everything left but trailing NUL
    if (bufferEnd > 1)
      client.write(buffer, bufferEnd - 1);
  }
```

## Discussion

The logic used to create the web page is similar to that used in the previous recipes. The form here is based on Recipe 15.12, but it has more elements in the table and uses embedded graphical objects to represent the state of the pins. If you have ever created a web page, you may be familiar with the use of JPEG images within the page. The Arduino Ethernet libraries do not have the capability to handle images in *.jpg* format.

Images need to be encoded using one of the internet standards such as Multipurpose Internet Mail Extensions (MIME). This provides a way to represent graphical (or other) media using text. The sketch in this recipe's Solution shows what the LED images look like when they are MIME-encoded. Many web-based services will MIME-encode your images; the ones in this recipe were created using the this service.

Even the small LED images used in this example are too large to fit into AVR RAM. Program memory (flash) is used; see Recipe 17.3 for an explanation of the `P(name)` expression. The sketch only employes this feature if running under an AVR. 32-bit Arduino boards are able to store more, and the compiler is smarter about storage of static strings in general.

The images representing the LED on and off states are stored in a sequence of characters; the LED on array begins like this:

```
P(led_on) =  "<img src=\"data:image/jpg;base64,"
```

P(led_on) = defines `led_on` as the name of this array. The characters are the HTML tags identifying an image followed by the MIME-encoded data comprising the image.

This example is based on code produced for the Webduino web server. Although it is not actively maintained at the time of this writing, you may find Webduino helpful in building web pages if your application is more complicated than the examples shown in this chapter.

## See Also

See Recipe 17.4 for more on using the `F("text")` construct for storing text in flash memory.

The Webduino web page

# 15.14 Sending Twitter Messages

## Problem

You want Arduino to send messages to Twitter; for example, when a sensor detects some activity that you want to monitor via Twitter.

## Solution

This sketch sends a Twitter message when a switch is closed. It uses a proxy service from ThingSpeak to provide authorization so you will need to register on that site to get a (free) API key. Click the Sign Up button on the home page and fill in the form. By creating an account, you get a ThingSpeak API key. To use the ThingSpeak service, you'll need to authorize your Twitter account to allow ThingTweet to post messages to your account (start at the ThingTweets page). After that is set up, replace `"YourThingTweetAPIKey"` with the key string you are given and upload and run the following sketch:

> **NOTE**
>
> You must set up the three header files as described in Recipe 15.6 and uncomment one of the
> `#include` lines to choose which kind of network connection to use. If you use WiFiNINA or
> ESP8266, you'll need to change your SSID and password in the corresponding header file.

```
/*
 * ThingTweet Sketch: post tweet when switch on pin 2 is pressed
 */


// Uncomment only one of the following
//#include "USE_NINA.h"     // WiFiNINA boards
//#include "USE_Ethernet.h" // Ethernet
//#include "USE_ESP8266.h"  // ESP8266 boards

char *thingtweetAPIKey = "YourThingTweetAPIKey";  // your ThingTweet API key
char serverName[] = "api.thingspeak.com";


bool MsgSent = false;
const int btnPin = 2;


void setup()
{
  Serial.begin(9600);
  while (!Serial);
  if (!configureNetwork()) // Start the network
  {
    Serial.println("Failed to configure the network");
    while (1)
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
  }
  pinMode(btnPin, INPUT_PULLUP);
  delay(1000);
  Serial.println("Ready");
}


void loop()
{
  if (digitalRead(btnPin) == LOW) // here if button is pressed
  {
    if (MsgSent == false) // check if message already sent
    {
```

```
        MsgSent = sendMessage("I pressed a button on something #thingspeak");
        if (MsgSent)
          Serial.println("Tweeted successfully");
        else
          Serial.println("Unable to Tweet");
      }
    }
    else
    {
      MsgSent = false;  // Button is not pressed
    }
    delay(100);
}

bool sendMessage(char *message)
{
  bool result = false;

  const int tagLen = 16; // the number of tag character used to frame the
message
  int msgLen = strlen(message) + tagLen + strlen(thingtweetAPIKey);
  Serial.println("Connecting ...");
  if (client.connect(serverName, 80) )
  {
    Serial.println("Making POST request...");
    client.println("POST /apps/thingtweet/1/statuses/update HTTP/1.1");
    client.print("Host: "); client.println(serverName);
    client.println("Connection: close");
    client.println("Content-Type: application/x-www-form-urlencoded");
    client.print("Content-Length: "); client.println(msgLen);
    client.println();
    client.print("api_key=");         // msg tag
    client.print(thingtweetAPIKey);   // api key
    client.print("&status=");         // msg tag
    client.print(message);            // the message
    client.println();
  }
  else

  {
    Serial.println("Connection failed");
  }
  // response string
  if (client.connected())
  {
    Serial.println("Connected");
```

```
    if (client.find("HTTP/1.1") && client.find("200 OK") ) {
      result = true;
    }
    else
      Serial.println("Dropping connection - no 200 OK");
  }
  else
  {
    Serial.println("Disconnected");
  }
  client.stop();
  client.flush();

  return result;
}
```

## Discussion

The sketch waits for a pin to go `LOW` and then posts your message to Twitter via the ThingTweet API.

The web interface is handled by the `sendMessage();` function, which will tweet the given message string. In this sketch it attempts to send the message string "Mail has been delivered" to Twitter and returns `true` if it is able to connect.

See the documentation on the ThingTweet site for more details.

## See Also

This ThingSpeak Arduino tutorial

This information on an Arduino Twitter library that communicates directly with Twitter

# 15.15 Exchanging Data for the Internet of Things

## Problem

You want to exchange data between devices connected via the internet.

## Solution

Use the Message Queue Telemetry Transport (MQTT) protocol on an internet connected Arduino to send or receive data. MQTT is a fast and lightweight protocol for sending (publishing) and receiving (subscribing to) data. It runs well on Arduino and is easy to use, which makes it suitable for Internet of Things projects.

MQTT relies on an internet-connected server (called a broker) to relay published data to clients. Data producers send (publish) messages on a topic to a broker. Consumers connect to the broker and subscribe to one or more topics. Brokers match received messages published on a topic and deliver these to all subscribers to that topic.

Although boards with limited memory like the Uno can publish and subscribe, these boards do not have enough horsepower to run a broker. You can run your own broker on a computer running Windows, Linux, or macOS, or you can use one of the cloud-based public brokers that are free to use such as mqtt.eclipse.org and test.mosquitto.org. More are listed here.

The recipes that follow show how to connect to the Eclipse IoT broker. You will find information on connecting to other public brokers on their respective sites.

If you want to install a broker on your computer, a popular broker is the open source Mosquitto project.

## Discussion

Arduino communicates with MQTT brokers using a library. Two popular solutions that can be installed using the Library Manager are:

- PubSubClientby Nick O'Leary
- Adafruit MQTT library

The following two recipes show how to publish and subscribe using the PubSubClient MQTT library.

## See Also

You can read more about MQTT at .

# 15.16 Publishing Data to an MQTT Broker

## Problem

You want to publish data to an MQTT broker.

## Solution

This sketch uses the PubSubClient library by Nick O'Leary to publish the value of analog pin 0 to a topic named "esp/alog". You can install this library using the Arduino Library Manager. You must set up the three header files as described in Recipe 15.6 and uncomment one of the #include lines to choose which kind of network connection to use. If you use WiFiNINA or ESP8266, you'll need to change your SSID and password in the corresponding header file:

```cpp
/*
 * Basic MQTT publish sketch
 */

// Uncomment only one of the following
//#include "USE_NINA.h"      // WiFiNINA boards
//#include "USE_Ethernet.h" // Ethernet
//#include "USE_ESP8266.h"   // ESP8266 boards

#include <PubSubClient.h>

const char* broker = "mqtt.eclipse.org"; // the address of the MQTT broker

const int interval = 5000;    // milliseconds between events
unsigned int timePublished;  // millis time of most recent publish

PubSubClient psclient(client);
```

```
void setup()
{
  Serial.begin(9600);
  if (!configureNetwork()) // Start the network
  {
    Serial.println("Failed to configure the network");
    while(1)
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
  }
  psclient.setServer(broker, 1883);
}


void loop(void)
{
  if (millis() - timePublished > interval)


  {
    if (!psclient.connected())
      psclient.connect("arduinoCookbook3Pub");
    if (psclient.connected())


    {
      int val = analogRead(A0);
      psclient.publish("arck3/alog", String(val).c_str());
      timePublished = millis();
      Serial.print("Published "); Serial.println(val);
    }
  }
  if (psclient.connected())
    psclient.loop();
}
```

## Discussion

The variable named `broker` is set to the address of the MQTT broker you want to connect to. This example connects to the public `mqtt.eclipse.org` broker. This code sets the broker address and the port to use (port 1883 is the default port for MQTT connections see the documentation for your broker to check if a different port is required):

```
const char* broker = "mqtt.eclipse.org"; // the address of the MQTT broker

psclient.setServer(broker, 1883);
```

The loop code checks if enough time has elapsed for another sample to be published and if so, the variable `val` is set to the value of analog pin 0. The library function to publish data is called with the topic string and string value:

```
psclient.publish("arck3/alog", String(val).c_str());
```

Because the `publish` method expects a C string (a null-terminated sequence of characters), the expression `String(val).c_str()` converts the integer value from analog read to an Arduino `String` and returns the value as a C string.

To see the published data, you need an MQTT client that is subscribed to the esp/alog topic. A suitable client is described in the next recipe.

## See Also

See Recipe 2.5 for more on Arduino `Strings`.

# 15.17 Subscribing to Data on an MQTT Broker

## Problem

You want to subscribe to data that has been published on an MQTT broker.

## Solution

This sketch uses the PubSubCLient library mentioned in the previous recipe to subscribe to the data published in that recipe. You must set up the three header files as described in Recipe 15.6 and uncomment one of the `#include` lines to choose which kind of network connection to use. If you

use WiFiNINA or ESP8266, you'll need to change your SSID and password in the corresponding header file:

```
/*
 *  Basic MQTT subscribe sketch
 */


// Uncomment only one of the following
//#include "USE_NINA.h"      // WiFiNINA boards
//#include "USE_Ethernet.h" // Ethernet
//#include "USE_ESP8266.h"  // ESP8266 boards

#include <PubSubClient.h>

const char* broker = "mqtt.eclipse.org"; // the address of the MQTT broker
const int interval = 5000;                // milliseconds between events
unsigned int timePublished;               // time of most recent publish

PubSubClient psclient(client);

void callback(char* topic, byte* payload, unsigned int length)

{
  Serial.print("Message on topic [");
  Serial.print(topic);
  Serial.print("] ");
  for (int i=0; i < length; i++)

  {
    Serial.write(payload[i]);
  }
  Serial.println();
}

void setup()
{
  Serial.begin(9600);
  if (!configureNetwork()) // Start the network
  {
    Serial.println("Failed to configure the network");
    while(1)
      delay(0); // halt; ESP8266 does not like ∞ loop without a delay
```

```
  }
  psclient.setServer(broker, 1883);
  psclient.setCallback(callback);
}


void loop(void)
{

  if (!psclient.connected())


  {
    if (psclient.connect("arduinoCookbook3Sub"))


    {

      Serial.println("Subscribing to arck3/alog");
      psclient.subscribe("arck3/alog");
    }
  }
  if (psclient.connected())
    psclient.loop();
}
```

The connection to the broker is similar to the previous sketch. The main differences are:


- A callback function is defined that will be called when an event for a subscribed topic has been published.
- The sketch sends a different identifier (`arduinoCookbook3Sub`) when calling `connect` so that it can be uniquely identified.
- The `loop` function subscribes to a topic. In the previous recipe the `loop` published data to a topic.

## Discussion

A callback function is a function passed to a second function that is executed when determined by the second function. In this case, the second

function is the MQTT library that will execute the callback when something is published for a subscribed topic.

The callback function receives the topic and payload. The topic is a null-terminated string; however, the payload data could include binary information with null values so the length of the data is provided to define the end of the message.

# 15.18 Getting the Time from an Internet Time Server

**Problem**

You want to get the current time from an internet time server; for example, to synchronize clock software running on Arduino.

**Solution**

This sketch gets the time from a Network Time Protocol (NTP) server and prints the results as seconds since January 1, 1900 (NTP time) and seconds since January 1, 1970:

```
/*
 * UdpNtp sketch
 * Get the time from an NTP time server
 * Demonstrates use of UDP sendPacket and ReceivePacket
 */

#include <SPI.h>
#include <Ethernet.h>

#include <EthernetUDP.h>

byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };   // MAC address to use

unsigned int localPort = 8888;        // local port to listen for UDP packets
```

```
IPAddress timeServer(192, 43, 244, 18); // time.nist.gov NTP server
const int NTP_PACKET_SIZE= 48; // NTP time stamp is in the first 48
                                // bytes of the message
byte packetBuffer[ NTP_PACKET_SIZE]; // buffer to hold incoming/outgoing
packets

// A UDP instance to let us send and receive packets over UDP
EthernetUDP Udp;

void setup()
{
  Serial.begin(9600);
  // start Ethernet and UDP
  if (Ethernet.begin(mac) == 0) {
    Serial.println("Failed to configure Ethernet using DHCP");
    while(1); // halt
  }
  Udp.begin(localPort);
}

void loop()
{
  sendNTPpacket(timeServer); // send an NTP packet to a time server
  // wait to see if a reply is available
  delay(1000);
  if ( Udp.parsePacket() )

  {

    Udp.read(packetBuffer,NTP_PACKET_SIZE);  // read packet into buffer

    //the timestamp starts at byte 40, convert four bytes into a long integer
    unsigned long hi = word(packetBuffer[40], packetBuffer[41]);
    unsigned long low = word(packetBuffer[42], packetBuffer[43]);
    unsigned long secsSince1900 = hi << 16 | low;  // this is NTP time
     unsigned long secsSince1900 = hi << 16 | low;  // this is NTP time
    unsigned long secsSince1900 = hi << 16 | low;  // this is NTP time
                                                    // (seconds since Jan 1
1900)

    Serial.print("Seconds since Jan 1 1900 = " );
    Serial.println(secsSince1900);
```

```
    Serial.print("Unix time = ");
    // Unix time starts on Jan 1 1970
    const unsigned long seventyYears = 2208988800UL;

    unsigned long epoch = secsSince1900 - seventyYears;   // subtract 70 years
    Serial.println(epoch);                                // print Unix time

    // print the hour, minute and second:
    // UTC is the time at Greenwich Meridian (GMT)
    Serial.print("The UTC time is ");

    // print the hour (86400 equals secs per day)
    Serial.print((epoch  % 86400L) / 3600);

    Serial.print(':');
    if ( ((epoch % 3600) / 60) < 10 )

    {
      // Add leading zero for the first 10 minutes of each hour
      Serial.print('0');
    }
    // print the minute (3600 equals secs per minute)
    Serial.print((epoch  % 3600) / 60);
    Serial.print(':');
    if ( (epoch % 60) < 10 )

    {
      // Add leading zero for the first 10 seconds of each minute
      Serial.print('0');
    }
    Serial.println(epoch %60); // print the second
  }
  // wait ten seconds before asking for the time again
  delay(10000);
}

// send an NTP request to the time server at the given address
unsigned long sendNTPpacket(IPAddress& address)
{
  memset(packetBuffer, 0, NTP_PACKET_SIZE);  // set all bytes in the buffer to
0

  // Initialize values needed to form NTP request
  packetBuffer[0] = B11100011;   // LI, Version, Mode
```

```
  packetBuffer[1] = 0;      // Stratum
  packetBuffer[2] = 6;      // Max Interval between messages in seconds
  packetBuffer[3] = 0xEC;   // Clock Precision
  // bytes 4 - 11 are for Root Delay and Dispersion and were set to 0 by
memset
  packetBuffer[12]  = 49;   // four byte reference ID identifying
  packetBuffer[13]  = 0x4E;
  packetBuffer[14]  = 49;
  packetBuffer[15]  = 52;


  // all NTP fields have been given values, now
  // you can send a packet requesting a timestamp:
  Udp.beginPacket(address, 123); //NTP requests are to port 123
  Udp.write(packetBuffer,NTP_PACKET_SIZE);
  Udp.endPacket();
}
```

## Discussion

NTP is a protocol used to synchronize time through internet messages. NTP servers provide time as a value of the number of seconds that have elapsed since January 1, 1900. NTP time is UTC (Coordinated Universal Time, similar to Greenwich Mean Time) and does not take time zones or daylight saving time into account.

NTP servers use UDP messages; see Recipe 15.3 for an introduction to UDP. An NTP message is constructed in the function named sendNTPpacket and you are unlikely to need to change the code in that function. The function takes the address of an NTP server; you can use the IP address in the preceding example or find a list of many more by using "NTP address" as a search term in Google. If you want more information about the purpose of the NTP fields, see the documentation.

The reply from NTP is a message with a fixed format; the time information consists of four bytes starting at byte 40. These four bytes are a 32-bit value (an unsigned long integer), which is the number of seconds since January 1, 1900. This value (and the time converted into Unix time) is printed. If you want to convert the time from an NTP server to the friendlier format using hours, minutes, and seconds and days, months, and years, you can use the

Arduino Time library (see Chapter 12). Here is a variation on the preceding code that prints the time as `14:32:56 Monday 18 Jan 2010`:

```
/*
 * Time_NTP sketch
 * Example showing time sync to NTP time source
 * This sketch uses the Time library
 * and the Arduino Ethernet library
 */


#include <TimeLib.h> // see text
#include <SPI.h>
#include <Ethernet.h>
#include <EthernetUDP.h>


byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 192, 168, 1, 44 }; // set this to a valid IP address (or use
DHCP)

unsigned int localPort = 8888;      // local port to listen for UDP packets


IPAddress timeServer(192, 43, 244, 18); // time.nist.gov NTP server


const int NTP_PACKET_SIZE= 48; // NTP time stamp is in first 48 bytes of
message
byte packetBuffer[ NTP_PACKET_SIZE]; // buffer to hold incoming/outgoing
packets

time_t prevDisplay = 0;    // when the digital clock was displayed


// A UDP instance to let us send and receive packets over UDP
EthernetUDP Udp;


void setup()
{
  Serial.begin(9600);
  Ethernet.begin(mac,ip);
  Udp.begin(localPort);
  Serial.println("waiting for sync");
  setSyncProvider(getNtpTime);
  while(timeStatus()== timeNotSet)
```

```
    ;  // wait until the time is set by the sync provider
}

void loop()
{
  if( now() != prevDisplay)   //update the display only if the time has
changed
  {
    prevDisplay = now();
    digitalClockDisplay();
  }
}

void digitalClockDisplay(){
  // digital clock display of the time
  Serial.print(hour());
  printDigits(minute());
  printDigits(second());
  Serial.print(" ");
  Serial.print(dayStr(weekday()));
  Serial.print(" ");
  Serial.print(day());
  Serial.print(" ");
  Serial.print(monthShortStr(month()));
  Serial.print(" ");
  Serial.print(year());
  Serial.println();
}

void printDigits(int digits){
  // utility function for digital clock display: prints preceding
  // colon and leading 0
  Serial.print(":");
  if(digits < 10)
    Serial.print('0');
  Serial.print(digits);
}

/*-------- NTP code ----------*/

unsigned long getNtpTime()
{
  sendNTPpacket(timeServer); // send an NTP packet to a time server
  delay(1000);
  if ( Udp.parsePacket() ) {
```

```
    Udp.read(packetBuffer,NTP_PACKET_SIZE);  // read packet into buffer


    //the timestamp starts at byte 40, convert four bytes into a long integer
    unsigned long hi = word(packetBuffer[40], packetBuffer[41]);
    unsigned long low = word(packetBuffer[42], packetBuffer[43]);
    // this is NTP time (seconds since Jan 1 1900
    unsigned long secsSince1900 = hi << 16 | low;


    // Unix time starts on Jan 1 1970
    const unsigned long seventyYears = 2208988800UL;


    unsigned long epoch = secsSince1900 - seventyYears;  // subtract 70 years
    return epoch;
  }
  return 0; // return 0 if unable to get the time
}


// send an NTP request to the time server at the given address
unsigned long sendNTPpacket(IPAddress address)
{
  memset(packetBuffer, 0, NTP_PACKET_SIZE);  // set all bytes in the buffer to
0

  // Initialize values needed to form NTP request
  packetBuffer[0] = B11100011;   // LI, Version, Mode
  packetBuffer[1] = 0;        // Stratum
  packetBuffer[2] = 6;        // Max Interval between messages in seconds
  packetBuffer[3] = 0xEC;   // Clock Precision
  // bytes 4 - 11 are for Root Delay and Dispersion and were set to 0 by
memset
  packetBuffer[12]  = 49;  // four-byte reference ID identifying
  packetBuffer[13]  = 0x4E;
  packetBuffer[14]  = 49;
  packetBuffer[15]  = 52;


  // send the packet requesting a timestamp:
  Udp.beginPacket(address, 123); //NTP requests are to port 123
  Udp.write(packetBuffer,NTP_PACKET_SIZE);
  Udp.endPacket();
}
```

> ### Error Message That "Something" Was Not Declared in This Scope
>
> The Arduino IDE is telling you that it does not recognize something. If the missing item is a library function, such as `setSyncProvider`, you have not included or not installed the library. If you see this message, see Chapter 12 for information on installing the Time library.

## See Also

Chapter 12 provides more information on using the Arduino Time library.

Details on NTP

The NTP code by Jesse Jaggars that inspired the sketch used in this recipe

If you are running an Arduino release prior to 1.0, you can download this UDP library.

# Using, Modifying, and Creating Libraries

## 16.0 Introduction

Libraries add functionality to the Arduino environment. They extend the commands available to provide capabilities not available in the core Arduino language. Libraries provide a way to add features that can be accessed from any of your sketches once you have installed the library.

The Arduino software distribution includes built-in libraries that cover common tasks. These libraries are discussed in Recipe 16.1.

Libraries are also a good way for people to share code that may be useful to others. Many third-party libraries provide specialized capabilities; these can be downloaded from the Arduino Library Manager but also from GitHub. Libraries are often written to simplify the use of a particular piece of hardware. Many of the devices covered in earlier chapters use libraries to make it easier to connect to the devices.

Libraries can also provide a friendly wrapper around complex code to make it easier to use. An example is the Wire library distributed with Arduino, which hides much of the complexity of low-level hardware communications (see Chapter 13).

This chapter explains how to use and modify libraries. It also gives examples of how to create your own libraries.

## 16.1 Using the Built-in Libraries

### Problem

You want to use the libraries provided with the Arduino distribution in your sketch.

## Solution

This recipe shows you how to use Arduino library functionality in your sketch.

To see the list of available libraries from the IDE menu, click Sketch→Include Library. A list will drop down showing all the available libraries. The first dozen or so are the libraries distributed with Arduino. A horizontal line separates that list from the libraries that you download and install yourself.

Clicking a library will add that library to the current sketch, by adding the following line to the top of the sketch:

```
#include <nameOfTheLibrarySelected.h>
```

This results in the functions within the library becoming available to use in your sketch.

> **NOTE**
>
> The Arduino IDE updates its list of available libraries only when the IDE is first started on your computer. If you manually install a library after the IDE is running, you need to close the IDE and restart for that new library to be recognized. If you install a library through the Library Manager, you won't need to restart the IDE.

The Arduino libraries are documented in the Arduino Reference and each library includes example sketches demonstrating their use. Chapter 1 has details on how to navigate to the examples in the IDE.

The libraries that are included with Arduino as of version 1.8.10 are:

Adafruit Circuit Playground

Supports Adafruit's Circuit Playground board, which includes many sensors and outputs for quick and easy prototyping.

Bridge

This is used by the now-discontinued Arduino Yun, Yun Shield, and TRE. It allows communication between the Linux system and microcontroller on those boards.

Esplora

This is used by the now-discontinued Arduino Esplora board, an all-in-one board that includes sensors, joystick, LEDs, buzzers, and other inputs and outputs.

EEPROM

Used to store and read information in memory that is preserved when power is removed; see Chapter 18.

Ethernet

Used to communicate with the Arduino Ethernet shield, compatible modules such as the Adafruit Ethernet FeatherWing, or for use with the Arduino Ethernet board; see Chapter 15.

Firmata

A protocol used to simplify serial communication and control of the board.

GSM

Supports the now-discontinued Arduino GSM shield, which connects Arduino to cellular data networks.

HID

Allows certain boards, such as the Arduino Leonardo and SAMD-based boards, to function as a mouse or keyboard. You won't use this library directly, but the Keyboard and Mouse libraries depend on it.

Keyboard

Allows certain boards, such as the Arduino Leonardo and SAMD-based boards, to function as a USB keyboard.

LiquidCrystal

For controlling compatible LCD displays; see <span style="color:#8B0000">Chapter 11</span>.

Mouse

Allows certain boards, such as the Arduino Leonardo and SAMD-based boards, to function as a USB mouse.

Robot Control

Supports operation of the now-discontinued Arduino Robot's control board.

Robot IR Remote

Supports operation of the now-discontinued Arduino Robot's infrared remote.

Robot Motor

Supports operation of the now-discontinued Arduino Robot's motor board.

SD

Supports reading and writing files to an SD card using external hardware.

Servo

Used to control servo motors; see <span style="color:#8B0000">Chapter 8</span>.

SoftwareSerial

Enables additional serial ports.

SpacebrewYun

This is used by the now-discontinued Arduino Yun to enable WebSocket-based communications.

SPI

Used for Ethernet and SPI hardware; see <span style="color:#8B0000">Chapter 13</span>.

Stepper

For working with stepper motors; see Chapter 8.

Temboo

Connects Arduino to Temboo, a platform for connecting to APIs, databases, and code utilities.

TFT

A library to support the now-discontinued Arduino LCD screen. Third-party boards and modules are available along with custom libraries.

WiFi

Supports the now-discontinued Arduino WiFi shield, which has been replaced by Arduino boards with built-in WiFi as well as third-party boards and modules.

Wire

Works with I2C devices attached to the Arduino; see Chapter 13.

The following two libraries can be found in releases prior to Arduino 1.0 but are no longer included with the Arduino distribution:

Matrix

Helps manage a matrix of LEDs; see Chapter 7.

Sprite

Enables the use of sprites with an LED matrix.

## Discussion

Libraries that work with specific hardware within the Arduino controller chip only work on predefined pins. The Wire and SPI libraries are examples of this kind of library. Libraries that allow user selection of pins usually have this specified in `setup`; Servo, LiquidCrystal, and Stepper are examples of that kind of library. See the library documentation for specific information on how to configure the library.

Including a library adds the library code to your sketch behind the scenes. This means the size of your sketch, as reported at the end of the compilation process, will increase, but the Arduino build process is smart enough to only include the code your sketch is actually using from the library, so you don't have to worry about the memory overhead for methods that are not being used. Therefore, you also don't have to worry about unused functions reducing the amount of code you can put into your sketch.

Libraries included with Arduino (and many contributed libraries) include example sketches that show how to use the library. They are accessed from the File → Examples menu.

## See Also

The Arduino reference for libraries
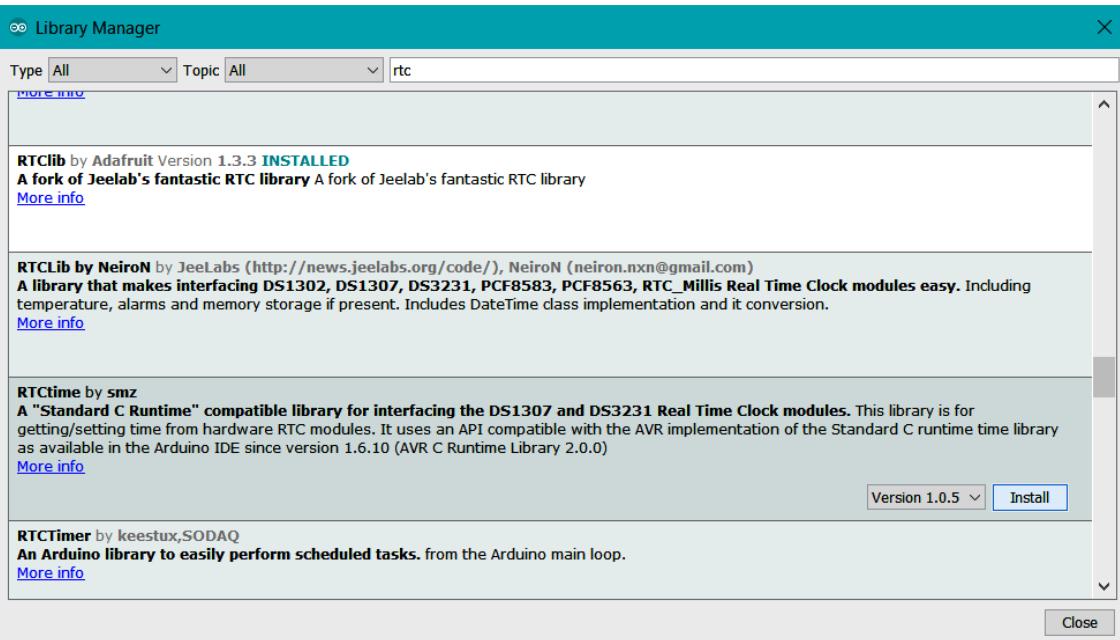
# 16.2 Installing Third-Party Libraries

## Problem

You want to use a library created for use with Arduino but not in the standard distribution.

## Solution

First, check the Library Manager to see if the library is available. Select Tools → Manage Libraries and search for the library you are looking for (or for the name of a component; you can often find libraries that are designed for specific components). Hover over its entry in the list of results, and click Install (see Figure 16-1). It will be ready to use right away.

*Installing a library with the Library Manager*

If the library is not available in the Library Manager, you will need to download the library. If the library is available on GitHub, check the *README* carefully. In most cases, you should be able to download it from the Releases tab just above the list of files or click the Clone or Download button and choose Download Zip. If the library is made available some other way, it will often be a *.zip* file. Unzip it and you will have a folder that has the same title as the name of the library. This folder needs to be put inside a folder called *libraries* inside your Arduino document folder. To find the Arduino document folder, open Preferences (Arduino→Preferences on macOS; File→Preferences on Windows or Linux) and note the sketchbook location. Navigate to that directory in a filesystem browser (such as Windows Explorer or macOS Finder) or at the terminal. If no *libraries* folder exists, create one and put the folder you unzipped inside it.

If the Arduino IDE is still running, quit and restart it. The IDE scans this folder to find libraries only when it is launched. If you now go to the menu Sketch→Import Library, at the bottom, below the gray line and the word *Contributed*, you should see the library you have added.

If the libraries provide example sketches, you can view these from the IDE menu; click File → Examples, and the libraries examples will be under the libraries name in a section between the general examples and the Arduino distributed library example listing.

## Discussion

A large number of libraries are provided by third parties. Many are very high quality, are actively maintained, and provide good documentation and example sketches. Arduino Libraries has a great, regularly updated list of available libraries. The Arduino Playground, although no longer accepting updates, is also a good place to look for libraries.

Look for libraries that have clear documentation and examples. Check out the Arduino forums to see if there are any threads (discussion topics) that discuss the library. Libraries that were designed to be used with early Arduino releases may have problems when used with the latest Arduino version, so you may need to read through a lot of material (some threads for popular libraries contain hundreds of posts) to find information on using an older library with the latest Arduino release.

If the library examples do not appear in the Examples menu or you get a message saying "Library not found" when you try to use the library, check that the *libraries* folder is in the correct place with the name spelled correctly. A library folder named *<LibraryName>* (where *<LibraryName>* is the name for the library) must contain a file named *<LibraryName>.h* with the same spelling and capitalization. Check that additional files needed by the library are in the folder.

# 16.3 Modifying a Library

## Problem

You want to change the behavior of an existing library, perhaps to extend its capability. For example, the TimeAlarms library in Chapter 12 only supports six alarms and you need more (see Recipe 12.5).

## Solution

The Time and TimeAlarms libraries are described in Chapter 12, so refer to Recipe 12.5 to familiarize yourself with the standard functionality. The libraries can be installed using the Library Manager. If you have trouble finding the Time library, try searching the Library Manager for "timekeeping."

Once you have the Time and TimeAlarms libraries installed, compile and upload the following sketch on an AVR-based board, which will attempt to create seven alarms—one more than the libraries support (on AVR that is; ARM and ESP8266 boards support more). Each `Alarm` task simply prints its task number:

```
/*
 * multiple_alarms sketch
 * Has more timer repeats than the library supports out of the box -
 * you will need to edit the header file to enable more than 6 alarms
 */

#include <TimeLib.h>
#include <TimeAlarms.h>

int currentSeconds = 0;

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

  // create 7 alarm tasks
  Alarm.timerRepeat(1, repeatTask1);
  Alarm.timerRepeat(2, repeatTask2);
  Alarm.timerRepeat(3, repeatTask3);
  Alarm.timerRepeat(4, repeatTask4);
  Alarm.timerRepeat(5, repeatTask5);
  Alarm.timerRepeat(6, repeatTask6);
  Alarm.timerRepeat(7, repeatTask7);  // 7th timer repeat
}

void repeatTask1()
{
  Serial.print("task 1  ");
}
```

```
void repeatTask2()
{
   Serial.print("task 2  ");
}
void repeatTask3()
{
   Serial.print("task 3  ");
}

void repeatTask4()
{
   Serial.print("task 4  ");
}

void repeatTask5()
{
   Serial.print("task 5  ");
}

void repeatTask6()
{
   Serial.print("task 6  ");
}

void repeatTask7()
{
   Serial.print("task 7  ");
}

void  loop()
{
  if(second() != currentSeconds)
  {
    // print the time for each new second
    // the task numbers will be printed when the alarm for that task is
triggered
    Serial.println();
    Serial.print(second());
    Serial.print("->");
    currentSeconds = second();
    Alarm.delay(1); // Alarm.delay must be called to service the alarms
  }
}
```

Open the Serial Monitor and watch the output being printed. After nine seconds of output, you should see this:

```
1->task 1
2->task 1   task 2
3->task 1   task 3
4->task 1   task 2   task 4
5->task 1   task 5
6->task 1   task 2   task 3   task 6
7->task 1
8->task 1   task 2   task 4
9->task 1   task 3
```

The task scheduled for seven seconds did not trigger because the library only provides six timer "objects" that you can use.

You can increase this by modifying the library. Go to the *libraries* folder in your Arduino *Documents* folder.

> **NOTE**
>
> You can locate the directory containing the sketchbook folder by clicking the menu item File → Preferences (on Windows or Linux) or Arduino → Preferences (on macOS) in the IDE. A dialog box will open, showing the sketchbook location.

If you have installed the Time and TimeAlarms libraries (both libraries are in the file you downloaded), navigate to the *Libraries\TimeAlarms* folder. Open the *TimeAlarms.h* header file (for more details about header files, see Recipe 16.4). You can edit the file with any text editor; for example, Notepad on Windows or TextEdit on a Mac.

You should see the following at the top of the *TimeAlarms.h* file:

```
#ifndef TimeAlarms_h
#define TimeAlarms_h

#include <Arduino.h>
#include "TimeLib.h"

#if defined(__AVR__)
#define dtNBR_ALARMS 6    // max is 255
#else
#define dtNBR_ALARMS 12   // assume non-AVR has more memory
#endif
```

The maximum number of alarms is specified by the value defined for `dtNbr_ALARMS`.

Change:

```
#define dtNBR_ALARMS 6
```

to:

```
#define dtNBR_ALARMS 7
```

and save the file.

Upload the sketch to your Arduino again, and this time the serial output should read:

```
1->task 1
2->task 1  task 2
3->task 1  task 3
4->task 1  task 2  task 4
5->task 1  task 5
6->task 1  task 2  task 3  task 6
7->task 1  task 7
8->task 1  task 2  task 4
9->task 1  task 3
```

You can see that task 7 now activates after seven seconds.

## Discussion

Capabilities offered by a library are a trade-off between the resources used by the library and the resources available to the rest of your sketch, and it is often possible to change these capabilities if required. For example, you may need to decrease the amount of memory used for a serial library so that other code in the sketch has more RAM. Or you may need to increase the memory usage by a library for your application. The library writer generally creates the library to meet typical scenarios, but if your application needs capabilities not catered to by the library writer, you may be able to modify the library to accommodate them.

In this example, the TimeAlarms library allocates room (in RAM) for six alarms. Each of these consumes around a dozen bytes and the space is reserved even if only a few are used. The number of alarms is set in the library header file (the header is a file named *TimeAlarms.h* in the *TimeAlarms* folder).

In the TimeAlarms library, the maximum number of alarms is set using a `#define` statement. Because you changed it and saved the header file when you recompiled the sketch to upload it, it uses the new upper limit.

Sometimes define statements (or constants) are used to define characteristics such as the clock speed of the board, and when used with a board that runs at a different speed, you will get unexpected results. Editing this value in the header file to the correct one for the board you are using will fix this problem.

If you edit the header file and the library stops working, you can always download the library again and replace the whole library to return to the original state.

## See Also

Recipe 16.4 has more details on how you can add functionality to libraries.

# 16.4 Creating Your Own Library

## Problem

You want to create your own library. Libraries are a convenient way to reuse code across multiple sketches and are a good way to share with other users.

## Solution

A library is a collection of methods and variables that are combined in a format that enables users to access functions and variables in a standardized way.

Most Arduino libraries are written as a class. If you are familiar with C++ or Java, you will be familiar with classes. However, you can create a library without using a class, and this recipe shows you how.

This recipe explains how you can transform the sketch from Recipe 7.1 to move the `BlinkLED` function into a library.

See Recipe 7.1 for the wiring diagram and an explanation of the circuit. The library will contain the `blinkLED` function from that recipe. Here is the sketch that will be used to test the library:

```
/*
 * blinkLibTest
 */

#include "blinkLED.h"

const int firstLedPin  = 3;        // choose the pin for each of the LEDs
const int secondLedPin = 5;
const int thirdLedPin  = 6;

void setup()
{
  pinMode(firstLedPin, OUTPUT);    // declare LED pins as output
  pinMode(secondLedPin, OUTPUT);   // declare LED pins as output
  pinMode(thirdLedPin, OUTPUT);    // declare LED pins as output
}

void loop()
{
  // flash each of the LEDs for 1,000 ms (1 second)
  blinkLED(firstLedPin, 1000);
  blinkLED(secondLedPin, 1000);
  blinkLED(thirdLedPin, 1000);
}
```

The `blinkLED` function from Recipe 7.1 should be removed from the sketch and moved into a separate file named *blinkLED.cpp* (see the Discussion for more details about *.cpp* files):

```
/* blinkLED.cpp
 * simple library to light an LED for a duration given in milliseconds
 */
```

```
#include "Arduino.h"    // use: Wprogram.h for Arduino versions prior to 1.0
#include "blinkLED.h"

// blink the LED on the given pin for the duration in milliseconds
void blinkLED(int pin, int duration)
{
  digitalWrite(pin, HIGH);      // turn LED on
  delay(duration);
  digitalWrite(pin, LOW);       // turn LED off
  delay(duration);
}
```

> ### TIP
>
> Most library authors are programmers who use their favorite programming editor, but you can use any plain-text editor to create these files.

Create the *blinkLED.h* header file as follows:

```
/*
 * blinkLED.h
 * Library header file for BlinkLED library
 */
#include "Arduino.h"

void blinkLED(int pin, int duration);  // function prototype
```

## Discussion

The library will be named "blinkLED" and will be located in the *libraries* folder (see Recipe 16.2); create a subdirectory named *blinkLED* in the *libraries* folder and move *blinkLED.h* and *blinkLED.cpp* into it. Next, create a subdirectory of that folder called *examples* and a subdirectory under that folder called *blinkLibTest*. Next, put the contents of the sketch shown earlier into a file named *examples/blinkLibTest/blinkLibTest.ino*.

The `blinkLED` function from Recipe 7.1 is moved out of the sketch and into a library file named *blinkLED.cpp* (the *.cpp* extension stands for "C plus plus" and contains the executable code).

The *blinkLED.cpp* file contains a `blinkLED` function that is identical to the code from Recipe 7.1 with the following two lines added at the top:

```
#include "Arduino.h"  // Arduino include
#include "blinkLED.h"
```

The `#include "Arduino.h"` line is needed by a library that uses any Arduino functions or constants. Without this, the compiler will report errors for all the Arduino functions used in your sketch.

The next line, `#include "blinkLED.h"`, contains the function definitions (also known as *prototypes*) for your library. The Arduino build process creates prototypes for all the functions within a sketch automatically when a sketch is compiled—but it does not create any prototypes for library code, so if you make a library, you must create a header with these prototypes. It

is this header file that is added to a sketch when you import a library from the IDE (see Recipe 16.1).

> **NOTE**
>
> Every library must have a file that declares the names of the functions to be exposed. This file is called a *header file* (also known as an *include* file) and has the form *<LibraryName>.h* (where *<LibraryName>* is the name for your library). In this example, the header file is named *blinkLED.h* and is in the same folder as *blinkLED.cpp*.

The header file for this library is simple. It declares the one function:

```
void blinkLED(int pin, int duration);  // function prototype
```

This looks similar to the function definition in the *blinkLED.cpp* file:

```
void blinkLED(int pin, int duration)
```

The difference is subtle but vital. The header file prototype contains a trailing semicolon. This tells the compiler that this is just a declaration of the form for the function but not the code. The source file, *blinkLED.cpp*, does not contain the trailing semicolon and this informs the compiler that this is the actual source code for the function.

> **TIP**
>
> Libraries can have more than one header file and more than one implementation file. But there must be at least one header and that must match the library name. It is this file that is included at the top of the sketch when you import a library.

A good book on C++ can provide more details on using header and *.cpp* files to create code modules. This recipe's See Also section lists some popular choices.

With the *blinkLED.cpp, blinkLED.h*, and *blinkLibTest.ino* files in the correct place within the *libraries* folder, close the IDE and reopen it. The directory structure should look like this:

```
libraries/
└── blinkLED/
      ├── blinkLED.cpp
      ├── blinkLED.h
      └── examples/
            └── blinkLibTest/
                  └── blinkLibTest.ino
```

---

**NOTE**

The Arduino IDE updates its list of available libraries only when the IDE is first started on your computer. If you create a library after the IDE is running, you need to close the IDE and restart for that library to be recognized. Although you need to close and restart the IDE when you first add the library to the *libraries* folder, you do not need to do so after subsequent changes to the library.

---

Click File → Examples (Examples from Custom Libraries) → blinkLED → blinkLibTest to open the example sketch. Upload the blinkLibTest sketch and you should see the three LEDs blinking.

It's easy to add additional functionality to the library. For example, you can add some constant values for common delays so that users of your libraries can use the descriptive constants instead of millisecond values.

Add the three lines with constant values, traditionally put just before the function prototype, to your header file as follows:

```cpp
// constants for duration
const int BLINK_SHORT = 250;
const int BLINK_MEDIUM = 500;
const int BLINK_LONG = 1000;

void blinkLED(int pin, int duration);  // function prototype
```

Change the code in `loop` as follows and upload the sketch to see the different blink rates:

```
void loop()
{
  blinkLED(firstLedPin, BLINK_SHORT);
  blinkLED(secondLedPin, BLINK_MEDIUM);
  blinkLED(thirdLedPin, BLINK_LONG);
}
```

New functions can be easily added. This example adds a function that
continues blinking for the number of times given by the sketch. Here is the
`loop` code:

```
void loop()
{
  blinkLED(firstLedPin, BLINK_SHORT, 5);    // blink 5 times
  blinkLED(secondLedPin, BLINK_MEDIUM, 3);  // blink 3 times
  blinkLED(thirdLedPin, BLINK_LONG);        // blink once
}
```

To add this functionality to the library, add the prototype to *blinkLED.h* as
follows:

```
/*
 * blinkLED.h
 * Header file for BlinkLED library
 */
#include "Arduino.h"

// constants for duration
const int BLINK_SHORT = 250;
const int BLINK_MEDIUM = 500;
const int BLINK_LONG = 1000;

void blinkLED(int pin, int duration);

// new function for repeat count
void blinkLED(int pin, int duration, int repeats);
```

Add the function into *blinkLED.cpp*:

```
/*
 * blinkLED.cpp
 * simple library to light an LED for a duration given in milliseconds
 */
```

```
#include "Arduino.h"
#include "blinkLED.h"

// blink the LED on the given pin for the duration in milliseconds
void blinkLED(int pin, int duration)
{
  digitalWrite(pin, HIGH);      // turn LED on
  delay(duration);
  digitalWrite(pin, LOW);       // turn LED off
  delay(duration);
}

/* function to repeat blinking */
void blinkLED(int pin, int duration, int repeats)
{
  while(repeats)
  {
    blinkLED(pin, duration);
    repeats = repeats -1;
  }
}
```

You can create a *keywords.txt* file if you want to add *syntax highlighting* (coloring the keywords used in your library when viewing a sketch in the IDE). This is a text file that contains the name of the keyword and the keyword type—each type uses a different color. The keyword and type must be separated by a tab (not a space). For example, save the following file as *keywords.txt* in the *blinkLED* folder (you'll need to quit and restart the IDE when you add or modify a *keywords.txt* file):

```
#######################################
# Methods and Functions (KEYWORD2)
#######################################
blinkLED   KEYWORD2
#######################################
# Constants (LITERAL1)
#######################################
BLINK_SHORT     LITERAL1
BLINK_MEDIUM    LITERAL1
BLINK_LONG      LITERAL1
```

## See Also

See Recipe 16.5 for more examples of writing a library.

This "Writing a Library for Arduino" reference document

Also see the following books on C++:

- *Practical C++ Programming* by Steve Oualline (O'Reilly)

- *C++ Primer Plus* by Stephen Prata (Sams)

- *C++ Primer* by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo (Addison-Wesley Professional)

# 16.5 Creating a Library That Uses Other Libraries

## Problem

You want to create a library that uses functionality from one or more existing libraries. For example, to use the Wire library to get data from a Wii nunchuck game controller.

## Solution

This recipe uses the functions described in Recipe 13.6 to communicate with a Wii nunchuck using the Wire library.

Create a folder named *Nunchuck* in the *libraries* directory (see Recipe 16.4 for details on the file structure for a library). Create a file named *Nunchuck.h* with the following code:

```
/*
 * Nunchuck.h
 * Arduino library to interface with wii Nunchuck
 */

#ifndef Nunchuck_included
#define Nunchuck_included

// identities for each field provided by the wii nunchuck
enum nunchuckItems { wii_joyX, wii_joyY, wii_accelX, wii_accelY, wii_accelZ,
                     wii_btnC, wii_btnZ, wii_ItemCount };
```

```
// uses pins adjacent to I2C pins as power & ground for Nunchuck
void nunchuckSetPowerpins();

// initialize the I2C interface for the nunchuck
void nunchuckInit();

// Request data from the nunchuck
void nunchuckRequest();

// Receive data back from the nunchuck,
// returns true if read successful, else false
bool nunchuckRead();

// Encode data to format that most wiimote drivers accept
char nunchuckDecode (uint8_t x);

// return the value for the given item
int nunchuckGetValue(int item);

#endif
```

Create a file named *Nunchuck.cpp* in the *Nunchuck* folder as follows:

```
/*
 * Nunchuck.cpp
 * Arduino library to interface with wii Nunchuck
 */

#include "Arduino.h"    // Arduino defines

#include "Wire.h"       // Wire (I2C) defines
#include "Nunchuck.h"   // Defines for this library

// Constants for Uno board (use 19 and 18 for mega)
const int vccPin = A3; // +v and gnd provided through these pins
const int gndPin = A2;

const int dataLength = 6;            // number of bytes to request
static byte rawData[dataLength];     // array to store nunchuck data

// uses pins adjacent to I2C pins as power & ground for Nunchuck
void nunchuckSetPowerpins()
{
  pinMode(gndPin, OUTPUT); // set power pins to the correct state
  pinMode(vccPin, OUTPUT);
```

```
  digitalWrite(gndPin, LOW);
  digitalWrite(vccPin, HIGH);
  delay(100); // wait for power to stabilize
}

// initialize the I2C interface for the nunchuck
void nunchuckInit()
{
  Wire.begin();                    // join i2c bus as master
  Wire.beginTransmission(0x52);// transmit to device 0x52
  Wire.write((byte)0x40);       // sends memory address
  Wire.write((byte)0x00);       // sends sent a zero.
  Wire.endTransmission();       // stop transmitting
}

// Request data from the nunchuck
void nunchuckRequest()
{
  Wire.beginTransmission(0x52);// transmit to device 0x52
  Wire.write((byte)0x00);// sends one byte
  Wire.endTransmission();// stop transmitting
}

// Receive data back from the nunchuck,
// returns true if read successful, else false
bool nunchuckRead()
{
  byte cnt=0;
  Wire.requestFrom (0x52, dataLength);// request data from nunchuck
  while (Wire.available ())
  {
    byte x = Wire.read();
    rawData[cnt] = nunchuckDecode(x);
    cnt++;
  }
  nunchuckRequest();  // send request for next data payload
  if (cnt >= dataLength)
    return true;   // success if all 6 bytes received
  else
    return false; // failure
}

// Encode data to format that most wiimote drivers accept
char nunchuckDecode (byte x)
{
  return (x ^ 0x17) + 0x17;
}
```

```
// return the value for the given item
int nunchuckGetValue(int item)
{
  if( item <= wii_accelZ)
    return (int)rawData[item];
  else if(item  == wii_btnZ)
    return bitRead(rawData[5], 0) ? 0: 1;
  else if(item  == wii_btnC)
    return bitRead(rawData[5], 1) ? 0: 1;
}
```

Connect the nunchuck as shown in but use the following sketch to test the library (if Arduino was running while you created the previous two files, quit and restart it so it will see the new library). If you'd like, you can create this as the file *WiichuckSerial.ino* in the folder *examples/WiichuckSerial* underneath the *Nunchuck* library folder to make it available as an example program:

```
/*
 * WiichuckSerial
 *
 *  Uses Nunchuck library to send sensor values to serial port
 */

#include <Wire.h>
#include "Nunchuck.h"

void setup()
{
  Serial.begin(9600);
  nunchuckSetPowerpins();
  nunchuckInit(); // send the initialization handshake
  nunchuckRead(); // ignore the first time
  delay(50);
}

void loop()
{
  nunchuckRead();
  Serial.print("H,");         // header
  for(int i=0; i <  5; i++) // print values of accelerometers and buttons
  {
      Serial.print(nunchuckGetValue(wii_accelX+ i), DEC);
      Serial.write(',');
```

```
  }
  Serial.println();
  delay(20); // the time in milliseconds between sends
}
```

## Discussion

To include another library, use its `include` statement in your code as you would in a sketch. It is sensible to include information about any additional libraries that your library needs in documentation if you make it available for others to use, especially if it requires a library that is not distributed with Arduino.

The major difference between the library code and the sketch from Recipe 13.6 is the addition of the *Nunchuck.h* header file that contains the function prototypes (Arduino sketch code silently creates prototypes for you, unlike Arduino libraries, which require explicit prototypes).

Here is another example of creating a library; this one uses a C++ class to encapsulate the library functions. A class is a programming technique for grouping functions and variables together and is commonly used for most Arduino libraries.

This library can be used as a debugging aid by sending print output to a second Arduino board using the Wire library. This is particularly useful when the hardware serial port is not available and software serial solutions are not appropriate due to the timing delays they introduce. Here the core Arduino print functionality is used to create a new library that sends printed output to I2C. The connections and code are covered in Recipe 13.5. The following description shows how that code can be converted into a library.

Create a folder named *i2cDebug* in the *libraries* directory (see Recipe 16.4 for details on the file structure for a library). Create a file named *i2cDebug.h* with the following code:

```
/*
 * i2cDebug.h
 */
#ifndef i2cDebug_included
#define i2cDebug_included
```

```
#include <Arduino.h>
#include <Print.h>     // the Arduino print class

class i2cDebugClass : public Print
{
  private:
    int i2cAddress;
    byte count;
    size_t write(byte c);
  public:
    i2cDebugClass();
    bool begin(int id);
};

extern i2cDebugClass i2cDebug;    // the i2c debug object
#endif
```

Create a file named *i2cDebug.cpp* in the *i2cDebug* folder as follows:

```
/*
 * i2cDebug.cpp
 */

#include <i2cDebug.h>

#include <Wire.h>      // the Arduino I2C library

i2cDebugClass::i2cDebugClass()
{
}

bool i2cDebugClass::begin(int id)
{
  i2cAddress = id;  // save the slave's address
  Wire.begin(); // join I2C bus (address optional for master)
  return true;
}

size_t i2cDebugClass::write(byte c)
{
  if( count == 0)
  {
    // here if the first char in the transmission
    Wire.beginTransmission(i2cAddress); // transmit to device
  }
```

```
    Wire.write(c);
    //  if the I2C buffer is full or an end of line is reached, send the data
    // BUFFER_LENGTH is defined in the Wire library
    if(++count >= BUFFER_LENGTH || c == '\n')
    {
      // send data if buffer full or newline character
      Wire.endTransmission();
      count = 0;
    }
    return 1; // one character written
}

i2cDebugClass i2cDebug;    // Create an I2C debug object
```

> ## NOTE
>
> The `write` method returns `size_t`, a value that enables the print function to return the number of characters printed. This is new in Arduino 1.0—earlier versions did not return a value from `write` or `print`. If you have a library that is based on Stream or Print, then you will need to change the return type to `size_t`.

Load this example sketch into the IDE:

```
/*
 * i2cDebug
 * example sketch for i2cDebug library
 */

#include <Wire.h>    // the Arduino I2C library
#include <i2cDebug.h>

const int address = 4;    // the address to be used by the communicating
devices
const int sensorPin = 0;  // select the analog input pin for the sensor
int val;                  // variable to store the sensor value


void setup()
{
  Serial.begin(9600);
  i2cDebug.begin(address);
}
```

```
void loop()
{
  // read the voltage on the pot(val ranges from 0 to 1023)
  val = analogRead(sensorPin);
  Serial.println(val);
  i2cDebug.println(val);
}
```

Remember that you need to restart the IDE after creating the library folder. See Recipe 16.4 for more details on creating a library.

Upload the slave I2C sketch onto another Arduino board and wire up the boards as described in Recipe 13.5, and you should see the output from the Arduino board running your library displayed on the second board.

The following references provide an introduction to classes if C++ classes are new to you:

- *Programming Interactivity* by Joshua Noble (O'Reilly)

- *C++ Primer* by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo (Addison-Wesley Professional)

# 16.6 Updating Third-Party Libraries for Arduino 1.0

## Problem

You want to use a third-party library created for Arduino releases previous to 1.0.

## Solution

Most libraries should only require the change of a few lines to work under Arduino 1.0. For example, any one or more of these header file includes:

```
#include "wiring.h"
  #include "WProgram.h"
  #include "WConstants.h"
  #include "pins_arduino.h"
```

should be changed to a single include of:

```
#include "Arduino.h"
```

> **TIP**
>
> The filenames may be enclosed in either angle brackets or quotes.

## Discussion

Older libraries that don't compile under Arduino 1.0 will usually generate one or more of these error messages:

```
source file: error: wiring.h: No such file or directory
source file: error: WProgram.h: No such file or directory
source file: error: WConstants.h: No such file or directory
source file: error: pins_arduino.h: No such file or directory
```

`source file` is the full path of the library file that needs to be updated. There will be a list of other errors following this due to the indicated file not being found in the 1.0 release, but these should disappear after you have replaced the old header names with *Arduino.h*. The definitions in these files are now included in *Arduino.h* and the solution is to replace includes for all of the preceding files with a single include for *Arduino.h*.

If you want to run current versions of Arduino alongside earlier versions, you can use a conditional define (see Recipe 17.6):

```
#if ARDUINO >= 100
#include "Arduino.h"
#else
// These are the filenames that are used in the original version of library
#include "wiring.h"
#include "pins_arduino.h"
#endif
```

Third-party libraries that use serial, Ethernet, or other functionality that has changed syntax in Arduino 1.0 may require additional code changes.

# Advanced Coding and Memory Handling

## 17.0 Introduction

As you do more with your Arduino, your sketches need to become more efficient. The techniques in this chapter can help you improve the performance and reduce the code size of your sketches. If you need to make your sketch run faster or use less RAM, the recipes here can help. The recipes here are more technical than most of the other recipes in this book because they cover things that are usually concealed by the friendly Arduino wrapper.

The Arduino build process was designed to hide complex aspects of C and C++, as well as the tools used to convert a sketch into the bytes that are uploaded and run on an Arduino board. But if your project has performance and resource requirements beyond the capability of the standard Arduino environment, you should find the recipes here useful.

The Arduino board uses memory to store information. It has three kinds of memory: program memory, random access memory (RAM), and EEPROM. Each has different characteristics and uses. Many of the techniques in this chapter cover what to do if you do not have enough of one kind of memory.

Program memory (also known as *flash*) is where the executable sketch code is stored. The contents of program memory can only be changed by the *bootloader* in the upload process initiated by the Arduino software running on your computer. After the upload process is completed, the memory cannot be changed until the next upload. There is far more program memory on an Arduino board than RAM, so it can be beneficial to store values that don't change while the code runs (e.g., constants) in program memory. The bootloader takes up some space in program memory. If all other attempts to minimize the code to fit in program memory have failed,

the bootloader can be removed to free up space, but an additional hardware programmer is then needed to get code onto the board.

If your code is larger than the program memory space available on the chip, the upload will not work and the IDE will warn you that the sketch is too big when you compile.

RAM is used by the code as it runs to store the values for the variables used by your sketch (including variables in the libraries used by your sketch). RAM is *volatile*, which means it can be changed by code in your sketch. It also means anything stored in this memory is lost when power is switched off. Arduino has much less RAM than program memory. If you run out of RAM while your sketch runs on the board (as variables are created and destroyed while the code runs) the board will misbehave (crash).

EEPROM (electrically erasable programmable read-only memory) is memory that code running on Arduino can read and write, but it is nonvolatile memory that retains values even when power is switched off. EEPROM access is significantly slower than for RAM, so EEPROM is usually used to store configuration or other data that is read at startup to restore information from the previous session.

To understand these issues, it is helpful to understand how the Arduino IDE prepares your code to go onto the chip and how you can inspect the results it produces.

## Preprocessor

Some of the recipes here use the *preprocessor* to achieve the desired result. Preprocessing is a step in the first stage of the build process in which the source code (your sketch) is prepared for compiling. Various find and replace functions can be performed. Preprocessor commands are identified by lines that start with `#`. You have already seen them in sketches that use a library—`#include` tells the preprocessor to insert the code from the named library file. Sometimes the preprocessor is the only way to achieve what is needed, but its syntax is different from C and C++ code, and it can introduce bugs that are subtle and hard to track down, so use it with care.

## See Also

AVRfreaks is a website for software engineers that is a good source for technical detail on the controller chips used by Arduino.

Technical details on the C preprocessor

The memory specifications for all of the official boards can be found on the Arduino website.

# 17.1 Understanding the Arduino Build Process

### Problem

You want to see what is happening under the covers when you compile and upload a sketch.

### Solution

You can choose to display all the command-line activity that takes place when compiling or uploading a sketch through the Preferences dialog. Select File → Preferences (Linux, Windows) or Arduino → Preferences (macOS) to display the dialog box to check or uncheck the boxes to enable verbose output for compile or upload messages. You can also choose whether to enable compiler warnings and how verbose you want those warnings (None, Default, More, All).

### Discussion

When you click Compile or Upload, a lot of activity happens that is not usually displayed on screen. The command-line tools that the Arduino IDE was built to hide are used to compile, link, and upload your code to the board.

First your sketch file(s) are transformed into a file suitable for the compiler (*AVR-GCC*) to process. All source files in the sketch folder that have the default Arduino (*.ino*) file extension are joined together to make one file.

All files that end in *.c* or *.cpp* are compiled separately. Header files (with an *.h* extension) are ignored unless they are explicitly included in the files that are being joined.

`#include "Arduino.h"` (`WProgram.h` in previous releases) is added at the top of the file to include the header file with all the Arduino-specific code definitions, such as `digitalWrite()` and `analogRead()`. If you want to examine the contents of that file, change to the directory where Arduino was installed; from there, you can navigate to *hardware/arduino/avr/cores/arduino* to find the header files

On the Mac, right-click the Arduino application icon and select Show Package Contents from the drop-down menu. A folder will open; from the folder, navigate to *Contents/Java/*. You'll be able to find the *hardware/arduino/avr/cores/arduino* folder there.

---

**TIP**

The Arduino directory structure may change in new releases, so check the documentation for the release you are using.

---

To make the code valid C++, the prototypes of any functions declared in your code are generated next and inserted.

Finally, the setting of the board menu is used to insert values (obtained from the *boards.txt* file) that define various constants used for the controller chips on the selected board.

This file is then compiled by AVR-GCC, which is included as part the Arduino IDE installation. It is in the *hardware/tools/avr/bin* folder under the Arduino installation (on macOS, the *hardware* directory is under the *Contents/Java/* folder within the Arduino app as described earlier).

The compiler produces a number of object files (files with an extension of *.o* that will be combined by the link tool). These files are stored in your temporary directory. In there, you'll find directories such as *arduino_build_137218* that contain all the build artifacts. You can

determine your temporary directory on Windows by checking the value of the `TEMP` environment variable: open a command prompt and run the command `echo %TEMP%`. On macOS, open a Terminal shell and run the command `echo $TMPDIR`. On Linux, you should be able to find build artifacts in *tmp*.

The object files are then linked together to make a hex file to upload to the board. Avrdude, a utility for transferring files to the Arduino controller, is used to upload to the board.

The tools used to implement the AVR build process can be found in the *hardware\tools\avr\bin\* directory.

Another useful tool for experienced programmers is `avr-objdump`, which you can find in the *hardware/tools/avr/bin* folder under the Arduino install. It lets you see how the compiler turns the sketch into code that the controller chip runs. This tool produces a disassembly listing of your sketch that shows the object code intermixed with the source code. It can also display a memory map of all the variables used in your sketch. To use the tool, compile the sketch and navigate to the Arduino build folder (which will be a subdirectory of your temporary directory as described earlier). You can also find this by enabling and viewing verbose compiler output and looking for the directory name there. Navigate to the folder that contains the file with the *.elf* file extension. The file used by `avr-objdump` is the one with the extension *.elf*. For example, if you compile the Blink sketch you could view the compiled output (the machine code) by executing the following on the Windows command line (note the use of the `PATH` command to add the Arduino *bin* folder to the head of your `PATH` for this session only). You will need to change `706012` to the suffix of the *arduino_build* folder for the sketch you just compiled:

```
cd %TEMP%
cd arduino_build_706012
PATH "\Program Files (x86)\Arduino\hardware\tools\avr\bin\";%PATH%
avr-objdump -S Blink.ino.elf
```

On Linux, it will be more like this:

```
cd /tmp/arduino_build_700798/
PATH=~/arduino-1.8.9/hardware/tools/avr/bin/:$PATH
avr-objdump -S Blink.ino.elf
```

And for macOS:

```
cd $TMPDIR
cd arduino_build_97987/
PATH=/Applications/Arduino.app/Contents/Java/hardware/tools/avr/bin/:$PATH
avr-objdump -S Blink.ino.elf
```

It is convenient to direct the output to a file that can be read in a text editor. You can do this as follows:

```
avr-objdump -S Blink.ino.elf > blink.txt
```

You can add the `-h` option to add a list of section headers (helpful for determining memory usage):

```
avr-objdump -S -h Blink.ino.elf > blink.txt
```

For non-AVR boards, such as ARM-based boards or ESP8266, the Arduino core (a collection of tools, header files, and other files that support compilation on a particular hardware architecture) are stored under your home directory somewhere. On macOS, cores are stored in *~/Library/Arduino15/packages*. On Windows, it's *%LOCALAPPDATA%\Arduino15\packages*. On Linux, *~/.arduino15/packages*. For example, at the time of this writing, the *objdump* file for SAMD boards is located in the *arduino/tools/arm-none-eabi-gcc/4.8.3-2014q1/arm-none-eabi/bin/* folder beneath the *packages* folder:

```
PATH=~/.arduino15/packages/arduino/tools/arm-none-eabi-gcc/4.8.3-
2014q1/bin:$PATH
arm-none-eabi-objdump -S Blink.ino.elf
```

If you have many cores installed, including cores from Adafruit, SparkFun, or the ESP8266 community, there may be several ARM toolchains in there. The best way to determine which one goes with which core is to compile your sketch verbosely and look for the full path to the tools. When you see a command like `arm-none-eabi-g++ -mcpu=cortex-m0plus` or `xtensa-lx106-elf-gcc -CC -E -P -DVTABLES_IN_FLASH` scroll by, make note of the full path to the tool.

## See Also

This information on the Arduino build process

# 17.2 Determining the Amount of Free and Used RAM

## Problem

You want to be sure you have not run out of RAM. A sketch will not run correctly if there is insufficient memory, and this can be difficult to detect.

## Solution

This recipe shows you how you can determine the amount of free memory available to your sketch. This sketch contains a function called `memoryFree` that reports the amount of available RAM on an AVR:

```
/*
 * Free memory sketch for AVR
 */

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

void loop()
{
  Serial.println(memoryFree());  // print the free memory
```

```
    delay(3000);
  }

  // Variable created by the build process when compiling the sketch
  extern int *__brkval; // Pointer to last address allocated on the heap (or 0)

  // function to return the amount of free RAM
  int memoryFree()
  {
    int freeValue; // This will be the most recent object allocated on the stack
    if((int)__brkval == 0) // Heap is empty; use start of heap
    {
      freeValue = ((int)&freeValue) - ((int)__malloc_heap_start);
    }
    else // Heap is not empty; use last heap address
    {
      freeValue = ((int)&freeValue) - ((int)__brkval);
    }
    return freeValue;
  }
```

## Discussion

The `memoryFree` function first declares a variable, `freeValue`, which is
allocated on the stack because it's local to this function. There are two
primary areas of managed memory: the stack and heap. The stack resides at
the end of free memory, and when your sketch calls functions, the stack
usage grows (downward). The stack memory is used for function calls and
storage for local variables. As your functions finish running, stack memory
is released, and so stack usage shrinks. The heap resides at the beginning of
free memory, and when your sketch (or a library) allocates memory (such as
when you create a `String` object), the heap usage grows (upward). The
stack grows toward the heap, and the heap grows toward the stack.

So, if you know the distance (in bytes) between the stack and heap, you
know how much free memory Arduino has. The address of `freeValue`
(`&freeValue`) is the address of the last byte of free memory (the start of the
stack). The system variable `__brkval` contains the address of the first byte
of free memory (aka the end of the heap), unless there is nothing currently
allocated on the heap. In that case, it's `0` (null), and won't be of any help to

us because there are a lot of other things in memory before the heap. However, there's another system variable, `__malloc_heap_start`, that gives you the address of the start of the heap. If you know the heap is empty (0), use `__malloc_heap_start` instead.

> **NOTE**
>
> You do not need to take the address of `__brkval` or `__malloc_heap_start` with the & like you do with `freeValue`. That's because `__brkval` and `__malloc_heap_start` contain numbers whose job it is to contain an address, while `freeValue` is an integer value whose address we are interested in. In case you are wondering where `__malloc_heap_start` and `__brkval` come from, they are created by the compilation process, and are available as symbols to your sketch. `__malloc_heap_start` is automatically available to all sketches, but `__brkval` requires an `extern` declaration to access it.

For ARM, the sketch is a bit simpler. The general idea is the same, except for ARM, you can use the `sbrk` function, which is a low-level function for managing memory. If you call it with an argument of 0, it just returns the starting address of free RAM. The distance in bytes between the address of `freeValue` and the return value of `sbrk(0)` gives you the free RAM. `sbrk` is declared as a char pointer, so the sketch has to cast it to an int pointer to subtract it from the address of `freeValue`. The sketch uses `reinterpret_cast` rather than a regular cast [`(int *)`] because `reinterpret_cast` allows the use of conversions that may be considered unsafe in some configurations:

```
/*
 * Free memory sketch for ARM
 */

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

void loop()
{
```

```
  Serial.println(memoryFree());  // print the free memory
  delay(3000);
}

// Variable created by the build process when compiling the sketch
extern "C" char *sbrk(int incr); // Call with 0 to get start address of free
ram

// function to return the amount of free RAM
int memoryFree()
{
  int freeValue; // This will be the most recent object allocated on the stack
  freeValue = &freeValue - reinterpret_cast<int*>(sbrk(0));

  return freeValue;
}
```

The number of bytes your code uses changes as the code runs. The important thing is to ensure that you don't consume more memory than you have. Your stack and heap will grow as your program runs. If they crash into each other, you're out of memory! If you make heavy usage of the heap, it will become fragmented, and there may be holes in it. The methods shown in this recipe will not take these holes into account, so you may have a little bit more memory than it reports.

Here are the main ways RAM memory is consumed:

- When you initialize constants or define a preprocessor macro:

  ```
  #define ERROR_MESSAGE "an error has occurred"
  ```

- When you declare global variables:

  ```
  char myMessage[] = "Hello World";
  ```

- When you make a function call:

  ```
  void myFunction(int value)
  {
     int result;
     result = value * 2;
  ```

```
        return result;
    }
```

If you create recursive functions (functions that call themselves), you can end up with very high stack usage if your nesting level gets too deep and/or you have a lot of local variables.

- When you dynamically allocate memory:

```
    String stringOne = "Arduino String";
```

The Arduino `String` class uses dynamic memory to allocate space for strings. You can see this by adding the following line to the very top of the code in the Solution:

```
  String s = "\n";
```

and the following lines just before the `delay` in the `loop` code:

```
  s = s + "Hello I am Arduino\n";
  Serial.println(s);              // print the string value
```

You will see the memory value reduce as the size of the string is increased each time through the `loop`. If you run the sketch long enough, the memory will run out—don't endlessly try to increase the size of a string in anything other than a test application.

Writing code like this that creates a constantly expanding value is a sure way to run out of memory. You should also be careful not to create code that dynamically creates different numbers of variables based on some parameter while the code runs, as it will be very difficult to be sure you will not exceed the memory capabilities of the board when the code runs.

Constants and global variables are often declared in libraries as well, so you may not be aware of them, but they still use up RAM. The Serial library, for example, has a 128-byte global array that it uses for incoming serial data. This alone consumes one-eighth of the total memory of an old Arduino 168 chip.

## See Also

# 17.3 Storing and Retrieving Numeric Values in Program Memory

## Problem

You have a lot of constant numeric data and don't want to allocate this to RAM.

## Solution

Store numeric variables in program memory (the flash memory used to store Arduino programs).

This sketch adjusts a fading LED for the nonlinear sensitivity of human vision. It stores the values to use in a table of 256 values in program memory rather than RAM.

The sketch is based on Recipe 7.2; see Chapter 7 for a wiring diagram and discussion on driving LEDs. Running this sketch results in a smooth change in brightness with the LED on pin 5 compared to the LED on pin 3:

```
/*
 * ProgmemCurve sketch
 * uses table in program memory to convert linear to exponential output
 */

#include <avr/pgmspace.h>  // needed for PROGMEM

// table of exponential values
// generated for values of i from 0 to 255 -> x=round( pow( 2.0, i/32.0) - 1);

const byte table[]PROGMEM = {
  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
  0,   0,   0,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
1,
  1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   2,   2,   2,   2,
```

```
  2,
    2,   2,   2,   2,   2,   2,   2,   2,   2,   2,   3,   3,   3,   3,   3,
  3,
    3,   3,   3,   3,   3,   3,   4,   4,   4,   4,   4,   4,   4,   4,   4,
  5,
    5,   5,   5,   5,   5,   5,   5,   6,   6,   6,   6,   6,   6,   6,   7,
  7,
    7,   7,   7,   8,   8,   8,   8,   8,   9,   9,   9,   9,   9,  10,  10,
 10,
   10,  11,  11,  11,  11,  12,  12,  12,  12,  13,  13,  13,  14,  14,  14,
 15,
   15,  15,  16,  16,  16,  17,  17,  18,  18,  18,  19,  19,  20,  20,  21,
 21,
   22,  22,  23,  23,  24,  24,  25,  25,  26,  26,  27,  28,  28,  29,  30,
 30,
   31,  32,  32,  33,  34,  35,  35,  36,  37,  38,  39,  40,  40,  41,  42,
 43,
   44,  45,  46,  47,  48,  49,  51,  52,  53,  54,  55,  56,  58,  59,  60,
 62,
   63,  64,  66,  67,  69,  70,  72,  73,  75,  77,  78,  80,  82,  84,  86,
 88,
   90,  91,  94,  96,  98, 100, 102, 104, 107, 109, 111, 114, 116, 119, 122,
124,
  127, 130, 133, 136, 139, 142, 145, 148, 151, 155, 158, 161, 165, 169, 172,
176,
  180, 184, 188, 192, 196, 201, 205, 210, 214, 219, 224, 229, 234, 239, 244,
250
};

const int rawLedPin  = 3;          // this LED is fed with raw values
const int adjustedLedPin = 5;      // this LED is driven from table

int brightness = 0;
int increment = 1;

void setup()
{
  // pins driven by analogWrite do not need to be declared as outputs
}

void loop()
{
  if (brightness > 254)
  {
    increment = -1; // count down after reaching 255
  }
  else if (brightness < 1)
  {
```

```
   increment =  1; // count up after dropping back down to 0
  }
  brightness = brightness + increment; // increment (or decrement sign is
minus)

  // write the brightness value to the LEDs
  analogWrite(rawLedPin, brightness);  // this is the raw value
  int adjustedBrightness = pgm_read_byte(&table[brightness]);  // adjusted
value
  analogWrite(adjustedLedPin, adjustedBrightness);

  delay(10); // 10 ms for each step change means 2.55 secs to fade up or down
}
```

## Discussion

When you need to use a complex expression to calculate a range of values
that regularly repeat, it is often better to precalculate the values and include
them in a table of values (usually as an array) in the code. This saves the
time needed to calculate the values repeatedly when the code runs. The
disadvantage concerns the memory needed to place these values in RAM.
RAM is limited on Arduino and the much larger program memory space
can be used to store constant values. This is particularly helpful for sketches
that have large arrays of numbers.

At the top of the sketch, the table is defined with the following expression:

```
const byte table[]PROGMEM = {
  0, . . .
```

PROGMEM tells the compiler that the values are to be stored in program
memory rather than RAM. If you were to delete PROGMEM from the sketch
on an Arduino Uno, you'd see the global variable use balloon from 13 bytes
to 269 bytes (the sketch would not work, though, because pgm_read_byte
will not work correctly without PROGMEM there). The remainder of the
expression is similar to defining a conventional array (see Chapter 2).

The low-level definitions needed to use PROGMEM are contained in a file
named *pgmspace.h* and the sketch includes this as follows:

```
#include <avr/pgmspace.h>
```

Although "avr" is in the path to this header file, you can still include it on 32-bit architectures because it is included for backward compatibility with AVR boards. The implementation is a bit simpler, and in fact `PROGMEM` is defined empty on ARM-based (SAM, SAMD) boards. This is because the ARM compiler will generally store data structures declared as `const` in program memory. On ARM-based boards, you can confirm its location by adding this code to `setup()`:

```
Serial.begin(9600);
while(!Serial); // for Leonardo and 32-bit boards
Serial.print("Address of table: 0x");
Serial.println((int)&table, HEX);
```

If the value displayed is between 0x0000 and 0x3FFFF, then it is stored in program memory! If you remove `const` from the declaration of `table` and run the sketch, you'll see that it is stored at a much higher address (0x2000000 or higher).

To adjust the brightness to make the fade look uniform, this recipe adds the following lines to the LED output code used in Recipe 7.2:

```
int adjustedBrightness = pgm_read_byte(&table[brightness]);
  analogWrite(adjustedLedPin, adjustedBrightness);
```

The variable `adjustedBrightness` is set from a value read from program memory. The expression `pgm_read_byte(&table[brightness]);` means to return the address of the entry in the `table` array at the index position given by `brightness`.

## See Also

Adafruit Industries' Memories of an Arduino contains a lot of useful insights and techniques for working with Arduino memory.

See for the technique introduced in Arduino 1.0 to store strings in flash memory.

# 17.4 Storing and Retrieving Strings in Program Memory

## Problem

You have lots of strings and they are consuming too much RAM. You want to move string constants, such as menu prompts or debugging statements, out of RAM and into program memory.

## Solution

This sketch creates a string in program memory and prints its value to the Serial Monitor using the F("text") expression. The technique for printing the amount of free RAM is described in . This sketch combines both the ARM and AVR methods. If you are using something other than an ARM or AVR board, this sketch probably won't compile correctly:

```
/*
 * Write strings using Program memory (Flash)
 */

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

void loop()
{
  Serial.println(memoryFree());  // print the free memory

  Serial.println(F("Arduino"));  // print the string
  delay(1000);
}

#ifdef __arm__
// Variable created by the build process when compiling the sketch
extern "C" char *sbrk(int incr); // Call with 0 to get start address of free
```

```
  ram
#else
extern int *__brkval; // Pointer to last address allocated on the heap (or 0)
#endif

// function to return the amount of free RAM
int memoryFree()
{
  int freeValue; // This will be the most recent object allocated on the stack
#ifdef __arm__
  freeValue = &freeValue - reinterpret_cast<int*>(sbrk(0));
#else
  if((int)__brkval == 0) // Heap is empty; use start of heap
  {
    freeValue = ((int)&freeValue) - ((int)__malloc_heap_start);
  }
  else // Heap is not empty; use last heap address
  {
    freeValue = ((int)&freeValue) - ((int)__brkval);
  }
#endif
  return freeValue;
}
```

## Discussion

Strings are particularly hungry when it comes to RAM. Each character uses a byte, so it is easy to consume large chunks of RAM if you have lots of words in strings in your sketch. Inserting your text in the `F("text")` expression stores the text in the much larger flash memory instead of RAM.

If you remove the `F` from before the string, you'll see the amount of free memory is lower than if you use it, at least on AVR. Depending on how the ARM compiler optimizes things, it may put strings in flash without the `F` expression.

## See Also

See Recipe 15.13 for an example of flash memory used to store web page strings.

# 17.5 Using #define and const Instead of Integers

**Problem**

You want to minimize RAM usage by telling the compiler that the value is constant and can be optimized.

## Solution

Use `const` to declare values that are constant throughout the sketch.

For example, instead of:

```
int ledPin = 2;
```

use:

```
const int ledPin = 2;
```

## Discussion

We often want to use a constant value in different areas of code. Just writing the number is a really bad idea. If you later want to change the value used, it's difficult to work out which numbers scattered throughout the code also need to be changed. It is best to use named references.

Here are three different ways to define a value that is a constant:

```
int ledPin =  2;          // a variable, but this wastes RAM
const int ledPin =  2;    // a const does not use RAM

#define ledPin LED_BUILTIN // with a define, the preprocessor replaces
                           // ledPin with the value of LED_BUILTIN

pinMode(ledPin, OUTPUT);
```

Although the first two expressions look similar, the term `const` tells the compiler not to treat `ledPin` as an ordinary variable. Unlike the ordinary

`int`, no RAM is reserved to hold the value for the `const`, as it is guaranteed not to change. The compiler will produce exactly the same code as if you had written:

```
pinMode(2, OUTPUT);
```

That said, if your sketch uses `ledPin` as though it were a constant (such as if you never modify it), the compiler is very likely to notice this and optimize it away, producing the preceding code even if you forgot to add `const`.

You will sometimes see `#define` used to define constants in older Arduino code, but `const` is a better choice than `#define`. This is because a `const` variable has a *type*, which enables the compiler to verify and report if the variable is being used in ways not appropriate for that type. The compiler will also respect C rules for the scope of a `const` variable. A `#define` value will affect all the code in the sketch, which may be more than you intended. Another benefit of `const` is that it uses familiar syntax—`#define` does not use the equals sign, and no semicolon is used at the end. One exception to this is if you need to perform conditional compilation of code, where the compiler ignores or includes code based on the values of one or more `#defines` (see Recipe 17.6).

## See Also

See Recipe 17.0 for more on the preprocessor.

# 17.6 Using Conditional Compilations

## Problem

You want to have different versions of your code that can be selectively compiled. For example, you may need code to work differently when debugging or when running with different boards.

## Solution

You can use the conditional statements aimed at the preprocessor to control how your sketch is built.

This example, using the sketch from Recipe 5.6, displays some debug statements only if DEBUG is defined:

```
/*
 Pot_Debug sketch
 blink an LED at a rate set by the position of a potentiometer
 Uses Serial port for debug if DEBUG is defined
 */

const int potPin = 0;      // select the input pin for the potentiometer
const int ledPin = 13;     // select the pin for the LED
int val = 0;               // variable to store the value coming from the sensor

#define DEBUG

void setup()
{
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);     // declare the ledPin as an OUTPUT
}

void loop() {
  val = analogRead(potPin);    // read the voltage on the pot
  digitalWrite(ledPin, HIGH);  // turn the ledPin on
  delay(val);                  // blink rate set by pot value
  digitalWrite(ledPin, LOW);   // turn the ledPin off
  delay(val);                  // turn LED off for same period
#if defined DEBUG
  Serial.println(val);
#endif
}
```

## Discussion

This recipe uses the preprocessor used at the beginning of the compile process to change what code is compiled. The sketch tests if DEBUG is defined, and if so, the file incorporates the debugging output. Expressions

that begin with the # symbol are processed before the code is compiled—see this chapter's introduction for more on the preprocessor.

You can have a conditional compile based on the controller chip selected in the IDE. For example, the following code will produce different code when compiled for a Mega board that reads the additional analog pins that it has:

```
/*
 * ConditionalCompile sketch
 * This sketch recognizes the controller chip using conditional defines
 */

int numberOfSensors;
int val = 0;                    // variable to store the value coming from the
sensor

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

#if defined(__AVR_ATmega2560__)   // defined when selecting Mega in the IDE
  numberOfSensors = 16;           // the number of analog inputs on the Mega
  #pragma message ( "Using 16 sensors" )
#else                             // if not Mega then assume a standard board
  numberOfSensors = 6;            // analog inputs on a standard Arduino board
  #pragma message ( "Using 6 sensors" )
#endif

  Serial.print("The number of sensors is ");
  Serial.println(numberOfSensors);
}

void loop() {
  for(int sensor = 0; sensor < numberOfSensors; sensor++)
  {
    val = analogRead(sensor);     // read the sensor value
    Serial.print(sensor); Serial.print(": ");
    Serial.println(val);          // display the value
  }
  Serial.println();
  delay(1000);          // delay a second between readings
}
```

The #pragma directive will display a message in the output area at the bottom of the IDE:

```
C:\Sketches\conditional.ino:15:40: note: #pragma message: Using 16 sensors
        #pragma message ( "Using 16 sensors" )
                                            ^
```

## See Also

Technical details on the C preprocessor

# Using the Controller Chip Hardware

## 18.0 Introduction

The Arduino platform simplifies programming by providing easy-to-use function calls to hide complex, low-level hardware functions. But some applications need to bypass the friendly access functions to get directly at hardware, either because that's the only way to get the needed functionality or because higher performance is required. This chapter shows how to access and use hardware functions that are not fully exposed through the Arduino programming environment.

> **NOTE**
>
> Changing register values can change the behavior of some Arduino functions (e.g., `millis`). The low-level capabilities described in this chapter require care, attention, and testing if you want your code to function correctly.

There are four key hardware features that you need to understand before getting deeper into the hardware. *Registers* are memory locations in the microcontroller that can be used to alter its behavior. When you work with the other hardware features, you will often be manipulating registers to configure them. *Interrupts* are signals generated by the microcontroller, generally in response to an external event, allowing Arduino sketches to respond immediately when something happens. *Timers* can generate a signal after a predetermined delay, or repeatedly generate a signal based on a duration you specify. As with interrupts, you can take an action in your sketch in response to a timer. You've already seen how to work with *analog* and *digital pins,* but the recipes in this chapter will show you how to work with them at much faster speeds.

## Registers

Registers are variables that refer to hardware memory locations. They are used by the chip to configure hardware functions or for storing the results of hardware operations. The contents of registers can be read and written by your sketch. Changing register values will change the way the hardware operates, or the state of something (such as the output of a pin). Some registers represent a numerical value (the number a timer will count to). Registers can control or report on hardware status; for example, the state of a pin or if an interrupt has occurred. Registers are referenced in code using their names—these are documented in the datasheet for the microcontrollers. Setting a register to a wrong value usually results in a sketch functioning incorrectly, so carefully check the documentation to ensure that you are using registers correctly.

## Interrupts

Interrupts are signals that enable the controller chip to stop the normal flow of a sketch and handle a task that requires immediate attention before continuing with what it was doing. Arduino core software uses interrupts to handle incoming data from the serial port, to maintain the time for the `delay` and `millis` functions, and to trigger the `attachInterrupt` function. Libraries, such as Wire and Servo, use interrupts when an event occurs, so the code doesn't have to constantly check to see if the event has happened. This constant checking, called *polling*, can complicate the logic of your sketch. Interrupts can be a reliable way to detect signals of very short duration. Recipe 18.2 explains how to use interrupts to determine if a digital pin has changed state.

Two or more interrupts may occur before the handling of the first interrupt is completed; for example, if two switches are pressed at the same time and each generates a different interrupt. The interrupt handler for the first switch must be completed before the second interrupt can get started. Interrupts should be brief, because an interrupt routine that takes too much time can cause other interrupt handlers to be delayed or to miss events.

## Timers

The Arduino Uno (and compatible boards based on the ATmega328) has three hardware timers for managing time-based tasks (the Mega has six). The timers are used in a number of Arduino functions:

Timer0

>   Used for `millis` and `delay`; also `analogWrite` on pins 5 and 6

Timer1

>   `analogWrite` functions on pins 9 and 10; also driving servos using the Servo library

Timer2

>   `analogWrite` functions on pins 3 and 11

The Mega has three additional 16-bit timers and uses different pin numbers with `analogWrite`:

Timer0

>   `analogWrite` functions on pins 4 and 13

Timer1

`analogWrite` functions on pins 11 and 12

Timer2

`analogWrite` functions on pins 9 and 10

Timer3

`analogWrite` functions on pins 2, 3, and 5

Timer4

`analogWrite` functions on pins 6, 7, and 8

Timer5

`analogWrite` functions on pins 45 and 46

---

**NOTE**

The megaAVR-based boards such as the Arduino WiFi Rev2 and the Arduino Nano Every use a different timer system than the ATmega-based boards like the Uno and Mega. This application note from Microchip explains how timers are implemented on the megaAVR. ARM-based Arduino boards use an entirely different approach to handling timers based on two facilities: Timer Counter (TC) and Timer Counter for Control Applications (TCC). The following application notes cover each of those for the SAMD chips: Timer Counter (TC) Driver and Timer Counter for Control Applications (TCC) Driver. Although this library is not intended (or supported) for use by anyone outside Adafruit, they have published their Adafruit_ZeroTimer library on GitHub, which includes wrapper code for Timer Control modules 3, 4, and 5 on both the SAMD21 (ARM Cortex-M0 core) and SAMD51 (Arm Cortex-M4 core) platforms.

---

Timers are counters that count pulses from a time source, called a *timebase*. The timer hardware consists of 8-bit or 16-bit digital counters that can be programmed to determine the mode the timer uses to count. The most common mode is to count pulses from the timebase on the Arduino board, usually 16 MHz derived from a crystal; 16 MHz pulses repeat every 62.5 nanoseconds, and this is too fast for many timing applications, so the timebase rate is reduced by a divider called a *prescaler*. Dividing the timebase by 8, for example, increases the duration of each count to half a microsecond. For applications in which this is still too fast, other prescale values can be used (see Table 18-1).

Timer operation is controlled by values held in registers that can be read and written by Arduino code. The values in these registers set the timer frequency (the number of system timebase pulses between each count) and the method of counting (up, down, up and down, or using an external signal).

Here is an overview of the timer registers ($n$ is the timer number):

Timer Counter Control Register A (TCCR$n$A)

> Determines the operating mode

Timer Counter Control Register B (TCCR$n$B)

> Determines the prescale value

Timer Counter Register (TCNT$n$)

> Contains the timer count

Output Compare Register A (OCR$n$A)

> Interrupt can be triggered on this count

Output Compare Register B (OCR$n$B)

> Interrupt can be triggered on this count

Timer/Counter Interrupt Mask Register (TIMSK$n$)

> Sets the conditions for triggering an interrupt

Timer/Counter 0 Interrupt Flag Register (TIFR$n$)

> Indicates if the trigger condition has occurred

Table 18-1 is an overview of the bit values used to set the timer precision. Details of the functions of the registers are explained in the recipes where they are used.

*Timer prescale values (16 MHz clock)*

| Prescale factor | CSx2, CSx1, CSx0 | Precision | Time to overflow | |
|---|---|---|---|---|
| | | | 8-bit timer | 16-bit timer |
| 1 | B001 | 62.5 ns | 16 µs | 4.096 ms |

| Prescale factor | CSx2, CSx1, CSx0 | Precision | Time to overflow | |
|---|---|---|---|---|
| | | | **8-bit timer** | **16-bit timer** |
| 8 | B010 | 500 ns | 128 µs | 32.768 ms |
| 64 | B011 | 4 µs | 1,024 µs | 262.144 ms |
| 256 | B100 | 16 µs | 4,096 µs | 1048.576 ms |
| 1,024 | B101 | 64 µs | 16,384 µs | 4194.304 ms |
| | B110 | External clock, falling edge | | |
| | B111 | External clock, rising edge | | |

All timers are initialized for a prescale of 64.

Precision in nanoseconds is equal to the CPU period (time for one CPU cycle) multiplied by the prescale.

## Analog and Digital Pins

Chapter 5 described the standard Arduino functions to read and write (to/from) digital and analog pins. This chapter explains how you can control pins faster than using the Arduino read and write functions and make changes to analog methods to improve performance.

Some of the code in this chapter is more difficult to understand than the other recipes in this book, as it is moving beyond Arduino syntax and closer to the underlying hardware. These recipes work directly with the tersely named registers in the chip and use bit shifting and masking to manipulate bits in them. The benefit from this complexity is enhanced performance and functionality.

## See Also

# 18.1 Storing Data in Permanent EEPROM Memory

## Problem

You want to store values that will be retained even when power is switched off.

## Solution

Use the EEPROM library to read and write values in EEPROM memory. This sketch blinks an LED using values read from EEPROM and allows the values to be changed using the Serial Monitor:

```
/*
 * EEPROM sketch based on Blink without Delay
 * uses EEPROM to store blink values
 */

#include <EEPROM.h>

// these values are saved in EEPROM
const byte EEPROM_ID = 0x99;    // used to identify if valid data in EEPROM
byte ledPin =  LED_BUILTIN;     // the LED pin
int interval = 1000;            // interval at which to blink (milliseconds)

// variables that do not need to be saved
int ledState = LOW;                 // ledState used to set the LED
long previousMillis = 0;            // will store last time LED was updated

//constants used to identify EEPROM addresses
const int ID_ADDR = 0;          // the EEPROM address used to store the ID
```

```arduino
const int PIN_ADDR = 1;       // the EEPROM address used to store the pin
const int INTERVAL_ADDR = 2;  // the EEPROM address used to store the interval

void setup()
{
  Serial.begin(9600);
  byte id = EEPROM.read(ID_ADDR); // read the first byte from the EEPROM
  if( id == EEPROM_ID)
  {
    // here if the id value read matches the value saved when writing eeprom
    Serial.println("Using data from EEPROM");
    ledPin = EEPROM.read(PIN_ADDR);
    byte hiByte  = EEPROM.read(INTERVAL_ADDR);
    byte lowByte = EEPROM.read(INTERVAL_ADDR+1);
    interval =  word(hiByte, lowByte); // see word function in Recipe 3.15
  }
  else
  {
    // here if the ID is not found, so write the default data
    Serial.println("Writing default data to EEPROM");
    EEPROM.write(ID_ADDR,EEPROM_ID); // write the ID to indicate valid data
    EEPROM.write(PIN_ADDR, ledPin); // save the pin in eeprom
    byte hiByte = highByte(interval);
    byte loByte = lowByte(interval);
    EEPROM.write(INTERVAL_ADDR, hiByte);
    EEPROM.write(INTERVAL_ADDR+1, loByte);
  }
  Serial.print("Setting pin to ");
  Serial.println(ledPin,DEC);
  Serial.print("Setting interval to ");
  Serial.println(interval);

  pinMode(ledPin, OUTPUT);
}

void loop()
{
  // this is the same code as the BlinkWithoutDelay example sketch
  if (millis() - previousMillis > interval)
  {
    previousMillis = millis();     // save the last time you blinked the LED
    // if the LED is off turn it on and vice versa:
    if (ledState == LOW)
      ledState = HIGH;
    else
      ledState = LOW;
    digitalWrite(ledPin, ledState);  // set LED using value of ledState
  }
  processSerial();
```

```
  }

  // function to get duration or pin values from Serial Monitor
  // value followed by  i is interval, p is pin number
  int value = 0;
  void processSerial()
  {
    if( Serial.available())
   {
     char ch = Serial.read();
     if(ch >= '0' && ch <= '9') // is this an ascii digit between 0 and 9?
     {
        value = (value * 10) + (ch - '0'); // yes, accumulate the value
     }
     else if (ch == 'i')  // is this the interval
     {
        interval = value;
        Serial.print("Setting interval to ");
        Serial.println(interval);
        byte hiByte = highByte(interval);
        byte loByte = lowByte(interval);
        EEPROM.write(INTERVAL_ADDR, hiByte);
        EEPROM.write(INTERVAL_ADDR+1, loByte);
        value = 0; // reset to 0 ready for the next sequence of digits
     }
     else if (ch == 'p')  // is this the pin number
     {
        ledPin = value;
        Serial.print("Setting pin to ");
        Serial.println(ledPin,DEC);
        pinMode(ledPin, OUTPUT);
        EEPROM.write(PIN_ADDR, ledPin); // save the pin in eeprom
        value = 0; // reset to 0 ready for the next sequence of digits
     }
   }
  }
```

Open the Serial Monitor. As the sketch starts, it tells you whether it is using values previously saved to EEPROM or defaults, if this is the first time the sketch is started.

You can change values by typing a number followed by a letter to indicate the action. A number followed by the letter *i* changes the blink interval; a number followed by a *p* changes the pin number for the LED.

## Discussion

Arduino contains EEPROM memory that will store values even when power is switched off. There are 1,024 bytes of EEPROM on the Arduino Uno and 4K bytes in a Mega. The Arduino Uno WiFi R2 and the Nano Every have only 256 bytes of EEPROM memory. Most ARM-based boards do not have EEPROM memory.

The sketch uses the EEPROM library to read and write values in EEPROM memory. Once the library is included in the sketch, an EEPROM object is available that accesses the memory. The library provides methods to `read`, `write`, and `clear`. `EEPROM.clear()` is not used in this sketch because it erases all the EEPROM memory.

The EEPROM library requires you to specify the address in memory that you want to read or write. This means you need to keep track of where each value is written so that when you access the value it is from the correct address. To write a value, you use `EEPROM.write(address, value)`. The address is from 0 to 1,023 (on the Uno), and the value is a single byte. To read, you use `EEPROM.read(address)`. The byte content of that memory address is returned.

The sketch stores three values in EEPROM. The first value stored is an `ID` value that is used only in `setup` to identify if the EEPROM has been previously written with valid data. If the value stored matches the expected value, the other variables are read from EEPROM and used in the sketch. If it doesn't match, this sketch has not been run on this board (otherwise, the `ID` would have been written), so the default values are written, including the `ID` value.

The sketch monitors the serial port, and new values received are written to EEPROM. The sketch stores the `ID` value in EEPROM address 0, the pin number in address 1, and the two bytes for the interval start in address 2. The following line writes the pin number to EEPROM. The variable `ledPin` is a byte, so it fits into a single EEPROM address:

```
EEPROM.write(PIN_ADDR, ledPin); // save the pin in eeprom
```

Because interval is an `int`, it requires two bytes of memory to store the value:

```
byte hiByte = highByte(interval);
byte loByte = lowByte(interval);
EEPROM.write(INTERVAL_ADDR, hiByte);
EEPROM.write(INTERVAL_ADDR+1, loByte);
```

The preceding code splits the value into two bytes that are stored in two consecutive addresses. Any additional variables to be added to EEPROM would need to be placed in addresses that follow these two bytes.

Here is the code used to rebuild the `int` variable from EEPROM:

```
ledPin = EEPROM.read(PIN_ADDR);
byte hiByte  =  EEPROM.read(INTERVAL_ADDR);
byte lowByte =  EEPROM.read(INTERVAL_ADDR+1);
interval =  word(hiByte, lowByte);
```

See Chapter 3 for more on using the `word` expression to create an integer from two bytes.

For more complicated use of EEPROM, it is advisable to draw out a map of what is being saved where, to ensure that no address is used by more than one value, and that multibyte values don't overwrite other information.

## See Also

Recipe 3.14 provides more information on converting 16- and 32-bit values into bytes.

# 18.2 Take Action Automatically When a Pin State Changes

## Problem

You want to perform some action when a digital pin changes value and you don't want to have to constantly check the pin state.

## Solution

This sketch monitors pulses on pin 2 and stores the duration in an array. When the array has been filled (32 pulses have been received), the duration of each

pulse is displayed on the Serial Monitor:

```
/*
 * Interrupts sketch
 * see Recipe 10.1 for connection diagram
 */

const int pin = 2; // pin the receiver is connected to
const int numberOfEntries = 32; // set this number to any convenient value

volatile unsigned long microseconds;
volatile byte idx = 0;
volatile unsigned long results[numberOfEntries];

void setup()
{
  pinMode(pin, INPUT_PULLUP);
  Serial.begin(9600);
  // Use the pin's interrupt to monitor for changes
  attachInterrupt(digitalPinToInterrupt(pin), analyze, CHANGE);
  results[0]=0;
}

void loop()
{
  if(idx >= numberOfEntries)
  {
    Serial.println("Durations in Microseconds are:") ;
    for( byte i=0; i < numberOfEntries; i++)
    {
      Serial.print(i); Serial.print(": ");
      Serial.println(results[i]);
    }
    idx = 0; // start analyzing again
  }
  delay(1000);
}

void analyze()
{
  if(idx < numberOfEntries)
```

```
    {
      if(idx > 0)
      {
        results[idx] = micros() - microseconds;
      }
      idx = idx + 1;
    }
    microseconds = micros();
  }
```

If you have an infrared receiver module, you can use the wiring in Recipe 10.1 to measure the pulse width from an infrared remote control. You could also use the wiring in Recipe 6.12 to measure pulses from a rotary encoder or connect a switch to pin 2 (see Recipe 5.1) to test with a pushbutton.

## Discussion

In `setup`, the `attachInterrupt(digitalPinToInterrupt(pin), analyze, CHANGE);` call enables the sketch to handle interrupts. The first argument in the call specifies which interrupt to initialize. The actual interrupt for a given pin varies from board to board, so you should use the `digitalPinToInterrupt` function to determine it, rather than hardcoding the interrupt value in your sketch.

The next parameter specifies what function to call (sometimes called an *interrupt handler*) when the interrupt event happens; `analyze` in this sketch.

The final parameter specifies what should trigger the interrupt. `CHANGE` means whenever the pin level changes (goes from low to high, or high to low). The other options are:

`HIGH`

When the pin is high (Due, Zero, and MKR ARM boards only)

`LOW`

When the pin is low

`RISING`

When the pin goes from low to high

`FALLING`

When the pin goes from high to low

When reading code that uses interrupts, bear in mind that it may not be obvious when values in the sketch change because the sketch does not directly call the interrupt handler; it's called when the interrupt conditions occur.

In this sketch, the main `loop` checks the `index` variable to see if all the entries have been set by the interrupt handler. Nothing in `loop` changes the value of `index`. `index` is changed inside the `analyze` function when the interrupt condition occurs (pin 2 changing state). The `index` value is used to store the time since the last state change into the next slot in the `results` array. The time is calculated by subtracting the last time the state changed from the current time in microseconds. The current time is then saved as the last time a change happened. (Chapter 12 describes this method for obtaining elapsed time using the `millis` function; here `micros` is used to get elapsed microseconds instead of milliseconds.)

The variables that are changed in an interrupt function are declared as `volatile`; this lets the compiler know that the values could change at any time (by an interrupt handler). Without using the `volatile` keyword, the compiler may store the values in registers that can be accidentally overwritten by an interrupt handler. To prevent this, the `volatile` keyword tells the compiler to store the values in RAM rather than registers.

Each time an interrupt is triggered, `index` is incremented and the current time is saved. The time difference is calculated and saved in the array (except for the first time the interrupt is triggered, when `index` is 0). When the maximum number of entries has occurred, the inner block in `loop` runs, and it prints out all the values to the serial port. The code stays in the `while` loop at the end of the inner block, so you need to reset the board when you want to do another run.

## See Also

Recipe 6.12 has an example of external interrupts used to detect movement in a rotary encoder.

# 18.3 Perform Periodic Actions

## Problem

You want to do something at periodic intervals, and you don't want to have your code constantly checking if the interval has elapsed. You would like to have a simple interface for setting the period.

## Solution

The easiest way to use a timer is through a library. The uTimerLib library can generate a pulse with a regular period. You can install it with the Library Manager. This sketch flashes the built-in LED at a rate that can be set using the Serial Monitor:

```
/*
 * Timer pulse with uTimerLib
 * pulse the onboard LED at a rate set from serial input
 */

#include "uTimerLib.h"

const int pulsePin = LED_BUILTIN;

int period = 100; // period in milliseconds
volatile bool output = HIGH; // the state of the pulse pin

void setup()
{
  pinMode(pulsePin, OUTPUT);
  Serial.begin(9600);
  TimerLib.setInterval_us(flash, period/2 * 1000L);
}

void loop()
{
  if( Serial.available())
  {
    int period = Serial.parseInt();
    if (period)
    {
      Serial.print("Setting period to "); Serial.println(period);
      TimerLib.setInterval_us(flash, period/2 * 1000L);
    }
  }
}
```

```
void flash()
{
  digitalWrite(pulsePin, output);
  output = !output;  // invert the output
}
```

## Discussion

Enter digits for the desired period in milliseconds using the Serial Monitor and press Return/Enter to send it to the sketch. The sketch uses `parseInt` to read the digits and divides the received value by 2 to calculate the duration of the on and off states (the period is the sum of the on time and off time, so the smallest value you can use is 2). Bear in mind that an LED flashing very quickly may not appear to be flashing to the human eye. Because `TimerLib.setInterval_us` specifies an interval in microseconds, we multiply the period (specified in milliseconds) by 1,000.

> **NOTE**
>
> On ATmega-based boards such as the Uno, this library uses Timer2, so it will interfere with the operation of `analogWrite` on pins 3 and 11 (pins 9 and 10 for the Arduino Mega). At the time of this writing, uTimerLib does not work with the Arduino Uno WiFi Rev 2 or the Arduino Nano Every, which are based on a Mega AVR chip that uses a different timer architecture. It does support several 32-bit platforms, including SAM (Arduino Due), SAMD21 (Zero and M0 boards from Adafruit and SparkFun), ESP8266, and more.

This library enables you to use a timer by providing the timing interval and the name of the function to call when the interval has elapsed:

```
TimerLib.setInterval_us(flash, period/2 * 1000L);
```

This sets up the timer. The first parameter is the function to call when the timer gets to the end of that time (the function is named `flash` in this recipe). The second parameter is the time for the timer to run in microseconds.

As in Recipe 18.2, the sketch code does not directly call the function to perform the action. The LED is turned on and off in the `flash` function that is

called by the timer each time it gets to the end of its time setting. The code in `loop` deals with any serial messages and changes the timer settings based on it.

Using a library to control timers is much easier than accessing the registers directly. Here is an overview of the inner workings of this library on ATmega: Timers work by constantly counting to a value, signaling that they have reached the value, then starting again. Each timer has a *prescaler* that determines the counting frequency. The prescaler divides the system timebase by a factor such as 1, 8, 64, 256, or 1,024. The lower the prescale factor, the higher the counting frequency and the quicker the timebase reaches its maximum count. The combination of how fast to count, and what value to count to, gives the time for the timer. Timer2 is an 8-bit timer; this means it can count up to 255 before starting again from 0. (Timer1 and Timers 3, 4, and 5 on the Mega use 16 bits and can count up to 65,535.)

On AVR, for intervals over 4,095 microseconds, the library uses a prescale factor of 64. On a 16 MHz Arduino board, each CPU cycle is 62.5 nanoseconds long, and when this is divided by the prescale factor of 64, each count of the timer will be 4,000 nanoseconds (62.5 * 64 = 4,000, which is four microseconds).

### See Also

The uTimerLib GitHub repository

# 18.4 Setting Timer Pulse Width and Duration

### Problem

You want Arduino to generate pulses with a duration and width that you specify.

### Solution

This sketch generates pulses within the frequency range of 1 MHz to 1 Hz using Timer1 PWM on pin 9. The library it uses supports only Arduino Uno, Mega, Leonardo, and a number of Teensy boards. See this Arduino page to determine which pins are available for use with this library on your board.

```
/*
 * Pulse width duration sketch
 * Set period and width of pulses
 */

#include <TimerOne.h>

const int outPin = 9; // pin; use 3-4 on Teensy 3.x, 11-13 on Arduino Mega

long period = 40; // 40-microsecond period (25 KHz)
long width  = 20; // 20-microsecond width (50% duty cycle)

void setup()
{
  Serial.begin(9600);
  pinMode(outPin, OUTPUT);
  Timer1.initialize(period); // initialize timer1, 10000 microseconds
  int dutyCycle = map(width, 0,period, 0,1023);
  Timer1.pwm(outPin, dutyCycle); // PWM on output pin
}

void loop()
{
}
```

## Discussion

You set the pulse period to a value from 1 to 1 million microseconds by setting the value of the period at the top of the sketch. You can set the pulse width to any value in microseconds that is less than the period by setting the value of `width`.

The sketch uses the Timer1 library.

Timer1 is a 16-bit timer (it counts from 0 to 65,535). On the Arduino Uno, it's the same timer used by `analogWrite` to control pins 9 and 10 (so you can't use this library and `analogWrite` on those pins at the same time). The sketch generates a pulse on pin 9 with a period and pulse width given by the values of the variables named `period` and `pulseWidth`.

`OCR1A` and `OCR1B` are constants that are defined in the code included by the Arduino core software (OCR stands for Output Compare Register).

On AVR, the Timer1 library uses a variety of registers. Many different hardware registers in the Arduino hardware are not usually needed by a sketch

(the friendly Arduino commands hide the actual register names). But when you need to access the hardware directly to get at functionality not provided by Arduino commands, these registers need to be accessed. Full details on the registers are in the Microchip datasheet for the chip. On ARM-based Teensy boards, the high-level concepts are more or less the same, but ARM uses an entirely different scheme for generating pulses.

`ICR1` (Input Compare Register for Timer1) determines the period of the pulse. This register contains a 16-bit value that is used as the maximum count for the timer. When the timer count reaches this value it will be reset and start counting again from 0. In the sketch in this recipe's Solution, if each count takes 1 microsecond and the `ICR1` value is set to `1000`, the duration of each count cycle is 1,000 microseconds.

`OCR1A` (or `OCR1B` depending on which pin you want to use) is the Output Compare Register for Timer1. When the timer count reaches this value (and the timer is in PWM mode as it is here), the output pin will be set low—this determines the pulse width. For example, if each count takes one microsecond and the `ICR1` value is set to `1000` and `OCR1A` is set to `100`, the output pin will be `HIGH` for 100 microseconds and `LOW` for 900 microseconds (the total period is 1,000 microseconds).

The duration of each count is determined by the Arduino controller timebase frequency (typically 16 MHz) and the prescale value. The prescale is the value that the timebase is divided by. For example, with a prescale of 64, the timebase will be four microseconds.

You can initialize the Timer1 library with a period (`Timer1.initialize(period);`). It is over this period that the pulse width is expressed. You set the pulse width (indirectly) by using the `Timer.pwm` function to set the duty cycle as a value between 0 and 1,023, which is exactly how you work with `analogWrite` (see Recipe 7.2). The difference here is that you're able to use Timer1 to control the period. So, if you want 20-microsecond pulses spaced evenly apart, a 40-microsecond period with a 50% duty cycle will provide this. The sketch allows you to specify the period and pulse width, and it uses the `map` function to calculate the value between 0 and

1,023 to pass to `Timer1.pwm()`. A 40-microsecond period is equivalent to 25 kHz (1,000,000/40).

## See Also

See the "See Also" for links to datasheets and other references for timers.

# 18.5 Creating a Pulse Generator

## Problem

You want to generate pulses from Arduino and control the characteristics from the Serial Monitor.

## Solution

This is an enhanced version of Recipe 18.4 that enables the frequency, period, pulse width, and duty cycle to be set from the serial port:

```
/*
 * Configurable Pulse Generator sketch
 */
#include <TimerOne.h>

const char SET_PERIOD_HEADER      = 'p';
const char SET_FREQUENCY_HEADER   = 'f';
const char SET_PULSE_WIDTH_HEADER = 'w';
const char SET_DUTY_CYCLE_HEADER  = 'c';

const int outPin = 9; // pin; use 3-4 on Teensy 3.x, 11-13 on Arduino Mega

long period = 40; // 40-microsecond period (25 KHz)
int duty = 512;    // duty as a range from 0 to 1023, 512 is 50% duty cycle

void setup()
{
  Serial.begin(9600);
  pinMode(outPin, OUTPUT);
  Timer1.initialize(period); // initialize timer1, 1000 microseconds
  Timer1.pwm(outPin, duty); // PWM on output pin
}

void loop()
{
```

```
    processSerial();
}

void processSerial()
{
  static long val = 0;

  if (Serial.available())
  {
    val = Serial.parseInt(); // Find the first number
    if (val)
    {
      char ch = Serial.read();
      switch(ch)
      {
        case SET_PERIOD_HEADER:
          period = val;
          Serial.print("Setting period to ");
          Serial.println(period);
          Timer1.setPeriod(period);
          Timer1.pwm(outPin, duty);
          show();
          break;
        case SET_FREQUENCY_HEADER:
          if(val > 0)
          {
            Serial.print("Setting frequency to ");
            Serial.println(val);
            period = 1000000 / val;
            Timer1.setPeriod(period);
            Timer1.pwm(outPin, duty);
          }
          show();
          break;
        case SET_PULSE_WIDTH_HEADER:
          if( val < period && val > 0) {
            long width = val;
            Serial.print("Setting Pulse width to ");
            Serial.println(width);
            duty = map(width, 0,period, 0,1023);
            Timer1.pwm(outPin, duty);
          }
          else
            Serial.println("Pulse width too long for current period");
          show();
          break;
        case SET_DUTY_CYCLE_HEADER:
          if( val >0 && val < 100)
          {
```

```
            Serial.print("Setting Duty Cycle to ");
            Serial.println(val);
            duty = map(val, 0,99, 0,1023);
            Timer1.pwm(outPin, duty);
            show();
          }
        }
      }
    }
}

void show()
{
  Serial.print("The period is ");
  Serial.println(period);
  Serial.print("Duty cycle is ");
  Serial.print(map(duty, 0,1023, 0,99));
  Serial.println("%");
  Serial.println();
}
```

## Discussion

This sketch is based on Recipe 18.4, with the addition of serial code to
interpret commands to receive and set the frequency, period, pulse width, and
duty cycle percent. Chapter 4 explains the technique used to accumulate the
variable `val` that is then used for the desired parameter, based on the command
letter.

You can add this function and invoke it from `setup` or `show` if you want to
print instructions to the serial port:

```
void instructions()
{
  Serial.println("Send values followed by one of the following tags:");
  Serial.println(" p - sets period in microseconds");
  Serial.println(" f - sets frequency in Hz");
  Serial.println(" w - sets pulse width in microseconds");
  Serial.println(" c - sets duty cycle in %");
  Serial.println("\n(duty cycle can have one decimal place)\n");
}
```

## See Also

Recipe 18.4

See the "See Also" for links to datasheets and other references for timers.

# 18.6 Changing a Timer's PWM Frequency

## Problem

You need to increase or decrease the Pulse Width Modulation (PWM) frequency used with `analogWrite` (see Chapter 7). For example, you are using `analogWrite` to control a motor speed and there is an audible hum because the PWM frequency is too high, or you are multiplexing LEDs and the light is uneven because PWM frequency is too low.

## Solution

You can adjust the PWM frequency by changing a register value. The register values and associated frequencies are shown in Table 18-2. This solution will work on ATmega-based boards such as the Arduino Uno, but not on ARM-based boards.

*Adjustment values for PWM*

| Timer0 (Uno pins 5 and 6, Mega pins 4 and 13) | | |
|---|---|---|
| **TCCR0B value** | **Prescale factor (divisor)** | **Frequency** |
| 32 (1) | 1 | 62500 |
| 33 (2) | 8 | 7812.5 |
| **34** | **64** | **976.5625** |
| 35 | 256 | 244.140625 |
| 36 | 1,024 | 61.03515625 |

| Timer1 (Uno pins 9 and 10, Mega pins 11 and 12) | | |
|---|---|---|
| **TCCR1B prescale value** | **Prescale factor (divisor)** | **Frequency** |

| Timer1 (Uno pins 9 and 10, Mega pins 11 and 12) | | |
| --- | --- | --- |
| **TCCR1B prescale value** | **Prescale factor (divisor)** | **Frequency** |
| 1 | 1 | 312500 |
| 2 | 8 | 3906.25 |
| **3** | **64** | **488.28125** |
| 4 | 256 | 122.0703125 |
| 5 | 1,024 | 30.517578125 |

| Timer2 (Uno pins 11 and 3, Mega pins 9 and 10) | | |
| --- | --- | --- |
| **TCCR2B value** | **Prescale factor (divisor)** | **Frequency** |
| 1 | 1 | 312500 |
| 2 | 8 | 3906.25 |
| **3** | **64** | **488.28125** |
| 4 | 256 | 122.0703125 |
| 5 | 1,024 | 30.517578125 |

All frequencies are in hertz and assume a 16 MHz system timebase. The default prescale factor of 64 is shown in bold.

This sketch enables you to select a timer frequency from the Serial Monitor. Enter a digit from 1 to 7 using the value in the lefthand column of Table 18-2 and follow this with character *a* for Timer0, *b* for Timer1, and *c* for Timer2:

```
/*
 * Set PWM Frequency sketch.
 * Frequency is set via the serial port.
 * A digit from 1-7 followed by a, b, or c Timer1, 2, 3 adjusts frequency
 */

const byte mask = B11111000; // mask bits that are not prescale values
int prescale = 0;
```

```
void setup()
{
  Serial.begin(9600);
  analogWrite(3,128);
  analogWrite(5,128);
  analogWrite(6,128);
  analogWrite(9,128);
  analogWrite(10,128);
  analogWrite(11,128);
}

void loop()
{
  if (Serial.available())
  {
    char ch = Serial.read();
    if(ch >= '1' && ch <= '7') // is ch a valid digit?
    {
      prescale = ch - '0';
    }
    else if(ch == 'a' && prescale)  // timer 0;
    {
      TCCR0B = (TCCR0B & mask)  | prescale;
    }
    else if(ch == 'b' && prescale)  // timer 1;
    {
      TCCR1B = (TCCR1B & mask)  | prescale;
    }
    else if(ch == 'c' && prescale)  // timer 2;
    {
      TCCR2B = (TCCR2B & mask)  | prescale;
    }
  }
}
```

---

**TIP**

Avoid changing the frequency of Timer0 (used for `analogWrite` pins 5 and 6) because it will result in incorrect timing using `delay` and `millis`.

---

# Discussion

If you just have LEDs connected to the analog pins in this sketch, you will not see any noticeable change to the brightness as you change the PWM speed.

You are changing the speed as they are turning on and off, not the ratio of the on/off time. If this is unclear, see the introduction to Chapter 7 for more on PWM.

You change the PWM frequency of a timer by setting the `TCCR`*n*`B` register, where *n* is the register number. On a Mega board you also have `TCCR3B`, `TCCR4B`, and `TCCR5B` for timers 3 through 5.

> **NOTE**
>
> All analog output (PWM) pins on a timer use the same frequency, so changing timer frequency will affect all output pins for that timer.

## See Also

See the "See Also" for links to datasheets and other references for timers.

Teensy 3.x boards support an `analogWriteFrequency` function that allows you to specify PWM frequency in Hz. See this PJRC page.

# 18.7 Counting Pulses

### Problem

You want to count the number of pulses occurring on a pin. You want this count to be done completely in hardware without any software processing time being consumed.

### Solution

Use the pulse counter built into the Timer1 hardware. This technique works on ATmega-based boards such as the Uno:

```
/*
 * HardwareCounting sketch
 *
 * uses pin 5 on 168/328
 */
```

```
const int hardwareCounterPin = 5;    // input pin fixed to internal Timer
const int ledPin              = LED_BUILTIN;

const int samplePeriod = 1000; // the sample period in milliseconds
unsigned int count;

void setup()
{
  Serial.begin(9600);
  pinMode(ledPin,OUTPUT);
  // hardware counter setup (see ATmega datasheet for details)
  TCCR1A=0;           // reset timer/counter control register A
}

void loop()
{
  digitalWrite(ledPin, LOW);
  delay(samplePeriod);
  digitalWrite(ledPin, HIGH);
  // start the counting
  bitSet(TCCR1B, CS12);   // Counter Clock source is external pin
  bitSet(TCCR1B, CS11);   // Clock on rising edge
  delay(samplePeriod);
  // stop the counting
  TCCR1B = 0;
  count = TCNT1;
  TCNT1 = 0;   // reset the hardware counter
  if(count > 0)
      Serial.println(count);
}
```

## Discussion

You can test this sketch by connecting the serial receive pin (pin 0) to the input pin (pin 5 on a the Uno). Each character sent should show an increase in the count—the specific increase depends on the number of pulses needed to represent the ASCII value of the characters (bear in mind that serial characters are sandwiched between start and stop pulses). Some interesting character patterns are:

```
'u' = 01010101
'3' = 00110011
'~' = 01111110
'@' = 01000000
```

If you have two Arduino boards, you can run one of the pulse generator sketches from previous recipes in this chapter and connect the pulse output (pin 9) to the input. The pulse generator also uses Timer1 (the only 16-bit timer on the Uno and other boards based on the ATmega328), so you can combine the functionality using a single board.

> **NOTE**
>
> Hardware pulse counting uses a pin that is internally wired within the hardware and cannot be changed. Use pin 5 on an Arduino Uno. The Mega uses Timer5 that is on pin 47; change `TCCR1A` to `TCCR5A` and `TCCR1B` to `TCCR5B`.

The timer's `TCCR1B` register controls the counting behavior, setting it so 0 stops counting. The values used in the `loop` code enable count in the rising edge of pulses on the input pin. `TCNT1` is the Timer1 register declared in the Arduino core code that accumulates the count value.

In `loop`, the current count is printed once per second. If no pulses are detected on pin 5, the values will be `0`.

## See Also

The FreqCount library uses the method discussed in this recipe.

You can use an interrupt to take an action when a pin state changes. See Recipe 18.2.

See the "See Also" for links to datasheets and other references for timers.

# 18.8 Measuring Pulses More Accurately

### Problem

You want to measure the period between pulses or the duration of the on or off time of a pulse. You need this as accurate as possible, so you don't want any delay due to calling an interrupt handler (as in Recipe 18.2), as this will affect the measurements.

## Solution

Use the hardware pulse measuring capability built into the Timer1 hardware. This solution uses AVR-specific facilities, and will only work on ATmega-based boards like the Arduino Uno:

```
/*
 * InputCapture Sketch
 * uses timer hardware to measure pulses on pin 8 on 168/328
 */

/* some interesting ASCII bit patterns:
 u 01010101
 3 00110011
 ~ 01111110
 @ 01000000
*/

const int inputCapturePin = 8;      // input pin fixed to internal Timer
const int ledPin          = LED_BUILTIN;

const int prescale = 8;              // prescale factor (each tick 0.5 us @16MHz)
const byte prescaleBits = B010;     // see Table 18-1 or Datasheet
// calculate time per counter tick in ns
const long  precision = (1000000/(F_CPU/1000.0)) * prescale;

const int numberOfEntries  = 64;    // the max number of pulses to measure
const int gateSamplePeriod = 1000;  // the sample period in milliseconds

volatile byte index = 0; // index to the stored readings
volatile byte gate  = 0; // 0 disables capture, 1 enables
volatile unsigned int results[numberOfEntries]; // note this is 16-bit value

/* ICR interrupt vector */
ISR(TIMER1_CAPT_vect)
{
  TCNT1 = 0;                              // reset the counter
  if(gate)
  {
    if( index != 0 || bitRead(TCCR1B ,ICES1) == true)   // wait for rising edge
    {                                     // falling edge was detected
      if(index < numberOfEntries)
      {
        results[index] = ICR1;            // save the input capture value
        index++;
      }
    }
  }
}
```

```
    TCCR1B ^= _BV(ICES1);           // toggle bit to trigger on the other edge
}

void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
  pinMode(inputCapturePin, INPUT); // ICP pin (digital pin 8 on Arduino) as input

  TCCR1A = 0 ;                      // Normal counting mode
  TCCR1B = prescaleBits ;          // set prescale bits
  TCCR1C = 0;
  bitSet(TCCR1B,ICES1);            // init input capture
  bitSet(TIFR1,ICF1);              // clear pending
  bitSet(TIMSK1,ICIE1);            // enable

  Serial.println("pulses are sampled while LED is lit");
  Serial.print( precision);        // report duration of each tick in microseconds
  Serial.println(" microseconds per tick");
}

// this loop prints the duration of pulses detected in the last second
void loop()
{
  digitalWrite(ledPin, LOW);
  delay(gateSamplePeriod);
  digitalWrite(ledPin, HIGH);
  index = 0;
  gate = 1; // enable sampling
  delay(gateSamplePeriod);
  gate = 0;  // disable sampling
  if(index > 0)
  {
    Serial.println("Durations in Microseconds are:") ;
    for( byte i=0; i < index; i++)
    {
      long duration;
      duration = results[i] * precision; // pulse duration in nanoseconds
      if(duration > 0) {
        Serial.println(duration / 1000);   // duration in microseconds
        results[i] = 0; // clear value for next reading
      }
    }
    index = 0;
  }
}
```

## Discussion

This sketch uses a timer facility called Input Capture to measure the duration of a pulse. Only 16-bit timers support this capability and this only works with pin 8 on an Arduino Uno or compatible board.

> **TIP**
>
> Input Capture uses a pin that is internally wired within the hardware and cannot be changed. Use pin 8 on an Uno and pin 48 on a Mega (using Timer5 instead of Timer1).

Because Input Capture is implemented entirely in the controller chip hardware, no time is wasted in interrupt handling, so this technique is more accurate for very short pulses (less than tens of microseconds).

The sketch uses a `gate` variable that enables measurements (when nonzero) every other second. The LED is illuminated to indicate that measurement is active. The input capture interrupt handler stores the pulse durations for up to 64 pulse transitions.

The edge that triggers the timer measurement is determined by the `ICES1` bit of the `TCCR1B` timer register. The line:

```
TCCR1B ^= _BV(ICES1);
```

toggles the edge that triggers the handler so that the duration of both high and low pulses is measured.

If the count goes higher than the maximum value for the timer, you can monitor overflow to increment a variable to extend the counting range. The following code increments a variable named `overflow` each time the counter overflows:

```
volatile int overflows = 0;

/* Overflow interrupt vector */
ISR(TIMER1_OVF_vect)                    // here if no input pulse detected
{
  overflows++;                          // increment overflow count
}
```

Change the code in `setup` as follows:

```
TIMSK1 =  _BV(ICIE1);              // enable input capture interrupt for timer 1
  TIMSK1 |= _BV(TOIE1);             // Add this line to enable overflow interrupt
```

### See Also

See the "See Also" for links to datasheets and other references for timers.

# 18.9 Measuring Analog Values Quickly

### Problem

You want to read an analog value as quickly as possible without decreasing the accuracy.

### Solution

You can increase the `analogRead` sampling rate by changing register values that determine the sampling frequency. This sketch will work on ATmega-based boards such as the Arduino Uno:

```
/*
 * Sampling rate sketch
 * Increases the sampling rate for analogRead
 */
const int sensorPin = 0;              // pin the receiver is connected to
const int numberOfEntries = 100;

unsigned long microseconds;
unsigned long duration;

int results[numberOfEntries];

void setup()
{
  Serial.begin(9600);
  while(!Serial); // Needed for Leonardo

  // standard analogRead performance (prescale = 128)
  microseconds = micros();
  for(int i = 0; i < numberOfEntries; i++)
  {
```

```
      results[i] = analogRead(sensorPin);
    }
    duration = micros() - microseconds;
    Serial.print(numberOfEntries);
    Serial.print(" readings took ");
    Serial.println(duration);

    // running with high speed clock (set prescale to 16)
    bitClear(ADCSRA,ADPS0);
    bitClear(ADCSRA,ADPS1);
    bitSet(ADCSRA,ADPS2);
    microseconds = micros();
    for(int i = 0; i < numberOfEntries; i++)
    {
      results[i] = analogRead(sensorPin);
    }
    duration = micros() - microseconds;
    Serial.print(numberOfEntries);
    Serial.print(" readings took ");
    Serial.println(duration);
}

void loop()
{
}
```

Running the sketch on a 16 MHz Arduino will produce output similar to the following:

```
100 readings took 11308
100 readings took 1704
```

## Discussion

analogRead takes around 110 microseconds to complete a reading. This may not be fast enough for rapidly changing values, such as capturing the higher range of audio frequencies. The sketch measures the time in microseconds for the standard analogRead and then adjusts the timebase used by the analog-to-digital converter (ADC) to perform the conversion faster. With a 16 MHz board, the timebase rate is increased from 125 kHz to 1 MHz. The actual performance improvement is slightly less than eight times because there is some overhead in the Arduino analogRead function that is not improved by

the timebase change. The reduction of time from 113 microseconds to 17 microseconds is a significant improvement.

The `ADCSRA` register is used to configure the ADC, and the bits set in the sketch (`ADPS0`, `ADPS1`, and `ADPS2`) set the ADC clock divisor to 16.

## See Also

Microchip has an application note that provides a detailed explanation of performance aspects of the ADC.

On ARM-based Arduino boards such as the Zero and compatibles, `analogRead` is quite a bit slower than on AVR. The AnalogReadFast library can read about 5 times faster than the AVR's `analogRead` and about 20 times faster than a SAMD21-based board like the Arduino Zero. You can install it from the Library Manager and you can find it on this GitHub page.

# 18.10 Reducing Battery Drain

## Problem

You want to reduce the power used by your application by shutting down Arduino until a period of time has elapsed or until an external event takes place.

## Solution

This Solution uses Adafruit's SleepyDog library, which supports Uno, Mega, Zero, Adafruit M0 boards, Leonardo, and (partial support) Teensy 3.X:

```
/*
 * Low power sketch
 * Reduce power usage with the Adafruit SleepyDog library
 */
#include <Adafruit_SleepyDog.h>

void setup()
{
  Serial.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT);
}
```

```
void loop()
{
  digitalWrite(LED_BUILTIN, HIGH);
  int sleepTimeMillis1 = Watchdog.sleep(500);

  digitalWrite(LED_BUILTIN, LOW);
  int sleepTimeMillis2 = Watchdog.sleep(500);

// Try to restore USB connection on Leonardo and other boards
// with Native USB.
#if defined(USBCON) && !defined(USE_TINYUSB)
  USBDevice.attach();
#endif

  Serial.print("Slept for "); Serial.print(sleepTimeMillis1);
  Serial.print("ms and "); Serial.print(sleepTimeMillis2);
  Serial.println("ms");
  delay(100); // Give the serial buffer time to transmit
}
```

> **WARNING**
>
> If you use this on an M0-based board, such as the Arduino Zero or MKR series, you may not be able to restore the USB connection to your board after the first time it powers down. If you need to re-flash the board, you can quickly double-click the reset button to put it in a bootloader mode (the sketch will not be running in this mode).

## Discussion

The Arduino Uno would run down a 9-volt alkaline battery in a few weeks. Significant power savings can be achieved if your application can suspend operation for a period of time—Arduino hardware can be put to sleep for a preset period of time, and this reduces the power consumption of the chip to less than one one-hundredth of 1 percent (from around 15 mA to around 0.001 mA) during sleep.

The library used in this recipe provides easy access to the hardware sleep function. The sleep time can range from 15 to 8,000 ms (eight seconds). To sleep for longer periods, you can repeat the delay intervals until you get the period you want:

```
unsigned long longDelay(long milliseconds)
{
  unsigned long sleptFor = 0;
  while(milliseconds > 0)
  {
    if(milliseconds > 8000)
    {
      milliseconds -= 8000;
      sleptFor += Watchdog.sleep(8000);
    }
    else
    {
      sleptFor += Watchdog.sleep(milliseconds);
      break;
    }
  }
  return sleptFor;
}
```

For each interval, the library will round the requested sleep period down to the closest sleep time supported by the underlying hardware. So, for example, if you were to request 8,600 ms on AVR, the first time through the loop, you'd get 8,000, but on the second, you'd get 500 because 500 is a valid sleep period on AVR followed by the next highest (1,000). See these *.cpp files* for the specific implementations.

Sleep mode can reduce the power consumption of the controller chip, but if you are looking to run for as long as possible on a battery, you should minimize current drain through external components such as inefficient voltage regulators, pull-up or pull-down resistors, LEDs, and other components that draw current when the chip is in sleep mode.

## See Also

The narcoleptic library can sleep for a period of time, and can also sleep until the board receives input on a particular pin. However, it will only work with AVR-based boards.

The Arduino Low Power library supports sleep modes on ARM SAMD-based boards as well as NRF52-based boards such as the Nano 33 BLE and Nano 33 BLE Sense. You can install it from the Library Manager and find more information at this Arduino page.

For an example of very low power operation, see this .

# 18.11 Setting Digital Pins Quickly

**Problem**

You need to set or clear digital pins much faster than enabled by the Arduino `digitalWrite` command.

**Solution**

Arduino `digitalWrite` provides a safe and easy-to-use method of setting and clearing pins, but it is more than 30 times slower than directly accessing the controller hardware. You can set and clear pins by directly setting bits on the hardware registers that are controlling digital pins.

This sketch uses direct hardware I/O to send Morse code (the word *arduino*) to an AM radio tuned to approximately 1 MHz. The technique used here is 30 times faster than Arduino `digitalWrite`:

```
/*
 * Morse sketch
 *
 * Direct port I/O used to send AM radio carrier at 1MHz
 */

const int sendPin = 2;

const byte WPM = 12;                      // sending speed in words per minute
const long repeatCount = 1200000 / WPM; // count determines dot/dash duration
const byte dot = 1;
const byte dash = 3;
const byte gap = 3;
const byte wordGap = 7;byte letter = 0; // the letter to send

char *arduino = ".- .-. -.. ..- .. -. ---";

void setup()
{
  pinMode(sendPin, OUTPUT);
  Serial.begin(9600);
}

void loop()
```

```
{
  sendMorse(arduino);
  delay(2000);
}

void sendMorse(char * string)
{
  letter = 0 ;
  while(string[letter]!= 0)
  {
    if(string[letter] == '.')
    {
      sendDot();
    }
    else if(string[letter] == '-')
    {
      sendDash();
    }
    else if(string[letter] == ' ')
    {
      sendGap();
    }
    else if(string[letter] == 0)
    {
      sendWordGap();
    }
    letter = letter+1;
  }
}

void  sendDot()
{
  transmitCarrier(dot *  repeatCount);
  sendGap();
}

void  sendDash()
{
  transmitCarrier(dash *  repeatCount);
  sendGap();
}

void  sendGap()
{
  transmitNoCarrier(gap *  repeatCount);
}

void  sendWordGap()
{
```

```
      transmitNoCarrier(wordGap *  repeatCount);
    }

    void transmitCarrier(long count)
    {
      while(count--)
      {
        bitSet(PORTD, sendPin);
        bitSet(PORTD, sendPin);
        bitSet(PORTD, sendPin);
        bitSet(PORTD, sendPin);
        bitClear(PORTD, sendPin);
      }
    }

    void transmitNoCarrier(long count)
    {
      while(count--)
      {
        bitClear(PORTD, sendPin);
        bitClear(PORTD, sendPin);
        bitClear(PORTD, sendPin);
        bitClear(PORTD, sendPin);
        bitClear(PORTD, sendPin);
      }
    }
```

Connect one end of a piece of wire to pin 2 and place the other end near the antenna of a medium wave AM radio tuned to 1 MHz (1,000 kHz).

## Discussion

The sketch generates a 1 MHz signal to produce dot and dash sounds that can be received by an AM radio tuned to this frequency. The frequency is determined by the duration of the `bitSet` and `bitClear` commands that set the pin `HIGH` and `LOW` to generate the radio signal. `bitSet` and `bitClear` are not functions, they are *macros*. Macros substitute an expression for executable code—in this case, code that changes a single bit in register `PORTD` given by the value of `sendPin`.

Digital pins 0 through 7 are controlled by the register named `PORTD`. Each bit in `PORTD` corresponds to a digital pin. Pins 8 through 13 are on register `PORTB`, and pins 14 through 19 are on `PORTA`. The sketch uses the `bitSet` and

`bitClear` commands to set and clear bits on the port (see Recipe 3.12). Each register supports up to eight bits (although not all bits correspond to Arduino pins). If you want to use Arduino pin 13 instead of pin 2, you need to set and clear `PORTB` as follows:

```
const int sendPin = 13;

bitSet(PORTB, sendPin - 8);
bitClear(PORTB, sendPin - 8);
```

You subtract 8 from the value of the pin because bit 0 of the `PORTB` register is pin 8, bit 1 is pin 9, and so on, to bit 5 controlling pin 13.

Setting and clearing bits using `bitSet` is done in a single instruction of the Arduino controller. On a 16 MHz Arduino, that is 62.5 nanoseconds. This is around 30 times faster than using `digitalWrite`.

The transmit functions in the sketch actually need more time updating and checking the `count` variable than it takes to set and clear the register bits, which is why the `transmitCarrier` function has four `bitSet` commands and only one `bitClear` command—the additional `bitClear` commands are not needed because of the time it takes to update and check the `count` variable.

# 18.12 Uploading Sketches Using a Programmer

## Problem
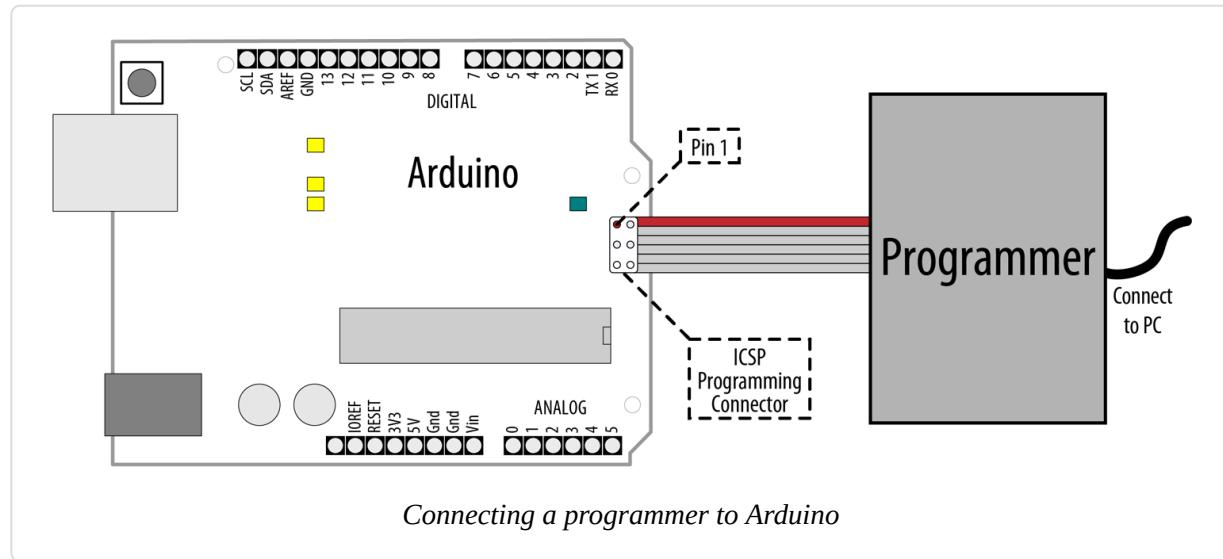
You want to upload sketches to an AVR-based Arduino (such as the Uno) using a programmer instead of the bootloader. Perhaps you want the shortest upload time, or you don't have a serial connection to your computer suitable for bootloading, or you want to use the space normally reserved for the bootloader to increase the program memory available to your sketch.

## Solution

Connect an external in-system programmer (ISP) to the Arduino programming ICSP (In-Circuit Serial Programming) connector. Programmers intended for

use with Arduino have a 6-pin cable that attaches to the 6-pin ICSP connector as shown in Figure 18-1.

Ensure that pin 1 from the programmer (usually marked with different color than the other wires) is connected to pin 1 on the ICSP connector. The programmer may have a switch or jumper to enable it to power the Arduino board; read the instructions for your programmer to ensure that the Arduino is powered correctly.



*Connecting a programmer to Arduino*

Select your programmer from the Tools menu (AVRISP, AVRISPII, USBtinyISP, Parallel programmer, or Arduino as ISP) and double-check that you have the correct Arduino board selected. From the File menu, select Upload Using Programmer to perform the upload.

## Discussion

There are a number of different programmers available, including expensive devices aimed at professional developers that offer various debugging options, low-cost self-build kits, and utilizing an additional Arduino board to perform this function. The programmer may be a native USB device, or appear as a serial port. Check the documentation for your device to see what kind it is, and whether you need to install drivers for it.

> **NOTE**
>
> The serial Rx and Tx LEDs on the Arduino will not flicker during upload because the programmer is not using the hardware serial port.

Uploading using a programmer removes the bootloader code from the chip. This frees up the space the bootloader occupies and gives a little more room for your sketch code. WIth your ISP connected, select Tools → Burn Bootloader to restore it.

## See Also

Code to convert an Arduino into an ISP programmer can be found in the sketch example named ArduinoISP. The comments in the sketch describe the connections to use.

Recipe 18.13

Suitable hardware programmers include:

- USBtinyISP
- Microchip AVRISP mkII

# 18.13 Replacing the Arduino Bootloader

## Problem

You want to replace the bootloader on an AVR-based board such as the Arduino Uno. Perhaps you can't get the board to upload programs and suspect the bootloader is not working. Or you want to replace an old bootloader with one with higher performance or different features.

## Solution

Connect a programmer and select it as discussed in Recipe 18.12. Double-check that you have the correct board selected and click Burn Bootloader from the Tools menu.

A message will appear in the IDE saying "Burning bootloader to I/O board (this may take a minute)…". Programmers with status lights should indicate that the bootloader is being written to the board. On the Uno, you should see the built-in LED flash as the board is programmed (pin 13 happens to be connected to one of the ICSP signal pins). If all goes well, you should get a message saying "Done Loading Bootloader."

Disconnect the programmer and try uploading code through the IDE to verify it is working.

## Discussion

The bootloader is a small program that runs on the chip and briefly checks each time the chip powers up to see if the IDE is trying upload code to the board. If so, the bootloader takes over and replaces the code on the chip with new code being uploaded through the serial port. If the bootloader does not detect a request to upload, it relinquishes control to the sketch code already on the board.

If you have used a serial programmer, you will need to switch the serial port back to the correct one for your Arduino board as described in Recipe 1.4.

## See Also

Optiloader, maintained by Bill Westfield, is another way to update or install the bootloader. It uses an Arduino connected as an ISP programmer, but all the bootloaders are included in the Arduino sketch code. This means an Arduino with Optiloader can program another chip automatically when power is applied —no external computer needed. The code identifies the chip and loads the correct bootloader onto it.

Optiboot is an upgrade to the Arduino bootloader that increases available sketch size, improves sketch upload speeds, and supports a number of other features. Typically, you do not install Optiboot directly to your board. Consult Optiboot's README file for a list of Optiboot-based Arduino cores you can install with the Boards Manager and then flash using Tools→Burn Bootloader.

# 18.14 Move the Mouse Cursor on a PC or Mac

**Problem**

You want Arduino to interact with an application on your computer by moving the mouse cursor. Perhaps you want to move the mouse position in response to Arduino information. For example, suppose you have connected input devices such as potentiometers or a Wii nunchuck (see Recipe 13.6) to your Arduino and you want to control the position of the mouse cursor in a program running on a PC.

## Solution

ARM-based boards such as the Arduino Zero, Adafruit Metro M0 Express, and SparkFun RedBoard Turbo, as well as ATmega32u4-based boards like the Leonardo can appear like a USB mouse to your computer using the built-in Mouse library. This will not work on the Uno or directly compatible boards. Here is a sketch that moves the mouse cursor based on the position of two potentiometers.

Wire up two potentiometers (see Recipe 5.6), one to analog input 4 (A4) and the other to analog input 5 (A5). Connect a switch to digital pin 2 (as described in Recipe 5.2) to act as the left mouse button, then run the sketch:

```
/*
 * Mouse Emulation sketch
 * Use the Mouse library to emulate a USB mouse device
 */

#include "Mouse.h"

const int buttonPin = 2; // left click
const int potXPin = A4;   // analog pins for pots
const int potYPin = A5;

int last_x = 0;
int last_y = 0;

void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT_PULLUP);
```

```
  // Get initial potentiometer positions. Range is -127 to 127
  last_x = (512 - (int) analogRead(potXPin)) / 4;
  last_y = (512 - (int) analogRead(potYPin)) / 4;

  Mouse.begin();
}

void loop()
{
  // Get current positions.
  int x = (512 - (int) analogRead(potXPin)) / 4;
  int y = (512 - (int) analogRead(potYPin)) / 4;

  Serial.print("last_x: "); Serial.println(last_x);
  Serial.print("last_y: "); Serial.println(last_y);
  Serial.print("x: "); Serial.println(x);
  Serial.print("y: "); Serial.println(y);

  // calculate the movement distance based on the potentiometer state
  int xDistance = last_x - x;
  int yDistance = last_y - y;

  // Update last known positions of the potentiometer
  last_x = x;
  last_y = y;

  // if X or Y movement is greater than 3, move:
  if (abs(xDistance) > 3 || abs(yDistance) > 3)
  {
    Serial.print("x move: "); Serial.println(xDistance);
    Serial.print("y move: "); Serial.println(yDistance);
    Mouse.move(xDistance, yDistance, 0);
  }

  // if the mouse button is pressed:
  if (digitalRead(buttonPin) == LOW)
  {
    if (!Mouse.isPressed(MOUSE_LEFT))
    {
      Mouse.press(MOUSE_LEFT); // Click
    }
  }
  else
  {
    if (Mouse.isPressed(MOUSE_LEFT))
    {
      Mouse.release(MOUSE_LEFT); // Release
    }
  }
```

```
    Serial.println();
    delay(10);
}
```

## Discussion

This technique for controlling applications running on your computer is easy to implement and should work with any operating system that can supports USB mice. If you need to invert the direction of movement on the x- or y-axis, you can do this by changing the sign of `xDistance` and `yDistance`:

```
Mouse.move(-xDistance, -yDistance, 0);
```

> **WARNING**
>
> A runaway `Mouse` object has the ability to make it difficult to reprogram the board. On most ARM-based boards, you can double-click the reset button to put it in a bootloader mode.

## See Also

Reference for the Mouse library

The ArduinoJoystick library allows ATmega32u4-based boards to emulate a USB joystick.

The MIDIUSB library allows ATmega32u4-based and ARM boards to emulate a USB MIDI device.

The built-in Keyboard library allows ATmega32u4-based and ARM boards to emulate a USB keyboard.

The HID library allows offers emulation of a variety of USB devices.

# Electronic Components
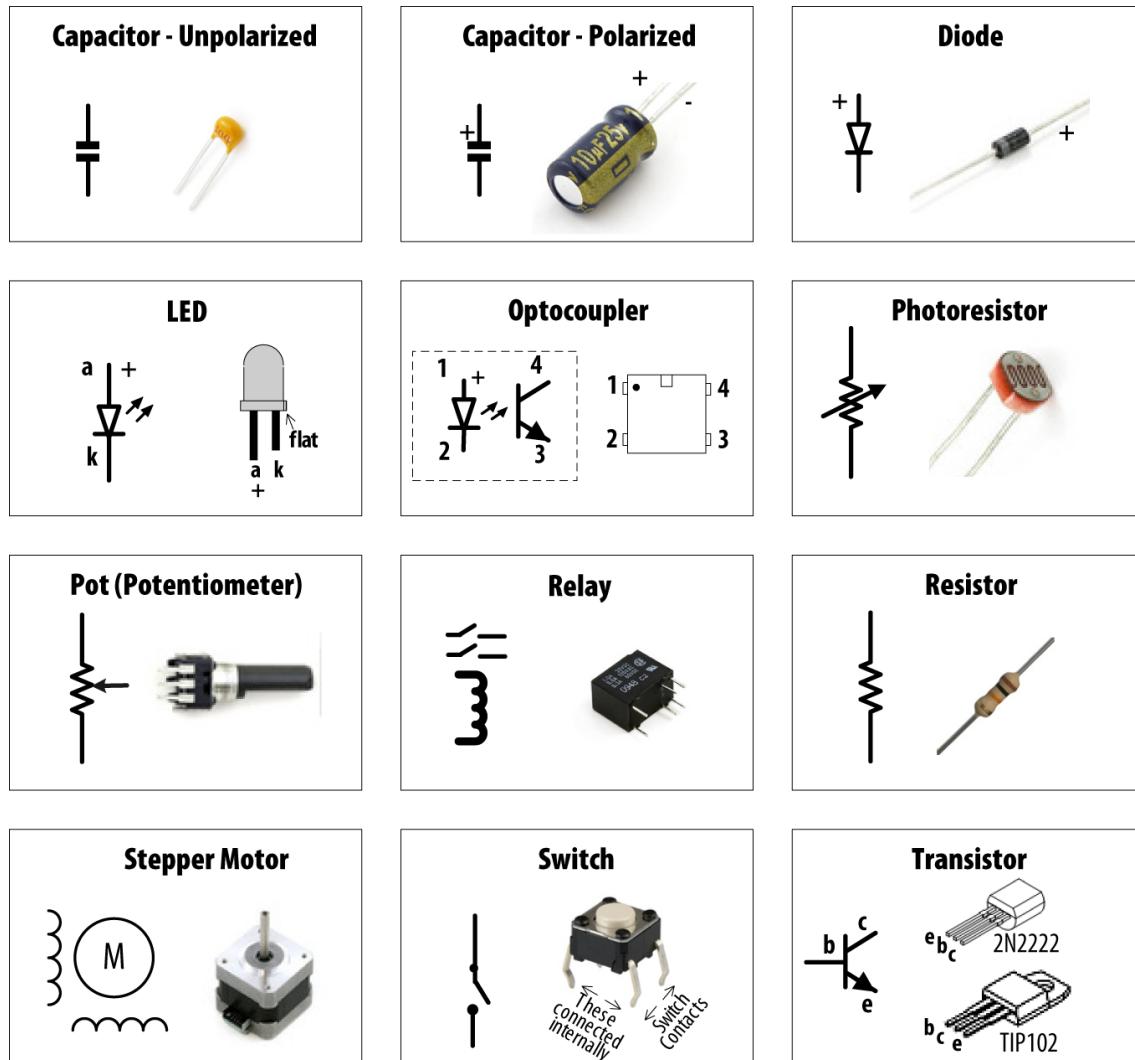
If you are just starting out with electronic components, you may want to purchase a beginner's starter kit that contains the basic components needed for many of the recipes in this book. These usually include the most common resistors, capacitors, transistors, diodes, LEDs, and switches.

Here are some popular choices:

- Arduino Starter Kit
- Getting Started with Arduino Kit
- SparkFun Tinker Kit>
- Adafruit MetroX Classic Kit
- ARDX: The starter kit for Arduino

You can also purchase the individual components for your project, as shown in Figure A-1. The following sections provide an overview of common electronic components.

| Capacitor - Unpolarized | Capacitor - Polarized | Diode |
| LED | Optocoupler | Photoresistor |
| Pot (Potentiometer) | Relay | Resistor |
| Stepper Motor | Switch | Transistor |

*Schematic representation of common components*

# Capacitor

Capacitors store an electrical charge for a short time and are used in digital circuits to filter (smooth out) dips and spikes in electrical signals. The most commonly used capacitor is the nonpolarized ceramic capacitor; for example, a 100 nF disc capacitor used for decoupling (reducing noise spikes). Electrolytic capacitors can generally store more charge than ceramic caps and are used for higher-current circuits, such as power supplies and motor circuits. Electrolytic capacitors are usually polarized,

and the negative leg (marked with a minus sign) must be connected to ground (or to a point with lower voltage than the positive leg). Chapter 8 contains examples showing how capacitors are used in motor circuits.

# Diode

Diodes permit current to flow in one direction and block it in the other direction. Most diodes have a band (see Figure A-1) to indicate the cathode (negative) end.

Diodes such as the 1N4148 can be used for low-current applications such as the levels used on Arduino digital pins. The 1N4001 diode is a good choice for higher currents (up to 1 amp).

# Integrated Circuit

Integrated circuits contain electronic components packaged together in a convenient chip. These can be complex, like the Arduino controller chip that contains thousands of transistors, or as simple as the optical isolator component used in Chapter 10 that contains just two semiconductors. Some integrated circuits (such as the Arduino chip) are sensitive to static electricity and should be handled with care.

# Keypad

A keypad is a matrix of switches used to provide input for numeric digits. See Chapter 5.

# LED

An LED (light-emitting diode) is a diode that emits light when current flows through the device. As they are diodes, LEDs only conduct electricity in one direction. See Chapter 7.

# Motor (DC)

Motors convert electrical energy into physical movement. Most small direct current (DC) motors have a speed proportional to the voltage, and you can reverse the direction they move by reversing the polarity of the voltage across the motor. Most motors need more current than the Arduino pins provide, and a component such as a transistor is required to drive the motor. See Chapter 8.

# Optocoupler

Optocouplers (also called optoisolators) provide electrical separation between devices. This isolation allows devices that operate with different voltage levels to work safely together. See Chapter 10.

# Photocell (Photoresistor)

Photocells are variable resistors whose resistance changes with light. See Chapter 6.

# Piezo

A small ceramic transducer that produces sound when pulsed, a Piezo is polarized and may have a red wire indicating the positive end and a black wire indicating the side to be connected to ground. See Chapter 9.

# Pot (Potentiometer)

A potentiometer (pot for short) is a variable resistor. The two outside terminals act as a fixed resistor. A movable contact called a wiper (or slider) moves across the resistor, producing a variable resistance between the center terminal and the two sides. See Chapter 5.

# Relay

A relay is an electronic switch—circuits are opened or closed in response to a voltage on the relay coil, which is electrically isolated from the switch. Most relay coils require more current than Arduino pins provide, so they need a transistor to drive them. See Chapter 8.

# Resistor

Resistors resist the flow of electrical current. A voltage flowing through a resistor will limit the current proportional to the value of the resistor (see Ohm's law). The bands on a resistor indicate the resistor's value. Chapter 7 contains information on selecting a resistor for use with LEDs.

# Solenoid

A solenoid produces linear movement when powered. Solenoids have a metallic core that is moved by a magnetic field created when passing current through a coil. See Chapter 8.

# Speaker

A speaker produces sound by moving a diaphragm (the speaker cone) to create sound waves. The diaphragm is driven by sending an audio frequency electrical signal to a coil of wire attached to the diaphragm. See Chapter 9.

# Stepper Motor

A stepper motor rotates a specific number of degrees in response to control pulses. See Chapter 8.

# Switch

A switch makes and breaks an electrical circuit. Many of the recipes in this book use a type of pushbutton switch known as a *tactile switch*. Tactile switches have two pairs of contacts that are connected together when the button is pushed. The pairs are wired together, so you can use either one of the pair. Switches that make contact when pressed are called Normally Open (NO) switches. See Chapter 5.

## Transistor

Transistors are used to switch on high currents or high voltages in digital circuits. In analog circuits, transistors are used to amplify signals. A small current through the transistor base results in a larger current flowing through the collector and emitter.

For currents up to .5 amperes (500 mA) or so, the 2N2222 transistor is a widely available choice. For currents up to 5 amperes, you can use the TIP120 transistor.

See Chapters 7 and 8 for examples of transistors used with LEDs and motors.
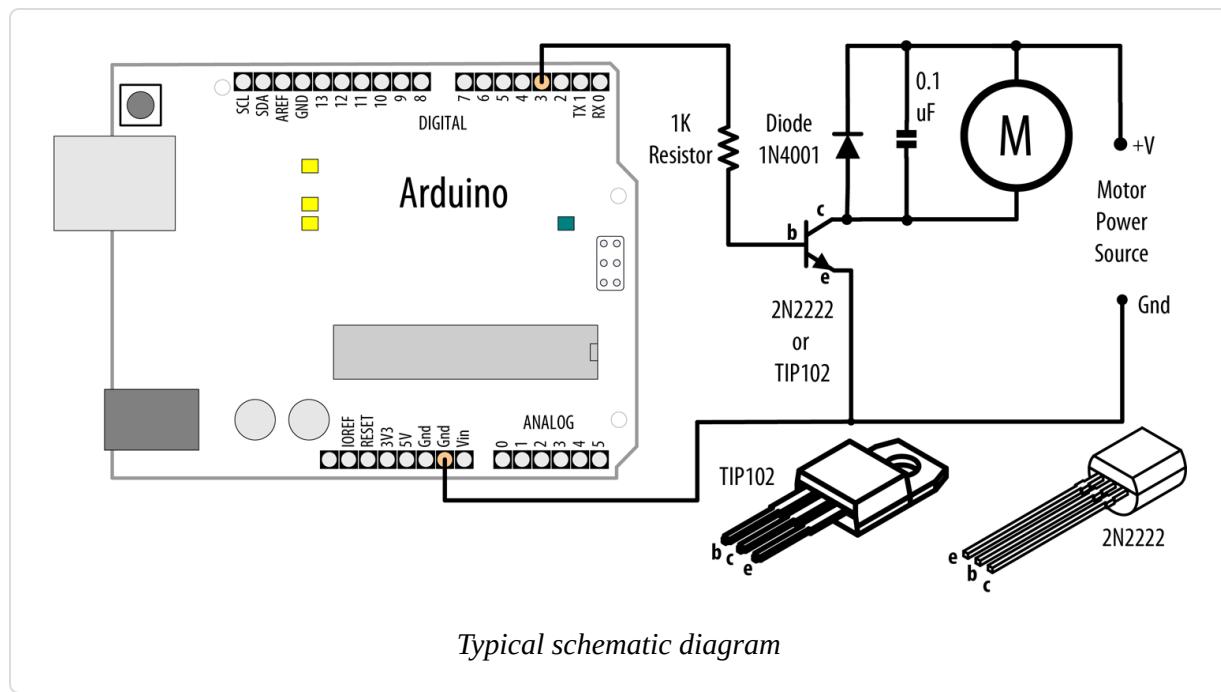
## See Also

For more comprehensive coverage of basic electronics, see the list of books in "What Was Left Out".

# Using Schematic Diagrams and Datasheets

A *schematic diagram,* also called a *circuit diagram,* is the standard way of describing the components and connections in an electronic circuit. It uses iconic symbols to represent components, with lines representing the connections between components.

Figure A-1 in Appendix A shows some of the most common components, and the symbols used for them in circuit diagrams. Figure B-1 is a schematic diagram from Recipe 8.8 that illustrates the symbols used in a typical diagram.



*Typical schematic diagram*
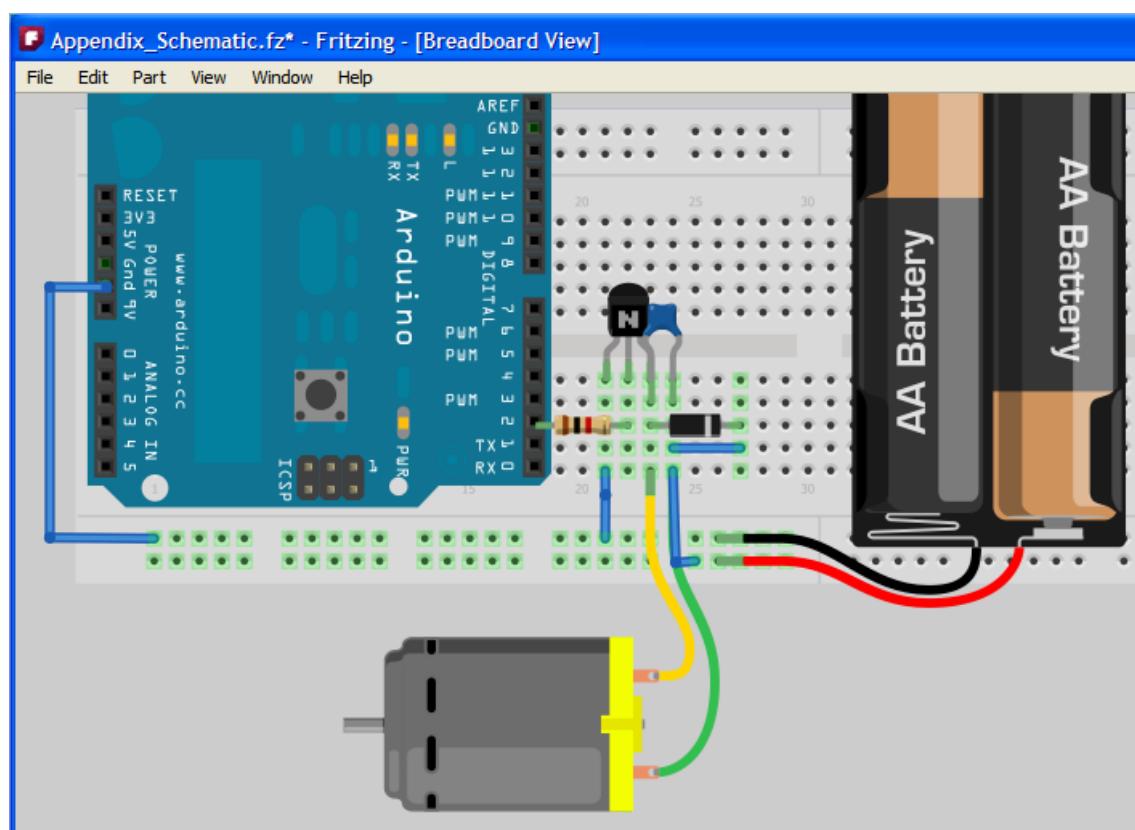
A circuit diagram represents the connections of a circuit, but it is not a drawing of the actual physical layout. Although you may initially find that drawings and photos of the physical wiring can be easier to understand than a schematic, in a complicated circuit it can be difficult to clearly see where each wire gets connected. Circuit diagrams are like maps. They have

conventions that help you to orient yourself once you become familiar with their style and symbols. For example, inputs are usually to the left, outputs to the right; 0V or ground connections are usually shown at the bottom of simple circuits, the power at the top.

Components such as the resistor and capacitor used here are not polarized—they can be connected either way around. Transistors, diodes, and integrated circuits are polarized, so it is important that you identify each lead and connect it according to the diagram.

Figure B-2 shows how the wiring could look when connected using a breadboard. This drawing was produced using a tool called Fritzing that enables the drawing of electronic circuits.
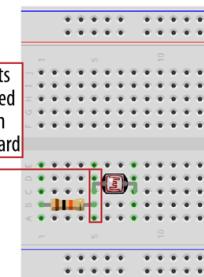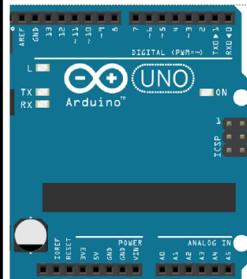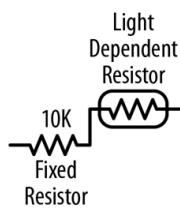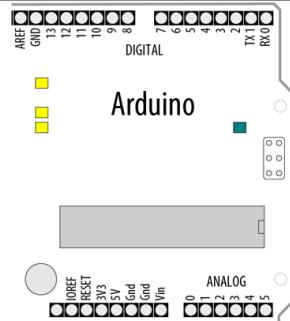


*Physical layout of the circuit shown in Figure B-1*
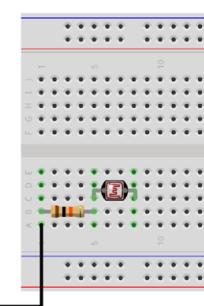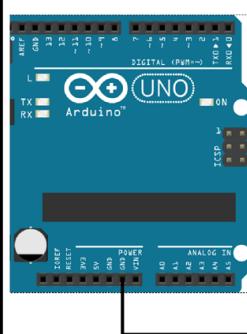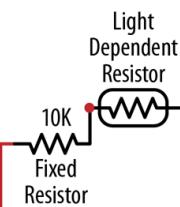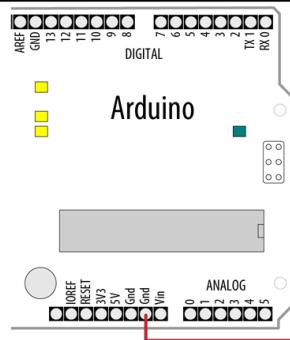
Wiring a working breadboard from a circuit diagram is easy if you break the task into individual steps. Figure B-3 shows how each step of breadboard

construction is related to a circuit diagram. The circuit shown is from
Recipe 1.6.
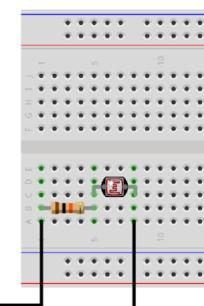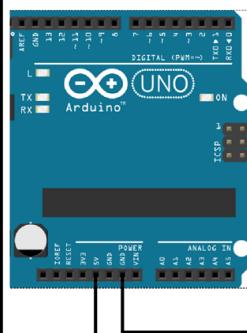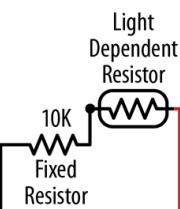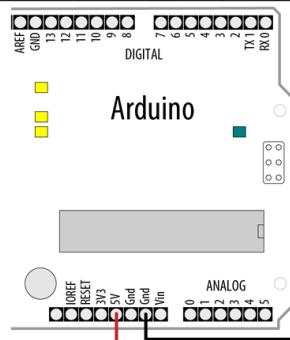
This Recipe uses a 10K ohm resistor (Brown Black and Orange bands) and a Light Dependent Resistor (LDR).
The following figures show the relationship between the schematic diagram and the physical connections.



These points are connected together in the breadboard
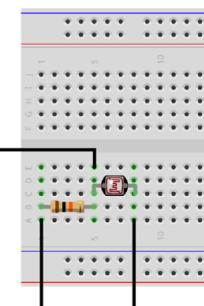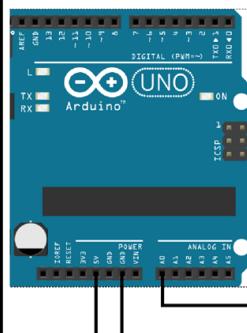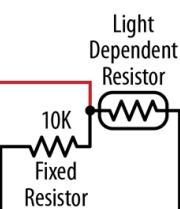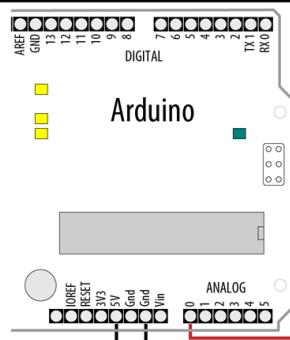
Start by plugging the 10K resistor and Light Dependent Resistor into the breadboard. Connect one side of each component together.



Connect a wire from an Arduino Ground pin (any one of the pins labelled Gnd) to the side of the 10K resistor not connected to the LDR.



Connect a wire from an Arduino 5V pin to the side of the LDR not connected to the 10K resistor.



Connect a wire from Arduino Analog input 0 to the junction of the LDR and 10K resistor.

# How to Read a Datasheet

Datasheets are produced by the manufacturers of components to summarize the technical characteristics of a device. Datasheets contain definitive information about the performance and usage of the device; for example, the minimum voltage needed for the device to function and the maximum voltage that it can reliably tolerate. Datasheets contain information on the function of each pin and advice on how to use the device.

For more complicated devices, such as LCDs, the datasheet covers how to initialize and interact with the device. Very complex devices, such as the Arduino controller chip, require hundreds of pages to explain all the capabilities of the device.

Datasheets are written for design engineers, and they usually contain much more information than you need to get most devices working in an Arduino project. Don't be intimidated by the volume of technical information; you will typically find the important information in the first couple of pages. There will usually be a circuit diagram symbol labeled to show how the connections on the device correspond to the symbols. This page will typically have a general description of the device (or family of devices) and the kinds of uses they are suitable for.

After this, there is usually a table of the electrical characteristics of the device.

Look for information about the maximum voltage and the current the device is designed to handle to check that it is in the range you need. For components to connect directly to a standard Arduino board, devices need to operate at +5 volts. To be powered directly from the pin of the Arduino, they need to be able to operate with a current of 40 mA or less.

# Choosing and Using Transistors for Switching

The Arduino Uno output pins are designed to handle currents up to 40 mA (milliamperes), which is only 1/25 of an amp. Other boards may be rated even lower. For example, the Uno WiFi Rev 2 board is rated at 20 mA, the Zero at 7 mA. You can use a transistor to switch larger currents. This section provides guidance on transistor selection and use.

The most commonly used transistors with Arduino projects are bipolar transistors. These can be of two types (named NPN and PNP) that determine the direction of current flow. NPN is more common for Arduino projects and is the type that is illustrated in the recipes in this book. For currents up to .5 amperes (500 mA) or so, the 2N2222 transistor is a widely available choice; the TIP120 transistor is a popular choice for currents up to 5 amperes.

Figure B-1 shows an example of a transistor connected to an Arduino pin used to drive a motor. See Chapter 8 for some recipes that use transistors.

Transistor datasheets are usually packed with information for the design engineer, and most of this is not relevant for choosing transistors for Arduino applications. Table B-1 shows the most important parameters you should look for (the values shown are for a typical general-purpose transistor). Manufacturing tolerances result in varying performance from different batches of the same part, so datasheets usually indicate the minimum, typical, and maximum values for parameters that can vary from part to part.

Here's what to look for:

Collector-emitter voltage

Make sure the transistor is rated to operate at a voltage higher than the voltage of the power supply for the circuit the transistor is controlling. Choosing a transistor with a higher rating won't cause any problems.

Collector current

This is the absolute maximum current the transistor is designed to handle. It is a good practice to choose a transistor that is rated at least 25% higher than what you need.

DC current gain

This determines the amount of current needed to flow through the base of the transistor to switch the output current. Dividing the output current (the maximum current that will flow through the load the transistor is switching) by the gain gives the amount of current that needs to flow through the base. Use Ohm's law (Resistance = Voltage / Current) to calculate the value of the resistor connecting the Arduino pin to the transistor base. For example, if the desired collector current is 1 amp and the gain is 100, you need at least 0.01 amps (10 mA) through the transistor base. For a 5-volt Arduino: 5 / .01 = 500 ohms (500 ohms is not a standard resistor value so 470 ohms would be a good choice).

Collector-emitter saturation voltage

This is the voltage level on the collector when the transistor is fully conducting. Although this is usually less than 1 volt, it can be significant when calculating a series resistor for LEDs or for driving high-current devices.

*Example of key transistor datasheet specifications*

| Absolute maximum ratings | | | | |
| --- | --- | --- | --- | --- |
| Parameter | Symbol | Rating | Units | Comment |

**Absolute maximum ratings**

| Parameter | Symbol | Rating | Units | Comment |
|---|---|---|---|---|
| Collector-emitter voltage | Vceo | 40 | Volts | The maximum voltage between the collector and emitter |
| Collector current | Ic | 600 | mA or A | The maximum current that the transistor is designed to handle |

**Electrical characteristics**

| Parameter | Symbol | Rating | Units | Comment |
|---|---|---|---|---|
| DC current gain | Ic | 90 @ 10 mA | | Gain with 10 mA current flowing |
| | Ic | 50 @ 500 mA | | Gain with 500 mA current flowing |
| Collector-emitter saturation voltage | Vce (sat) | 0.3 @ 100 mA  1.0 @ 500 mA | Volts  Volts | Voltage drop across collector and emitter at various currents |
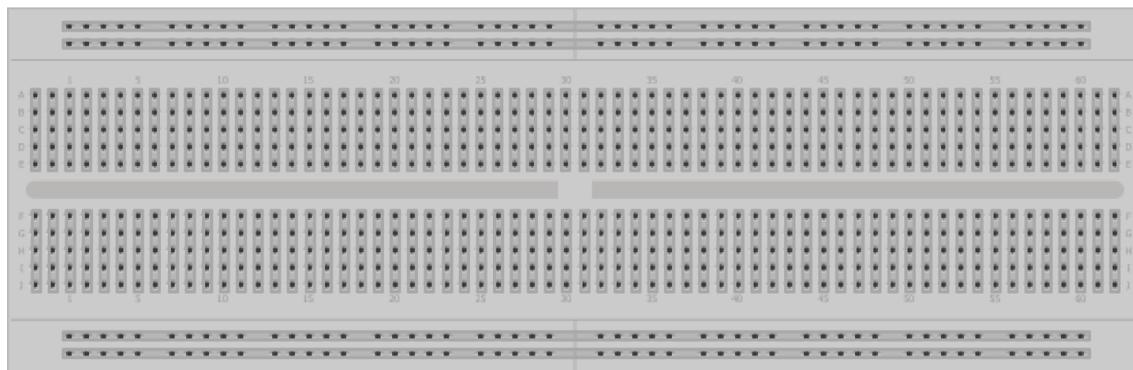
# Building and Connecting the Circuit

There are a variety of ways to connect components when you build a circuit. The most common practice, especially during the prototyping phase, is to use a solderless breadboard. As you build your circuit, you will need to power it, and with power comes the need to manage power fluctuations. This appendix explains the basics of breadboarding and offers some tips on working with power supplies.

## Using a Breadboard

A breadboard enables you to prototype circuits quickly, without having to solder the connections. Figure C-1 shows an example of a breadboard.



*Breadboard for prototyping circuits*

Breadboards come in various sizes and configurations. The simplest kind is just a grid of holes in a plastic block. Inside are strips of metal that provide electrical connections between holes in the shorter rows. Pushing the legs of two different components into the same row joins them together electrically. A deep channel running down the middle indicates that there is a break in

connections there, meaning you can push a chip in with the legs at either side of the channel without connecting them together.

Some breadboards have two strips of holes running along the long edges of the board that are separated from the main grid. These have strips running down the length of the board inside, and provide a way to connect a common voltage. They are usually in pairs for +5 volts and ground. These strips are referred to as *rails* and they enable you to connect power to many components or points in the board.

While breadboards are great for prototyping, they have some limitations. Because the connections are push-fit and temporary, they are not as reliable as soldered connections. If you are having intermittent problems with a circuit, it could be due to a poor connection on a breadboard.

# Connecting and Using External Power Supplies and Batteries

The Arduino can be powered from an external power source rather than through the USB lead. You may need more current than the USB connection can provide (the maximum USB current is 500 mA; some USB hubs only supply 100 mA), or you may want to run the board without connection to the computer after the sketch is uploaded.

The Arduino Uno board has a socket for connecting external power. This can be an AC-powered power supply or a battery pack.

> **NOTE**
>
> These details relate to the Uno and Mega boards. Other Arduino and compatible boards may not protect the board from reverse connections, or they may automatically switch to use external power and may not accept higher voltages. If you are using a different board, check before you connect power or you may damage the board.

If you are using an AC power supply, you need one that produces a DC voltage between 7 and 12 volts. Choose a power supply that provides at

least as much current as you need (there is no problem in using a power supply with a higher current than you need). Wall wart–style power supplies come in two broad types: regulated and unregulated. A regulated power supply has a circuit that maintains the specified voltage, and this is a good choice to use with Arduino. An unregulated power supply will produce a higher voltage when run at a lower current and can sometimes produce twice the rated voltage when driving low-current devices such as Arduino. Voltages higher than 12 volts can overheat the regulator on the Arduino, and this can cause intermittent operation or even damage the board.

Battery voltage should also be in the range of 7 to 12 volts. Battery current is rated in mAh (the amount of milliamperes the battery can supply in one hour). A battery with a rating of 500 mAh (a typical alkaline 9V battery) should last around 20 hours with an Arduino board drawing 25 mAh. If your project draws 50 mA, the battery life will be halved, to around 10 hours. How much current your board uses depends mostly on the devices (such as LEDs and other external components) that you use. Bear in mind that the Uno board is designed to be easy to use and robust, but it is not optimized for low power use with a battery. See Recipe 18.10 for advice on reducing battery drain.

The positive (+) connection from the power supply should be connected to the center pin of the Arduino power plug. If you connect it the wrong way around on an Uno or Mega, the board will not break, but it will not work until the connection is reversed. These boards automatically detect that an external power supply is connected and use that to power the board. You can still have the USB lead plugged in, so serial communication and code uploading will still work.

# Using Capacitors for Decoupling

Digital circuits switch signals on and off quickly, and this can cause fluctuations in the power supply voltage that can disrupt proper operation of the circuit. Properly designed digital circuits use decoupling capacitors to filter these fluctuations. Decoupling capacitors should be connected across

the power pins of each IC in your circuit with the capacitor leads kept as short as possible. A ceramic capacitor of 0.1 uF is a good choice for decoupling—that value is not critical (20% tolerance is OK).

## Using Snubber Diodes with Inductive Loads

Inductive loads are devices that have a coil of wire inside. This includes motors, solenoids, and relays. The interruption of current flow in a coil of wire generates a spike of electricity. This voltage can be higher than +5 volts and can damage sensitive electronic circuits such as Arduino pins. Snubber diodes are used to prevent that by conducting the voltage spikes to ground. Figure B-1 in Appendix B shows an example of a snubber diode used to suppress voltage spikes when driving a motor.

## Working with AC Line Voltages

When working with an AC line voltage from a wall socket, the first thing you should consider is whether you can avoid working with it. Electricity at this voltage is dangerous enough to kill *you,* not just your circuit, if it is used incorrectly. It is also dangerous for people using whatever you have made if the AC line voltage is not isolated properly.

Hacking controllers for devices that are manufactured to work with mains voltage, or using devices designed to be used with microcontrollers to control AC line voltages, is safer (and often easier) than working with mains voltage itself. See Chapter 10 for recipes on controlling external devices for examples of how to do this.

# Tips on Troubleshooting Software Problems

As you write and modify code, you will get code that doesn't work for some reason (this reason is usually referred to as a *bug*). There are two broad areas of software problems: code that won't compile and code that compiles and uploads to the board but doesn't behave as you want.

## Code That Won't Compile

Your code might fail to compile when you click the Verify (checkbox) button or the Upload button (see Chapter 1). This is indicated by red error messages in the black console area at the bottom of the Arduino software window and a yellow highlight in the code if there is a specific point where the compilation failed. Often the problem in the code is in the line immediately before the highlighted line. The error messages in the console window are generated by the command-line programs used to compile and link the code (see Recipe 17.1 for details on the build process). This message may be difficult to understand when you first start.

One of the most common errors made by people new to Arduino programming is omission of the semicolon at the end of a line. This can produce various different error messages, depending on the next line. For example, this code fragment:

```
void loop()
{
  digitalWrite(ledPin, HIGH)    // <- BUG: missing semicolon
  delay(1000);
}
```

produces the following error message:

```
In function 'void loop()':
   error: expected ';' before 'delay
```

A less obvious error message is:

```
expected ',' or ';' before 'void'
```

Although the cause is similar, a missing semicolon after a constant results in the preceding error message, as in this fragment:

```
const int ledPin =  LED_BUILTIN     // <- BUG: missing semicolon after constant
void loop()
```

The combination of the error message and the line highlighting provides a good starting point for closer examination of the area where the error has occurred.

Another common error is misspelled words, resulting in the words not being recognized. This includes incorrect capitalization—LedPin is different from ledPin. This fragment:

```
const int ledPin =  LED_BUILTIN;

digitalWrite(LedPin, HIGH);    // <- BUG: the capitalization is different
```

results in the following error message:

```
note: suggested alternative: 'ledPin':
   error: 'LedPin' was not declared in this scope
```

The fix is to use exactly the same spelling and capitalization as the variable declaration, as the suggestion indicates.

You must use the correct number and type of parameters for function calls (see Recipe 2.10). The following fragment:

```
digitalWrite(ledPin);    // <- BUG: this is missing the second parameter
```

generates this error message:

```
error: too few arguments to function 'void digitalWrite(uint8_t, uint8_t)'
 error: at this point in file
```

The cursor in the IDE will point to the line in the sketch that contains the error.

Functions in sketches that are missing the return type will generate an error. This fragment:

```
loop()                  // <- BUG: loop is missing the return type
{
}
```

produces this error:

```
expected constructor, destructor, or type conversion before ';' token
```

The error is fixed by adding the missing return type:

```
void loop()                 // <- return type precedes function name
{
}
```

Incorrectly formed comments, such as this fragment that is missing the second "/":

```
digitalWrite(ledPin, HIGH);   / set the LED on (BUG: missing //)
```

result in this error:

```
error: expected primary-expression before '/' token
```

It is good to work on a small area of code, and regularly verify/compile to check the code. You don't need to upload to check that the sketch compiles (just click the Verify button in the IDE). The earlier you become aware of a problem, the easier it is to fix it, and the less impact it will have on other

code. It is much easier to fix code that has one problem than it is to fix a large section of code that has multiple errors in it.

# Code That Compiles but Does Not Work as Expected

There is always a feeling of accomplishment when you get your sketch to compile without errors, but correct syntax does not mean the code will do what you expect.

This is usually a subtler problem to isolate. You are now in a world where software and hardware are interacting. It is important to try to separate problems in hardware from those in software. Carefully check the hardware (see Appendix E) to make sure it is working correctly.

> ### TROUBLESHOOTING INTERRELATED HARDWARE/SOFTWARE PROBLEMS
>
> Some problems are not due strictly to software or hardware errors, but to the interplay between them.
>
> The most common of these is connecting the circuit to one pin and in software reading or writing a different pin. Hardware and software are both correct in isolation—but together they don't work. You can change either the hardware or the software to fix this: change the pin in software or move the connection to the pin number declared in your sketch.

If you are sure the hardware is wired and working correctly, the first step in debugging your sketch is to carefully read through your code to review the logic you used. Pausing to think carefully about what you have written is usually a faster and more productive way to fix problems than diving in and adding debugging code. It can be difficult to see faulty reasoning in code you have just written. Walking away from the computer not only helps prevent repetitive strain injury, but it also refreshes your troubleshooting abilities. On your return, you will be looking at the code afresh, and it is very common for the cause of the error to jump out at you where you could not see it before.

If this does not work, move on to the next technique: use the Serial Monitor to watch how the values in your sketch are changed when the program runs and whether conditional sections of code run. Chapter 4 explains how to use Arduino serial print statements to display values on your computer.

To troubleshoot, you need to find out what is actually happening when the code runs. `Serial.print()` lines in your sketch can display what part of the code is running and the values of your variables. These statements are temporary, so you should remove them once you have fixed your problem. The following sketch reads an analog value and is based on the Solution from Recipe 5.6. The sketch should change the blink rate based on the setting of a variable resistor (see the Discussion for Recipe 5.6 for more details on how this works). If the sketch does not function as expected, you can see if the software is working correctly by using a `serial.print()` statement to display the value read from the analog pin:

```
const int potPin = A0;
const int ledPin = LED_BUILTIN;
int val = 0;

void setup()
{
  Serial.begin(9600);        // <- add this to initialize Serial
  pinMode(ledPin, OUTPUT);
}

void loop() {
  val = analogRead(potPin);    // read the voltage on the pot
  Serial.println(val);         // <- add this to display the reading
  digitalWrite(ledPin, HIGH);
  delay(val);
  digitalWrite(ledPin, LOW);
  delay(val);
}
```

If the value displayed on the Serial Monitor does not vary from 0 to 1,023 when the pot (variable resistor) is changed, you probably have a hardware problem—the pot may be faulty or not wired correctly. If the value does change but the LED does not blink, the LED may not be wired correctly.

# Tips on Troubleshooting Hardware Problems

Hardware problems can have more immediate serious ramifications than software problems because incorrect wiring can damage components. The most important tip is *always disconnect power when making or changing connections, and double-check your work before connecting power*.

> **NOTE**
>
> Unplug Arduino from power while building and modifying circuits.

Applying power is the last thing you do to test a circuit, not the first.

For a complicated circuit, build it a bit at a time. Often a complicated circuit consists of a number of separate circuit elements, each connected to a pin on the Arduino. If this is the case, build one bit and test it, then the other bits, one at a time. If you can, test each element using a known working sketch such as one of the example sketches supplied with Arduino or on the Arduino Playground. It usually takes much less time getting a complex project working if you test each element separately.

For some of the techniques in this appendix, you will need a multimeter (any inexpensive digital meter that can read volts, current, and resistance should be suitable).

The most effective test is to carefully inspect the wiring and check that it matches the circuit you are trying to build. Take particular care that power connections are the correct way around and there are no short circuits, +5 volts accidentally connected to 0 volts, or legs of components touching where they should not. If you are unsure how much current a device connected to an Arduino pin will draw, test it with a multimeter before

connecting it to a pin. If the circuit draws more than 40 mA (20 mA on the WiFi Rev2/Nano Every and 7 mA on most ARM-based boards), the pin on the Arduino can get damaged. (See this video tutorial and PDF for details on how to use a multimeter.)

You may be able to test output circuits (LEDs or motors) by connecting to the positive power supply instead of the Arduino pin. If the device does not function, it may be faulty or not wired correctly.

If the device tests OK, but when you connect to the pin and run the code you don't get the expected behavior, the pin might be damaged or the problem is in software.

To test a digital output pin, hook up an LED with a resistor (see Chapter 7) or connect a multimeter to read the voltage and run the Blink sketch on that pin. If the LED does not flash, or doesn't jump between 0 volts and 5 volts (or 3.3 on a 3.3V board) on the multimeter, the output pin is probably damaged.

Take care that your wiring does not accidentally connect the power line to ground. If this happens on a board that is powered from USB, all the lights will go out and the board will become unresponsive. The board has a component, called a *polyfuse*, that protects the computer from excessive current being drawn from the USB port. If you draw too much current, it will "trip" and switch off power to the board. You can reset it by unplugging the board from the USB hub (you may also need to restart your computer). Before reconnecting the power, check your circuits to find and fix the faulty wiring; otherwise, the polyfuse will trip again when you plug it back in.

# Still Stuck?

After trying everything you can think of, you still may not be able to get your project to work. If you know someone who is using Arduino or similar boards, you could ask them for help. But if you don't, use the internet—particularly the Arduino forum site. This is a place where people of all experience levels can ask questions and share knowledge. Use the forum

search box (it's in the top-right corner) to try to find information relating to your project. A related site is the Arduino Playground, a wiki for user-contributed information about Arduino.

If a search doesn't yield the information you need, you can post a question to the Arduino forum. The forum is very active, and if you ask your question clearly, you are likely to get a quick answer.

To ask your question well, identify which forum section the question should go in and choose a title for your thread that reflects the specific problem you want to solve. Post in only one place—most people who are likely to answer will check all the sections that have new posts, and multiple posts will irritate people and make it less likely that you will get help.

Explain your problem, and the steps you have taken to try to fix it. It's better to describe what happens than to explain why you think it is happening. Include all relevant code, but try to produce a concise test sketch that does not contain code that you know is not related to the problem. If your problem relates to a device or component that is external to the Arduino board, post a link to the datasheet. If the wiring is complex, post a diagram or photo showing how you have connected things up.

# Digital and Analog Pins

Tables F-1 and F-2 show the digital and analog pins for the Arduino Uno board and the Mega board. The "Arduino" column is for the ATmega168/328, and the "Mega" column is for the ATmega1280/2560.

The Port column lists the physical port used for the pin—see Recipe 18.11 for information on how to set a pin by writing directly to a port. The introduction to Chapter 18 contains more details on timer usage. The table shows:

- USART RX is hardware serial receive

- USART TX is hardware serial transmit

- Ext Int is external interrupt (followed by the interrupt number)

- PWM TnA/B is the Pulse Width Modulation (`analogWrite`) output on timer n

- MISO, MOSI, SCK, and SS are SPI control signals

- SDA and SCL are I2C control signals

*Analog and digital pin assignments common to popular Arduino boards*

| | Arduino 168/328 | | | Arduino Mega (pins 0–19) | | |
|---|---|---|---|---|---|---|
| Digital pin | Port | Analog pin | Usage | Port | Analog pin | Usage |
| 0 | PD 0 | | USART RX | PE 0 | | USART0 RX, Pin Int 8 |
| 1 | PD 1 | | USART TX | PE 1 | | USART0 TX |
| 2 | PD 2 | | Ext Int 0 | PE 4 | | **PWM** T3B, INT4 |
| 3 | PD 3 | | **PWM** T2B, Ext Int 1 | PE 5 | | **PWM** T3C, INT5 |

| | Arduino 168/328 | | | Arduino Mega (pins 0–19) | | |
|---|---|---|---|---|---|---|
| **Digital pin** | **Port** | **Analog pin** | **Usage** | **Port** | **Analog pin** | **Usage** |
| 4 | PD 4 | | | PG 5 | | **PWM** T0B |
| 5 | PD 5 | | **PWM** T0B | PE 3 | | **PWM** T3A |
| 6 | PD 6 | | **PWM** T0A | PH 3 | | **PWM** T4A |
| 7 | PD 7 | | | PH 4 | | **PWM** T4B |
| 8 | PB 0 | | Input capture | PH 5 | | **PWM** T4C |
| 9 | PB 1 | | **PWM** T1A | PH 6 | | **PWM** T2B |
| 10 | PB 2 | | **PWM** T1B, SS | PB 4 | | **PWM** T2A, Pin Int 4 |
| 11 | PB 3 | | **PWM** T2A, MOSI | PB 5 | | **PWM** T1A, Pin Int 5 |
| 12 | PB 4 | | SPI MISO | PB 6 | | **PWM** T1B, Pin Int 6 |
| 13 | PB 5 | | SPI SCK | PB 7 | | **PWM** T0A, Pin Int 7 |
| 14 | PC 0 | 0 | | PJ 1 | | USART3 TX, Pin Int 10 |
| 15 | PC 1 | 1 | | PJ 0 | | USART3 RX, Pin Int 9 |
| 16 | PC 2 | 2 | | PH 1 | | USART2 TX |
| 17 | PC 3 | 3 | | PH 0 | | USART2 RX |
| 18 | PC 4 | 4 | I2C SDA | PD 3 | | USART1 TX, Ext Int 3 |
| 19 | PC 5 | 5 | I2C SCL | PD 2 | | USART1 RX, Ext Int 2 |

## Assignments for additional Mega pins

| Arduino Mega (pins 20–44) | | | Arduino Mega (pins 45–69) | | | |
| --- | --- | --- | --- | --- | --- | --- |
| Digital pin | Port | Usage | Digital pin | Port | Analog pin | Usage |
| 20 | PD 1 | I2C SDA, Ext Int 1 | 45 | PL 4 | | **PWM** 5B |
| 21 | PD 0 | I2C SCL, Ext Int 0 | 46 | PL 3 | | **PWM** 5A |
| 22 | PA 0 | Ext Memory addr bit 0 | 47 | PL 2 | | T5 external counter |
| 23 | PA 1 | Ext Memory bit 1 | 48 | PL 1 | | ICP T5 |
| 24 | PA 2 | Ext Memory bit 2 | 49 | PL 0 | | ICP T4 |
| 25 | PA 3 | Ext Memory bit 3 | 50 | PB 3 | | SPI MISO |
| 26 | PA 4 | Ext Memory bit 4 | 51 | PB 2 | | SPI MOSI |
| 27 | PA 5 | Ext Memory bit 5 | 52 | PB 1 | | SPI SCK |
| 28 | PA 6 | Ext Memory bit 6 | 53 | PB 0 | | SPI SS |
| 29 | PA 7 | Ext Memory bit 7 | 54 | PF 0 | 0 | |
| 30 | PC 7 | Ext Memory bit 15 | 55 | PF 1 | 1 | |
| 31 | PC 6 | Ext Memory bit 14 | 56 | PF 2 | 2 | |
| 32 | PC 5 | Ext Memory bit 13 | 57 | PF 3 | 3 | |
| 33 | PC 4 | Ext Memory bit 12 | 58 | PF 4 | 4 | |
| 34 | PC 3 | Ext Memory bit 11 | 59 | PF 5 | 5 | |
| 35 | PC 2 | Ext Memory bit 10 | 60 | PF 6 | 6 | |
| 36 | PC 1 | Ext Memory bit 9 | 61 | PF 7 | 7 | |
| 37 | PC 0 | Ext Memory bit 8 | 62 | PK 0 | 8 | Pin Int 16 |

| Arduino Mega (pins 20–44) | | | Arduino Mega (pins 45–69) | | | |
|---|---|---|---|---|---|---|
| Digital pin | Port | Usage | Digital pin | Port | Analog pin | Usage |
| 38 | PD 7 | | 63 | PK 1 | 9 | Pin int 17 |
| 39 | PG 2 | ALE Ext Mem | 64 | PK 2 | 10 | Pin Int 18 |
| 40 | PG 1 | RD Ext Mem | 65 | PK 3 | 11 | Pin Int 19 |
| 41 | PG 0 | Wr Ext Mem | 66 | PK 4 | 12 | Pin Int 20 |
| 42 | PL 7 | | 67 | PK 5 | 13 | Pin Int 21 |
| 43 | PL 6 | | 68 | PK 6 | 14 | Pin Int 22 |
| 44 | PL 5 | PWM 5C | 69 | PK 7 | 15 | Pin Int 23 |

Table F-3 lists timer modes showing the pins used with popular Arduino chips.

*Timer modes*

| Timer | Arduino 168/328 | Mega |
|---|---|---|
| Timer 0 mode (8-bit) | Fast PWM | Fast PWM |
| Timer0A `analogWrite` pin | Pin 6 | Pin 13 |
| Timer0B `analogWrite` pin | Pin 5 | Pin 4 |
| Timer 1 (16-bit) | Phase correct PWM | Phase correct PWM |
| Timer1A `analogWrite` pin | Pin 9 | Pin 11 |
| Timer1B `analogWrite` pin | Pin 10 | Pin 12 |
| Timer 2 (8-bit) | Phase correct PWM | Phase correct PWM |

| Timer | Arduino 168/328 | Mega |
| --- | --- | --- |
| Timer2A analogWrite pin | Pin 11 | Pin 10 |
| Timer2B analogWrite pin | Pin 3 | Pin 9 |
| Timer 3 (16-bit) | N/A | Phase correct PWM |
| Timer3A analogWrite pin | | Pin 5 |
| Timer3B analogWrite pin | | Pin 2 |
| Timer3C analogWrite pin | | Pin 3 |
| Timer 4 (16-bit) | N/A | Phase correct PWM |
| Timer4A analogWrite pin | | Pin 6 |
| Timer4B analogWrite pin | | Pin 7 |
| Timer4C analogWrite pin | | Pin 8 |
| Timer 5 (16-bit) | N/A | Phase correct PWM |
| Timer5A analogWrite pin | | Pin 46 |
| Timer5B analogWrite pin | | Pin 45 |
| Timer5C analogWrite pin | | Pin 44 |

Full details of these Arduino controller chips can be found in their datasheets:

- The datasheet for Uno-compatible boards (ATmega328)
- The mega (ATmega2560) datasheet

# ASCII and Extended Character Sets

ASCII stands for American Standard Code for Information Interchange. It is the most common way of representing letters and numbers on a computer. Each character is represented by a number—for example, the letter *A* has the numeric value 65, and the letter *a* has the numeric value 97 (lowercase letters have a value that is 32 greater than their uppercase versions).

Values below 32 are called *control codes*—they were defined as nonprinting characters to control early computer terminal devices. The most common control codes for Arduino applications are listed in Table G-1.

*Common ASCII control codes*

| Decimal | Hex | Escape code | Description |
|---------|-----|-------------|-------------|
| 0 | 0x0 | `'\0'` | Null character (used to terminate a C string) |
| 9 | 0x9 | `'\t'` | Tab |
| 10 | 0xA | `'\n'` | New line |
| 13 | 0xD | `'\r'` | Carriage return |
| 27 | 0x1B | | Escape |

Table G-2 shows the decimal and hexadecimal values of the printable ASCII characters.

*ASCII table*

| | Dec | Hex | | Dec | Hex | | Dec | Hex |
|---|-----|-----|---|-----|-----|---|-----|-----|
| Space | 32 | 20 | @ | 64 | 40 | ` | 96 | 60 |

|   | Dec | Hex |   | Dec | Hex |   | Dec | Hex |
|---|-----|-----|---|-----|-----|---|-----|-----|
| ! | 33 | 21 | A | 65 | 41 | a | 97 | 61 |
| " | 34 | 22 | B | 66 | 42 | b | 98 | 62 |
| # | 35 | 23 | C | 67 | 43 | c | 99 | 63 |
| $ | 36 | 24 | D | 68 | 44 | d | 100 | 64 |
| % | 37 | 25 | E | 69 | 45 | e | 101 | 65 |
| & | 38 | 26 | F | 70 | 46 | f | 102 | 66 |
| ' | 39 | 27 | G | 71 | 47 | g | 103 | 67 |
| ( | 40 | 28 | H | 72 | 48 | h | 104 | 68 |
| ) | 41 | 29 | I | 73 | 49 | i | 105 | 69 |
| * | 42 | 2A | J | 74 | 4A | j | 106 | 6A |
| + | 43 | 2B | K | 75 | 4B | k | 107 | 6B |
| , | 44 | 2C | L | 76 | 4C | l | 108 | 6C |
| - | 45 | 2D | M | 77 | 4D | m | 109 | 6D |
| . | 46 | 2E | N | 78 | 4E | n | 110 | 6E |
| / | 47 | 2F | O | 79 | 4F | o | 111 | 6F |
| 0 | 48 | 30 | P | 80 | 50 | p | 112 | 70 |
| 1 | 49 | 31 | Q | 81 | 51 | q | 113 | 71 |
| 2 | 50 | 32 | R | 82 | 52 | r | 114 | 72 |
| 3 | 51 | 33 | S | 83 | 53 | s | 115 | 73 |
| 4 | 52 | 34 | T | 84 | 54 | t | 116 | 74 |
| 5 | 53 | 35 | U | 85 | 55 | u | 117 | 75 |
| 6 | 54 | 36 | V | 86 | 56 | v | 118 | 76 |
| 7 | 55 | 37 | W | 87 | 57 | w | 119 | 77 |

|     | Dec | Hex |     | Dec | Hex |     | Dec | Hex |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8   | 56  | 38  | X   | 88  | 58  | x   | 120 | 78  |
| 9   | 57  | 39  | Y   | 89  | 59  | y   | 121 | 79  |
| :   | 58  | 3A  | Z   | 90  | 5A  | z   | 122 | 7A  |
| ;   | 59  | 3B  | [   | 91  | 5B  | {   | 123 | 7B  |
| <   | 60  | 3C  | \   | 92  | 5C  | \|  | 124 | 7C  |
| =   | 61  | 3D  | ]   | 93  | 5D  | }   | 125 | 7D  |
| >   | 62  | 3E  | ^   | 94  | 5E  | ~   | 126 | 7E  |
| ?   | 63  | 3F  | _   | 95  | 5F  |     |     |     |

Characters above 128 are non-English characters or special symbols and are displayed in the Serial Monitor using the characters shown in Table G-3.

*Extended characters*

|     | Dec | Hex |       | Dec | Hex |     | Dec | Hex |     | Dec | Hex |
| --- | --- | --- | ----- | --- | --- | --- | --- | --- | --- | --- | --- |
| €   | 128 | 80  | Space | 160 | A0  | À   | 192 | C0  | à   | 224 | E0  |
|     | 129 | 81  | ¡     | 161 | A1  | Á   | 193 | C1  | á   | 225 | E1  |
| ,   | 130 | 82  | ¢     | 162 | A2  | Â   | 194 | C2  | â   | 226 | E2  |
| ƒ   | 131 | 83  | £     | 163 | A3  | Ã   | 195 | C3  | ã   | 227 | E3  |
| „   | 132 | 84  | ¤     | 164 | A4  | Ä   | 196 | C4  | ä   | 228 | E4  |
| …   | 133 | 85  | ¥     | 165 | A5  | Å   | 197 | C5  | å   | 229 | E5  |
| †   | 134 | 86  | ¦     | 166 | A6  | Æ   | 198 | C6  | æ   | 230 | E6  |
| ‡   | 135 | 87  | §     | 167 | A7  | Ç   | 199 | C7  | ç   | 231 | E7  |
| ˆ   | 136 | 88  | ¨     | 168 | A8  | È   | 200 | C8  | è   | 232 | E8  |
| ‰   | 137 | 89  | ©     | 169 | A9  | É   | 201 | C9  | é   | 233 | E9  |
| Š   | 138 | 8A  | ª     | 170 | AA  | Ê   | 202 | CA  | ê   | 234 | EA  |

| | Dec | Hex | | Dec | Hex | | Dec | Hex | | Dec | Hex |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ‹ | 139 | 8B | « | 171 | AB | Ë | 203 | CB | ë | 235 | EB |
| Œ | 140 | 8C | ¬ | 172 | AC | Ì | 204 | CC | ì | 236 | EC |
| | 141 | 8D | | 173 | AD | Í | 205 | CD | í | 237 | ED |
| Ž | 142 | 8E | ® | 174 | AE | Î | 206 | CE | î | 238 | EE |
| | 143 | 8F | ¯ | 175 | AF | Ï | 207 | CF | ï | 239 | EF |
| | 144 | 90 | ° | 176 | B0 | Ð | 208 | D0 | ð | 240 | F0 |
| ' | 145 | 91 | ± | 177 | B1 | Ñ | 209 | D1 | ñ | 241 | F1 |
| ' | 146 | 92 | ² | 178 | B2 | Ò | 210 | D2 | ò | 242 | F2 |
| " | 147 | 93 | ³ | 179 | B3 | Ó | 211 | D3 | ó | 243 | F3 |
| " | 148 | 94 | ´ | 180 | B4 | Ô | 212 | D4 | ô | 244 | F4 |
| • | 149 | 95 | µ | 181 | B5 | Õ | 213 | D5 | õ | 245 | F5 |
| – | 150 | 96 | ¶ | 182 | B6 | Ö | 214 | D6 | ö | 246 | F6 |
| — | 151 | 97 | · | 183 | B7 | × | 215 | D7 | ÷ | 247 | F7 |
| ˜ | 152 | 98 | ¸ | 184 | B8 | Ø | 216 | D8 | ø | 248 | F8 |
| ™ | 153 | 99 | ¹ | 185 | B9 | Ù | 217 | D9 | ù | 249 | F9 |
| š | 154 | 9A | º | 186 | BA | Ú | 218 | DA | ú | 250 | FA |
| › | 155 | 9B | » | 187 | BB | Û | 219 | DB | û | 251 | FB |
| œ | 156 | 9C | ¼ | 188 | BC | Ü | 220 | DC | ü | 252 | FC |
| | 157 | 9D | ½ | 189 | BD | Ý | 221 | DD | ý | 253 | FD |
| ž | 158 | 9E | ¾ | 190 | BE | Þ | 222 | DE | þ | 254 | FE |
| Ÿ | 159 | 9F | ¿ | 191 | BF | ß | 223 | DF | ÿ | 255 | FF |

You can view the entire character set in the Serial Monitor using this sketch:

```
/*
 * display characters from 1 to 255
 */

void setup()
{
  Serial.begin(9600);
  while(!Serial); // For Leonardo and 32-bit boards
  for(int i=1; i < 256; i++)
  {
    Serial.write(i);
    Serial.print(", dec: ");
    Serial.print(i, DEC);
    Serial.print(", hex: ");
    Serial.println(i, HEX);
  }
}

void loop()
{
}
```

Note that some devices, such as LCD displays (see Chapter 11), may use different symbols for the characters above 128, so check the datasheet for your device to see the actual characters supported.

# Index

## B

## C

## F

## G

## N

## S

## T

## V

## W

XML-formatted data,

## Z

## About the Authors

**Michael Margolis** is a technologist in the field of real-time computing with expertise in developing and delivering hardware and software for interacting with the environment. He has more than 30 years of experience at senior levels with Sony, Microsoft, and Lucent/Bell Labs. He has written libraries and core software that are part of the official Arduino 1.0 distribution.

**Brian Jepson** is a content manager at LinkedIn Learning, where he manages design and engineering courses. He is also the co-organizer of Providence Geeks, a founding member of the National Maker Faire planning and production team, and coproducer of the Providence Mini Maker Faire. He shares and spreads knowledge of electronics and digital fabrication through hands-on events and workshops, working closely with AS220, a nonprofit community arts center, and with the Rhode Island Computer Museum (both Rhode Island–based nonprofits).

**Nick Weldin** works at the Rix Centre based at the University of East London, looking into technology that may help people with learning difficulties to get involved in what is happening online and on the computer in front of them. He is also senior technologist for Tinker It! working on various technology projects, often related to Arduino, an open source electronics prototyping platform based on flexible, easy-to-use hardware and software. Other projects include work with Paddington Arts, Oily Cart Theatre Company, and Deafinitely Theatre.

## Colophon

The animal on the cover of *Arduino Cookbook,* Third Edition, is a toy rabbit. Mechanical toys like this one move by means of springs, gears, pulleys, levers, or other simple machines, powered by mechanical energy. Such toys have a long history, with ancient examples known from Greece, China, and the Arab world.

Mechanical toy making flourished in early modern Europe. In the late 1400s, German inventor Karel Grod demonstrated flying wind-up toys. Prominent scientists of the day, including Leonardo da Vinci, René Descartes, and Galileo Galilei, were noted for their work on mechanical toys. Da Vinci's famed mechanical lion, built in 1509 for Louis XII, walked up to the king and tore open its chest to reveal a fleur-de-lis.

The art of mechanical toy making is considered to have reached its pinnacle in the late eighteenth century, with the famed "automata" of the Swiss watchmaker Pierre Jaquet-Droz and his son Henri-Louis. Their human figures performed such lifelike actions as dipping a pen in an inkwell, writing full sentences, drawing sketches, and blowing eraser dust from the page. In the nineteenth century, European and American companies turned out popular clockwork toys that remain collectible today.

Because these original toys, which had complicated works and elaborate decorations, were costly and time-consuming to make, they were reserved for the amusement of royalty or the entertainment of adults. Only since the late nineteenth century, with the appearance of mass production and cheap materials (tin, and later, plastic), have mechanical toys been considered playthings for children. These inexpensive moving novelties were popular for about a century, until battery-operated toys superseded them.

The cover illustration is by Karen Montgomery, based on a black and white engraving from the Dover Pictorial Archive. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.