1 Getting Started and Understanding the Arduino Landscape

What You'll Need for This Chapter:

Arduino Uno or Adafruit METRO 328

USB cable (Type A to B for Uno, Type A to Micro-B for METRO)

CODE AND DIGITAL CONTENT FOR THIS CHAPTER
Code downloads, videos, and other digital content for this chapter can be found at:
`exploringarduino.com/content2/ch1`

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:
`wiley.com/go/exploringarduino2e`

Whether you are a weekend warrior looking to learn something new, an aspiring electrical engineer, or a software developer looking to better understand the hardware that runs your code, getting your first Arduino project up and running is sure to be an energizing experience. The preface of this book should have already given you some perspective on the Arduino platform and its capabilities; now it's time to explore your options in the world of Arduino. In this chapter, you will examine the available hardware, learn about the programming environment and language, and get your first program up and running. Once you understand the functionality that the Arduino can provide, you'll write your first program and get the Arduino to blink!

NOTE To follow along with a video that introduces the Arduino platform, visit the Chapter 1 content page for the second edition of this book at `exploringarduino.com/content2/ch1`.

Exploring the Arduino Ecosystem

In your adventures with the Arduino, you'll depend on three main components for your projects:

- First-party or third-party Arduino boards
- External hardware (including both shields and manually created circuits, which you'll explore throughout this book)
- The Arduino integrated development environment, or Arduino IDE

All these system components work in tandem to enable you to accomplish just about anything with your Arduino.

You have a lot of options when it comes to Arduino development boards. Most of this book uses Arduino boards designed by Arduino. Some of the final chapters leverage Arduino-compatible hardware that is designed by third parties to add features like Bluetooth and Wi-Fi to the standard Arduino offerings. Many third-party Arduino boards are directly compatible with Arduino software, libraries, hardware, etc. Some of these boards are designed to be exact clones of official Arduino boards, while others add their own features or capabilities. All the boards used in this book are programmable via the same IDE. When relevant, this book will list caveats about using different boards for various projects. Most of the projects in this book use the Arduino Uno because it has become the de facto introductory board for learning Arduino. You can freely substitute the Adafruit METRO 328 board in places where the Uno is called for—it is functionally identical. You'll see it used in place of the Uno in some of the photos and videos that accompany this book. Most introductory tutorials that you'll find on the web use the Uno or a variant of it. If you do use the Adafruit METRO 328, you may need to install the drivers for it to be detected as an Arduino Uno on Windows. Download and run the installer from `blum.fyi/adafruit-windows-drivers`.

WARNING **Beware of Counterfeits.** Only buy Arduino boards and Arduino-compatible boards from reputable sources (such as those listed throughout this book). There are many companies that manufacture clones of popular Arduino boards with inferior components.

You will start by exploring the basic functionality that is found in every Arduino board. Then you will examine the differences between each modern board so that you can make an informed decision when choosing a board to use for your next project.

THE GREAT ARDUINO SCHISM AND REFORMATION

Before you jump into understanding the available options in the Arduino ecosystem, I need to talk about the elephant in the room: the Great Arduino Schism and Reformation (not an official name). Between the release of the first edition of this book and the release of the second edition, the people behind the Arduino hardware and software had a falling out. I won't go into the details here, or pick a side. Basically, Arduino split into two entities represented by two websites: `Arduino.cc` and `Arduino.org`. Each group started producing slightly different hardware offerings, forked the codebase, and made conflicting claims about which hardware was genuine. Thankfully, the two sides of this battle have since reconciled their differences and we're back to one Arduino again. Throughout this book, I'll generally talk about the hardware offerings from Arduino.cc, though by the time you get this book, the two Arduinos should be one again. If you'd like to learn more about this nerdy drama, `Hackaday.com` did a series of reports on it. You can read about the resolution at `blum.fyi/arduino-vs-arduino`.

Arduino Functionality

All Arduino boards have a few key capabilities and functions. Take a moment to examine the Arduino Uno shown in Figure 1-1; it will be your base configuration. These are some functional groups that you'll be concerning yourself with:

- **Microcontroller:** At the heart of every Arduino is a microcontroller. This is the brain of your Arduino.
- **Programming:** Programming interfaces enable you to load software onto your Arduino.
- **I/O:** Input/Output (I/O) circuitry is what enables your Arduino interface with sensors, actuators, etc.
- **Power:** There are a variety of ways to supply power to an Arduino. Most Arduino boards can automatically switch between power from multiple sources (such as USB and a battery).

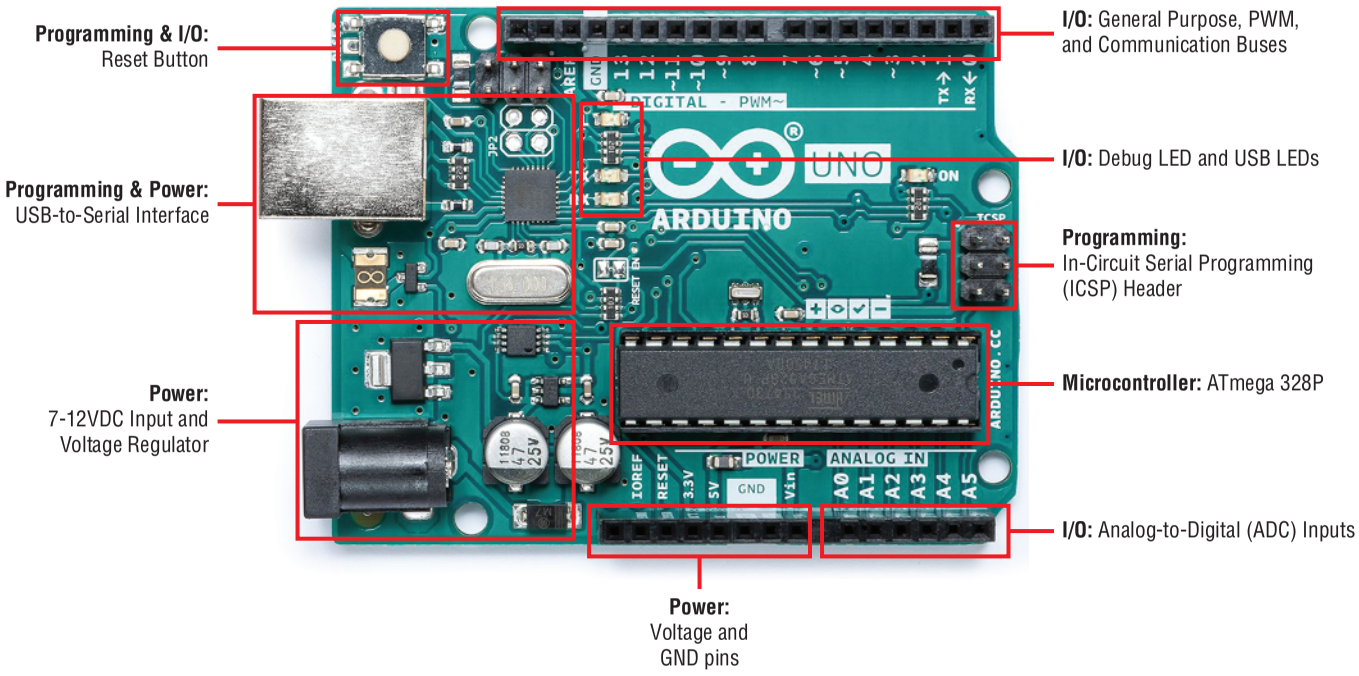

Figure 1-1: Arduino Uno components

Credit: Arduino, arduino.cc; annotations by author

The Microcontroller

At the heart of every Arduino is a microcontroller unit (MCU). All the original Arduino boards, including the Arduino Uno, used an 8-bit Atmel® ATmega microcontroller based on the AVR® architecture. The Arduino Uno in Figure 1-1, for example, uses an ATmega 328P. For most projects that you'll want to build, a simple 8-bit MCU like this one will be more than enough for your needs, so that's what you'll use throughout most of the exercises in this book.

NOTE MICROCHIP AND ATMEL Microchip, a chip manufacturer famous for making the PIC series of microcontrollers, recently acquired Atmel. ATmega chip production has continued under this new brand. Therefore, you may see Microchip and Atmel used interchangeably in reference to the manufacturer of ATmega microcontroller chips. The chips are functionally identical if they have the same part number.

NON-AVR MICROCONTROLLER ARCHITECTURES

But what about when you want to start doing crazy things like synthesizing music, running a web server, and driving massive LED displays? Though possible with clever and efficient programming on an 8-bit MCU, some of these needs are better served by faster and more capable processors.

As an answer to this, in recent years, Arduino has been expanding the range of available Arduino boards to include some that run on Intel (x86 and ARC—Argonaut RISC Core) architectures, and some that use Arm® (Advanced RISC Machine) architectures. The Arduino Due, for example, uses a 32-bit Arm Cortex®-M3 microprocessor. This Cortex processor is faster and contains more peripherals than the 8-bit AVR MCU, thus enabling the Due to do things like play music. Other new Arduino boards add functionality like built-in Wi-Fi and Bluetooth, which is also facilitated by faster and more capable processors. I'll touch on some of these boards later in this chapter, and you'll also get the opportunity to build projects with them later in this book.

You don't need to understand the intricacies of processor architectures to program or use an Arduino—it's all abstracted away for you. However, some people like to know what underlies their hardware projects. The following list will help clarify the buzzwords you just read:

- 8-bit architecture—An MCU architecture type where all addresses, integers, and other key data types are represented as 8-bit numbers.
- 32-bit architecture—An MCU architecture type where all the addresses, integers, and other key data types are represented as 32-bit numbers.

(Continued)

(Continued)

- Microchip (previously Atmel)—A company that makes microcontrollers. Microchip/Atmel makes both AVR MCUs and Arm processors. Most Arduinos use processors that are made by Microchip/Atmel.
- AVR—A microcontroller architecture developed by Atmel for their ATmega MCUs.
- Arm—A collection of 32/64-bit processor architectures developed by a company of the same name. Arm licenses its embedded architecture designs to be used by companies like Microchip and others.
- Cortex-M Series—Cortex M0, M3, and so on represent microprocessor Arm architectures.

The Arduino's microcontroller is responsible for holding all your compiled code and executing the commands you specify. The Arduino programming language gives you access to microcontroller peripherals, including analog-to-digital converters (ADCs), general-purpose input/output (GPIO or just I/O) pins, communication buses (including I^2C, SPI, UART, and others), and serial/USB interfaces. Utilizing copper wires etched into the Arduino's printed circuit board, all of this useful functionality is routed from the tiny pins on the microcontroller to accessible headers on the Arduino that you can plug wires or shields into. In the case of the Uno, a 16 MHz ceramic resonator or oscillating crystal is wired to the ATmega's clock pins, which serves as the reference by which all program commands execute. You can use the Reset button to restart the execution of your program. Most Arduino boards come with a debug LED already connected to pin 13, which enables you to run your first program (blinking an LED) without connecting any additional circuitry.

Programming Interfaces

Ordinarily, microcontroller programs are written in C or assembly, and programmed via the In-Circuit Serial Programming™ (ICSP™) interface using a dedicated programmer (see Figure 1-2). Perhaps the most important characteristic of an Arduino is that you can program it directly using only an ordinary USB cable. This functionality is made possible by the Arduino bootloader. The bootloader is loaded onto the microcontroller at the factory (using the ICSP header), which allows a serial USART (Universal Synchronous/Asynchronous Receiver/Transmitter) to load your program on the Arduino without using a separate programmer. (You can learn more about how the bootloader functions in the sidebar, "The Arduino Bootloader and Firmware Setup.")

In the case of the Arduino Uno and Mega 2560, a secondary microcontroller (an ATmega16U2 or ATmega8U2, depending on your revision) serves as an interface

between a USB cable and the serial USART pins on the main microcontroller. In the Adafruit METRO 328, a Silicon Labs bridge chip is used in place of the ATmega 16U2, but its function is equivalent. The Arduino Leonardo, which uses an ATmega32U4 as the main microcontroller, has USB incorporated, so a secondary microcontroller is not needed. The Arduino M0 uses a Cortex M0 that also includes USB functionality, so it doesn't need a secondary USB chip. In older Arduino boards, an FTDI brand USB-to-serial chip was used as the interface between the ATmega's serial USART port and a USB connection. It's still a popular solution when creating your own Arduino-compatible product.

Figure 1-2: AVRISP mkII programmer

Credit: © Microchip Technology Incorporated. Used with permission.

Input/Output: GPIO, ADCs, and Communication Busses

The part of the Arduino that you'll care most about during your projects is the general-purpose Input/Output (GPIO) and ADC pins. All of these pins can be individually addressed via the programs you'll write. These pins can serve as digital inputs and outputs. The ADC pins can also act as analog inputs that can measure voltages between 0V and 5V (usually from sensors). Many of these pins are also multiplexed to serve special functions, which you will explore later in this book. These special functions include various communication interfaces, serial interfaces, pulse-width-modulated outputs, and external interrupts.

Power

For most of your projects, you will simply use the 5V power that is provided over your USB cable. However, when you're ready to untether your project from a computer, you have other power options. Most Arduinos can accept between 6V and 20V (7V to 12V is the recommended voltage supply range) via the direct current (DC) barrel jack connector,

or into the VIN pin. Some Arduinos operate at 5V logic levels, and others operate at 3.3V logic levels. For 5V Arduinos, like the Uno, the power is configured as follows:

- 5V is used for all the logic on the Uno board. In other words, when you toggle a digital I/O pin, you are toggling it between 5V and 0V.
- 3.3V is broken out to a pin to accommodate 3.3V shields and external circuitry.

For most projects in this book, you can generally assume the use of a 5V Arduino, unless I explicitly specify otherwise.

THE ARDUINO BOOTLOADER AND FIRMWARE SETUP

A *bootloader* is a chunk of code that lives in a reserved space in the program memory of the Arduino's main MCU. In general, AVR microcontrollers are programmed with an ICSP, which talks to the microcontroller via a Serial Peripheral Interface (SPI). Programming via this method is straightforward, but necessitates the user having a hardware programmer such as an STK500 or an AVRISP mkII (see Figure 1-2).

When you first boot the Arduino board, it enters the bootloader, which runs for a few seconds. If it receives a programming command from the IDE over the MCU's UART (serial interface) in that time period, it loads the program that you are sending it into the rest of the MCU's program memory. If it does not receive a programming command, it starts running your most recently uploaded sketch, which resides in the rest of the program memory.

When you send an "upload" command from the Arduino IDE, it instructs the USB-to-serial chip (an ATmega 16U2 or 8U2 in the case of the Arduino Uno) to reset the main MCU, thus forcing it into bootloader mode. Then, your computer immediately begins to send the program contents, which the MCU is ready to receive over its UART connection (facilitated by the USB-to-serial converter).

Bootloaders are great because they enable simple programming via USB with no external hardware. However, they do have two downsides:

- They take up valuable program space. If you have written a complicated sketch, the approximately 2 KB of space taken up by the bootloader might be really valuable.
- Using a bootloader means that your program will always be delayed by a few seconds at bootup as the bootloader checks for a programming request.

If you have a programmer (or another Arduino that can be programmed to act as a programmer), you can remove the bootloader from your ATmega and program it directly by connecting your programmer to the ICSP header and using the *File* ➤ *Upload Using Programmer* command from within the IDE.

Arduino Boards

This book cannot possibly cover all the available Arduino boards; there are many, and manufacturers are constantly releasing new ones with various features. I will focus on a subset of the most commonly used Arduino boards. The following section highlights some of the features in these boards.

The Uno (see Figure 1-3) is the flagship introductory-level Arduino and will be used heavily in this book. It uses an ATmega328P as the main MCU.

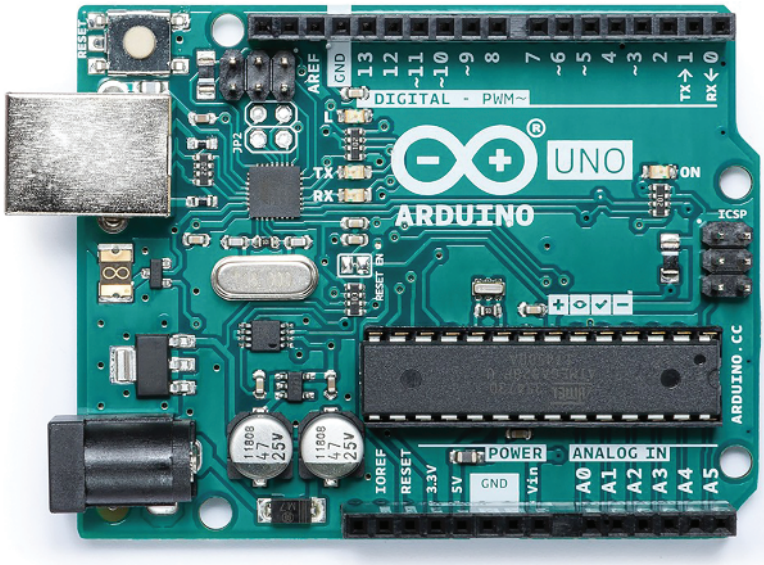

Figure 1-3: The Arduino Uno

Credit: Arduino, arduino.cc

The Mega 2560 (see Figure 1-4) employs an Microchip/Atmel ATmega2560 as the main MCU, which has 54 general I/Os to enable you to interface with many more devices. Think of the Mega as a supercharged version of the Uno—it's faster, has more memory, exposes more ADC channels, and has four hardware serial interfaces (unlike the one serial interface found on the Uno). It costs approximately 50 percent more than the Uno.

The Arduino Leonardo and Arduino Micro (see Figure 1-5 and Figure 1-6) both use the ATmega32U4 as the main microcontroller, which has a USB interface built in. Therefore, they don't need a secondary MCU to perform the serial-to-USB conversion. This cuts down on the cost and enables you to do unique things like emulate a joystick or a keyboard instead of a simple serial device. You will learn how to use these features in Chapter 8, "Emulating USB Devices". The Micro is functionally identical to the Leonardo, but is a smaller form factor that is designed to be plugged into a solderless or soldered breadboard.

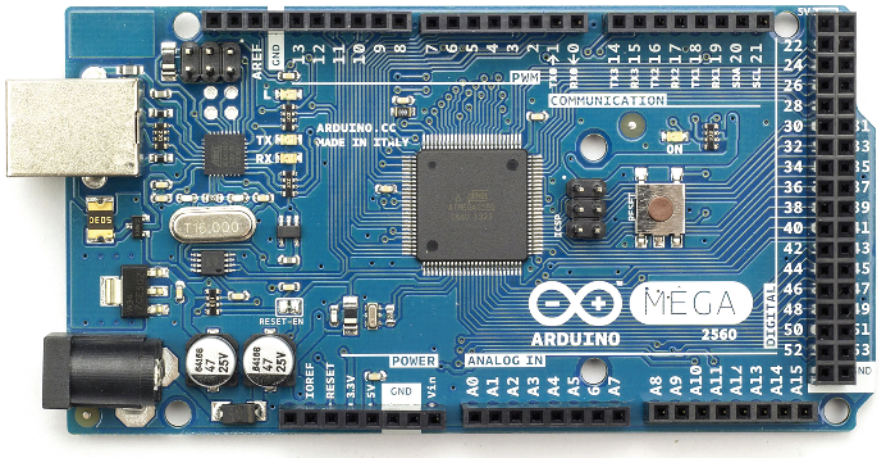

Figure 1-4: The Arduino Mega 2560
Credit: Arduino, arduino.cc

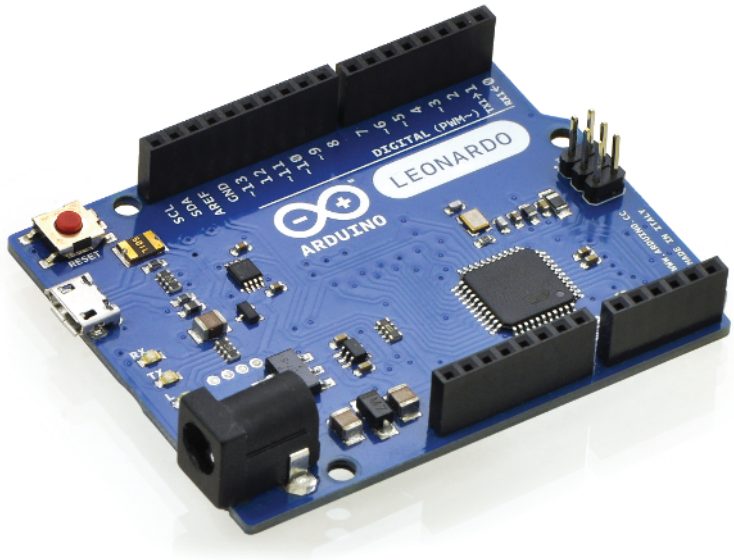

Figure 1-5: The Arduino Leonardo
Credit: Pololu Robotics & Electronics, pololu.com

Figure 1-6: The Arduino Micro
Credit: Arduino, arduino.cc

The Due (see Figure 1-7) was Arduino's first foray into using the Arm microarchitecture. It uses a 32-bit Arm Cortex-M3 SAM3X. The Due offers higher-precision ADCs, selectable-resolution pulse-width modulation (PWM), digital-to-analog converters (DACs), a USB host connector, and an 84 MHz clock speed.

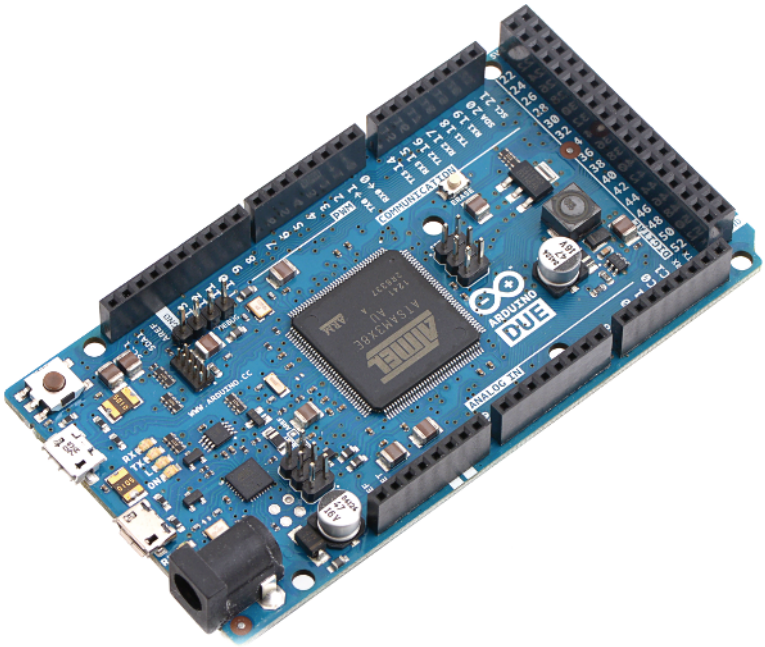

Figure 1-7: The Arduino Due
Credit: Pololu Robotics & Electronics, pololu.com

There are a variety of other Arduino boards as well. As you go through the chapters of this book, you may want to consider using some of those boards for more sophisticated projects that you dream up. As your needs get more specific, you may consider using some of the third-party Arduino-compatible boards that are available from companies like SparkFun, Adafruit, Pololu, and others. Because Arduino is an open-source platform, literally hundreds of clones and derivatives are available. The products and companies that I specifically call out in this book are ones that I have tested personally and can confirm work well. Use caution when buying generic Arduino clones online; read the reviews to find out if they work the way they are intended to. When in doubt, buy official Arduino products, or products from well-trusted companies like the ones I've mentioned.

When it comes to things like Bluetooth and Wi-Fi interoperability, the official Arduino offerings are a bit lacking at the time of this writing, so my recommended route is to check out the extremely well-documented Arduino-compatible Feather boards from `Adafruit.com`. You'll learn how to use these boards for building wireless Bluetooth and Wi-Fi projects in the final chapters of this book. Figure 1-8 shows a Bluetooth-enabled Arduino board from Adafruit.

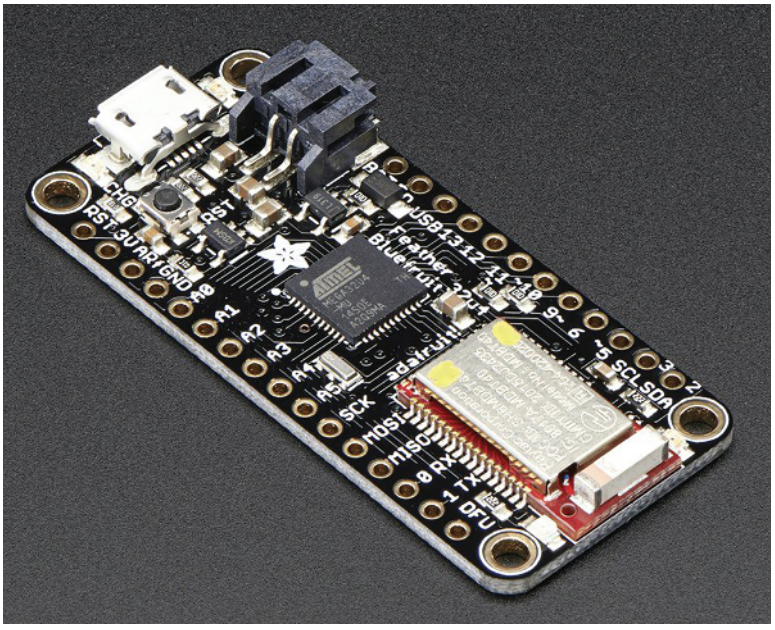

Figure 1-8: The Adafruit Feather 32u4 Bluefruit LE

Credit: Adafruit, `adafruit.com`

The skills you learn from this book will also easily transfer to a variety of Arduino-inspired platforms that use an Arduino-like programming interface coupled with their own hardware. The Photon (see Figure 1-9) from Particle is a great example of a Wi-Fi enabled microcontroller that uses a programming language inspired by the Arduino language. I use Particle Photons in my apartment to control my reading lamps and window shades from my phone.

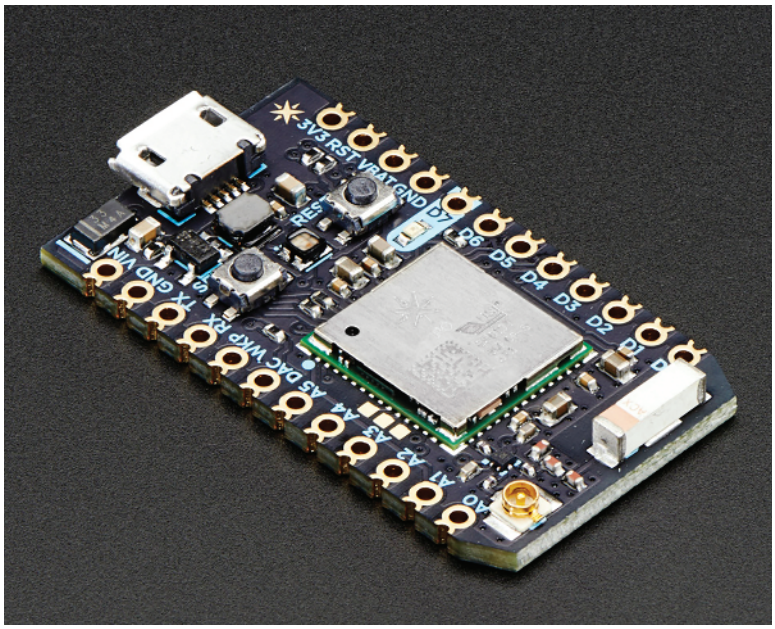

Figure 1-9: The Particle Photon
Credit: Adafruit, `adafruit.com`

Creating Your First Program

Now that you understand the hardware you'll be using throughout this book, you can install the software or access the Arduino web IDE and run your first program. Throughout this book, you'll generally use the downloaded desktop IDE. Start by downloading the Arduino software to your computer.

THE ARDUINO CLOUD IDE

The Arduino Cloud IDE is not explicitly used in this book's tutorials, but you can use it instead of the desktop IDE if you prefer. Simply set up an account at `arduino.cc`, and navigate to the editor, at `create.arduino.cc/editor`. Follow the instructions to install the plug-in and to start uploading code.

Downloading and Installing the Arduino IDE

Go to the Arduino website at `arduino.cc` and click the Software tab to display the Software page (see Figure 1-10). From there, you can download the newest version of the IDE that corresponds to your operating system.

Figure 1-10: The Arduino.cc page where you can download the Arduino IDE

If you're on Windows, download the installer instead of the Zip file. The installer will handle loading the necessary drivers for you. Run the installer and follow the onscreen directions. All the default options should be fine. For macOS or Linux, download the

compressed folder and extract it. On Mac OS X, simply drag the application into your Applications folder.

Running the IDE and Connecting to the Arduino

Now that you have the IDE downloaded and ready to run, you can connect the Arduino to your computer via USB, as shown in Figure 1-11. Linux and macOS machines usually install the drivers automatically.

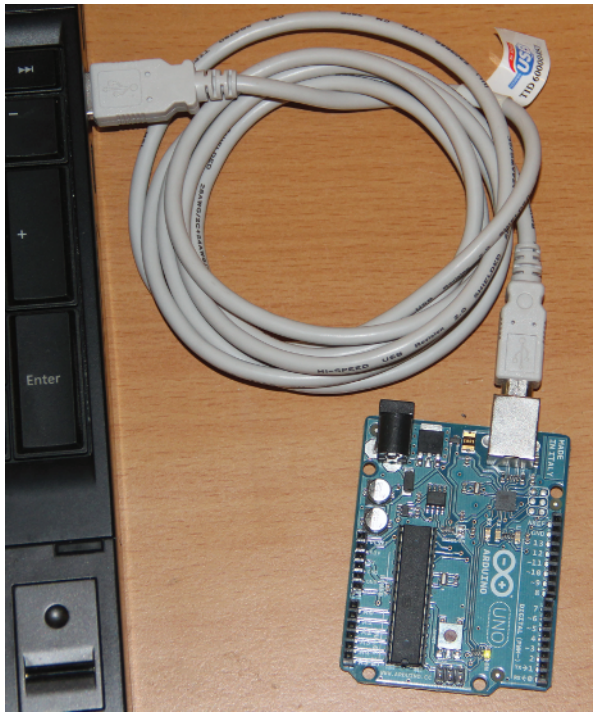

Figure 1-11: Arduino Uno connected to a computer via USB

NOTE Having trouble getting the IDE installed, or connecting to your board? Arduino.cc provides great troubleshooting instructions for all operating systems and Arduino hardware. Check out `blum.fyi/install-arduino`.

Now, launch the Arduino IDE. You're ready to load your first program onto your Arduino. To ensure that everything is working as expected, you'll load the Blink example program, which will blink the onboard LED. Most Arduinos have an onboard LED

(connected to pin 13 in the case of the Arduino Uno). Navigate to *File* ➤ *Examples* ➤ *Basic*, and click the Blink program. This opens a new IDE window with the Blink program already written for you. First, you'll program the Arduino with this example sketch, and then you'll analyze the program to understand the important components so that you can start to write your own programs in the next chapter.

Before you send the program to your Arduino board, you need to tell the IDE what kind of Arduino you have connected and what port it is connected to. Go to *Tools* ➤ *Board* and ensure that the right board is selected. This example uses the Uno, but if you are using a different board, select that one (assuming that it also has an onboard LED—most do).

The last step before programming is to tell the IDE what port your board is connected to. Navigate to *Tools* ➤ *Serial Port* and select the appropriate port. On Windows machines, this will be `COM*`, where * is some number representing the serial port number.

TIP If you have multiple serial devices attached to your computer, try unplugging your board to see which COM port disappears from the menu; then plug it back in and select that COM port.

On Linux and macOS computers, the serial port looks something like `/dev/tty.usbmodem*` or `/dev/tty.usbserial*`, where * is a string of alphanumeric characters.

You're finally ready to load your first program. Click the Upload button ( ) in the top-left corner of the IDE. The status bar at the bottom of the IDE shows a progress bar as it compiles and uploads your program. The TX/RX LEDs on your Arduino will flash as it is programming. These LEDs show that data is being transferred to the board from your computer. When the upload completes, the onboard LED on your Arduino should be blinking once per second. Congratulations! You've just uploaded your first Arduino program.

Breaking Down Your First Program

Take a moment to deconstruct the Blink program so that you understand the basic structure of programs written for the Arduino. Consider Figure 1-12.

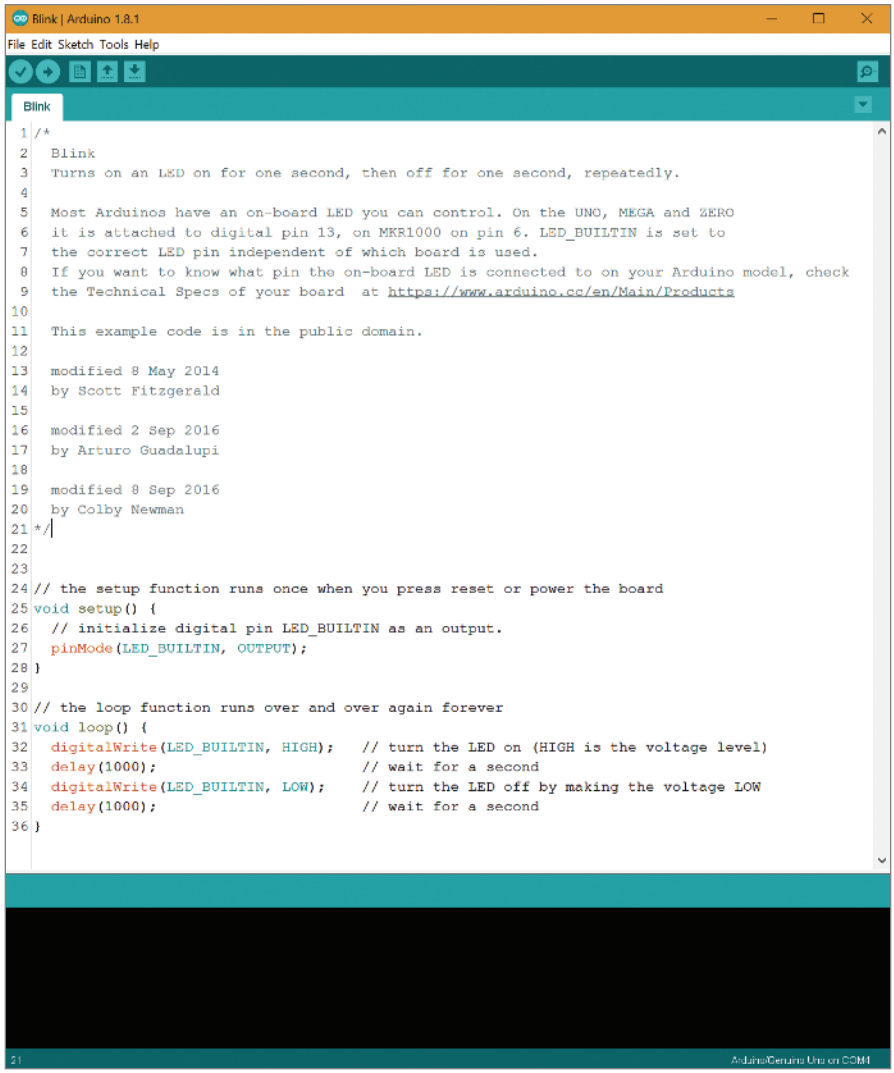

Figure 1-12: The Blink program (with line numbers)

Here's how the code works, piece by piece:

1. **Lines 1–21:** This is a multiline comment. Comments are important for documenting your code. Whatever you write between these symbols will not be compiled or even seen by your Arduino. Multiline comments start with /* and end with */. Multiline comments are generally used when you have to say a lot (like the description of this program).

2. **Line 24:** This is a single-line comment. When you put // on any line, the compiler ignores all text after that symbol on the same line. This is great for annotating specific lines of code or for "commenting out" a particular line of code that you believe might be causing problems.
3. **Line 25:** `void setup()` is one of two functions that must be included in every Arduino program. A *function* is a piece of code that does a specific task. Code within the curly braces of the `setup()` function is executed once at the start of the program. This is useful for one-time settings, such as setting the direction of pins, initializing communication interfaces, and so on. In this program, it will configure the pin that connects to the LED as an output, because you will be telling the pin to do something, instead of querying the pin to determine its state.
4. **Line 27:** The Arduino's digital pins can all function as inputs or outputs. To configure their direction, use the command `pinMode()`. All pins default to inputs unless you explicitly tell the Arduino to treat them as outputs. Defining a pin as an output during the `setup()` will mean that the pin stays configured as an output for the duration of the program execution (unless you explicitly change it again in the main loop). Set a pin as an output to assign a value to it (5V or 0V in the case of a digital pin on a 5V board like the Uno). Set a pin as an input if you want to "read" the value being applied to it. You'll explore these concepts more in the next chapter.

   The `pinMode()` command takes two arguments. An *argument* gives commands information on how they should operate. Arguments are placed inside the parentheses following a command. The first argument to `pinMode()` determines which pin is having its direction set. For instance, you could simply specify `13` as the first argument, because the onboard LED is connected to pin 13 on the Uno. However, the Arduino language has a number of built-in defined words. These words enable one Arduino program to be abstracted to a variety of different hardware based on what board you've told the IDE you are using. The Arduino compiler converts these special words to specific instructions depending on your hardware. For instance, `LED_BUILTIN` is a special word that the compiler converts to the pin number of the built-in LED on your board. On the Uno, this gets converted to "13." On the MKR1000, this gets converted to "6" because the LED is connected to those pin numbers on those boards. By using this special word instead of just writing the pin number, you ensure that your program is *portable*, meaning it can be executed on various types of Arduino hardware. In the next chapter, you'll learn about variables, which are special words that you define yourself to assign a meaningful name to numbers, text, and other data.

   The second argument to `pinMode()` sets the direction of the pin: `INPUT` or `OUTPUT`. These are additional special predefined words that the compiler uses to

configure the MCU onboard your Arduino. Because you want to light an LED, you have set the LED pin to an output (when configured as an output, a pin can "source" or "sink" current by toggling internal switches called transistors).

5. **Line 31:** The second required function in all Arduino programs is `void loop()`. The contents of the `loop` function repeat forever as long as the Arduino is on. If you want your Arduino to do something once at boot only, you still need to include the `loop` function, but you can leave it empty.
6. **Line 32:** `digitalWrite()` is a command that is used to set the state of an output pin. It can set the pin to either 5V or 0V. When an LED is connected to a pin (through a current-limiting resistor), setting it to 5V will enable you to light up the LED. (You will learn more about this in the next chapter.) The first argument to `digitalWrite()` is the pin you want to control. The second argument is the value you want to set it to, either `HIGH` (5V) or `LOW` (0V). The pin remains in this state until it is changed later in the code.
7. **Line 33:** The `delay()` function accepts one argument: a delay time in milliseconds. When calling `delay()`, the Arduino stops doing anything for the amount of time specified. In this case, you are delaying the program for 1000 ms, or 1 second. This results in the LED staying on for 1 second before you execute the next command.
8. **Line 34:** Here, `digitalWrite()` is used to turn the LED off, by setting the pin state to `LOW`.
9. **Line 35:** Again, you delay for 1 second to keep the LED in the off state before the loop repeats and switches to the on state again.

That's all there is to it. Don't be intimidated if you don't fully understand all the code yet. As you put together more examples in the following chapters, you'll become more proficient at understanding program flow, and writing your own code.

Summary

In this chapter, you learned about the following:

- All of the components that comprise an Arduino board
- How the Arduino bootloader allows you to program Arduino firmware over a USB connection
- The differences between the various Arduino boards
- How to connect and install the Arduino with your system
- How to load and run your first program