



TECHNOLOGY IN ACTION™

Beginning Arduino Programming

**Writing Code for the Most Popular
Microcontroller Board in the World**



Brian Evans

For your convenience Apress has placed some of the front matter material after the index. Please use the [Bookmarks](#) and [Contents at a Glance](#) links to access them.



Contents at a Glance

About the Author	xv
About the Technical Reviewer	xvi
Acknowledgments	xvii
Introduction	xviii
■ Chapter 1: Getting Started	1
■ Chapter 2: Sketching in Code	17
■ Chapter 3: Working with Variables	33
■ Chapter 4: Making Decisions	47
■ Chapter 5: Digital Ins and Outs	61
■ Chapter 6: Analog In, Analog Out	79
■ Chapter 7: Advanced Functions	95
■ Chapter 8: Arrays and Memory	121
■ Chapter 9: Hardware Libraries	143
■ Chapter 10: Serial and I2C	175
■ Chapter 11: Continuing On	201
■ Chapter 12: Beginning Electronics	221
■ Appendix: Resources	239
Index	245

Introduction

This book will help you to develop working source code for the Arduino microcontroller. In these pages, we will primarily concern ourselves with the software aspect of physical computing—designing code to work with physical objects that exhibit behavior or interactivity through software. Starting with the basic context of the Arduino platform to getting up and running with our first code, we will discuss the structure and syntax of Arduino’s C-based programming language, looking at variables, control structures, arrays, and memory. This book will then go into many of the functions unique to Arduino development for controlling digital and analog input and output, timing, randomness, writing functions, and using many of the Arduino libraries for working with different kinds of hardware and communication protocols.

Arduino, like Processing before it, adopted the idea of a code sketchbook. We will carry on this metaphor as we talk about the process of sketching in code as an intuitive method for quickly testing out new ideas in code. Most of this book is written around this idea of developing programming skills through sketching. We will also provide some suggestions for new projects and hardware, new languages to try out, and ways to contribute back to the community. This book intentionally does not dwell too long on electronics theory, circuit design, hacking, or other specifically hardware-based practices, although we’ll revisit the hardware side of things in our last chapter to provide a small foundation for physical computing.

This book in many ways picks up where the *Arduino Programming Notebook* left off, with even more in-depth discussions about the Arduino environment; simple, no-frills code samples; and clear, easy-to-read schematics and illustrations. The *Notebook*, a little PDF booklet, was my first experience writing about the Arduino and was never meant to be more than a brief guide for my students when I first introduced a class of 15 college art and design majors to the Arduino in 2007. Best laid plans and all, this little booklet has now been translated into Spanish, Russian, and Dutch (that I know of), is hosted in so many different places that it is impossible to keep track of, and it’s been used in workshops and classes around the world. I haven’t updated the *Notebook* over the last few years, and in all honesty I am not entirely sure what to do with it now, so hopefully this new book will fill a void and find a similar, widespread adoption that the little booklet has enjoyed all these years.

Who This Book is For

This book is written for the primary audience of the Arduino platform: artists, designers, students, tinkerers, and the makers of things. While you might have some programming experience that you want to bring to the Arduino platform, we will assume no prior knowledge of writing code. With that said, a healthy familiarity of the computer is helpful, as is the willingness and inquisitive curiosity to look beyond this book for certain answers.

The majority of Arduino users just want to get things done and often don’t care about the little details—they just want their projects to work. I understand this, as I am one of those people. I first discovered programmable microcontrollers when I was an art student, and at the time, art school was not generally the most conducive environment for learning how to write code and wire up motors—at

least it wasn't before the Arduino came along. Likewise, I was never one for a love of mathematics, which thankfully is not a prerequisite to deeply enjoy the process of writing code.

Reading This Book

Our process in each chapter will be to focus on some fundamental projects that build on the primary concepts presented in that chapter. For each project, we will begin with a project description and discuss the specific hardware needed for that project. We will also provide diagrams and illustrations for making these simple circuits and interfacing them to the Arduino board. As you read through each project, you should take notes and write in the margins—we won't be offended. Experiment, try new things, and see what happens.

The projects demonstrated in this book are meant to be prototypes, or fundamental proof-of-concept designs for a new device. We will adhere to a degree of minimalism, keeping to simple and easily obtainable hardware that supports the development of sophisticated written code. Once you have built the prototype, it can be incorporated into a final project later. We won't actually be doing that here so that we can focus on actually writing and developing code. Our examples will borrow and build on each other throughout the book, revisiting past examples when we need to as our understanding of writing code develops.

The intent with our code samples is to write compartmentalized or modular code wherever possible to allow for easy adaptability and future development. We will spend a lot of time developing our coding skills so that when it comes time to develop a new project independently, you will know where to begin. The sketches are meant to be fluid—you are encouraged to hack them—changing values, timing, pin assignments, ranges, and so on—until it no longer works. Then try to fix it. We will stick to a particular style of writing code in our samples, although we urge you to develop your own writing style that reflects the way you think and the way you want to see your code.

Wiring up the circuits for our projects is as simple and straightforward as possible, with little to no understanding of electronics necessary. As a way to reconnect our discussions of programming to the physical electronics used throughout the book, Chapter 12 will provide a brief review of some basic electronics, including how circuits work, reading schematics, and an introduction to soldering. If you find that you are struggling with hooking up the projects in the earlier chapters, you might want to jump to Chapter 12 for a refresher. Otherwise, this chapter will serve as a good summary that could help answer some questions you might not even know you had. While this might at first seem a little backwards, it has worked pretty well in my classes over the last few years.

Arduino 1.0

At the time of this writing, the Arduino developers are hard at work on a more stable, more efficient, and generally improved version of the Arduino software called Arduino 1.0. The final release version of Arduino 1.0 should be available right about the same time that this book is published. This is important because in the process of making things better, some things had to be broken. This means that some older code written under the alpha release of the Arduino software will no longer work on Arduino 1.0.

Conversely, the code in this book and images of the Arduino development environment have all been prepared using a beta release of Arduino 1.0 (<http://code.google.com/p/arduino/wiki/Arduino1>), so images of the Arduino software may appear different from the final version, some of the code in this book may not work on older versions of the software, and still other features of 1.0 were not yet fully implemented—so I couldn't write about them. There may also be other growing pains with this upgrade that we are not fully aware of at this time, so if an unusual problem crops up, then you might want to blame 1.0 and start there to figure out what's wrong.

Conventions

We will use several conventions in this book, including fixed width fonts in line to denote specific code examples, **bold text** highlights new concepts or definitions, and anything with a parenthesis after it—as in `setup()` or `loop()`—will denote something called a function. Anytime there is a block of fixed-width font separated from the main text, it is a multiline code example, as in the following:

```
// this is a multiline  
// code example
```

■ **Note** Occasionally there will be areas separated as this sentence is, as a side note, general tip, or caution about something you will want to pay careful attention to.

Downloading the Code

The source code for this book is available from the Apress web site (www.apress.com) in the Source Code / Downloads section. If you are publishing examples that use code from this book, using attribution that includes the title, author, publisher, year, and ISBN is generally a nice thing to do.

Getting Started

One of the keys to the Arduino's success is the minimal amount of time that it takes for a complete novice to move from opening the little box containing the Arduino interface board to having their first source code, also known as a sketch, up and running on that board.

The Arduino software development environment is free to download and use with no lengthy registration procedures or end-user agreements, and there is little to no setup to get the board running with your computer, regardless of the platform, working equally well on Mac, Linux, and Windows-based PCs. The Arduino web site at www.arduino.cc provides easy-to-follow “Getting Started” tutorials and whenever you get stuck there is always the active, vocal, and generally helpful Arduino community that is willing to share its knowledge. This low barrier to programming embedded electronics means it's possible to make lights blink in ten minutes flat with little to no prior experience.

This chapter will walk you through the history behind the Arduino platform, with an eye towards what makes the Arduino such a success story. We will look at some of the things that can be done with the platform and get you started on the right foot with a firm understanding of its possibilities. This includes a brief walk-through of that first ten-minute experience so that we can quickly move into some of our first projects in the next chapter. But first, let's start with a little background.

Arduino is for Makers

Arduino is a flexible programmable hardware platform designed for artists, designers, tinkerers, and the makers of things. Arduino's little, blue circuit board, mythically taking its name from a local pub in Italy, has in a very short time motivated a new generation of DIYers of all ages to make all manner of wild projects found anywhere from the hallowed grounds of our universities to the scorching desert sands of a particularly infamous yearly arts festival and just about everywhere in between. Usually these Arduino-based projects require little to no programming skills or knowledge of electronics theory, and more often than not, this handiness is simply picked up along the way.

Central to the Arduino interface board, shown in Figure 1-1, is an onboard microcontroller—think of it as a little computer on a chip.

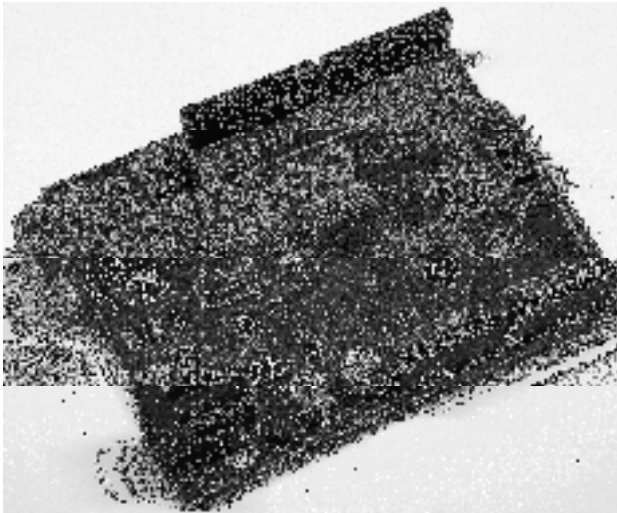


Figure 1-1. *The Arduino Uno interface board, 2011*

This microcontroller comes from a company called Atmel and the chip is known as an AVR. It is slow in modern terms, running at only 16Mhz with an 8-bit core, and has a very limited amount of available memory, with 32 kilobytes of storage and 2 kilobytes of random access memory. The interface board is known for its rather quirky design—just ask the die-hards about standardized pin spacing—but it also epitomizes the minimalist mantra of only making things as complicated as they absolutely need to be. Its design is not entirely new or revolutionary, beginning with a curious merger of two, off-the-shelf reference designs, one for an inexpensive microcontroller and the other for a USB-to-serial converter, with a handful of other useful components all wrapped up in a single board. Its predecessors include the venerable BASIC Stamp, which got its start as early as 1992, as well as the OOPic, Basic ATOM, BASIC-X24, and the PICAXE.

Where all of these precursors are generally closed proprietary products and often require a single computer platform to use, the Arduino development environment is free for all to use and will run on just about any kind of computer that supports Java. The actual hardware board costs a mere USD \$30 or EUR €22 and needs nothing more complex than a USB cable to get up and running. This affordable price, nearly half that of its closest competitor, and the board's durable design have led to numerous Arduino-like boards being stitched into embroidery (see Figure 1-2), embedded in pumpkins to be launched through the air by trebuchets, and even sent into outer space in weather balloons.



Figure 1-2. LilyPad Arduino embroidery, courtesy Becky Stern, sternlab.org

This at least tells us a little bit about what kind of person the Arduino was originally designed for and about the hardware used in its design, but these things alone do not begin to account for the huge degree of success enjoyed by the Arduino as a whole. To get a sense for this popularity, we need to look at the larger Arduino ecosystem and how some fairly divergent parts came together to create a movement.

The Arduino Ecosystem

The Arduino is not just one simple thing making it a little hard to define. It is a microcontroller platform, an open-source design that encourages modification and reuse, a community that has embraced and grown up around the Arduino, and a new crop of projects and devices that can trace their lineage to the Arduino and have in return contributed back to the development of various aspects of the entire Arduino ecosystem.

The Arduino Platform

The Arduino ecosystem begins with the Arduino platform, itself several layers of hardware and software working together to create a cohesive whole. We can start with the hardware interface board—that little, blue circuit board that you build into your projects. It has a fairly standard onboard microcontroller that can interact with the world around it by using its programmable inputs and outputs, as well as a USB

port and controller for easily communicating with your computer. This USB connectivity and friendly sockets for hookup wires to easily plug in to, contribute to the high level of usability in the interface board design.

Equally important to this ecosystem is the Arduino development environment, a program based on the Processing development environment (<http://processing.org>) that you use to write, edit, compile, and upload your Arduino source code to the interface board. The Arduino team made the general assumption that people don't really care about the myriad of technical specifics involved with microcontroller architecture—they just want it to do something cool. With that in mind, the Arduino development environment wraps up some of the more techie parts of programming AVR microcontrollers into a nice, simple library of Arduino-specific commands that are easier to use and easier to understand, and built right into every sketch written for the Arduino. This development environment is so versatile, that an Arduino interface board is not even needed to use it. Instead, we can use the very same AVR microcontroller as is built onto the interface board, but in an entirely different device—such as the example shown in Figure 1-3 of a microcontroller on a breadboard that has been programmed with an Arduino sketch.

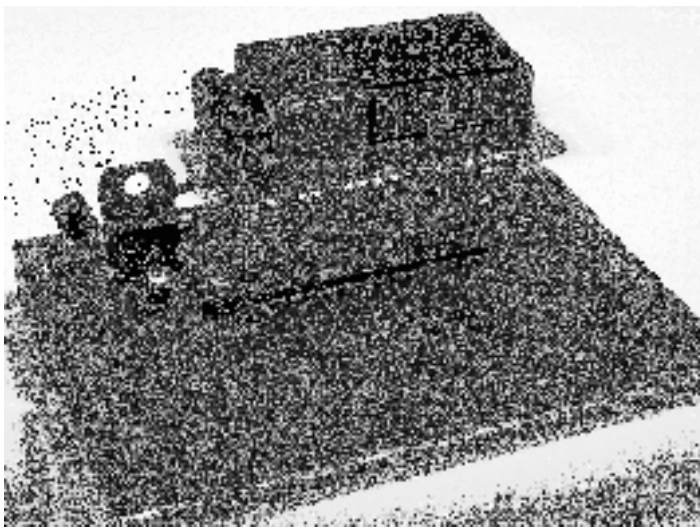


Figure 1-3. *Arduino compatible AVR microcontroller on breadboard*

Finally, we are brought to the Arduino bootloader, a little chunk of code residing on the microcontroller that allows it to be easily programmed with a simple serial connection rather than cranky, external hardware. As long as the bootloader has been programmed onto the microcontroller beforehand, we can use that chip in whatever device we choose, such as the breadboard in Figure 1-3, and still write code for it using the rest of the Arduino platform. It helps if we use one of the microcontrollers common to the Arduino, but this is not always a prerequisite, with many other Atmel microcontrollers working nearly as well with little or no modification.

Open-Source Hardware

The Arduino platform is itself pretty useful for microcontroller projects, but that alone is not enough to propel the popularity and widespread adoption of the platform. Instead of closing the design of the interface board and development environment, the entire Arduino project is deeply entrenched in the emerging practice of open-source hardware. Unlike open-source software, of which Linux is usually the often-cited example, open-source hardware seeks collaboration where physical objects are the outcome. It engages a distributed model of hardware development with contributors generally residing in different parts of the world. Rather than closed systems, open source projects allow an individual freedom to access the source files of a design, make improvements, and redistribute these improvements to a larger community.

The Arduino ecosystem fundamentally embodies this aspiration for openness in design, architecture, collaboration, and philosophy. You can see it for yourself as all of the design files, schematics, and software are freely available to download, use, modify, remake, and even resell. What started as a seemingly serendipitous decision to open the Arduino design and software to the greater community, spurred by the closing of the design school where the Arduino team was first formed, has lead to an entirely new movement in design. The practice of contributors having the liberty to use these designs freely (free as in speech) and with no obligation to buy anything (free as in beer) helps make the Arduino as endearing as a collection of silicon and copper can be. Not to mention that this creative feedback loop ensures that every inspired innovation derived from the Arduino platform is met with ever more imaginative uses for even more new things.

Community

Maybe even more important than the hardware platform itself, the Arduino community is most likely the single greatest cog in the Arduino machine. This community of makers has contributed to the Arduino ecosystem by developing code and libraries, designing new hardware, teaching workshops and classes, and sharing what they've made. The Arduino is now being taught in high schools, colleges, and universities everywhere and "Arduino Night" is a regular ritual at any of a number of hacker spaces around the world. People coming together around the Arduino, as shown in the image of a soldering workshop at Maker Faire in Figure 1-4, contributes to this global community that has made the Arduino such a success.

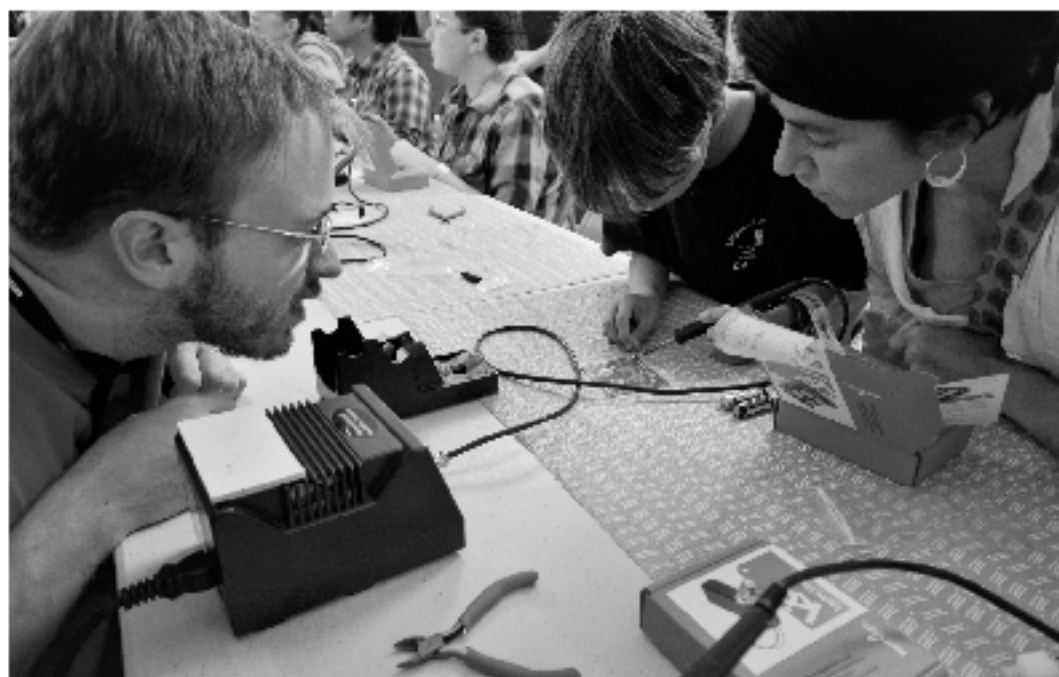


Figure 1-1. Soldering Workshop. Maker Faire, San Mateo, CA, 2011, courtesy SparkFun Electronics

Community engagement for a project like the Arduino is essential for the success or failure of the project. Fortuitous timing also helped in that the Arduino came on the scene at the same time as the return of the Maker and DIY movements also began again. This ever-increasing new crop of makers means that you never have to look hard to find willing co-conspirators to help on anything open source: whether it's through pockets of helpful individuals in the internet or your neighborhood hacker space, help is never that far away.

Arduinoland

This community-driven research and development seems to have a magical effect on any project to come out of the wake of the Arduino, and this proverbial kingdom of mystical wonder is something I'm going to call Arduinoland. Maybe it's because the hardware design works, is open, and is hackable, or maybe it's because of a communal desire to build upon each other's work, but whatever it is, in Arduinoland, so daring, so mercurial, difficult to nearly impossible to pull off in a garage just ten years ago, seem to happen every day. Anything from interactive electronic textiles, autonomous flying aircraft, networked beer keg refrigerators, photographic missions to near space, and immersive architectural installations have all sprouted from this open Arduino ecosystem.

Take the MakerBot Cup-size CNC for example, a sub-\$1,000, kit-based, plastic-extruding, 3D printer shown in Figure 1-5. Merging an electronics system to interpret code sent from an attached computer to drive the positioning of the printhead and the temperature of the extruder, the MakerBot team built off the Arduino hardware platform and the development environment to create an open platform for building plastic objects.

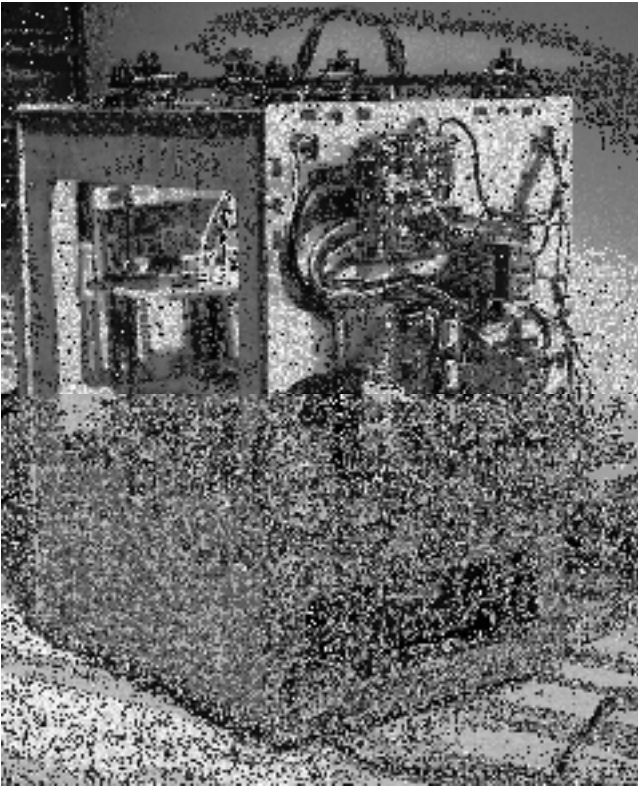


Figure 1-5. *MakerBot CupCake 3D printer with Arduino-compatible electronics*

Or for all of those brown thumbs out there that can never seem to water their house plants when it's most needed, a team of New York University students developed a tweeting plant moisture meter called Botanicalls, shown in Figure 1-6. Based on the Arduino reference design, combined with an off-the-shelf Ethernet module and a simple moisture probe all wrapped up in a leafy-green circuit board, the Botanicalls device will post a tweet and blink an LED whenever the soil moisture of the plant falls below a certain threshold, letting its owner know that they should get out their watering can.

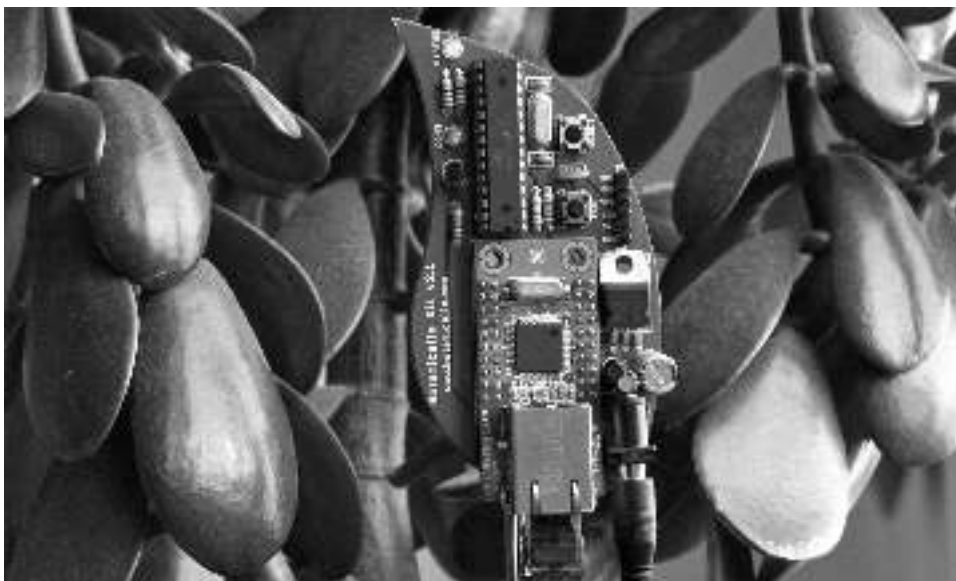


Figure 1-6. *Botanicalls Arduino-based tweeting plant water meter, courtesy Botanicalls*

These projects and many others have all benefited from the rapid development and reliable longevity brought to you by Arduinoland and the hard work put in by the Arduino team and community at large to create an open architecture that enables continued and accelerated growth and creativity.

Arduino is C... Mostly

In addition to the various aspects of the diverse Arduino ecosystem, we have the programming language of the Arduino platform, which is the central focus of this book. The core language used in the Arduino development environment is the C computer programming language first developed at the research institute of Bell Laboratories in the early 1970s for use with the UNIX operating system. C uses a procedural language syntax that needs to be processed by a compiler to map human-readable code to machine instructions. The long-standing popularity of C lends the Arduino some of its heritage, but the code that we are writing in this book is only mostly C.

Because there are aspects of the C language that look like it was written by dyslexic aliens, and with the language sometimes accused of being overly cryptic and difficult for beginners to pick up, the Arduino team has developed the standard Arduino library that provides a simple and targeted set of functions that make programming the Arduino interface board about as easy as it can get. Now, these libraries are themselves actually C++, itself a subset of the original C language, but we really don't need to go there.

What's important is that most of the code that we will write for the Arduino, including its syntax, structure, operators, control statements, and functions, remain fundamentally and functionally the same as C. What will be unique to the Arduino, however, are all sorts of functions that you will come to know and love, including `pinMode()`, `digitalWrite()`, and `delay()` that are specific to the standard Arduino library. For the purposes of this book, this basic framework of C combined with the additional Arduino library that is automatically a part of every sketch that we write, is what we will refer to as

Arduino C. To illustrate this point, Listings 1-1 and 1-2 provide two examples of the same source code to blink the onboard LED connected to digital pin 13.

Listing 1-1. *Blink LED with avr-libc*

```
#include <avr/io.h>
#include <util/delay.h>

int main(void) {
    while (1) {
        PORTB = 0x20;
        _delay_ms(1000);
        PORTB = 0x00;
        _delay_ms(1000);
    }
    return 1;
}
```

Listing 1-2. *Blink LED with Arduino*

```
void setup() {
    pinMode(13, OUTPUT);
}

void loop() {
    digitalWrite(13, HIGH);
    delay(1000);
    digitalWrite(13, LOW);
    delay(1000);
}
```

These two different listings show two functionally identical sketches, one written with the Arduino library and one written without. The really nifty thing here is that, if you want to geek out, the Arduino development environment is fully compatible and extensible using C/C++ code written using the `avr-libc` library, a subset of the standard C library, and the GCC compiler, both written for Atmel's standard 8-bit AVR microcontrollers. Listing 1-1 is written with `avr-libc` while Listing 1-2 is written using the Arduino library. They both are compatible with the Arduino development environment and can be uploaded the same way to the Arduino board. The first example also consumes a fifth of the amount of memory as the second example, coming in at 210 bytes as opposed to 1010 bytes.

What the Arduino example lacks in memory efficiency, however, it more than makes up for in usability and integration with the Arduino interface board. For example, referring to the digital pin that our LED is connected to as pin number 13 is generally easier for most people than the hexadecimal address 0x20 on PORTB. This simplicity is one of the benefits to writing code using Arduino C. That is not to say that one is better than the other, but simply that this scalability and flexibility is an often-overlooked benefit of learning on the Arduino platform because it allows budding code-monkeys the opportunity to develop into ever more powerful architectures later. We will focus on programming the Arduino using the standard Arduino libraries, although if you want to know more, full documentation on the `avr-libc` library package can be found at www.nongnu.org/avr-libc/. While we are at it, it is also worth mentioning that it is even possible to program the Arduino interface board using other development environments more often associated with computer development, such as Eclipse, NetBeans, or any other development package that you are familiar with ... or if you have an aversion to the color teal.

What's Needed

Now that we know why we are learning to write code and generally what this platform offers, we are going to need a little bit of hardware to complete the projects discussed in the rest of this book. Even though we will focus our discussions primarily on programming the Arduino, the hardware is what makes programming the Arduino so much fun. The best place to get started with prototyping electronic circuits with the Arduino is through one of the many starter kits available from various retailers for anywhere from \$40 to \$100 USD or about €30 to €70 EUR. SparkFun Electronics (www.sparkfun.com) offers the Starter Kit and Inventor's Kit; Adafruit Industries (www.adafruit.com) the Budget Pack or Starter Pack; *MAKE Magazine's* Maker SHED (www.makershed.com) with either a Getting Started Kit or the Mintronics Survival Pack; or the Arduino Store's (<http://store.arduino.cc>) Workshop Kit. Figure 1-7 shows what this assembled kit might look like.



Figure 1-7. *Getting started hardware kit*

While each of these kits will have more or less what you need, Table 1-1 provides a simple shopping list of some of the specific hardware that we will use in the following pages, which you might want to purchase individually or at least check whether or not the kit you are considering contains these items. Prices are only approximate and are listed just to give you an idea on the costs.

Keep in mind that this is just a general list of recommended parts; feel free to purchase something different—as long as it's somewhat similar to what's listed—and to order from vendors of your choice. For example, rather than using an Arduino Uno, the standard Arduino interface board, maybe you're interested in wearables or e-textiles and would like to use a LilyPad Arduino instead. That's fine, just be sure to read up on the differences between the recommended hardware and your own on the Arduino Hardware page at <http://arduino.cc/en/Main/Hardware>. Likewise, many of the items on the list, such as resistors, are kind of like horseshoes in that close enough will usually get the job done. For a more detailed list that includes all the parts used in this book, refer to the expanded hardware list in the appendix at the end of this book.

Table 1-1. *Abbreviated Hardware List*

Part	Description	Price (US / EUR)
Arduino Uno	Microcontroller interface board	\$30/€22
USB cable	A-B (often used for printers) for the Arduino Uno	\$4/€2
Solderless breadboard	Either standard (400 tie points) or mini (170 tie points)	\$6/€4
Hookup wire	Solid 22 AWG or pre-terminated M/M jumper wires	\$7/€5
Light emitting diodes	Standard 5mm LEDs and common cathode RGB LEDs	\$10/€7
Resistors	Handful each of 220, 330, 1K, 10K ohm ¼ watt or similar	\$10/€7
Capacitors	A couple 1 and .1 microfarad capacitors of any variety	\$4/€3
Analog sensors	10K ohm trimpot, 10K ohm photocell, TMP36 temp sensor	\$4/€3
Switches	Miniature momentary pushbuttons and tilt switches	\$3/€2
Transistors	2N2222 or 2N3904 NPN transistor or similar	\$1/€1

This minimal list of hardware will get you through about half of the book with other specific components necessary for the more advanced projects. When required, information on additional parts will be provided at the start of the chapter or section. While detailed in each chapter, these projects will include the use of temperature sensors, force/flex sensors, light sensors, accelerometers, wind sensors, liquid crystal displays, stepper motors, hobby servos, and other sorts of hardware. Although, rather than mandate a hefty toolbox of equipment, the projects and code will be written with as much flexibility as possible to allow the use of other sensors or actuators, depending on what you have available and your own personal interests and project ideas.

To keep things simple, we will power the Arduino Uno from the USB port of our computer using a USB cable for each one of the projects in this book. Once we upload the programs to our boards, however, the program is permanently stored on the board just like a USB flash drive. We could (if we wanted) power the Arduino board from an external power source, such as DC power supply or battery, and not need the USB cable at all. For more information, refer to the section “Connecting & Powering Arduino” on the Arduino Playground at <http://arduino.cc/playground/Main/ArduinoCoreHardware>.

Getting Up and Running

By this point, you have a pretty good sense for what the Arduino is, its history, and some of what you can do with it. It's really exciting that with a few carefully written commands, which you will learn in the upcoming chapters, you can make things light up or move, sense the world around you, and generally make things more fun.

And now that you have some basic hardware in hand, including an Arduino interface board, it's time to get up and running. We need to do the following before moving on to the next chapter:

1. Download and install the Arduino development environment
2. Connect the Arduino board with a USB cable and install drivers
3. Launch the Arduino application and open the Blink example
4. Select your board and serial port
5. Upload your first sketch

Don't worry. It's not that difficult to get going. In fact, it's generally hard to go wrong, because it is nearly impossible to burn the house down or cut off an arm with an Arduino. And even if you wire up something wrong, it's probably okay because the Arduino board is a tough little guy and can take some abuse. So don't worry that you might mess something up. Of course, we are going to make some mistakes, but hopefully they will teach us something and we will become better programmers and makers because of it. So let's get on with it.

Installing the Software

First things first, you need to download and install the Arduino software. Because the Arduino Team is always making updates to the software, you should head to the main download page on the Arduino web site (<http://arduino.cc/en/Main/Software>), shown in Figure 1-8.

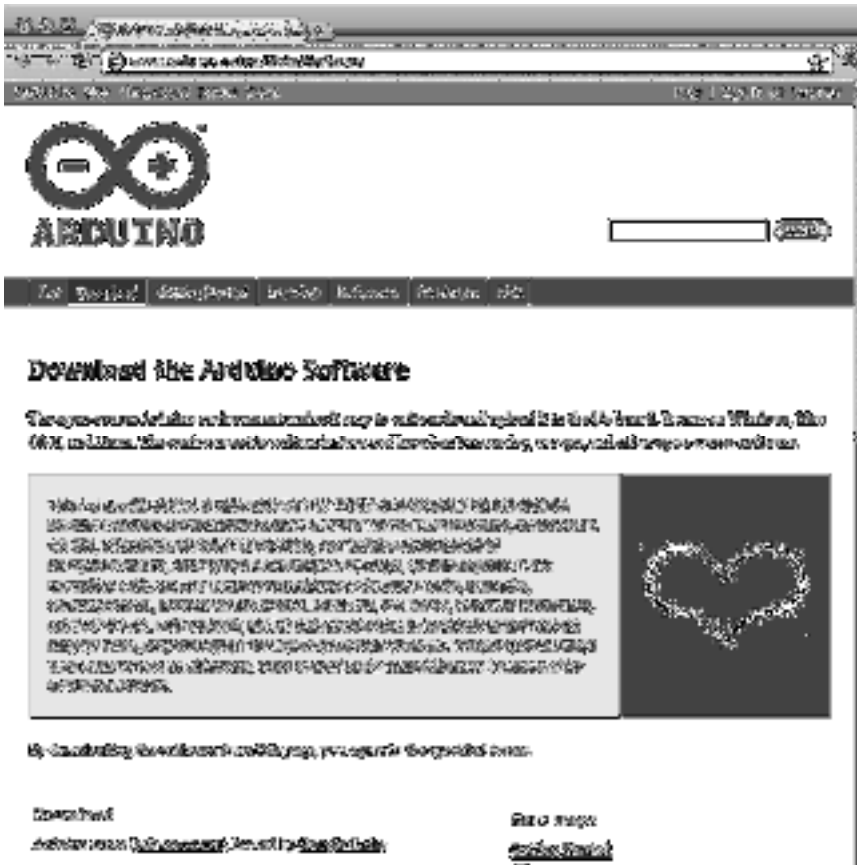


Figure 1-8. Arduino software download page

From here download and install the latest version of the software for your particular operating system. Full installation instructions are available on the Getting Started page at <http://arduino.cc/en/Guide/HomePage>. Linux can be a little tricky to get installed as of this writing, so be sure to carefully follow the instructions posted.

Connecting the Arduino

With the software installed, you should be able to connect your Arduino to the USB port on your computer using an A-B USB cable. The Arduino's power indicator LED will light up on the board, letting us know that it has power and is ready to go. With a brand-new Arduino Uno, the first time that it powers up, the pin 13 LED marked L will begin to blink rapidly, letting us know that the board has been fully tested. On Windows-based PCs or older Arduino boards, it is necessary to install a driver for the Arduino's onboard USB to Serial convertor, as shown in Figure 1-9. For the latest on how to install these drivers, be sure to follow the instructions on the drivers section of the Getting Started Guide at <http://arduino.cc/en/Guide/Windows#toc4>.

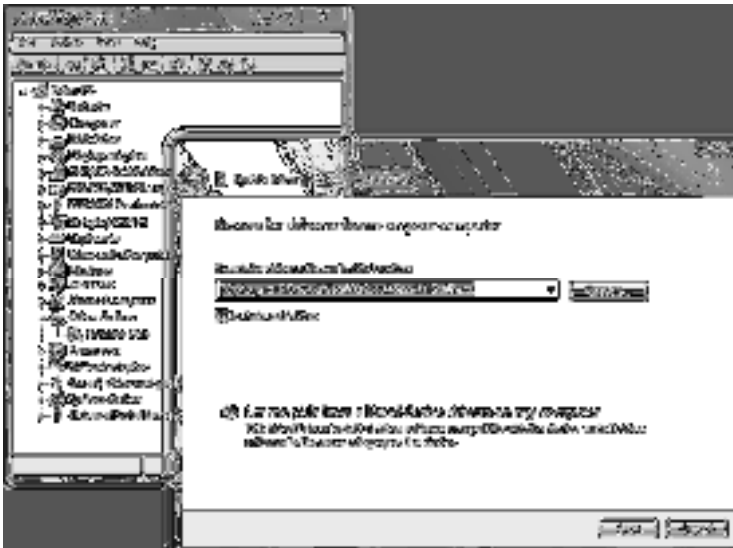


Figure 1-9. *Installing drivers on Windows 7*

Opening a Sketch

Now you can launch the Arduino development environment. This will bring up an empty window if this is your first time out. Open an example sketch by navigating to the File menu ► Examples ► 1.Basics and select the sketch named Blink, as shown in Figure 1-10. You should now see a very simple example sketch for blinking the onboard LED once every second.



Figure 1-10. Opening example sketch

Selecting the Board and Serial Port

Before we can upload our sample sketch, we need to select the correct board type and serial port that the board is attached to on our computer. Setting the correct board can be done in the Tools ► Board menu by selecting Arduino Uno or one of the other corresponding board names, as shown in Figure 1-11.



Figure 1-11. Selecting board type

Next, we need to choose the correct serial port under the Tools ► Serial Port menu, as shown in Figure 1-12. This port should be named COM3, or something similar, on a Windows PC; or something like `/dev/tty.usbmodem` or `/dev/tty.usbserial` on the Mac.

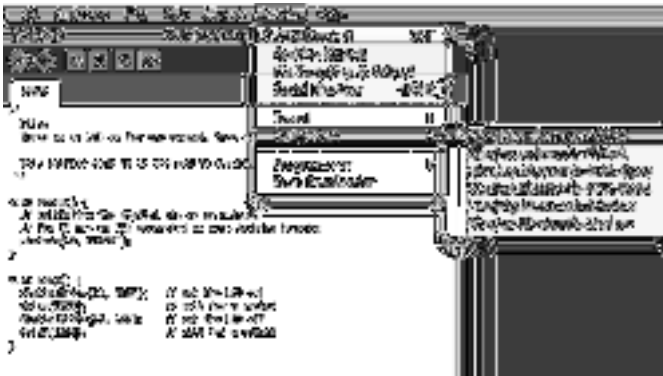


Figure 1-12. Selecting serial port

Uploading a Sketch

Once you have selected the proper board and serial port, it's time for the fun part. To upload a sketch onto the Arduino board, simply hit the Upload button on the toolbar, as shown in Figure 1-13. The onboard LEDs marked RX and TX will blink furiously and you will receive a message in the status bar that says, "Done uploading." That's all there is to it!



Figure 1-13. Uploading a sketch to the Arduino board

Summary

With all the installing, connecting, and uploading done and out of the way, you should now have a blinking LED on your Arduino Uno. That's not to say you already fully understand how it works because that would kind of defeat the entire purpose of this book. But now that we've got something to blink and know that we can make it work, we might as well jump into learning the basics of programming the Arduino with our first project. By jumping right into how the code works, we can move from hacking together lines of code that somebody else gave us to actually writing them ourselves.

Sketching in Code

In the last chapter, we managed to get our first sketch up and running with the famous Blink sketch, otherwise known as the “Hello world” example. This is often a good place to start because it has a long-standing tradition as the first program to write when learning a new language. In the case of programming hardware, we generally use the LED to blink hello. While that is a good beginning, we want to quickly get to some of the more interesting aspects of Arduino C, beginning with the focus of this chapter: the fundamental mechanics and control structures of sketching Arduino code. Our first project, RGB Blink, will explore the basic structure, syntax, and functions necessary for controlling the output of a three-color LED, taking our blinking LED to the next level.

We’ll start with a primer on what sketching means and how we can put the philosophy of sketching in code to use. As we will do throughout this book, we will then jump right into our project example and once you have this project wired up and the sketch uploaded to your Arduino board, we’ll back up and further explain how the sketch works and discuss most of the elements used in the project. In this way, when we look at each of the individual concepts in the chapter, you’ll already know how they were put to use. In this chapter, we’ll have a look at the basic anatomy of our project sketch and discuss how and why Arduino C does what it does. Don’t worry if it doesn’t make a tremendous amount of sense in the beginning, just work through it and it should get clearer as you go along.

What’s needed for this chapter:

- Arduino Uno
- 5mm RGB LED with common cathode
- 1x 330 ohm and 2x 220 ohm ¼ watt resistors or similar
- Hookup wires
- Solderless breadboard

What is Sketching in Code?

The idea of sketching in code is a way of thinking about writing code as a simple intuitive process, just like drawing in a sketchbook. In this way, an Arduino program is called a sketch and is saved in a folder called a sketchbook. Sketching means we can get our hands dirty and quickly try out a new idea. It is a skill available to all of us, not just artists and designers, and neither is it limited to pens, paper, pencils, or napkins.

So often, an idea in one of my classes begins with the simple words, “Wouldn’t it be cool if ...?” Quickly sketching out these ideas serves as a way to conceptualize this moment of inspiration. Writing code can be just like this; it is after all a creative process used to solve specific problems. Sketching implies a sense of directness in the application of materials, like a pencil to a piece of paper. The Arduino development environment takes this same approach to making code as simple and direct as is possible. It is also why we will begin with simple sketches that quickly get you making something right away rather than bogging down in page after page of complex theory or algorithms.

When sketching in code, it’s okay to write bad code. You won’t hurt anything. As long as you get in there and start somewhere, as doing anything is always better than doing nothing. Making mistakes and learning what does work and what doesn’t, is an important part of learning anything new. In our sketchbook, we will start with simple **source code**, the basic instructions that tell the Arduino what it should do, along with basic hardware, starting small and working our way through to more involved examples.

You should freely experiment with every example, changing values, and piecing things together in unexpected ways to see what happens. Don’t be discouraged if it doesn’t work the first time; stick with it and it will get clearer with time. Testing and iterations are important parts of successfully writing code. So, change existing code or write some new code to establish a basic framework and then verify that your changes compile correctly. Work in incremental steps one addition at a time as you add to this basic framework for your code so that when you finally have something substantial, you can load it on to the interface board to see what happens. If something didn’t quite work as expected, all you need to do is undo that last incremental step. In this way it’s a good idea when you make changes, to only change one thing at a time before verifying that it still works before moving on to the next change.

Project 1: RGB Blink

To really get started, we are going to use a nifty little component called a red-green-blue light emitting diode, or more simply an **RGB LED**. The RGB LED works off a similar principle as televisions and computer monitors. By using the same three colors of light that we are receptive to in our vision—specifically red, green, and blue—we can reproduce a vast array of colors through an additive color-mixing process. This form of color mixing should sound familiar from secondary school science, and is shown in the RGB color wheel in Figure 2-1. While it’s a little harder to make out in a black-and-white book like this one, if you use your imagination you’ll see that by combining two primary colors we end up with a secondary color. For example, add the color red to blue and we get magenta or if we add red and green we will get yellow. If we add all three primary colors together we will end up with white light.

- [download online Life After Loss: Conquering Grief and Finding Hope](#)
 - [download Criminal Destiny \(Masterminds, Book 2\)](#)
 - [**A Political History of The Gambia, 1816-1994 \(Rochester Studies in African History and the Diaspora\) pdf**](#)
 - [**The Hunted \(Tom Clancy's EndWar, Book 2\) book**](#)
 - [The Tale of the Heike pdf, azw \(kindle\)](#)
 - [click Trolls: An Unnatural History here](#)
-
- <http://aneventshop.com/ebooks/England-and-the-English.pdf>
 - <http://test1.batsinbelfries.com/ebooks/Punishing-Race--A-Continuing-American-Dilemma.pdf>
 - <http://aneventshop.com/ebooks/A-Political-History-of-The-Gambia--1816-1994--Rochester-Studies-in-African-History-and-the-Diaspora-.pdf>
 - <http://metromekanik.com/ebooks/Brazilian-Jiu-Jitsu--The-Ultimate-Guide-to-Dominating-Brazilian-Jiu-Jitsu-and-Mixed-Martial-Arts-Combat.pdf>
 - <http://www.mmastyles.com/books/The-Tale-of-the-Heike.pdf>
 - <http://www.khoi.dk/?books/Trolls--An-Unnatural-History.pdf>