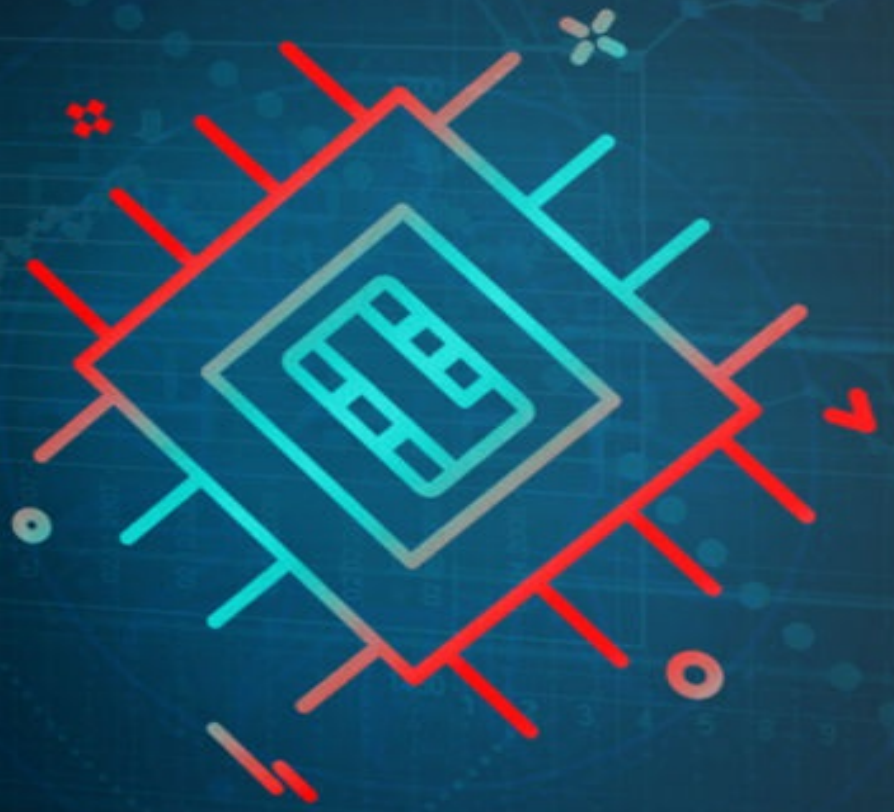
ARDUINO PROGRAMMING

THE ULTIMATE BEGINNER'S GUIDE TO LEARN ARDUINO PROGRAMMING STEP-BY-STEP



MARK REED

ARDUINO PROGRAMMING

THE ULTIMATE BEGINNER'S GUIDE TO LEARN ARDUINO PROGRAMMING STEP BY STEP

Mark Reed

TABLE OF CONTENTS

INTRODUCTION

Programming has always fascinated a lot of people. The idea of typing in a language that only a select few can understand alone seems cool enough. Add to that the ability to do so much more, such as creating softwares, games, robots, and working with artificial intelligence (AI), you can see why one would be so eager to get started. However, there is a problem; a lot of us who wish to learn programming do not have a relevant degree. What can be done then?

Well, there is good news, and there is even better news. The good news is that programming languages are many in number, meaning you can get started with any one and understand most right away. The better news, however, is that you can get started with something called Arduino; a perfect starting point for those interested in typing codes, creating programs, and physically programming things.

Now, I know a lot of you may not have heard about Arduino before, and if that is the case, do not worry. Arduino is essentially an open-source electronic interface that comprises two parts. The first one is a programmable circuit board (think a canvas), and the other part is the coding program you choose to work with on your computer.

“I get to choose a program?”

Yes! You do. There are many programs, better known as Integrated Development Environments or IDEs, that you can download right now and get started. Most of these, if not all, are absolutely free and do not require you to have a super-computer to operate. Even the most basic machines of today would get the job done nicely.

So, you have a circuit board, and you have an IDE. Can you connect the two together? Yes, you can. By doing so, you end up with a way to use your computer to write in a coded program that is then interpreted and installed

into whatever is on the circuit board. Makes sense, right?

For beginners, we generally prefer that they start with a simple setup and a program called “Hello World!” If you happen to be a student of computer sciences, you’ll probably be smiling right about now.

For any programmer, “Hello World!” marks the start of their programming journey. This is essentially the first ever successful output they get by creating some kind of program that, when run, displays these words out to the user. However, since we are using a circuit board, we may not be able to see the words printed on the screen. What we can do, however, is to create a light show. I kid you not, it may not be the most dazzling display of lights, but for a programmer, it equates to that. For a second, you can have the circuit turn an LED on and then have it turn itself off the next second. It’s not exactly the most interesting thing to do but when you get to look at the kind of code you write to do that, a sense of curiosity automatically takes over. Ready to see what a typical code would look and feel like? Here it is!

```
Const int PinkL = 13;
Void setup ()
{ pinMode (PinkL, OUTPUT); }
Void loop ()
{digitalWrite(PinkL, HIGH);
 delay (600);
digitalWrite(PinkL, LOW);
delay(600); }
```

“What on earth is that?” Intimidated? Don’t be.

For most of you, it may be just plain gibberish. Do not worry because later in the book, I will explain what each of those coded lines mean and what these symbols do. You may not be able to make much sense of it right now, but let me give you an idea of what will happen when you upload this program into a connected Arduino device.

It will essentially turn on an LED for 600 milliseconds, followed by another 600 milliseconds of it turning itself off. This will continue for as long as there is power in the Arduino.

Already, you can see that Arduino is a treasure chest of possibilities. You can

create an array of things using the Arduino setup. All you need is to get the right kind of Arduino. We will also be looking into that later within the book, but for now, let me quickly introduce you to some popular Arduinos in existence today. Each of these are designed to give you flexibility and programmability. These are:

- Arduino Uno
- LilyPad
- Redboard
- Arduino Mega
- Arduino Leonardo

We will go through these one by one, along with their market prices, features, and functionality. We will also look into how you can set these up and integrate them with your computer to get started.

What fascinates a lot of programmers about Arduino is the sheer number of possibilities that they can tap into. You can create simple digital clocks all the way up to complicated robots. Of course, this would mean that you must familiarize yourself well with the Arduino programming language. However, there is good news here, too.

Generally, a computer can only understand zeros and ones. We refer to this as binary language. If I tell a computer to turn itself off, it wouldn't be able to understand what I am trying to say. In order for a computer to do what I want it to do, the command must be converted into a binary code consisting of hundreds of zeros and ones. Only then will the computer actually carry out the command. Luckily for us, we do not have to worry about that because the folks at Arduino came up with their own programming language. This means that we can continue to type in English (well, mostly English) and the software we use (IDE) will convert the code itself into binary code. Add to that the fact that Arduino's programming language is well-documented. This means that if you find yourself lost or stuck, or have no idea on how to call a function, a variable, or execute specific commands, you can hop online and find what you are looking for instantly.

With all said and done, Arduino is a perfect stepping stone for beginners to get started on and learn to develop their programming skills. There is so

much that you can learn with Arduino and I intend to help you do just that. I will teach you all about Arduino, and I will do so in the simplest of languages to help everyone understand how it works, and how you, the reader, can go on to experiment with Arduino on your own.

Throughout the book, I will be using a bit of humor to make the otherwise monotonous world of programming come to life. I will also be encouraging you to try a few things out on your own, just to reinforce what you have learned so far in the book. I have ensured not to use terminologies that are too tough for beginners to understand. However, despite my efforts, expect a few to come your way. I promise you, I will explain each of these properly so that everything makes perfect sense. Even a tiny typo can cause the entire program to crash or your Arduino or microcontroller to die.

Some Basic Terms to Know

To get started, grab a notebook and a pen, and note down some of the most commonly used terminologies pertaining to Arduino. It is generally a great practice to write something because it helps you remember things more easily in the future.

- Microcontrollers – These are long black-colored chip-like things that have many legs (connector pins). These are integrated circuits that are basically tiny computers. By function, they store and run small and simple software programs. These are very low powered, meaning that they can run off batteries for days, but still offer computational power faster than any human mind can deliver.
- Loop – If you look at the code above, I essentially wrote what is called a 'loop' and it is exactly as it sounds. Whatever you put inside a loop, the program will continue running for as long as your Arduino is powered on.
- Potentiometer – This is a regulator or a device that allows you to choose specific output. Think about that circular switch you turn to slow down or speed up the ceiling fan.
- Circuit boards – Essentially, all Arduinos are circuit boards upon which you can place your microcontrollers and other hardware items and program them to function as per your requirements.
- Sketches – In the simplest terms, these are programs either created by you or by other developers. These sketches are uploaded to your Arduinos, allowing them to carry out the functions you want them to. You will come across this term quite a lot, so it is a good idea to note this one down in bold letters.
- Shields – These are essentially additional circuit boards that are plugged into your main Arduino, hence allowing you to do more. For example, you can use Adafruit Wave Shield to control motors and servos without designing additional motor circuitry.
- Breadboard – I assure you, it has nothing to do with bread. A

breadboard is essentially an additional component, a large one at that, that is used to build circuits upon and test them out before deciding to finalize any circuit design. It comprises many holes that allow ICs and resistors to be plugged in, allowing you to design your circuits and test them efficiently.

So far, so good. You now have an idea of what you are about to deal with. It may sound intimidating, but as we go along, things will start to make sense. There will be a lot of coding that will be involved, which is why I would highly recommend to practice as we move along. I will be talking in detail about how to download the related software, install it and the Arduino, and then run the first successful program. I will then start taking you deeper into understanding every component of the Arduino and how you can go on to write great programs and logic.

If, for any reason, you find yourself stuck or unable to understand, simply take a break and freshen up a little. I do not encourage going through this book, cover to cover, in a single go. It is humanly impossible for anyone to learn a programming language and be able to understand everything in a single go. It takes time, and above all, it takes consistent practice. It would also help if you refer back to previous sections or chapters, just to refresh your memory about certain concepts.

Now that you have your eye in, it is time to get started and understand what Arduino truly is. Of course, you may have questions at this point, but everything will start making sense as we move forward with the book. With that said, let’s get started and learn all things Arduino.

CHAPTER 1:

WHAT IS ARDUINO?

There is no denying the fact that with the advent of the internet, people from all walks of life are trying to develop skills and knowledge in the world of software and hardware. Gone are the days when we would look at an integrated circuit (IC) and immediately look the other way. Now, a lot of us find ourselves drawn to these little things. Now, we are far more fascinated about what goes on within these tiny chips and circuits not only because they are interesting, but also because the internet has allowed us to understand everything with the click of a button.

The race of technology certainly benefits a lot of mega corporations around the world, but it is us, the people, who are truly in for a life changing experience. We are passing through a time where we are stumbling upon something new every day. As it happens, Arduino is now one of them. While Arduino has existed for a while, people are now taking Arduino more seriously than ever before. Most of these people are hobbyists who would love to create something from scratch and impress their friends and family members. However, to some of us, it is a potential career in the making. Whatever your reason may be, we are all in this together, and together, we shall learn the basics of Arduino programming. But before we get started, we must address the obvious elephant in the room:

“What on earth is Arduino?”

To fully understand what Arduino is, I will have to create a new section because, quite frankly, Arduino refers to a lot of things.

History of Arduino

It all started in 2005, when the world was introduced to the first-ever Arduino board. This was created in a classroom of the Interactive Design Institute in the city of Ivera, Italy. From there, Arduino became an instant hit.

Today, Arduino is a brand name, a company that operates from Italy. It is an Open-Source microcontroller-based development board, and this, single-handedly, has brought creativity to a completely new level for electronics and creative engineers (CircuitsToday.com, n.d.).

The brilliant mind behind this was Hernando Barragan. This was submitted as a thesis with the title “Arduino- La rivoluzione dell’open hardware.” In English, that means “Arduino – The revolution of Open Hardware.” Well, it was indeed everything the title said it would be. For a thesis, the title may have sounded a little too optimistic, but what Barragan ended up doing was carve out a niche for a completely new technology that he created.

A total of five developers put their heads together and worked on this thesis by the brilliant student. This led them to install a new wiring platform. Once they completed that, they focused on ensuring that this technology was made available at a much more affordable price to the open-source community, and boy did the community welcome this.

Soon, Massimo Banzi, along with other founders, founded Arduino, the company that uses the same technology, and produces a variety of circuit boards for microcontrollers. These circuit boards, unsurprisingly, are called Arduinos. To make matters a little complicated, you have a massive line up of Arduinos sold by Arduino itself.

Now you might imagine, why should one use an Arduino and not program a microcontroller themselves? It is a good question, but it does have a great answer.

Not so long ago, if you wanted to program a microcontroller, you had to learn a lot of things. To begin with, you would need to type in thousands of binary codes and had to memorize a lot of registers and instructions, and let me tell you, it was no walk in the park. These were some of the most confusing and toughest lines of codes to write. Then, you would need specialized

programming hardware, that too with custom cables, allowing you to upload these programs into a microcontroller. If you ask me, this is far too intimidating and cumbersome.

Arduino stepped in and removed all of that. They created a software of their own that is able to work on Linux, Mac, and Windows, allowing easy access to anyone on any operating system. Then, they created their own programming language and released detailed documentation that helps any programmer on earth to know what they are supposed to type, how, and why. You can use the programming language to type in all kinds of instructions in a very easy way. While it may not be as easy as Python, which is a completely different programming language, it is still easy enough for beginners with no background knowledge about programming to understand and use.

The name ‘Arduino’ came from a bar that Banzi frequently visited. When working on the thesis, he was the one who suggested the name Arduino to this revolutionary new low-cost microcontroller board, as a way to honor the place.

Before Banzi went on to reserve a place in history, he was recruited by IDII as an associate professor in the year 2002. His job was to promote novel ways of interactive designing. In layman’s terms, he was working with physical computing. Since Arduino was not yet invented, he had to rely on something called BASIC Stamp. This was a microcontroller that was developed by Parallax, a company based in California. This was a popular choice at that point in time and was in use by engineers around the world for around a decade. The microcontroller could be programmed using BASIC programming language, and it offered a power supply, memory, input/output ports and a microcontroller. Other hardware could easily be attached to this unit.

Although Banzi was not adamant on using this microcontroller, owing to his lack of budget and limited hours in the class, he still soldiered on. However, he noticed two major issues with BASIC Stamp:

1. It did not offer enough computing power to power up some projects.
2. It was rather expensive

Just the board with some basic parts was up for $100, and that is a lot. Add to that the fact that designers used Macintosh systems and this wasn't exactly the kind of a unit that supported Mac. It was at this point that the concept of creating something new came into his mind. This is where the genesis of Arduino began.

Ironically, Hernando Barragan, the Colombian student who wrote the thesis, was Banzi's student. He took the first step in creating a software that was similar to processing, a new prototype platform that was called wiring. These included a very user-friendly software and a ready-to-go circuit board. The project laid the foundation upon which Arduino was built. He simply took the idea and wanted to make it more affordable and readily available within the open-source community.

In 2005, Banzi, using the same thesis, successfully created his first prototype board. At that point in time, it wasn't exactly called Arduino, but the name was coined the same year later on. At this point, both Banzi and his partners were inclined to create an open-source software. The idea of opening up their software for the entire world was a thrill because that would allow many developers around the world to chip in and collaborate in making this new IDE a major success. However, that posed a bit of a problem.

You see, the open-source model generally caters to software, not hardware. Here, Banzi and team had both to offer. To make things work, they had to find a licensing solution that could apply to their board. After much deliberation, the team decided to opt for a license from Creative Commons, which is a non-profit group that generally issued agreements for cultural works, such as music and writing. Banzi believed that a piece of hardware is a piece of culture, and that it must be promoted and shared with the rest of the world.

Once that was sorted, they got on the job of bringing the cost down. They targeted to sell their boards at $30 a piece, a price most students could comfortably pay. To further add to the value, they changed the color of the board to blue as opposed to the traditional green color. Unlike other manufacturers, who quite often saved on input and output pins, Banzi and company decided to provide these for their potential buyers. What they ended up creating was a board that was nothing like the traditional engineering

board but one that resembled a DIY project people could use.

To test the functionality of the board and see its potential, the team gave 300 blank printed circuit boards to their students in IDII. The objective was simple: understand the assembly instructions on the internet, build their own unique boards, and create something truly unique. What they ended up with was a number of impressive projects. This new piece of technology struck a chord with people instantly. Now, everyone wanted one for themselves. Everything started going their way, and that is when they decided to make it into an official business entity that produces these circuit boards for people around the world. They named it Arduino (CircuitsToday.com, n.d.).

Supervised by Massimo Banzi and Casey Reas, Barragán worked in the computer language called Processing to create the environment, IDE (Arduino's official coding environment and program). He fiddled with the Wiring platform technology to come up with the hardware called ATmega168, the first Arduino microcontroller.

Who Uses Arduino?

Frankly, Arduino is so much fun that practically anyone can use it. Considering the fact that you gain access to something completely new and exciting, the potential is virtually limitless. This means that you can be an engineer, a hobbyist, a beginner, or a veteran software developer; Arduino is fit for all. In fact, if you look it up, kids are using Arduino to create some truly fabulous projects and simple robots.

Due to the nature of Arduino, and it being simple and easy to use, a wide array of people use Arduino for various projects and hobbies, as well as for professional uses. For beginners, who may have no idea how to write code or program anything, Arduino offers deep and rich learning opportunities for beginners to learn from and grow.

Arduino was initially released for teachers and students, and indeed they still are the intended consumer base for the products, as Arduino offers a low-cost way to build scientific instruments. However, with time, hobbyists, YouTube streamers, and many others have joined in and taken up Arduino as a new and creative way to create elaborate designs and items such as alarm clocks, security systems, IP cameras, and some interesting robots.

On the other hand, we have designers and architects who use Arduino technologies to build their interactive models and prototypes of what they hope to develop to a full-scale project. If that wasn't enough, musicians and artists also use Arduino microcontrollers to experiment with new instruments or techniques in their art.

Therefore, stating that Arduino is only limited to a specific niche would be wrong. Everyone, from kids to adults, can use Arduino products and get started.

The Six Advantages of Arduino

- The driving force behind creating Arduino microcontrollers was cost-efficiency. As opposed to the $100 price tag that many boards would generally hold, a pre-assembled Arduino board costs less than $50, and then there are boards that can be manually put together that cost even less.
- The Arduino environment, IDE, works across different platforms. This takes away the limitation that most other competing products are stuck with. Whether you are running Windows, MacOS, or even Linux, Arduino IDE will work like a charm. This opens up the use of microcontrollers to the Apple user and the open-source Linux user.
- The software for Arduino is open-source. The tools, or strings of code that you use to instruct the microcontroller to accomplish certain functions, are accessible by anyone. This isn't something to be alarmed about. If anything, this is even more encouraging because the open-source community is always ready to help you out when you feel stuck. It also means that you do not have to purchase a license to use these tools either.
- The open-source tools are also extendable by the C++ libraries and the AVR-C coding language, meaning that those with more in-depth knowledge of code would be able to benefit from using these technologies as well. While there is depth for more experienced programmers to explore, it still remains fairly easy to use for beginners to tinker with and experiment.

- The IDE for Arduino, apart from being free of cost, is smooth and does not burden your computer with unnecessarily high demands. You can create programs upon programs and it will still work brilliantly.
- Due to the fact that Arduino is also an open-source hardware, anyone who desires to create their own designs and boards can do so and use them with Arduino software programming in the IDE. Even those who are not experienced circuit designers can use a breadboard to create their own Arduino circuit-board.

Next, let us look at some important things that we must understand in order to start learning about Arduino and how to program things.

CHAPTER 2:

KEY ASPECTS IN UNDERSTANDING ARDUINO

So far, we have learned what Arduino can theoretically do. I know that many of you are quite eager to get started and write some codes, but we will get to that bit soon. The reason I cannot jump right into the action sequence is because we still need to familiarize ourselves with a bundle of things Arduino-related.

Arduino, just like any other programming language, comes packed with terminology. Do not worry, as most of it makes sense the second you read it. However, just because it is easy to understand does not mean that we should skip it.

To begin with, you will need to understand certain terminology to help you choose a board. There are many that are available, and I have seen a lot of people ending up buying the wrong kind of board. Then, we will move onto coding, where we will see how we can write coded instructions for the computer to transfer to the Arduino, set up the microcontroller for use, and bring our programs to life. In this chapter, you will find some key terms that will aid you greatly in your endeavor to become an Arduino user.

Arduino is based on an easy-to-use hardware, which is the actual physical computer board that you will be working with. Additionally, the IDE is where all the coding magic will happen. These codes will instruct the hardware to perform a variety of tasks. The software is also known as code, and the individual pieces of instructions are called tools. You may want to remember that for future reference.

With that said, let us start looking into the Arduino board itself and see what the fuss is all about.

Anatomy of the Board

The board itself contains a good number of parts. The digital pins run along the edges of most Arduino microcontrollers and are used for input, or sensing of a condition, and output, which is essentially the response that the controller gives to the input. These are clearly marked with numbers starting from ‘0’ and often going to 13 (in the case of Arduino UNO), as well as the word ‘digital’ above a solid white line. Some of these numbers have a tilde (~) sign before them, but that is something we will skip for now as it will cause confusion. Just know that these pin numbers can access special features.

For the input and output pins, you will use them every time you use the Arduino board to input commands and gain results. For example, the input instruction might direct Arduino to dim the light intensity on the LED. The output will then do the following and dim the light as per instructions. What you end up with is a nightlight for the younger folks.

On most boards, there will be a pin LED, associated with a specific pin. This is generally next to pin 13, especially if you are using an Arduino UNO board. This pin LED is the only output possibility built into the board, and it will help you with your first project of a “blink sketch,” which will be explained later. The Pin LED is also used for debugging or ‘fixing’ the code you have written, in case you run into an error.

Then, we have the power LED, and that is exactly what you think it is: it lights up when the board is receiving power or is “turned on.” This can also be helpful in debugging your code.

Of course, just as every human body needs a mind to function, there exists on every board the microcontroller itself, called the ATmega microcontroller. This, for all intents and purposes, is the brain of the entire board. Take this away and your Arduino board is practically useless. This is the part that receives your instructions and acts accordingly.

Then, on the opposite edge of the digital pins, you have what are called “Analog in” pins. These are inputs into the Arduino system. Analog means that the signal is not constant but instead varies with time, such as audio

input.

Next, we have the GND and 5V pins, which are used to create additional power of 5V to the circuit, the microcontroller, and provide ground to your circuits.

The power connector is next, and this is most often on the edge of the Arduino board. It is used to provide power to the microcontroller when it is not connected to the computer via the USB. While the USB port can be used as a power source as well, its main function is to upload, or transfer, your sketch or set of instructions that you have coded, from your computer to the Arduino board.

Moving along, we have the TX and RX LEDs. These are used to indicate an active communication between the Arduino board and your computer. The lights will flicker rapidly when you upload your sketches from your computer to the Arduino. They are also extremely helpful for debugging purposes.

Next to that we have the USB port, and we already know what that does. It is through this port that you can attach a USB cable and connect it to your computer, exchange information, have sketches transferred and uploaded to the Arduino board, and even provide power.

Finally, we have the reset button. It simply resets the microcontroller to factory settings and erases any information you have uploaded to the Arduino.

That is it really; the entire Arduino board explained. However, that is not all. There are many other things that we will come across to further help us understand Arduino better.

Additional Things to Know

There are three types of memory in an Arduino system. For those who may not know what memory means, memory is space where information of any kind is stored. These are then recalled when required or instructed by the program itself. It is just like you meeting someone new. The first time, you may ask for their name, but the next time you meet the person, your mind will immediately call upon the stored information and the words, or the output, will come out automatically.

The first type is called flash memory. Flash memory is where the code for the program that you have written is stored. It is also called the “program space,” because it is used for the program automatically when you upload it to the Arduino. This type of memory remains intact when the power is cut off, or when the Arduino is turned off.

Next, we have SRAM. The term SRAM stands for Static Random-Access Memory, and it is the space used by the sketch or program you have created to create, store, and work with information from input sources to create an output. This type of storage is removed once the power is turned off.

Finally, we have the EEPROM. Electrically Erasable Programmable Read-Only Memory, or EEPROM, is like a tiny hard-drive that allows the programmer to store information other than the program itself when the Arduino is turned off. There are separate instructions for the EEPROM for reading, writing, and erasing, as well as other functions. This type of memory does not lose information when Arduino is powered off.

Apart from the memory types, there are other things to learn about the Arduino before jumping into the world of coding. Certain digital pins will be designated as Pulse Width Modulation or PWM pins, meaning that they can create analog results using digital means. Analog, as we remember, means that input (or output) is varied and not constant. Normally, digital pins can only create a constant flow of energy. However, PWM pins can vary the "pulse" of energy between 0 and 5 volts. Certain tasks that you program can only be carried out by using PWM pins.

In addition to the above, in comparing microcontroller boards, you will want to look at clock speed, which is the speed at which the microcontroller operates. The faster the speed, the more responsive the board will be, but do bear in mind that such boards will consume more battery or energy.

You will also come across something called UART. The UART is a serial port that is responsible for communicating between the computer and your Arduino board using serial communication lines. These lines are communication lines that transfer data serially, that is, in a line rather than in parallel or simultaneously. It requires less hardware to process things serially than in parallel.

Some projects will have you connecting devices to the Internet of Things,

which essentially describes the interconnectedness of devices, other than desktop and laptop computers, to various networks in order to share information. Everything from smart refrigerators, to smartphones, and smart TVs are all connected to the Internet of Things (IoT).

With that sorted, it is now time to get started with Arduino programming. First things first, we must find out what possible options we have before buying our very first Arduino board.

CHAPTER 3:

UNDERSTANDING YOUR CHOICES

We are pretty much set to get started. You might have even logged on to Amazon or visited Arduino's own website, and straight away, you would have found yourself in a big problem. There are tons of variants out there, each providing something unique or different. The obvious question here is which one to go for. To help you with that, let us look at some of the most popular choices in the market and see what's what.

The Types of Arduinos

Uno

This, perhaps, is the finest board to start with. Generally, every beginner starts here, but that isn't a rule of thumb that everyone must follow. It may not offer the smartest of features, but once you get started, it is very hard to outgrow it. You can use this over and over again to get most of your projects done.

It is on the smaller side in terms of memory, but it is very flexible in functionality and a great tool for those wanting to give Arduino a try. This model has a mini-USB port which allows you to upload directly to the board without using a breakout board or any other extra hardware. Here are the facts and specs you should know about the Arduino UNO.

Side note: All prices are verified from Arduino.cc which is the official website for Arduino. The prices reflected here are what were quoted at the time of writing this book and do not include any applicable taxes. I do not make any guarantee that the price will remain the same later.

Price: $23.00

Flash Memory: 32kB

SRAM: 2kB

EEPROM: 1kB

Processing Speed: 16MHz

Digital Pins: 14 pins

PWM Pins: 6 pins

Analog In: 6 pins

Operating Voltage: 5V

Input Voltage (Recommended): 7-12V

Leonardo

The Leonardo microcontroller board is fully functional right out-of-the-box.

All you need is a micro-USB cable and a computer to get started. In addition, the computer can recognize the Leonardo as a mouse or a keyboard due to its ATmega32U4 processor. This offers a little more pins, hence increasing the capacity of workload it can handle.

Price: $20.70

Flash Memory: 32kB

SRAM: 2.5kB

EEPROM: 1kB

Processing Speed: 16MHz

Digital Pins: 20 pins

PWM Pins: 7 pins

Analog In: 12 pins

Operating Voltage: 5V

Input Voltage (Recommended): 7-12V

Nano

While Arduino boards are actually small, there are those who still prefer something a little smaller, something a little more discrete. If you are such a person, Arduino Nano is the board for you. It is one of the smallest boards and still offers impressive features and functions. Do not be fooled by the size here either, because this board is also compatible with a breadboard. The only downside is that it does not offer you a DC power jack. This means that you will only have a Mini-B USB cable as opposed to a standard one.

Price: $20.70

Flash Memory: 32kB

SRAM: 2kB

EEPROM: 1kB

Processing Speed: 16MHz

Digital Pins: 22 pins

PWM Pins: 6 pins

Analog In: 8 pins

Operating Voltage: 5V

Input Voltage (Recommended): 7-12V

Micro

While this may be just a little bigger than nano, it is still quite small. However, unlike the nano, it comes with a built-in USB that allows Micro to be detected by your system as a mouse or a keyboard. This board is similar to Leonardo, but there are some differences.

To begin with, this board was developed in conjunction with Adafruit, another company that excels in creating Arduino-related products, such as shields. It offers an impressive number of digital input/output pins, a 16 MHz crystal oscillator, a reset button, and some other impressive features. It is, therefore, a good choice if you do not wish to settle for Arduino UNO.

Price: $20.70

Flash Memory:32kB

SRAM: 2.5kB

EEPROM: 1kB

Processing Speed: 16MHz

Digital Pins: 20

PWM Pins: 7

Analog In: 12

Operating Voltage: 5V

Input Voltage (Recommended): 7-12V

Mega 256O

The Mega is an 8-bit board that offers 54 digital pins, four serial ports, and 16 analog inputs, and that is quite a resume. However, before you jump on to this one, know that this microcontroller is primarily designed for larger

projects like robotics and 3D printers. If you have projects that require detailed instructions, and a lot of them, this may be the kind of board you are looking for. There is space for greater complexity and specificity in this Arduino board.

Price: $40.30

Flash Memory: 256kB

SRAM: 8kB

EEPROM: 4kB

Processing Speed: 16MHz

Digital Pins: 54 pins

PWM Pins: 15 pins

Analog In: 16 pins

Operating Voltage: 5V

Input Voltage (Recommended): 7-12V

UART: 4 lines

Zero

Do not be fooled by the name. This one packs quite a lot of enhanced features as well. It may look simple enough, but it has a powerful 32-bit extension of the same platform established by the renowned UNO. It increases performance with a vastly increased processing speed, 16 times the amount of SRAM and a high flash memory. You will pay for the extensions, at almost twice the price of the UNO, but you also double your capabilities with this hardware.

One other advantage of the Zero is that it has a built-in feature called Atmel's Embedded Debugger, abbreviated as EDBG, which helps you debug your code without using extra hardware and thereby increases your efficiency in software coding.

Price: $42.90

Flash Memory: 256kB

SRAM: 32kB

EEPROM: n/a

Processing Speed: 48MHz

Digital Pins: 20 pins

PWM Pins: All except ‘2’ and ‘7’

Analog In: 6 pins

Analog Out: 1 pin

Operating Voltage: 3.3V

Input Voltage (Recommended): 7-12V

UART: 2 lines

USB port: 2 micro-USB ports

Due

I wouldn’t be wrong to say that this is a novelty in the microcontroller board world, and for very good reasons, too. It is the first Arduino board that is based on a 32-bit ARM core microcontroller, hence there is a great deal of power and functionality. There is no questioning the extremely quick processor that it comes with and the four UARTs, giving it significant flexibility as well as availability to perform a multitude of functions. It is generally used for larger scale projects. While this may not be your first Arduino board, you can certainly keep tabs on this one, especially if you have something bigger planned for the future.

Price: $40.30

Flash Memory: 512kB

SRAM: 96kB

EEPROM: n/a

Processing Speed: 84MHz

Digital Pins: 54 pins

PWM Pins: 12 pins

Analog In: 12 pins

Analog Out: 2 pins

Operating Voltage: 3.3V

Input Voltage (Recommended): 7-12V

UART: 4 lines

Nano Every

This is perhaps the cheapest of the lot, and a perfect board if you are someone who is not willing to spend more. It is understandable that many may want to try these out first before opting to choose a higher-end model, and if that is the case, Nano Every is a perfect choice.

It is arguably the smallest board by a country mile. It is highly recommended for people looking to buy boards that are small and easy to use. This is also an ideal board to use if you are trying to work on wearable inventions, electronic musical instruments, and low-cost robots, among many other things. However, despite it being the smallest, it still offers you 50% more processing power than UNO. Add to that the fact that the RAM is 200% higher. Now that's what I call “interesting.”

Price: $10.90

Flash Memory: 48kB

SRAM: 6kB

EEPROM: 256 bytes

Processing Speed: 16MHz

Digital Pins: n/a

PWM Pins: 5 pins

Analog In: 8 pins

Operating Voltage: 5V

UART: 1 line

LED Built-in: 13

Arduino Nano 33 BLE (Without Headers)

Measuring in at 45x188mm, this is yet another tiny entry into the impressive lineup of Arduino products. What makes this one special is the fact that it comes loaded with a Bluetooth Low Energy and an embedded inertial sensor. You don't get that in any of the boards I have mentioned above, and that makes this one highly interesting. However, I wouldn't exactly call this one a beginner-friendly board.

While it does pack in quite some power, it is generally aimed for designers creating wearable inventions or for science projects that require short-distance communication. The main problem, however, is the fact that you will need to do quite a bit of soldering, and that means that once it's done, it's permanent.

Price: $20.20

Flash Memory: 1MB

SRAM: 256kB

EEPROM: None

Processing Speed: 64 MHz

Digital Pins: 14 pins

PWM Pins: All pins

Analog In: 8 pins

Operating Voltage: 3.3V

Input Voltage (limit): 21V

Arduino Nano 33 BLE Sense

If you were impressed by Nano 33 BLE alone, wait until you get to know this variant.

While the Nano 33 BLE is one of the finest boards in existence, the 'Sense'

variant takes things up a notch by adding the power of Artificial Intelligence to the board. Yes, not only do you enjoy the benefits of the Nano 33 BLE, but now you can get your hands on AI for your projects. This means that it comes pre-loaded with quite a few sensors. It has:

- An impressive 9-Axis inertial Sensor, making this an ideal board for those working with wearable devices
- Offers temperature and humidity sensors
- Comes with barometric sensors, allowing you to make simple weather stations (now that's an idea worth exploring)
- It even has a microphone, to allow and capture real-time audio for analysis
- Light intensity sensor
- Proximity sensor
- Light color sensor
- Gesture sensor

I know what you are thinking, once again this isn't ideal for beginners, but if you still wish to give it a try, go for it!

Price: $31.10

Flash Memory: 1MB

SRAM: 256kB

EEPROM: None

Processing Speed: 64 MHz

Digital Pins: 14 pins

PWM Pins: All pins

Analog In: 8 pins

Operating Voltage: 3.3V

Input Voltage (Limit): 21V

Arduino Starter Kit

For those of you who are still unable to figure out which Arduino board to settle for, you also do have an option that Arduino itself offers for new timers. It is the Arduino Starter Kit, and I assure you, it offers a massive number of things for you to tinker with. The package itself comes with all of the following items:

- Project Book (170 pages)
- One USB cable
- One Arduino UNO
- One Breadboard (400 points)
- 70 Solid core jumper wires
- One easy-to-assemble wooden base
- One 9V battery snap
- One stranded jumper wire (black)
- One stranded jumper wire (red)
- Six phototransistors
- Three potentiometers (10k Ohms)
- 10 push buttons
- One temperature sensor
- One tilt sensor
- One Alpha-numeric LCD
- One LED (bright white)
- One LED (RGB)
- Eight LEDs (red)
- Eight LEDs (yellow)
- Eight LEDs (green)
- Three LEDs (blue)
- One small DC motor 6/9V
- One small servo motor

- One piezo capsule
- One H-bridge motor driver
- One Optocoupler
- Two Mosfet transistors
- Five capacitors (100uF)
- Five Diodes
- Three transparent gels (green, red, blue)
- One male pins strip
- 20 resistors (220 ohms)
- Five resistors (1 kOhms)
- Five resistors (4.7 kOhms)
- 20 resistors (10 kOhms)
- Five resistors (1 MOhms)
- Five resistors (10 MOhms)

See what I mean? This is a perfect way to get started. Price: $91.90

Other Boards

You aren’t just limited to buying directly from Arduino’s own website. You can still find a lot of other ones in the market. However, some of these may have stopped production officially, but they still work and are still being used by many project makers. You can find these on Amazon, eBay, and many other places, and some other third-party manufacturers. Even if you settle for something that isn’t branded as Arduino, you should still be able to get the functionality that Arduino offers. The board and every component in every Arduino board is open-source, meaning that other manufacturers use these as well. It is only the name ‘Arduino’ that cannot be used by others. With that said, let’s explore some other viable options.

Side note: I have mentioned the word ‘retired’ for the rest of the list. This does not mean that these are no longer worth using, but they are no longer produced by Arduino.

Arduino M0 (Retired)

This board was an extension of Arduino Uno, giving the Uno the 32-bit power of an ARM Cortex M0 core. If you intend to get one, know that this will not be your first board, but it might be your most exciting one yet for any given project.

It will allow a creative mind to develop wearable technology, make objects with high-end automation, create yet-unseen robotics, come up with new ideas for the Internet of Things, or many other fantastic projects. Surely enough, it is a worthy extension of the technology Uno offers, and thus it has the flexibility to become almost anything you could imagine.

Price: $22.00 (may vary from seller to seller)

Flash Memory: 256 kB

SRAM: 32kB

Processing Speed: 48MHz

Digital Pins: 20 pins

PWM Pins: 12 pins

Analog In: 6 pins

Operating Voltage: 3.3V

Input Voltage: 5-15V

Arduino M0 Pro (Retired)

This was the same extended technology of the Uno as the Arduino M0, but it came with the added functionality and capability of debugging its own software with the Atmel's Embedded Debugger (EDBG) integrated into the board itself. That, for all intents and purposes, was a reason alone pushing a person like me to buy one.

If you intend to get one of these, it is worth the money. However, I would not recommend it if the board is not new in condition. There is no way of telling if the board works 100% through pictures.

Price: $42.90 (may vary)

Flash Memory: 256 kB

SRAM: 32kB

Processing Speed: 48MHz

Digital Pins: 20 pins

PWM Pins: 12 pins

Analog In: 6 pins

Operating Voltage: 3.3V

Input Voltage: 5-15V

Arduino YÚN (based on ATmega32U4) (Retired)

The Arduino YÚN is, by all means, a great board to use when connecting to the Internet of Things. It is an ideal board to go for if you want to design a device connected to a network. It has a built-in ethernet support, which can give you a wired connection to a network, as well as Wi-Fi capabilities, allowing you to connect wirelessly to the Internet.

Price: $68.20 (may vary)

Flash Memory: 32kB

SRAM: 2.5kB

EEPROM: 1kB

Processing Speed: 16MHz

Digital Pins: 20 pins

PWM Pins: 7 pins

Analog In: 12 pins

Operating Voltage: 5V

Input Voltage: 5V

Arduino Ethernet (Retired)

This Arduino board was based on the ATmega328, the same microcontroller as the Arduino Uno. Pins 10 through 13 are reserved for interacting with the Ethernet, and as such, this board has less input/output capabilities than the Uno and other Arduino microcontroller boards.

It does not connect via USB, but rather through the Ethernet cord itself, which also has the option to power the microcontroller. However, unlike any other board in the list, this one packs one feature that sets this board apart from the rest; you can add more storage by using the built-in micro-SD slot. Just insert a card in and you are all set.

The method in which you upload your sketches to this board is similar to the Arduino Pro, and that is via an FTDI USB cable or through an FTDI breakout board. This Arduino model is more complex than most of the boards that we have already gone through, but it has functionalities that are not possible on other boards. That may not be something to explore at this point, but once you know what you are doing, you can always try to find this board and unlock more creativity.

Price: $43.89 (may vary)

Flash Memory: 32kB

SRAM: 2kB (it is reasonable)

EEPROM: 1kB

Processing Speed: 16MHz

Digital Pins: 14 pins

PWM Pins: 4 pins

Analog In: 6 pins

Operating Voltage: 5V

Input Voltage (Recommended): 7-12V

Arduino Tian (Retired)

Calling the Arduino's Tian a board would be somewhat wrong because it is essentially a miniature computer. It packs the power of a microprocessor right on top of a microcontroller. Furthermore, this one has Wi-Fi capabilities as well as ethernet connectivity. Of course, you do pay a higher price, but the small price difference is just overshadowed by the sheer benefits this offers.

Make no mistake, this is significantly fast. At 560 MHz, it stands out as one of the fastest boards, and with Bluetooth capabilities. What makes this one different, however, is the fact that this is based on Linux and OpenWRT.

Price: $95.70 (may vary from store to store)

Flash Memory: 256kB and 16MB flash + 4GB from microprocessor

SRAM: 32kB +64MB RAM

Processing Speed: 48MHz and another 560 MHz from the microprocessor

PWM Output: 12 pins

Analog In: 6 pins

Operating Power: 3.3V

Input Power: 5V

Industrial 101 (Retired)

Despite its name, this one wasn't made for industrial purposes. However, it certainly comes with some perks and a relatively smaller size. The price is

lower, too. The board offers built-in Wi-Fi capabilities, a familiar USB connection port, and if that wasn't enough, it also has one Ethernet port.

Price: $38.50 (may vary)

Flash Memory: 16MB (processor)

SRAM: 2.5KB (additional 64 MB DDR2 from processor)

EEPROM: 1kB

Processing Speed: 16MHz (additional 400MHz from processor)

Digital Pins: 20 pins

PWM Pins: 7 pins

Analog In: 12 pins

Operating Voltage: 3.3V

Input Voltage: 5V

Arduino Leonardo ETH Retired

Leonardo was nice, but this was just better. The fact that this one offered the power of Leonardo and gave us the much-needed USB and ethernet connectivity was just phenomenal.

Just like any other Arduino board with ethernet connectivity, this one also allows you to power the board using the ethernet connection. This is quite helpful in some projects that may require you to be a bit more creative.

Price: $43.89 (may vary)

Flash Memory: 32kB

SRAM: 2.5kB

Processing Speed: 16MHz

Digital Pins: 20 pins

PWM Pins: 7 pins

Analog In: 12 pins

Operating Voltage: 5V

Input Voltage: 7-12V

Gemma (Retired)

This Arduino is made by Adafruit Technologies in the USA. The Arduino Gemma is a miniature microcontroller board that is wearable in nature, owing to its circular and small design.

It indeed has less space and room for functionality than the non-wearable boards, but for many wearable projects, you will not need the robustness of some of the other Arduino microcontroller boards. There is a micro-USB connection on this board, so you do not need a breakout board or TKDI cable. Instead, you simply upload a sketch via the micro-USB connection and then power the microcontroller by micro-USB or by battery connection.

Price: $9.95 (may vary)

Flash Memory: 8kB (2.75 kB used by bootloader)

SRAM: 512 Bytes

EEPROM: 512 Bytes

Processing Speed: 8 MHz

Digital Pins: 3 pins

PWM Pins: 2 pins

Analog In: 1 pin

Operating Voltage: 3.3V

Input Voltage: 4-16V

Lilypad Arduino USB (Retired)

This board is round and based on the ATmega32u4 Arduino microcontroller. It contains a micro-USB connection for ease of uploading sketches and for powering the board. It also has a JST connector built in so that, should you decide to power the board by battery, you can connect a 3.7V Lithium Polymer battery.

The difference between the Lilypad Arduino USB and the rest of the Lilypad Arduino models is that the USB model contains a micro-USB port standard,

eliminating the need for a breakout board or TKDI adapter. In addition, the board can be seen as a mouse or a keyboard by the computer, among other things.

This board is intended to be worn, like the Gemma. It can be sewn into clothing or otherwise be attached to one's body to perform whatever function you have programmed it to perform.

Price: $25.95 (available on SparkFun.com)

Flash Memory: 32kB (4 kB used by bootloader)

SRAM: 2.5kB

EEPROM: 1kB

Processing Speed: 8MHz

Digital Pins: 9 pins

PWM Pins: 4 pins

Analog In: 4 pins

Operating Voltage: 3.3V

Input Voltage: 3.8-5V

Lilypad Arduino Main Board (Retired)

This is another wearable Arduino microcontroller board. It can be sewn into a piece of fabric or combined with other sensors, actuators, and a power supply to be something to carry with you, with the functionality you have programmed.

It requires a breakout board and TKDI cable to upload your sketch to the microcontroller's flash memory, but once you have that taken care of, you have an inexpensive, wearable device that you have created yourself from a scratch.

Price: $15.95 (available on SparkFun.com)

Flash Memory: 16kB (2 kB used by the bootloader)

SRAM: 1kB

EEPROM: 512 Bytes

Processing Speed: 8 MHz

Digital Pins: 14 pins

PWM Pins: 6 pins

Analog In: 6 pins

Operating Voltage: 2.7-5.5V

Input Voltage: 2.7-5.5V

Lilypad Arduino Simple (Retired)

This Arduino microcontroller board model differs from the Lilypad Arduino Main Board. It possesses only nine digital input/output pins, about two-third the number of pins on the Main Board model.

This is a good board for simpler projects that do not require as many inputs and outputs. It is also more powerful than the Main Board, having twice the flash memory, SRAM, and EEPROM.

Price: $21.95 (available on SparkFun.com)

Flash Memory: 32 kB

SRAM: 2 kB

EEPROM: 1 kB

Processing Speed: 8 MHz

Digital Pins: 9 pins

PWM Pins: 5 pins

Analog In: 4 pins

Operating Voltage: 2.7-5.5V

Input Voltage: 2.7-5.5V

Lilypad Arduino Simple Snap (currently sold out at Arduino.cc)

This is a more expensive version of the Lilypad Arduino Simple and is designed to create wearable devices and e-textiles. It solves an essential problem of the previous versions: washing the textiles. Unlike the other

models of Lilypad Arduino and the Gemma, one simply removes the power source and then hand washes the material with the microcontroller still embedded. Then, one must wait for the entire circuit to dry before powering back up, or else they risk ruining the technology.

With the Lilypad Arduino Simple Snap, the 9 pins for input/output are snappable buttons such that the microcontroller board can be removed from the material to which it is attached. That is ingenuity right there.

The Lilypad Arduino Simple Snap also has a built-in lithium polymer battery (LiPo battery), which can be recharged by attaching power to the charging circuit. The way this board is designed, it has the advantage of being able to detach and attach to a new project easily.

Price: $29.95 (available on SparkFun.com)

Flash Memory: 32 kB

SRAM: 2 kB

EEPROM: 1 kB

Processing Speed: 8 MHz

Digital Pins: 9 pins

PWM Pins: 5 pins

Analog In: 4 pins

Operating Voltage: 2.7-5.5V

Input Voltage: 2.7-5.5V

Let's Go Shopping!

Finally, the long list of Arduinos, Lilypad, and others comes to an end. Well, there are a lot more, but they are not at all meant for beginners or even intermediate users, which is why I did not include them in the list. For a complete list, you can always visit Arduino.cc or Sparkfun.com and learn about them.

Now that you know what kind of boards are available within the market, it is time to make a decision. Hopefully, it shouldn't be a tough one. However, if you ask me, my answer would be fairly simple; either go for the starter kit or

settle for the Arduino UNO.

The Arduino UNO is highly recommended for beginners as it does not confuse beginners or intimidate them with far too many pins and modules attached. Besides, I have already given you a detailed account on what the Arduino UNO looks like and where each of the components can be found. This, therefore, would make more sense because you will no longer need to go through instructions manual all over again to figure out where the pins are or where the crystal oscillator may be.

With that said, it is now time to go out shopping, and when you come back, we will get started with setting up Arduino for first-time use. Exciting stuff awaits ahead!

CHAPTER 4:

CHOOSING AND SETTING UP ARDUINO

You may have already visited a few websites that I have recommended in the previous chapter. If you did, fabulous, and if you didn't, don't worry as there are just a few things you must decide before clicking on that golden "Add to Cart" button and placing your order.

This chapter, therefore, will look into choosing the right kind of Arduino board and then setting it up to work with your computer.

Choosing the Board

When looking at the options for Arduino Boards, there are a few factors you will want to consider before making a choice. As a perfect way to eliminate doubts and confusion, ask yourself the following questions:

How much power do I need to run the application I have in mind?

Of course, for anyone who is looking to get started, you might know how much power you need for your first few projects. You might not know the exact measure of flash memory and processing power that you require, but there is a clear difference between the functioning of a simple night light that changes colors and a robotic hand with many moving parts. The latter would require a more robust Arduino microcontroller board, with faster processing power, more flash memory, and more SRAM. As a general rule of thumb, know that the bigger and more complex your project will be, the more power and features you will need.

Now, write down whatever it is that you are hoping to create. Even if it is a simple circuit that lights up to show you are connected, it still needs power. As I mentioned earlier, Arduino UNO is generally what beginners go for as their projects, at least in the start, are not that complex.

How many digital and analog pins will I require to have the functionality that I desire?

Once again, you don’t need to know every detail, but knowing whether you need more pins or less will greatly influence your decision regarding which board you choose. If you are going for a simple first project, you could get away with having less digital, PWM, and analog pins, while if you are looking to do something more complex, you will want to consider boards with a higher number of pins. Once again, the UNO offers a good balance as it offers significant pins and just about enough power to get you started.

Do I want this to be a wearable device?

Okay, this one may be a little advanced, but if you see yourself creating something that will eventually become a wearable device, most of the

Arduinos are already out of the picture. You will only be left with Lilypad and other circular variants. However, remember that you can find these on Sparkfun.com as opposed to Arduino.cc.

Do I want to connect to the Internet of Things? If so, how?

Consult the list in the previous chapter and you will know right away that there are quite a few models that support connecting to the Internet. You can either settle for boards with built-in Wi-Fi capabilities or opt to go for boards with ethernet connectivity. Generally speaking, if your project does not require you to move the board, an ethernet is a good choice as it offers a more stable connection.

These are all the questions you need to ask yourself. Once sorted, go ahead and buy your first Arduino board.

Getting Started on Arduino IDE

Buying the Arduino board is just one step. Now, you will need to download the IDE, and this can be done quite easily. All you need to do is to visit Arduino.cc. Right on the top navigation bar, you will see "software." Drag your cursor on top of the name and a drop down will appear, providing you with two options. Click on 'Downloads' and choose the correct operating system in your case.

Once the file is downloaded, simply go ahead and install the program. If you are asked for administrator authorization or permission, click on 'Allow' to proceed. Now, the IDE should launch. That's pretty much it! You made it. All that remains now is to start coding.

Coding Your First Program

It is quite an exciting moment for any beginner to be able to finally write something and compile it into a program. However, do not start celebrating just yet. We still need to write a code for our Arduino to receive and perform.

Your Arduino IDE, once it opens, should have a simple look and feel. You should be able to see English letters and words, but they may not mean a lot at this point, and that is okay. We are not trying to learn to crack the code in this chapter anyway. Right now, my intention is to get you started and write

yourself your first program.

Right in front of you, you should have a cascaded or a maximized window showing the following:

```
void setup() {
/put your setup code here, to run once:
}
void loop() {
//put your main code here, to run repeatedly:
}
```

If you are able to see this, well done! You have successfully installed the IDE and are now ready. Let's not wait any more and connect our Arduino board.

Connecting to the Arduino Board

Some of the boards come with a built-in USB, a mini-USB, or micro-USB port. If you have taken up my recommendation, you will have a micro-USB port. To get started, connect the relevant end of the USB into your Arduino board and the other, more traditional looking side to the computer. If you have your Arduino IDE running, it should automatically recognize the device. Do not be alarmed if your computer thinks you have connected a mouse or a keyboard as some devices, as we saw earlier, have that effect. For any given reason, if the IDE fails to recognize your device, simply go to Tools, Board, and select the model that applies. To further confirm that the system can actually detect a valid connection, go to Tools and down to Port. You should be able to see a connection there. If 'Port' is not selectable, you may need to install drivers for the board, but that does not happen often.

For models that may not have a USB port of any kind, you will need to use a TKDI cable or a breakout board in order to make the Arduino connect with your computer. This means you will need to insert the TKDI into the TKDI port on your Arduino board and then connect it either to your computer or to another board. If you connect the TKDI cable to a breakout board, you will then need to connect a relevant end of a USB cable to the breakout board and the traditional USB end to your computer. Once again, your IDE should then be able to identify the board right away. If not, repeat the troubleshooting step I mentioned above.

Uploading to the Arduino Board

To upload any sketch to the board, you will need to select the correct board and port to which you would like to upload. While the board part is relatively easy, selecting the right port can often be confusing for beginners. Generally, once you connect your Arduino board with the computer, it should be able to detect it and install the relevant drivers on its own. For Windows, you should also be able to see what COM port it is attached to. However, if the notification fails to show you the COM port, there are a few ways you can ensure you choose the right port for the job and the board as shown below. Refer to the one that suits your situation. Simply browse to Tools, Port, and then select the one that applies.

Mac

Use /dev/tty.usbmodem241 for the Uno, Mega256O or Leonardo.

Use /dev/tty.usbserial-1B1 for Duemilanove or earlier Arduino boards.

Use /dev/tty.USA19QW1b1P1.1 for anything else connected by a USB-to-serial adapter.

Windows

Use COM1 or COM2 for a serial board.

Use COM4, COM5, or COM7 or higher for a USB-connected board.

Look in Windows Device Manager to determine which port the device you are using is utilizing.

Linux

Use /dev/ttyACMx for a serial port.

Use /dev/ttyUSBx or something like it for a USB port.

Now that the correct port is also selected, it is time to learn how to upload a sketch (a program) to the Arduino. We aren’t going to upload an empty file, and this is where Arduino IDE comes to the rescue.

Arduino IDE comes pre-loaded with numerous examples of sketches or programs, all of which are designed to help you as any other tutorial would. They have the steps mentioned, and they even tell you what pin to connect to

in order to get the required output. For now, we will settle for an example called Blink. This is perfect to use as it will let us know if we have successfully uploaded something into our Arduino and that everything works fine.

Go to File, Examples, Basics, and choose Blink. As soon as you click, a new window will pop up, complete with the code and instructions. I will shortly be telling you all about how to read code and know what a comment is, but for now, know that the grey fonts are comments that are written to help other users and programmers understand what that specific code will do.

The actual program is as under:

```
void setup() {
// initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}
// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);
// wait for a second
  digitalWrite(LED_BUILTIN, LOW);
// turn the LED off by making the voltage LOW
  delay(1000);
// wait for a second
}
```

You do not need to change anything here. It is time to upload these sets of instructions to the Arduino and watch the magic happen. Just underneath File, you should see a check mark. Right next to it, you should see an arrow pointing to the right. That is the upload button. With your Arduino board connected, click on the upload button.

As soon as you click the button, two things will happen. The ‘TX’ and ‘RX’ lights will start to blink. These simply mean that the device is transmitting and receiving data, or in simpler words, communicating with the computer.

The second thing that will happen is the progress bar that will appear underneath the code itself. This is to let you know that the code is being uploaded. Once it is done, the two LEDs will stop and the one at pin 13 will start to blink. The LED will blink for a second, followed by turning itself off for the next second.

Now, let's tinker a little with the code. Under the void loop() block, let's change the value of delay for both HIGH and LOW functions. Set it to 100 instead of 1000. Once done, use the upload button once again and see how that changes the results. Now, the LED will start blinking faster. Do not worry, it will not burn out. If you were to increase the value of delay, the LED would blink slower.

Now that was exciting, wasn't it? You just uploaded your very first sketch to your Arduino board. It doesn't matter if you just made it blink a little. What matters is that you have started moving forward. Every great programmer started in the exact same manner. However, how you go on from here is completely your call. Arduino can get a lot more fun and a lot more complex as you go further into the world of Arduino. With that said, let's move on to the next chapter and start learning what exactly these codes, semi-colons, curly brackets and comments mean, along with some additional words that you may have come across.

CHAPTER 5:

CODING FOR ARDUINO

I will not lie to you; learning Arduino is like learning a new language. It takes time to fully understand it, get used to it, and then use it properly. This means that you cannot expect to learn the entire language in a day or even a week. Take baby steps and only move forward once you have understood what's going on.

In this chapter, I intend to break the code down into smaller, more digestible chunks so that you know what I mean when I write loop() or why on earth am I attaching parentheses to the word. For that, we must look at how the entire Arduino coding language is structured.

The Structure

There will be quite a lot of things you will learn from here on out. I recommend you note these down separately and use the notes for future reference.

setup()

This is one of the only two basic functions that form up the entire structure of the program. This is called on when the sketch starts and will run only once after startup or reset. You can use it to start variables, pin modes, or use libraries (specific terms that you can download for extra functionality).

loop()

The loop function is the second and the last part of the main body. It requires the Arduino microcontroller board to repeat a set of instructions within the loop block multiple times, continuously or until certain conditions are met. You can set the conditions for the loop to stop or break once the condition is met or no longer true. We will learn that later on.

Control Structures

As a programmer, we know that there are a lot of things that we can make the computer do, including make decisions. So far, the only codes you have come across are the ones you saw in the Blink example and the one I mentioned earlier in the book (yes, I will explain that later on). However, these programs are far too simple. They only do a specific thing over and over again. What if we want to add a bit of control or set some kind of condition that, if met, forces the program to change and do something else without us interfering? That is possible, and that is exactly what control structures do.

The control structures allow you and your program to take actions based on conditions that have been set by the programmer. To give you an idea, let us assume we wish to run a mile without stopping. There are a few things that can happen here:

- We may start but stop before completing a mile
- We may finish the run

- We may not even start

In all of the above, our actions would change depending on what the situation is. Similarly, for computer programs, we may want a game to run until the score reaches 100, or until the player loses all lives, or until all checkpoints are cleared. We tell the program all this using control structures. After that, the program knows what needs to be done should one of these conditions be met, or become true.

If you have any programming experience, you would already know what these are. These are essentially conditional statements we use in Python or other languages. However, it is still a good idea to see how they work in Arduino.

if

The ‘if’ statement simply checks for a given condition and executes the set of instructions defined within its block if the condition is true.

if…else

This is just like the ‘if’ condition, but here, we provide the program with two different conditions and a set of instructions. For example, if condition ‘A’ is true, then we want the red LED to light up, otherwise (else) we would like the blue LED to be turned on.

switch case

When you have a lot of conditions, each with their own unique variable values, and you want to allow the program to take actions based on which of those values it matches in a given scenario, you use the switch case. It is just another extension of if…else, but with more scenarios defined. Each condition will have instructions, meaning that the switch case will only execute the set of instructions under a condition or value that matches.

for

The ‘for’ is a loop statement. This is used when you are trying to run a specific code again and again and continue adding an increment or otherwise to the values obtained in the end.

while

The ‘while’ is also a loop statement. For as long as a specific condition is

true, whatever set of instructions is defined will be executed. The program first checks for the condition. If it is true, it runs the code once, checks the condition again, and repeats until the condition is no longer true.

do…while

This is just like the above, however, unlike the ‘while’ statement, where the condition is first checked before execution of the code, the do…while runs the code first and then checks the condition. This means that do…while will always run the code at least once.

break

The ‘break’ is like an emergency shut off button, theoretically speaking. Quite often, we may find ourselves in a situation where the loop condition is always true, in which case it will continue to run until it overloads the system. If we can spot a specific code that has the tendency of doing that, we use the term ‘break’ at the end of the loop to break out of it. This helps us to ensure that our system does not crash.

continue

This simply skips any current iteration, and moves on with the rest of the loop.

return

This is used to terminate a function and return a value from a given function to the calling function. Do not be intimidated because you might not be using this one a lot.

goto

This piece of code tells the microcontroller to move to another place, or a labeled point. Its use is generally discouraged by C language programmers, but it can definitely simplify a program.

Syntax

Now, let’s look at some other important components that go into making a code functional.

; (semicolon)

This is used as a period in the English language - it ends a statement. For every line of code that you write, it must end with a semicolon. The semicolon is used at the end of every statement, but there are two exceptions to this rule.

1. You do not need to use a semicolon when writing comments
2. If you are typing the last line in a block of code that ends with a curly brace '}' you must use the semicolon before the brace itself, not after it.

{} (curly braces)

Curly braces are used for more complex operations, such as defining a block of code for control structures or loops, et cetera. With that said, whenever you type the opening brace, be sure to type in the closing brace right away. This is termed as balancing braces, and it can help you manage the code more effectively. It is very easy to lose track of braces when dealing with them on numerous occasions.

// (single-line comment)

Comments are used to inform the programmer what a specific code or line of code does, or what the programmer must do to ensure that the code works. While there are a few ways to write comments, the most basic is a single-line comment. It is placed before or after the line of code and starts with two forward slashes. Simply type the message and hit enter and it will be saved as a comment. The program itself will not be affected by any number of comments because the program simply ignores it.

/* */ (multi-line comment)

There will be situations where you may need to explain the entire code or block, and for that, you will need multiple lines of comments. It is not a good habit to start every line with // as that can make the code look messier. Instead, start with /* and then type in your comments. Once you are done, type */ to highlight the entire block between the starting and the ending point as comments.

#define

The #define is essentially a C++ component. This allows a programmer to give a specific name to any constant value before the compilation of a program. A defined constant, therefore, will not take up any memory space, and that is a good thing.

```
#define ledPin 3

//#define does NOT use a semicolon at the end.
```

#include

When you are trying to use libraries from outside, meaning the ones that are not a part of your Arduino IDE or program, you use the #include statement. To include a library, use it in the following manner:

```
#include <LibraryFile.h>

//#include does NOT use a semicolon at the end either.
```

Next, we have to learn about operators and how we can use these in Arduino to get the job done.

CHAPTER 6:

OPERATORS

There are a few types of operators in existence when it comes to Arduino. These are extremely important and helpful to guide us in executing the code better and gain the kind of results we want.

Arithmetic Operators

Programming is never complete without the use of arithmetic operators. They are pretty much standard and are very easy to understand. However, just because they are easy does not mean we skip right past them. To ensure we all know and understand how they work, let us look at them one by one.

= (assignment operator)

You might have thought I made an error there, and that this is an “equal to” sign. Well, you wouldn’t be half wrong here. While this may very well be an “equal to” symbol, it works differently in programming languages. Here, it is called an assignment operator. The general rule of thumb is that whatever is on the left of the ‘=’ operator is ‘assigned’ the value shown on the right side. Here is an example to further clarify that:

```
int valueOne = 10;
```

Here, the variable valueOne is assigned the value of 10. The ‘int’ stands for integer, and we will learn a little more about that later on.

+ (addition)

This does exactly what it would do ordinarily. It adds values, either numbers, variables, or results. With that said, however, there are some limitations here. Depending on the data type you are using, you may end up receiving a negative number, and if that happens, know that you have gone above the maximum possible value that data type can hold. To fix the issue, change the data type. We will learn more about data types shortly.

- (subtraction)

Just like the addition operator, the subtraction operator is used to subtract two values, variables, or results.

*** (multiplication)**

This operator is used to multiply values with each other or a given number. However, do note that you may end up exceeding the maximum limit of a specific data type.

/ (division)

Two forward slashes are meant for comments, but one forward slash is used for division.

% (modulo)

This operator is an interesting one. When you are trying to divide some values or numbers together, and you wish to see what the remainder will be, use this sign instead of the usual / division symbol.

Comparison Operators

Just as the name suggests, these operators are used to compare values, allowing the program to better understand the condition and deliver outputs accordingly. There are a few types of these that we, as programmers, use in our code. It is a good idea to familiarize yourself with these as they aren't quite what you might be used to seeing.

== (equal to)

Earlier, I mentioned that a single equal to symbol is an assignment operator, meaning that it assigns something with a value that follows it. However, when setting conditions, if you wish for a program to execute a block of code when an output or an input is equal to something, you use the double equal to symbol "==."

!= (not equal to)

This is just the opposite of what you learned above. If you want your program to execute a block of code when said value is not equal to something, you use the "!=" symbol.

< (less than)

This one is quite self-explanatory. You use the '<' sign to compare values and see if the left is less than the one mentioned on the right side of the symbol.

> (greater than)

The greater than '>' symbol does the opposite. It compares a value on the left to see if it is greater than the one on the right.

>= and <=

You can either use greater than or equal to (>=) or less than or equal to (<=), depending on the kinds of values you are trying to compare. We usually use these comparison operators with if, if…else statements as well as loops.

Constants

Besides Comparison operators, we have what are called as constants. These are predefined expressions used in the Arduino language. These are designed to make the program easier to read and understand. Arduino classifies constants in a few groups. These are:

Logical Level: Boolean Constants

Boolean is a data type, and it can only have one of the two values; true or false. False is easy to define as false is essentially zero. On the other hand, anything that is not zero, whether a positive number or a negative one, is defined and identified by the program as true.

Note: True and false constants are always typed using lowercases.

PIN Levels: HIGH and LOW

When we want to read or write to a digital pin, there are only two possible values that any pin can take or be set to. These are HIGH and LOW.

The HIGH is a bit tricky to get in the start, but once you fiddle around a little, you should soon know exactly when to use it. It depends on whether you are using a specific pin as an INPUT or an OUTPUT.

When a pin is set as an INPUT, using the pinMode() function, and is read using digitalRead() function, the Arduino will report HIGH if:

- The said pin has a voltage greater than 3.0V (on 5V boards)
- The said pin has a voltage greater than 2.0V (on 3.3V boards)

There are ways you can configure the same pin as INPUT using pinMode(), and then change it to HIGH using digitalWrite(). However, if your pin is configured to be an OUTPUT with the pinMode(), and is set to HIGH using the digitalWrite(), the pin holds:

- 5V on 5V boards
- 3.3V on 3.3V boards

If the pin is in such a state, it can be used to power an LED, or something else

entirely. Do not worry if this is confusing because once you start with the practical aspects, things will become a lot easier.

For LOW, it is almost the same as above. The Arduino will report a pin LOW if:

- The pin has a voltage less than 1.5V (5V boards)
- The pin has a voltage less than 1.0V (3.3V boards)

The major difference is that if a pin is configured to OUTPUT, using the pinMode() and then set to LOW using digitalWrite(), the pin will have 0 volts on both 5V and 3.3V boards.

Digital Pin Modes: INPUT, OUTPUT, INPUT_PULLUP

The pinMode() function is used to change a pin's function and set it to INPUT, INPUT_PULLUP or OUTPUT. Any change that is done will alter the electrical behavior of that pin.

When a pin is defined as INPUT, that pin is considered to be in something called 'high-impedance' state. It demands extremely small power from the circuit. This, as a result, makes the pin extremely useful to read sensors.

The ATmega microcontroller that is present on the Arduino comes with an internal pull-up resistor. This means that they have resistors that can easily connect to the power internally. You, as a user, can access these. You can use these if you do not wish to use external pull-up resistors, and to do so, you will add the INPUT_PULLUP argument within the pinMode() function. However, care must be taken when using INPUT or INPUT_PULLUP for pins configured as inputs. These pins can easily be destroyed if they are connected to voltages that are below ground, or negative voltages, or are above the positive rail (5V or 3V) (Arduino.cc, n.d).

The pins that are configured as OUTPUT are considered to be in a state of low-impedance. This means that they can provide significant current to other circuits with ease. However, do not connect such pins that are configured as OUTPUT to ground or positive power rails as that can destroy the pins.

Built-Ins: LED_BUILTIN

Quite a lot of Arduino boards come with built-in LEDs. This is generally connected to an on-board pin. In most cases, this is pin number 13, as it is in

the case of Arduino UNO. Using the LED_BUILTIN tells the Arduino board to use the built-in LED. We have already used that when we were tinkering with the Blink example.

Next in line, we have the data types. The next chapter looks into what these are and how they vary from one another.

CHAPTER 7:

DATA TYPES

Now that the constants are done with, it is time to start learning about data types. Of course, every program has to deal with a lot of data. Some of these are numeric in nature while others use text, or strings. To cater to each of these, we have what are called data types. We use these to ensure that the program runs smoothly.

Void

This is used in a function declaration to tell the microcontroller that no information is expected to be returned with this function. For example, you would use it with the setup() or loop() functions.

Boolean

Boolean data holds one of two values: true or false. We generally use a bool value of true or false when comparing statements or conditions. If we have more than one condition, and we want both of them to be either true or false, we use the && operator. If we want one of the two to be true, we use the ‘or’ operator ||.

Char

This is a character, such as a letter. It also has a numeric value, and that means you can perform arithmetic functions as well. If you want to use characters literally, you will use a single quote for a single character, ‘A’ and a double quote for multiple characters, “ABC.” All characters are enclosed in quotes. The numbers -128 to 127 are used to signify various signed characters.

Unsigned Char

This is the same as a character but uses the numbers 0 to 255 to signify characters instead of the “signed” characters which include negatives. This is the same as the byte data type.

Byte

This type of data stores a number from 0 to 255 in an 8-bit system of binary numbers. For example, B10010 is the number 18. Slightly complicated? I know! However, you do not have to worry about it for now.

Int

Integers are how you store numbers for the most part. Because most Arduinos have a 16-bit system, the minimum value is -32,768 and the maximum value of an integer is 32,767. The Arduino Due and a few other boards work on a 32-bit system, and thus can carry integers ranging from -2,147,483,648 to 2,147,483,647. Remember these numbers when you are attempting arithmetic operations with your program, as any numbers higher or lower than these values will cause errors.

Unsigned Int

This yields the ability to store numbers from 0 to 65,535 on the 8-bit boards with which you will likely be working. If you have higher values than the signed integers will allow, you can switch to unsigned integers and achieve the same amount of range but all in the positive realm.

Word

A word stores a 16-bit unsigned number on the Uno and on other boards with which you will likely be working. When using the Due and the Zero, you will be storing 32-bit numbers using words. Word is essentially the means by which integers and numbers are stored.

Long

If you need to store longer numbers, you can access 4-byte storage, or 32-bit storage using the long variable. You simply follow an integer in your coded math with the capital letter L, as shown here:

long distanceToMoon = 1800000L; // not the actual distance

This will achieve numbers from -2,147,483,648 to 2,147,483,647.

Unsigned Long

The way to achieve the largest numbers possible and store the largest integers possible is to direct the microcontroller using the unsigned long variables. This also gives you 32 bits or 4 bytes to work with, but being unsigned the 32nd bit is freed from indicating the positive or negative sign in order to give you access to numbers from 0 to 4,294,967,295.

Short

This is simply another way of indicating a 16-bit data type. On every type of Arduino, you can use short to indicate you are expecting or using integers from -32,768 to 32,767. This helps free up space on your Due or Zero by not wasting space on 0s for a small number and by halving the number of bits used to store that number.

Float

A float number is a single digit followed by 6 to 7 decimal places, multiplied by 10 to a power of up to 38. This can be used to store more precise numbers or just larger numbers. Float numbers take a lot more processing power to calculate and work with, and they only have 6 to 7 decimals of precision, so they are not useful in all cases.

Double

This is only truly relevant to the Due, in which doubling allows for double the precision of a float number. For all other Arduino boards, the floating-point number always takes up 32 bits, so floating does nothing to increase precision or accuracy.

Lowercase vs Uppercase

You may have already noticed that I am using lowercase, uppercase and at times both. Arduino uses all of these and treats them all separately. This means that the variable ‘A’ will not be the same as ‘a’ because the program will treat the two separately. Similarly, pinMode() function will work but Pinmode() or pinmode() will not as they are not defined in that order.

Pay close attention to how these words are defined and how you declare your own variables. A lot of the time, we end up using the wrong name or letters, causing our program to be non-responsive and full of errors.

CHAPTER 8:

DISSECTING THE CODE

Let us start from the beginning, literally. In the start of the book, when I was introducing you to the world of Arduino, I wrote a program. I did say that we would be looking into it later on. Well, now is the time to do just that. Let me break that code down slowly so that you get to know what I meant when I wrote the following code:

```
const int PinkL = 13;
void setup ()
{ pinMode (PinkL, OUTPUT); }
void loop ()
{digitalWrite(PinkL, HIGH);
 delay (600);
digitalWrite(PinkL, LOW);
delay(600); }
```

Let's go line-by-line and see what the code says:

Const int PinkL = 13;

In the first line, I am essentially declaring an integer called PinkL and assigning it a value of 13. However, unlike other variables, that can often change, this one remains constant, hence the 'const' in the start of the code.

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

Okay, these are two lines, but there is a reason for that. The void setup() is a complete code block and counted as one. Whatever you write here, it will be executed once upon start. Here, I have used the function pinMode() and passed two parameters; PinkL and OUTPUT. What this will do is to execute the pinMode function and pass both PinkL and OUTPUT through the function for us. The pinMode function simply allows Arduino to use the pin instead of taking data from it, or reading from it. Since we have already specified pin 13 as PinkL, Arduino will now be controlling pin 13.

```
Void loop()
{digitalWrite (PinkL, HIGH);
delay(600);
digitalWrite(PinkL, LOW);
Delay(600);}
```

Now, we are in the loop territory. The code within this block will continue to run over and over until Arduino is disconnected or reset. The word ‘HIGH’ means that the pin will have an output voltage of 5V. For ‘LOW,’ the same pin will have a voltage of 3V. The higher voltage will allow the light to blink while the lower one will allow the light to remain off.

Now, you might be wondering why not use the Blink example instead? Well, this is actually the kind of code you will use if you have something connected to the Arduino and a breadboard, perhaps a simple LED circuit where the LED will light up brightly for 0.6 seconds and then turn itself off for the next 0.6 seconds.

I do not expect you to create a circuit just yet, but if, for any reason, you have set one up, you can use this code to figure out if your LED is placed properly or in a reverse order. You see, the LED has something called polarity. This simply means that it will work if it is inserted or connected in a specific order. You cannot just insert it in the middle of a circuit and expect it to work. If you have your LED inserted in a reversed manner, it will not light up even after uploading the sketch to your Arduino. If that happens, ensure you flip it around and the rest should work just fine.

Things to Remember

- Tabs, blank lines, and white spaces are equivalent to a single space, making it easier for one to read.
- Code blocks are normally grouped using curly braces, i.e., “{“ and “}”
- All open parentheses have a corresponding closing parenthesis, i.e. “(“ and “)”
- Numbers don’t have commas. So instead of writing 1,000, ensure that you write 1000.
- All program statements MUST end with a semicolon. This means each statement, except for the following two cases:
    - In comments
    - after curly braces are placed “}”

If, for any given reason, you find yourself with a code or a function that you are unsure of, I highly encourage you to check out Arduino.cc and read the ‘Reference’ section. It is the official resource from where you can learn about each and every component and structure that goes into making a program work.

Now that you have some taste of how you can understand a code, it is time to fully understand how logic statements work in Arduino. Don’t worry, I have already given you a taste of what these do when I mentioned ‘if’ and ‘else’ statements. These are extremely important to understand as almost every complex program or project will need these to function properly.

CHAPTER 9:

LOGIC STATEMENTS AND ARRAYS

Logic statements are effective ways for you to check the value of a variable against another value. Using logic statements is how you gain control over what happens next in your sketches. To further reinforce the concept, and learn how this works, let's look at a similar sketch that deals with an input and affects the output.

To follow along in Arduino IDE, the path is:

File → Examples → 02.Digital → Button

Notice how similar this code looks to the last one? By now, you should be able to make out what the code is trying to do. However, let's break down the new elements that you haven't seen yet.

One of the variables has to do with the button's pin, and another is for the button's state (on or off). In setup, we see that we are again using the pinMode to initialize the pins, but this time, our button pin is set to INPUT. This tells the chip that this pin will have the current going in, not out.

Moving forward, we have the loop. This is where we get into the real program. The first line introduces another function called digitalRead() which is the opposite of digitalWrite(). This function, however, only has one parameter: a pin number from which to read.

The 'If' and the 'Else'

Next, we encounter our first piece of logic statement. The outcome of the logic expression will vary depending on whether or not certain conditions are met. The comment already explains what this condition will be, and what the

outcome will become if said condition is met. Check if the button is pressed. When pressed, it should show HIGH. The expression is an ‘if’ statement, and this piece of code will only execute the code within its curly braces when the condition in the brackets is true. In our example, if (buttonState == HIGH), we are telling the compiler that when our variable buttonState is pressed down, or High, it must do what is between the curly braces {}.

When you use two equals in a row, you are asking the compiler to check if a variable has a certain value recorded there. In other words, we tell a compiler to compare the value of a variable with the one mentioned after the two equals symbols. When this condition is true, the now familiar digitalWrite() function is used to turn the LED on.

Next, we see an ‘else’ statement with its own curly braces. In our current example, this again uses digitalWrite() to tell the chip to turn off the LED, just as in our last piece of code. Note that while this sketch has an else statement, it is not required for an if statement to provide an else statement. Instead, if the condition isn't met, it will not run that code, and it will go past it to proceed to the next instruction.

And that's it! That's all that's needed to make an LED blink at the push of a button. Now we've gotten some pretty simple circuits out of the way.

The ‘While’ Loop

Let us look at another sketch. To follow along, open up:

File → Examples → 05.Control → WhileStatementConditional

The first part of this sketch will look quite familiar to you. We are declaring the variables we will need, initializing them, and setting the pins to the correct settings which are either input or output. Once we hit the main loop of the program, we see our very first while statement. Let's take a look at it now:

```
while (digitalRead(buttonPin) == HIGH) {
  calibrate();
```

This statement is fairly complex so let's break it down piece by piece. First, while statements carry out the operation within the curly braces for as long as the condition is true. In our case, if the button is being pressed, meaning that

it is constantly on a High position, the loop will continue to work.

We simply check our pin associated with our button to see if it is high or pressed. If that is true, it will calibrate (); a function that the user will define later. As the compiler sees calibrate (); it will jump to the instructions for that function, execute the code, and then return to the point in the code where it was before jumping away.

Let's look at that function now since it is being called:

```
void calibrate() {
```

Right, so this might look pretty familiar to you. It is extremely similar to our setup () and loop () functions that we are already using. This statement, or line of code, is telling the compiler that you want to define a function with the name calibrate, it will return ‘void,’ and it takes no ‘arguments.’

Technical jargon may be hard, but let me quickly explain. In order for us to be good programmers, we must understand that not everything within Arduino, or any other programming language for that matter, is pre-programmed. This means that we can often create our own variables, functions, and methods, and we do so by first defining them. Once we define them, and use them later on in the script, the compiler will know exactly what you are talking about and execute the commands you defined without any additional efforts.

The term ‘void’ simply states that this function will not return any value. In simpler terms, when this function is executed, it will not return the user a value as a result. It will simply stop working.

Now, let us look at the code again. The function calibrate() is doing the following things:

- Turning the indicatorLED on - telling you, the user, that calibration is underway
- The value found on sensorPin is now being stored in sensorValue

So far so good. However, now we must compare the values that we have recorded and figure out if these values are higher or lower than the ones that may have been previously recorded. We do that using something called ‘if’ statements. For now, we will not be using the ‘else’ part, and that is because

most of our readings will already be a part of the range. All we need is to record the minimum and the maximum values and check if they are higher or lower. After all of that is sorted, the function simply calibrates itself and returns no value, or void.

After the calibrate function completes, we jump back to the previous location in the code, right after a while statement. Now that the calibration is stopped, it is time to turn the indicatorLED off using the friendly digitalWrite function. Then, we must assign the value of the sensor to sensorValue using the analogRead. Now, the next line introduces a new function we haven't seen before.

Do the following in Arduino IDE: Help > Reference

This will open a webpage for you in your favorite browser. Pay attention to this page, or better yet bookmark it. This webpage is perhaps the only source you will ever need to fully master Arduino programming language, the syntax and everything related. It gives a detailed explanation of each of the components and provides examples as well. For now, find ‘map’ and simply click on the name.

Okay, I do admit that this isn’t the easiest of the things for anyone to get, however, try not to be intimidated. The ‘map’ function simply maps one number from one range to another. The map function takes five parameters, and these are:

1. Value
2. Current low
3. Current high
4. Target low
5. Target high

The above must be in that order. By doing so, the map function is able to scale a value to a completely different value that falls between the range we have defined above. This, in turn, will ensure that the eventual number falls within our ideal scale.

This is important because there is no way for us to know what values the sensor will return. We also do not know within what range the data set will

fall. With that said, the Arduino chip is capable of changing the output incrementally for one of its pins. In simpler words, whatever the reading of the pins may be, it will be adjusted so that the values fall between 0 and 255 in order for the chip to respond appropriately. Any values beyond this range would not work. By doing the calibration routine, we learn of the high and low values in the data set, and are able to determine the current value. Using that, we are then able to scale the values between the 0 and 255 range.

Keep in mind that the map function will not change values that are outside of the specified range. This is because some of these values may be of interest for other uses.

Moving on, we come across something called constraint function. This function is a lot easier to understand. It simply accepts a given value, a minimum, and a maximum as its parameters. If the said value falls within range, it will be left alone. If it is anything else, it will be set to either min or max, whichever is closest to the value. This means that if the value is 300, it will be set to 255, and if it is 50, it will be set to 0. Since the Arduino chips are only able to deal with the 0 to 255 range, the constraint function limits values that fall beyond the range of 0 to 255.

```
do {
// Some code to execute goes between the curly braces
}
while ( conditional statement);
```

The above is called do...while loop. Unlike the while loop, which first checks the condition and then executes itself, the do...while loop executes the code first, then checks for the condition. This can often be useful in cases where you want to run a block of code first and then have the code check for the condition later on.

The World of For Loops

Just like there is a ‘while’ loop, we also have the ‘for’ loop. This type of loop is effective for counting through sequences, ranges, or even initializing a number of pins on the chip, as we will see shortly.

For loops have a unique attribute, and that is their ability to create a unique variable within themselves. They also change the value of the variable they

create every time, and this is done to change the condition each time a loop is run because if the value does not change, the loop will continue to work indefinitely until the program crashes or worse.

for (variable; condition; increment/decrement) {}

The variable you see here is essentially a local loop variable. This means that this variable can only be called upon within the block of code that resides under the for loop. However, the more important aspect is to ensure that you name this variable correctly. Use a naming convention that not only is appropriate, but is also easy to read for other programmers. If you are working with pins, naming it myVariable would not be ideal. Instead, use something within the lines of myPins as it gives a better understanding of what this variable is doing.

The for loop uses a variable, followed by a condition. This is where we use the greater than, equal to, lesser than, and all the comparison operators. Finally, we come to the increment or decrements part of the for loop.

In the Arduino coding language (which is based on C++) you can increase the value of a variable by one with the symbols ++, and decrease the value of a variable by one with the symbols - - symbols. These increments are extremely important because otherwise, you either risk not running the code at all or trap your program into a never-ending loop, and that too is dangerous.

This means that if you were to do the following, the code would simply continue running forever.

```
For (int myValue; myValue > 1; myValue++) {
digitalWrite (ledPin, HIGH);
}
```

Therefore, it is important that you first visualize the loop, write it, and ensure that the given condition is eventually met as only by doing so will the loop discontinue.

Arrays

There will be times when you may need a single variable that could record a list of numbers or values, all in one place. It is indeed cumbersome trying to remember all the variable names and their corresponding values. Fortunately, we have something called arrays that come to our rescue.

The Arduino.CC defines arrays as “a collection of variables that are accessed with an index number” (Arduino.cc, n.d.). To create or declare an array, use any of the following methods:

```
int myNums [6];  //This creates an array named myNums with a limit to store six values
int myPins [] = {2, 3, 4, 5, 6}; // creates array myPins with given values stored in it
int mySensVal [6] = {2, 3, 4, 5, 6};
char message[6] = “hello”;
```

In the above, the first value is always stored at index number ‘0.’ This means that if we were to call myNums array, more specifically its fourth value, we would call upon the index number ‘3.’ That happens because the first index position is zero. You may want to remember that as quite a lot of beginners tend to forget that.

Note: The square brackets are how you distinguish that this will be an array of data.

Let us assume that we have five LEDs, located on pins 2, 3, 4, 5, and 6. Later within the program, we would need to refer back to these in that specific sequence. Instead of creating five different variables, we can store all of these values in a single array. How? Just like this:

```
name ledPins[5] = {2, 7, 4, 6, 5};
```

Here, we have instructed the compiler to put together five values, and initialize the array using those values. Now, if we were to access pin 2, we would simply type the following:

```
ledPins[0]
```

With that in mind, try to figure out what this line of code would do:

```
digitalWrite(ledPins[0], HIGH);
```

Correct! This would turn pin 2 on or change its state to High.

If you know you are going to work with a variety of pins, and you aren't too sure which ones they will be, you can declare an array without putting any data within it. You can then add the data as you go along.

```
int ledPins[6]; // This declares an array that will be able to hold six values.
```

Now, if you were to find out what values you will be using, you can do the following:

```
ledPins[0] = 1; // first index position
ledPins[1] = 2;  // second index position
ledPins[2] = 3;  // third index
ledPins[3] = 4;  // fourth index
ledPins[4] = 5;  // fifth index
ledPins[5] = 6;  // sixth index
```

There will be times when you aren't even sure how many values you need in an array, and even then, you can declare one using the following method:

```
myArray[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // You can add as many values as you choose
```

To further understand the importance of arrays, let us explore the following file:

File → Examples → 05.Control → Arrays

Pretty much everything you see here is standard, except for a variable named timer. The array here is showcasing the number of pins to which LEDs are attached. Moving on, we have the variable called pinCount, and as the name implies, it does exactly what it says. It counts the number of pins, and that also happens to be the length of the array. Seems easy, right? However, declaring a separate variable to store the count of pins may seem to be overdoing things, but that would be us jumping to conclusions too soon. We will later see how this variable is brought into action.

Moving further ahead, we have the setup(). The for loop here will initialize pins for us. We then go on to create a counter variable, in order to access the pins we want, and that variable is initialized to 0. The loop will continue to work for as long as the variable thisPin remains less than the pinCount. After every iteration, thisPin will be increased by 1, and the entire code will once again run until the condition is no longer true.

A lot of programs will use arrays and for loops together by creating counter

variables and then using both the array itself and the counter variable to get the job done. It is a good idea to practice your for loops by creating some of your own.

Moving forward, we will now learn about sensors that we can use with the arduino.

CHAPTER 10:

WORKING WITH SENSORS

Sensors for Arduino range in uses and functions. Arduino is designed to be inclusive of multiple types of sensors, which can, in turn, be applied to the programming language C. Arduino is equipped to work with these sensors, and they can be purchased relatively cheaply from Arduino or through other sites. Some examples of Arduino sensors include:

- The ultrasonic module
- IR infrared obstacle avoidance sensor module
- Soil hygrometer detection module soil moisture sensor
- Microphone sensor
- Digital barometric pressure sensor board
- Photoresistor sensor module light detection light
- Digital thermal sensor module temperature sensor module
- Rotary encoder module brick sensor development board
- MQ-2gas sensor module smoke methane butane detection
- Motion sensor module vibration switch alarm
- Humidity and rain detection sensor module
- Speed sensor module
- IR infrared flame detection sensor module
- Accelerator module
- Wi-Fi module

While there are many others, these are just a few. Some sensors are easier than others to connect with different Arduino units, so be aware which will best fit your Arduino.

When it comes to sensors, there is one thing that we, as programmers, must understand, and that is to establish how important it is to read analog signals. These are wave-like signals that are visible at the Serial Plotter of the Arduino IDE. These can go higher when the input or output is powered up, and they can dip downwards when the power is being reduced.

Analog inputs like the voltage of some sensors are a result of changing some factors. A good example of that would be the photoresistor. It is a type of a resistor that changes its value depending on the amount of light present around it. The voltage, and any changes pertaining to it, can be gauged using the multimeter.

This, essentially, is how every sensor works. You can do the same for virtually any kind of sensor out there and get the results recorded, see it in action and fully understand how they are able to gauge changes.

The Arduino UNO (ATMega 328p) comes with six input pins dedicated for the analog signals, and these start from A0 to A5. These can measure voltages with 4.8 millivolts per unit, and that means these are fairly accurate.

Using Sensors

Let’s take the temperature sensor as an example. In fact, we will create a simple device that can measure temperature for us. We will not be using Arduino IDE for this one.

The temperature sensor comes with a sensitive transistor that is made out of silicon. Silicon is highly affected by the temperature, which is why it is the ideal choice.

The temperature sensor comprises the following elements:

1. Input leg (also known as vin) (2.2v to 5.5v).
2. Signal leg (the one in center/also termed as vout) - We use this to get measurements
3. The ground leg (abbreviated as GND) - This is used to connect the sensor with any ground point, hence completing the circuit.

Next, we will need the following:

- Multimeter (any type)
- Two AAA 1.5v batteries
- One temperature sensor (You can use TMP35 or TMP35 or LM35)

Steps

- Insert two AAA batteries in the battery holder. You should now have 3 volts.
- Connect the red wire of the battery holder to the sensor’s Vin leg.
- Use black wire of the battery holder and connect it to the sensor’s ground (GND) leg
- Set up your multimeter to voltage mode
- Using a black probe of the voltmeter, connect it with the ground (GND) leg of the sensor, and connect the other probe (red probe) to the Vin leg.
- If you set this up smoothly, the reading should say 0.76 volts.

- Now, bring your hand closer to the sensor and note how the reading of the multimeter changes as you get closer. It will jump higher or lower, depending on how close or far you are from the sensor.

This happens because the change in temperature causes the sensor to change the voltage value. This may be read digitally using a multimeter, but for Arduino, we need to rely on an analog approach. The good thing is that using Arduino, we are able to convert the digital signals into analog ones, and are then in a position to do more.

Let us look at another example, but this time, we will use Arduino. We will be controlling the amount of light using a potentiometer.

1. Arduino UNO board
2. Breadboard
3. LED
4. 560 ohm resistor
5. 10 k ohm potentiometer
6. Wires

Connect the components as mentioned:

- LED small leg on D25, and long leg on D26
- 560 ohm resistor (C26 and C30)
- 10k ohm potentiometer (C34, C35, C36)
- Black wire (B25, and – terminal on row 24)
- Black jumper lead in negative terminal on row 25, and ground to GND
- Blue jumper lead (A30, ~9)
- Red jumper lead (A34, 5V)
- Yellow jumper lead (A35, A0)
- Black jumper lead (A36, - terminal on row 37)

Now, onwards to coding:

Use the Arduino IDE to create a new file:

```
const int sensorPin = A0;
const int ledPin = 9;
int inValue = 0;
int outValue = 0;
void setup ()
{
 Serial.begin(9600);
}

void loop()
{
inValue = analogRead(analogPin);//reads value from potentiometer
Serial.print("Input: ");
Serial.println(inValue);
outValue = map(inValue, 0, 1023, 0, 255); // This converts numbers between 0-1023 to the number 0 to 255
Serial.print("Output: ");
Serial.println(outValue);
analogWrite(ledPin, outValue); //Turns LED on
delay(1000);
}
```

The analogRead() function is used here to read the voltage in an analog signal form. The microcontroller within the Arduino chip can measure voltages ranging between 4.8 millivolts to maximum of 5 volts. Moreover, it also has the ability to convert values to digital values, which range from 0 to 1,024. Such conversion is called analog to digital converting, or ADC for short.

To give you an idea of how such conversions work, here are some examples:

Assuming that the input voltage to the pin at A0 is as follows:

4.8millivolt - This will be equal to 1 in digital

49millivolt - This will be equal 10 in digital

480millivot - This will be equal to 100 in digital

1volt - This will equate to 208.33 in digital

2volt - This will equate to 416.66 in digital

5volt - This will be equal to 1024 in digital

It felt nice to be able to create another project, even thought this too is a basic one. However, the idea was to ensure that you understand the principles involved in creating these circuits and how they work. Try tinkering with these in your own preferred way. You can change the position of the pins or use different sockets within the breadboard. However, ensure that you connect the right kind of leads with the right kind of power or purpose. It takes practice, but once sorted, you will find it relatively easy to set up skeletal circuits that you can then use for something a little more complex in the future.

Putting Arduino to Work

So far, so good. It is now time to put on your thinking cap and start with some more complex and meaningful projects.

We will first begin by trying to create a simple device that can measure various temperatures easily. For this, you will need the following:

- Arduino UNO
- USB A-to-B cable
- Breadboard – Half size
- Temperature Sensor (TMP36)
- Jumper wires (five of them)

Start by creating the circuit first. To create the circuit, do the following:

- Connect a black wire to GND and the negative terminal on row 30 (blue, far right side)
- Connect a red wire to 5V and the positive terminal on row 30 (red, far right)
- Connect the temperature sensor to F5, F6, and F7
- Connect a piece of black wire to J5 and negative terminal on Row 1
- Connect a piece of red wire to J7 and positive terminal on 9
- Connect a yellow wire to J5 and A0

Connect the Arduino to the computer via the USB cable. Open your Arduino

IDE and create a new project.

Type in the following before void setup():

```
const int tempPin = A0;
```

Next, we will type in the following:

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

void loop()
{
float, degreesC, voltage, degreesF; //Declaring three variables at the same time
voltage = getVoltage(tempPin); // This measures voltage at analog pin 0
degreesC = (voltage – 0.5) * 100.0; // This converts voltage into degrees Celsius
degreesF = degreesC * (9.0/5.0) + 32.0; // This converts Celsius into Fahrenheit

Serial.print(“voltage is:”);
Serial.print(voltage);
Serial.print(“Deg C is:”);
Serial.print(degreesC);
Serial.print(“Deg F is:”);
Serial.print(degreesF);

delay(1000); // this repeats the sequence once per second (you can always change this)
}
float getVoltage(int pin) //This is a function to read true voltage on said analog pin
{
return (analogRead(pin) * 0.004882814);
//This helps convert the 0 and 1023 values that the analogread() returns into a 0.0 to 5.0 value that the true voltage is.
}
```

Now, upload the sketch and watch the results on the serial monitor. Well done! You just created yourself a simple temperature reading device.

Before we move on, note that I will provide you with 12 extremely brilliant projects for beginners at the end of the book. Be sure to check those out as they can really help you master the concepts of Arduino. For now, we will move to the next chapter and do a bit of revision, just to ensure our concepts are reinforced.

CHAPTER 11:

REVISITING THE CONCEPTS

This chapter is a quick reminder of all that we have learned so far. I cannot emphasize enough the importance of these lessons, because without having a thorough understanding, it is practically impossible for us to move forward and design more complex projects.

Foundations of C Programming

In order to work with Arduino properly, you need to have a bit of an idea about programming in C. As I said, we aren't going to spend a terribly long time going over everything in this chapter, but there are a number of essentials that it is important we cover for posterity's sake. We're going to talk about the basic concepts which build up programming in C so that you can work with it with immense ease and feel somewhat natural when you're finding your way around programming in Arduino.

Working with Variables and Values

For a computer, everything that we type in is a value. With these values, our computer can perform some mathematical operations. That means that when it comes to any computer, everything, and I do mean everything, the computer does has to do with these values and variables.

Values can be in any form. They can be text, called strings, integers, decimal numbers, or something else entirely. These are values that can be used by the computer to carry out functions, perform tasks, or even retrieve data from other parts of the memory. However, this does not mean that if we type the number '2' or a word 'hello' that the computer will actually understand it. The computer has no idea what we are referring to. In fact, the words you are

reading right now, the computer cannot compute unless these are first converted into binary codes and then fed to it. This is where the compiler comes in. It changes everything into a series of zeros and ones, and that allows the computer to know what we are trying to do.

When it comes to variables, there are a few types that exist in almost every other language that you will come across. Arduino uses these to remain as efficient as possible.

Byte - This type of data only represents values between 0 and 255. These are integers, meaning that these are whole numbers and not decimals. The byte only uses one byte of memory to store its value, and that is good news for Arduino, too.

Int - The integer takes around four bytes worth of memory to store numbers. However, unlike a byte, an integer can store larger values as compared to byte. With that said, try not to use values that exceed 2,000,000 as this may push you beyond the buffer limit.

Float - Of course, not all numbers are whole numbers. A float is a decimal number with up to 5 decimal places. It has the capability to store a value as large as 32,000. Floats are not generally used in Arduino, but there may be some special cases where you may need to use them.

Double - While a float is fairly accurate, a double is twice as accurate as a float. Doubles take up twice the space but offer twice the decimal precision as that of float.

Unsigned values - The difference between these and other numbers is largely the fact that unsigned values cannot be negative in number. This allows them to hold twice as large values as integers and floats. If you are certain that you will not be working with any negative numbers, these are great to remember.

Short - Short uses twice the memory of a byte and has half the values offered by integers. If you are trying to save on memory, and you do not need to work with larger numbers, these are a great choice.

Long - These are integer values that are twice the size of normal integers, which means they can hold numbers well into the two billion range, but there is no default value type large enough for any number bigger than that aside from unsigned longs, which can be roughly four and a half billion.

Char - These represent ASCII character values. These are essentially any symbols that can be parsed by a computer and generally are used in order to store and print characters. Characters can be anything from the symbolic representation of a number, like ‘7’, or an alphabetical character like ‘a,’ or a symbol like ‘?’. Essentially, if your computer can print it, it is probably a character.

With that, we’ve covered all of the major data types available for you to use in C and Arduino. There are more, don’t misunderstand, but these are the primary ones that you need to understand for right now.

Arrays

Sometimes, you need to store multiple values at once. We’ll be talking about arrays more when we start to talk about strings, but for right now, we can cover the bare essentials of arrays. We already covered them in passing in the book prior, so we aren’t really looking to establish an encyclopedic knowledge of them right now, anyway. Regardless, we are going to cover them enough such that you have a refresher on them.

Arrays offer a way for you to essentially group values together by a common idea. As we have established, values are stored at random places in the computer’s memory. They are then accessed in the memory whenever they’re needed. Arrays serve two purposes in this arena, then.

First, they tell the computer “hey, these things are alike, and they’re going to be referenced at about the same time pretty often, therefore, we should be putting them near each other so that the total time to get from one to another is less.”

The closer these values are stored to each other, the easier and more efficient it will be to retrieve them by the compiler. It also means that the time for retrieval will be significantly less.

Arrays essentially set up contiguous, or connected, areas of memory that are the size of n elements of the array times the size of a data type. This means that if an array consumes four bytes of memory, the compiler will store 16 bytes worth of memory immediately next to each other.

You can then go on to assign values individually, as per the data type that the

array is made up of.

```
dataType arrayName[size]; //syntax of how to declare an array
```

If you wish to populate, or partially populate, the array, you can do so as shown here:

```
int myArray[5] = {1, 2, 3, 4, 5};
```

Remember that arrays use zero indexing, meaning that the first position or value is stored at index zero. This means that if you were to call the value of 3, you would need to do the following:

```
myArray[2];
// this would fetch the value of '3' since it is the element store at index '2'
```

Despite the fact that these are quite easy to understand, it is good to know how to use these as you will come along arrays in a lot of programs, especially the ones that are meant to be complex in nature.

Truth and Logic

What is logic? How do you define logic? In the simplest terms, logic is the use of comparison of two or more things, to establish an end result.

You will also come across another definition, and this one applies more to programmers. It is the combination of both premises and conclusions to try and seek out truth. Both of these are not mutually exclusive in any way. You can use logic with many things that may not make sense generally, and still arrive at a logical conclusion, whether true or false.

All cars are blue.

I have a car.

My car is blue.

There is no denying that these statements are logically sound. They are not, however, true because the idea of all cars being blue is downright incorrect. However, many things can be established in a logical manner using logic that is based upon some kind of truthful premises. If that is the case, you will then arrive at a truthful conclusion.

We use comparison to determine or establish logic when it comes to programming. Comparisons essentially take pieces of information and then

compare them to something else in order to determine whether something is true.

For example, take 3 and 7; if I were to say that “3 is more than 7”, this is a logical comparison between these two values. This entire statement would be false, though, since 3 is not more than 7.

In programming, we can do this with any given variables and values that we want so long as they are comparable. You can even overload these operators in order to define new ways for things to be comparable when you start with C++ and similar languages.

For right now, though, let’s focus on the things which allow us to compare values. These are known as comparison operators. The comparison operators in C and, by extension, Arduino, are like so:

s == t

This checks to see whether or not s and t are equal.

s < t

This checks to see whether or not s is less than t.

s <= t

This checks to see whether or not s is less than or equal to t.

s > t

This checks to see whether or not s is more than t.

s >= t

This checks to see whether or not s is either greater than or equal to t.

s != t

This checks to see whether or not s is not equal to t.

There are many logical operators, but the ones in C that you mostly need to know are these:

A && B

Checks to see if both expression A and expression B are true.

A || B

Checks to see if either A or B is true.

!A

Checks to see if the statement A is not true. If it is not true, then we will return true since the statement “not A” is true.

If you're familiar at all with discrete mathematics or symbolic logic, then you'll see quite easily how many of the concepts carry over from it. However, even if you aren't familiar, they're still very easy for you to grasp nonetheless.

Conditionals

Now that we've talked a bit about logic and truth, we can build on that knowledge to talk about what are called conditional statements. Conditional statements are one major part of control flow. Control flow is extremely prominent in computer science, and almost every application you've ever used will have some degree of control flow built into it. Control flow is, after all, the basic way that you can give your program some sort of "intelligence," if we're defining intelligence as the capacity to make decisions based on given data.

These can take two forms: the passive and the active conditional. The passive conditional is the most basic form, so we're going to cover that first.

The passive conditional is called so because there is no obligation for the program to run the code of the conditional. The passive conditional is always established through the use of 'if' statements.

We already know how 'if' statements are used. The 'if' statement first checks to see if the condition is met, or it is true, and only executes the code block if that is the case. Otherwise, it will simply skip the code altogether.

As a reminder, here is how you use an 'if' statement.

```
if (condition) { // you get to define the condition here
// code goes within
}
```

If you have two conditions and you only wish to use one of the conditions, depending on the output, you can use the complimentary 'else' statement as shown below.

```
if (condition) { //First condition
// code goes within
} else { // if first condition is not met, it will execute this (does not need a condition)
// back-up code is here
```

```
}
```

There will be times when you may have to test out multiple conditions and have the program execute one that is more relevant to the findings. To do this, we use what is called ‘else if’ statements. There is no limit as to how many you can use, allowing you full control of how you want the program to behave. To do so, do the following:

```
if (condition) {        //primary condition

} else if (condition) {        // secondary condition

} else if (condition) {        // n-th condition

} else {        // if all conditions are not met, this will be executed

}
```

Loops

Loops, just like conditional statements and arrays, are a vital part of programming, be it in C, C++, C#, Python, Arduino, or any other language. Loops allow you to get a lot done in a fraction of time. Imagine trying to light up three LEDs individually, one after the other, over 100 times; you can do so using loops. All you need is to figure out which loop works for the situation and how to type the condition in so that the desired output is received.

While the above is just a simple example, you can expect significantly tougher and more complex pieces of code to come your way. It really helps to first plan your codes on a piece of paper, draw a flow chart that shows how the program works and executes certain commands, and then code that into your Arduino.

Loops come in two different types. The first one is the ‘while’ loop and the other is the ‘for’ loop. These two are derived from the C programming language directly.

Although you will be using ‘for’ loops more often, the ‘while’ loops are far easier to understand and work with. The downside, however, comes in knowing when to use ‘while’ loops because they perform similar operations.

With for loops, you have really obvious bounds, but with while loops, you

don’t. For this reason, while loops are best suited to those cases where you don’t have an actual finite number of times for a loop to run.

A while loop simply checks a condition and then runs the code within the body of the loop for as long as that condition is met. If that condition is ever not met, then the loop will exit. Easy enough!

The syntax for a while loop is like so:

```
while (condition) {
// loop’s internal code
}
```

While loops are best suited to the concept of the “game loop.” These loops aren’t exclusive to games, of course; they just express the idea of a game, because games will do the same thing over and over until a win or lose condition is met. When those conditions are met, the game is considered over.

The game loop is based on the idea of having either a true or false variable that is changed to the opposite when a certain condition is met. So, for example, the code may be:

```
#define TRUE 1
#define FALSE 0
int bHasWon = FALSE;
while (bHasWon == FALSE) {
// code goes here
}
```

Then have something that will change bHasWon to TRUE when the player wins. This will indicate that the loop should be terminated because the win condition has been met. Of course, this logic - again - can be used for many things aside from games. Anything with a central main menu will in fact likely use this sort of loop logic.

The other kind of loop is the for loop. The primary purpose of the for loop is to allow you to iterate through a given set of data with ease. For loops start with the creation of an iterator variable, which can be named whatever you want. The iteration step can be many things, but it normally is by 1 (iterator++ or iterator-- for +1 or -1 each time, respectively).

The syntax for a for loop is like so:

```
for (iterator declaration; condition; iteration step) {
// code goes within
}
```

So to print out every number in an array, we could do the following:

```
for (int i = 0; i < (sizeof(myArray)/sizeof(myArray[0]); i++)
{
printf("%s\n", myArray[i]);
}
```

Where sizeof(myArray)/sizeof(myArray[0]) gets the number of elements within the array altogether.

Functions

The last thing that we need to talk about and rehash before moving on to the next chapter is the concept of functions. Functions are a foundational concept in C programming and Arduino by extension. You're going to run into them a lot, so it is important that you understand exactly how they work. Fortunately, they aren't a terribly difficult concept to understand! Functions simply are based around the idea of breaking something down into code which can be reused over and over.

Functions may be familiar to you through things like past math classes, where you would have something like f(x) = y. X was the argument, and the function manipulated x in order to give you the value y. Functions in computer science are relatively similar (and are very much similar to higher level functions in mathematics, but I'm not so willing to assume that everybody who reads this book has already worked with those, so I'm going to pull back on that one).

Functions have a few basic parts. First, they have their declaration. In C, you either have to declare a new function at the start of the file, called prototyping, or you have to put it before your main function. For simplicity's sake, we'll go ahead and just prototype them at the start of our file and put them after our sketch's primary functions.

Functions also have a return type. This is the kind of value that they give

back at the end of the function. So, for example, a function called convertTemp would probably give back a decimal number. Therefore, the type of function would be a float or a double.

Functions can also be written which don't have a return type. These functions are called void functions. They are valid, especially for performing certain operations that are to be done over and over but aren't particularly mathematical in and of themselves, like printing text to a serial or spinning a motor or something of the like.

Functions also have arguments. A function doesn't have to have arguments, but you can give a function an argument. A function's arguments are defined in its declaration. You can tell the types of the arguments as well as the placeholder names. You can then treat the arguments as variables within the body of the function and feed in the actual values when you call the function later in the program.

Again, though, a function doesn't necessarily have to have an argument. This isn't a requirement for a working function. Remember that as you go forward!

The syntax for prototyping a function is like so:

functionType functionName(arguments, if any);

And the syntax for actually writing a function is like so:

```
functionType functionName(arguments, if any) {
// code within the function
// return valueName if necessary;
}
```

So, let's make a function which will return the volume of a cone as a float. This is going to need two arguments, radius and height. It will be a double since we're working with pi and want it to be as accurate as possible. We will feed in doubles as arguments, too.

We could prototype the function near the start of our program like so:

```
double volumeOfCone(double radius, double height);
```

Then at some point in the program afterward, but not within another function, we could include the actual body of our function like so:

```
double volumeOfCone(double radius, double height) {
return 0.33333333 * 3.141569 * (radius * radius) * height;
}
```

We can then treat the return value of this function as a value of itself. So, we could create a double variable and assign the return value of this function to it:

```
double volumeOfConeRThreeHFive = volumeOfCone(3.00, 5.00);
```

This would return the volume of a cone with a radius of three and height of five and save it to the variable we created. We can also insert it anywhere that value is accepted, like in a printf statement or into a formatted string or something similar.

With that, we've covered the last thing that we needed to go back over before we get into some of the dense and meaty concepts within this book. In the chapters to follow, we're going to be going over much more in-depth programming concepts as we try to figure out the world of programming in Arduino at a greater level than we have before.

CHAPTER 12:

IN-DEPTH ARDUINO

Working with computers, especially something so precise and hardware-limited as the Arduino, can be immensely rewarding, but if you want to be good at it, you have to have a very firm understanding of a lot of underlying concepts.

It is perfectly fine, for example, to understand what variables are, but if you don't understand how they work, you may end up wasting a lot of computing power with them when you really don't mean to. And again, when you're working with something like Arduino, that's the last thing that you want to do.

So, in this chapter, we're going to be building on some of our topics that we've already discussed so that you can be a better Arduino programmer and, in turn, a better C/C++ programmer.

Memory Management and Pointers

The first thing that we're going to talk about in this chapter is the concept of memory management and pointers. This is an immensely important topic, especially when we're talking about Arduino. You don't have a lot of onboard memory to work with, so you need to make the best of what you have.

Let's think back a second to our discussion about data and data types. We talked about how you could create variables and all of the things that you can do with them. One thing we didn't really talk about, however, is how these variables work in terms of the computer's memory.

Computers store variables for running the length of a process called the random-access memory. You can think of random-access memory as space that can be allocated dynamically and as needed in accordance with the current demands of the program. All values which are worked with by the programmer and the program are stored, to some degree, in the random-access memory. Variables can be defined, which allocate a set space of random-access memory that is the size of the defined variable.

Variables, too, by their very nature, are essentially references to the places that a value sits within the computer's memory. When you refer to a variable, though, you aren't actually working with that value necessarily. For example, when you pass a variable to a function, you aren't sending the variable itself. Instead, you're sending a copy of the variable's value to be manipulated by the newer function. This copy is then disposed of when the function is finished.

In the olden days of computing, this made a bit of sense. After all, you don't necessarily want to change the value of some variables every time that you send those to a function. It makes sense for such a case not to be the default. Additionally, functions are also stored in the computer's memory in a certain way such that it makes more sense to send them values directly than to send them references to values elsewhere in memory. It makes them, in a manner of speaking, run more efficiently.

However, there are certainly cases where you would want to refer to the value

of a variable itself and not just to the value that the variable refers to. For these cases, you'd want to use pointers. Note at the same time that it is certainly possible to work through most Arduino sketches without ever having to use pointers. However, there are times where you will be working with a data structure or need to create a data structure, and in these cases, a working knowledge of pointers is very useful.

Not to mention that whether we're talking in terms of general programming or Arduino programming specifically, pointers are something you need to know because they are an important concept of memory management. You have to understand memory management in order to write efficient programs, and you need to at the very least understand the concept - if not for Arduino, then for anything else that you want to program.

So, what are the key points to take away from all of this? Well, first and foremost, pointers offer a method for you to refer to a value by its place in memory rather than just by a copy of its value. This is important. Why? Because it allows you to pass and work with direct values rather than simply copies of those values that you may have through variables. This is foundational to programming in C and will also probably come up sometimes during Arduino programs. While generally, you can write entire sketches without ever even using pointers, it is still good knowledge to have for when you are looking through other people's sketches and learning from the code that they're writing.

So, how do pointers work then? Pointers work through a combination of operators called reference and dereference operators. You can use these in order to create new pointer variables and point them toward an already existing variable.

The first thing that you're going to do is create a new pointer to the type of value that you want to point to. You do this by using the reference operator *. For example, let's say we had this:

```
int apples = 3;
```

And we wanted to create a pointer that would point to this variable. The first thing that we would do is create a new pointer:

```
int *ptr;
```

You can put the reference operator wherever. It depends on the coding conventions of whatever you're working with, but generally putting it with the variable name is a safe bet.

Afterward, you're going to define the address you want it to point to. You do this through the dereference operator: &.

```
int *ptr;

ptr = &apples;
```

Then, whenever we go and modify the ptr variable through the reference operator, it will change the value stored at the address that we pointed it to.

```
*ptr = 4;

printf("%d", apples);

// this would print out four since we changed the value at the address referred to by the variable apples to be 4 rather than 3.
```

You can see pretty plainly how this would have a lot of utility to you as a programmer when you're trying to pass variables between functions and work with variables in a complex manner. Being able to directly manipulate pointers like this has a lot of useful perks, too. When you're working with a platform where memory is both as limited and as crucial as the Arduino, you're going to want to have at least the ability to work with memory directly. There are some more essential things that you will want to know, but they are part of an advanced topic and aren't within the scope of this book. They also aren't particularly useful to Arduino programming itself, such as direct memory allocation through the malloc function.

Regardless, knowing how to work with pointers will push you forward as an Arduino programmer because when you do encounter pointers in the wild or have to create functions that pass variables that need to be directly modified, you can do so with ease and not be pulling your hair out in confusion.

Stacks

Stacks are yet another extremely important computer science concept. They're important primarily because they work in a really crucial and integral way with things like pointers and arrays. So, what are they?

This is a bit more in-depth than Arduino, but you will still inevitably run into

the basic stack terminology in discussions on Arduino programming, so it is important that you have a solid idea of what the stack is and how it can be used.

"Stack" is actually a relatively versatile term. The idea of a "stack" simply refers to what it sounds like - a stack of values. You can add variables to this stack. Imagine a block tower. This is essentially how a stack works.

You can put things on top of the stack, and these things are also the first things to be removed from the stack. This is especially useful in algorithmic programming, but it does bear some use in Arduino programming as well. Why? Because when you're dealing with complex and limited memory structures, you're going to inevitably run into many occasions where the best path forward is to use a stack, owing to the fact that stacks are memory-friendly. Stacks do not demand a lot and are fairly easy to use and compute.

The stack is based on two functions, namingly pushing and popping. These are fairly easy to understand.

Pushing simply refers to adding something on top of the current stack. There is literally no rocket science here. The beauty about pushing a value on top of a stack is the fact that you do not have to worry about remembering the name of the variable. You simply demand the program fetch the value, and it will do so right away.

Popping is the opposite of pushing, and that simply means to remove something from the stack. If you use popping, it is a good idea to remember that it will only remove the lastest added value on the stack and nothing more. Once the value is removed and retrieved, you can then modify it easily.

To return the modified value back to the stack, you simply use the pushing method, and it will be back on top of the stack.

Structures

Just like every building needs to have a solid structure that can withstand the tests of time, programming also needs a structure that can shape the functionality of the program and help programmers create something more meaningful. A lot of beginners tend to overlook the importance of structures, which is why they face difficulties later on.

For anyone who may have worked with C programming language before, you may have come across the term object-oriented programming, or OOP. Structures within the C language are essentially precursors to the same idea. Although they aren't as extensively programmed as OOP concepts are, they still offer some effective ways for programmers to group together ideas into a single structure.

There are many situations that we, as programmers, come across, and some of these often involve the use of values or variables that are interlinked or applicable to a variety of things. Declaring them individually and initializing them, and then recalling them separately for numerous variables, is simply a cumbersome task. That is where structures can help us reduce the code and make it a lot more readable.

Let us assume that we have the following values.

```
catOneLegs = 4;
catTwoLegs = 4;
catThreeLegs = 4;
catOneColor = 'brown';
catTwoColor = 'black';
catThreeColor = 'white';
catOneBreed = 'tabby';
catTwoBreed = 'persian';
catThreeBreed = 'american shorthair';
```

See how difficult it is to type all of these variables and define each individually? Imagine if we had to type in such data for 50 cats, or 100.

Through structures, you can instantly define every one of these in one go. You will then be able to recall these values and work with them whenever you like. Unlike Stacks, where you are only able to work with the top most value, structures are far more effective. To define a structure, you do the following:

```
struct structName {          //Declare and name your structure here
// data
};
```

Now to implement the data we have from above:

```
struct theCat {
int countLegs;
String color;
String breed;
};
```

Now to define the cats:

```
theCat catOne = {4, ‘brown’, ‘tabby’};
theCat catTwo = {4, ‘black’, ‘persian’};
theCat catThree  = {4, ‘white’, ‘american shorthair’};
```

And that’s it! You just discovered an effective shortcut to carry out the extensive operation in a much more efficient manner.

You can see how this presents the programmer with a much easier way to group important data together. There’s a good chance you aren’t going to be using this data super often, but it does allow you to have such a way to do this. It is important that you know what it is because you will eventually see it.

All of these concepts are important to Arduino because Arduino is far more restricted in terms of memory and processing power than your home computer would be. It is important that you know how to use these concepts so that you can make the most out of your Arduino’s processing ability as well as write simpler and more elegant sketches than you would have otherwise. For example, you could create a structure that held three-byte variables called r, g, and b in order to program your RGB colors alongside the same lines in a simple manner. You could define new colors doing this to access them easily later on in your program, like so:

```
struct color {
byte r, g, b;
};
color blue = {0, 0, 255};
```

See how simple that is? But the utility of doing such a thing is pretty plainly obvious!

In this chapter, we've covered three major programming concepts that you're going to inevitably come across when you're working with other people's Arduino code and learning from what they've written, so it was important that I develop your ability to parse and work with these ideas.

Arduino API Functions

In this chapter, we're going to start going into a lot of detail on functions that are provided by the Arduino API. The first book had a lot to do with the bare fundamentals of programming. This is great and all, but we didn't really get too much experience with the Arduino API itself. Our goal now is to get some experience with the numerous functions that are provided by the Arduino interface for programmers to use.

The Arduino API is the rich set of different things that are provided to the hopeful Arduino programmer to give them more options in their programming. The Arduino team has done a fantastic job of providing a full-featured API that gives the programmer a large variety of different things that they can do within the context of their Arduino tinkering.

We're going to be dividing these by section and going into a lot more information on each function, so settle in tight. What you get from this chapter is a full reference on Arduino programming that you can swing back to whenever you need.

Digital Input and Output

There are a number of functions defined by the Arduino API in order to allow you to work with digital pins.

pinMode(pin, INPUT - OUTPUT - or INPUT_PULLUP)

The pinMode() function helps you to designate a role to any given pin. If you are trying to use a specific pin as an input or an output, or you are trying to enable certain pins through pullup resistors, you will need to use the pinMode() function, followed by the relevant attribute.

Analog Input and Output

The Arduino offers both digital pins and analog ones. If you are trying to read voltage from a given analog pin, or configure voltage, you can use the relevant functions as shown below.

analogRead(pin)

This is used to read the voltage at a given analog pin, and in return, it will fetch an integer value between 0 to 1023.

analogReference(type)

Depending on the type of Arduino you are using, this function helps in configuring the voltage and using that as a reference.

In most cases, you will specify these as either DEFAULT or INTERNAL. In rare exceptions, you can specify a completely different reference voltage.

analogWrite(pin, value)

This is used to write a given voltage valued between 0 and 1023 to the pin. Remember, 0 represents 0 volts whereas 1023 represents 4.99 volts.

Advanced Input and Output

If you are trying to create simpler projects, you will not be using a lot of these. However, for more complex ones, it is a good idea to familiarize yourself with the advanced input and output functions.

tone(pin, frequency, OPTIONAL duration)

You use the tone() function to specify a frequency and then convert that into a square wave on a given pin. The function takes three arguments at most: pin number, frequency, and the duration (optional).

noTone(pin)

To stop the tone that is generated through tone function, we use noTone().

pulseIn(pin, value)

To read the pulse on a given pin, we use pulseIn(). The function takes two arguments: the pin number and the value. If the selected pin is generating high to low pulses, or fluctuations, the function will return the time between the high and low in microseconds. Considering the fact that pulses are often not even, you can specify if you want to only read the high pulse or the low pulse.

pulseInLong(pin, value)

Much like the pulseIn() function, this one returns an integer value of the microseconds. However, unlike the previous one that returns an integer value,

this will return a long number.

shiftIn(dataPin, clockPin, bitOrder)

To send in a small, byte worth of data to the pin, we use shiftIn(). The function takes three arguments. The first one is dataPin and that is essentially the pin where you are trying to send the data. The clockPin simply designates when said pain has read the data. The bitOrder can be set to MSBFIRST (Most significant bit first) or LSBFIRST (Least significant bit first).

shiftOut(dataPin, clockPin, bitOrder, value)

This is the opposite of the function above. Through this function, you can send the data to a pin but only one bit at a time. It also takes an additional argument of value, which must be a type of byte.

Time

These functions are intended to help you in working with time-sensitive things in the Arduino scope.

delay(value)

This allows you to pause your sketch for a certain amount of time specified by the integer value in milliseconds.

delayMicroseconds(value)

This is functionally the same as the delay() function except for the fact that it uses microseconds instead of milliseconds.

Math

Believe it or not, programming sometimes involves a lot of math. The math functions in the Arduino API are similar in many ways to those math functions defined by the C math library, but they keep you from having to import any additional mathematical libraries. Even if you don’t use much math in your program, you’ll still benefit from knowing that these exist because you never know when you might need them.

abs(value)

This function returns the absolute value of a given number or the distance

between zero and a given number on a number line.

constrain(variant, lowerBound, upperBound)

This allows you to create a function such that a number will always be within the lower and upper bound.

map(number, fromMin, fromMax, toMin, toMax)

This will map a number from one range to another range.

max(number1, number2)

This will return the highest number of number1 or number2. Simple enough!

min(number1, number2)

Pretty much the exact opposite of the max function. This will return the lowest of the two numbers.

pow(base, exponent)

This will allow you to take a given number and then raise it to an exponent. C, which doesn't have a built in exponential operator, makes great use of this function.

sq(number)

This will return the square of a given number. A shorthand for pow(number, 2).

sqrt(number)

This will calculate the square root of a given number.

cos(angle)

This will compute the cosine of a given angle, with the angle to be specified in radians.

sin(angle)

This will compute the sin of a given angle, with the angle to be specified in radians.

tan(angle)

This will compute the tangent of a given angle, with the angle to be specified

in radians.

Characters

While they will be rare, it is important that you have a set of functions primed for you to use whenever you're working with character sets.

isAlpha(character)

This will return whether or not the character is alphabetical.

isAlphaNumeric(character)

This will return whether or not the character is either alphabetic or numeric.

isAscii()

This will return whether or not the character is an ASCII character.

isControl()

This will return whether or not a character is a control character.

isDigit()

This will return whether or not a character is a number.

isGraph()

This will return whether or not the character is something that has visual data. A space, for example, does not have visual data.

isHexadecimalDigit()

This will return whether or not the character is hexadecimal.

isLowerCase()

This will return whether or not the character is lowercase.

isPrinable()

This will return whether or not the character can be printed to the console.

isPunct()

This will return whether or not the character is a punctuation mark.

isSpace()

This will return whether or not the character is a space.

isUpperCase()

This will return whether or not the character is in upper case.

isWhiteSpace()

This will return whether or not the character is a whitespace character, like a tab, space, or line break.

Random Numbers

These functions will allow you to create random numbers in your program. Note that computers can never be truly random and spontaneous; all things are based on inputs, and nothing will ever be without these inputs in a computer. As a result, the random function must be seeded.

randomSeed(number)

This starts the random number generator. You feed a number in, and it starts at some random point within the sequence of the pseudo-random number generator's numerical sequence.

random(OPTIONAL minimum, maximum)

This will act as the bounds to your random number generation. The maximum value is the highest random number that you will allow, and the minimum input is the lowest value you will allow. If you don't specify a minimum, then the minimum will be assumed to be 0.

Bitwise Functions

These functions allow you to work with bits and bytes, which are the smallest pieces of data that a computer will work with. You can, theoretically, work with smaller values (in terms of overall computing power required), but these are the smallest practical values that you're going to work with while programming for Arduino.

bit(bitNum)

This will return the value of a given bit.

bitClear(variable, bit)

This will set the given bit of a specified numeric variable to 0.

bitRead(variable, bit)

This will give back the bit of a specified numeric variable.

bitSet(variable, bit)

This will set a given variable’s bit as position denoted by bit to 1.

bitWrite(variable, bit, 0 or 1)

This will set the bit at the given position within the variable to either 0 or 1, depending on what you say.

highByte(value)

This will return the highest byte of a given value.

lowByte(value)

This will return the lowest byte of a given value.

Well, that about wraps up this chapter. Next, we enter into our final chapter where we will look into the stream class and how to work with strings.

CHAPTER 13:

STREAM CLASS AND MORE

This deserved its own chapter. While this also deals heavily with using the Arduino API in an effective manner, this is such a broad lesson that we really needed to break it into its own chapter so that we could properly discuss it.

The stream class is a relatively simple concept to grasp. The stream class in itself is based on reading information from a certain source and using this within your sketch. Because the stream is about reading data, it is necessary that we also talk about working with the keyboard and mouse in this chapter even though these aren't related intrinsically to the stream class.

When you're working with data, especially reading in data, you're going to inevitably find at times that you're going to need to work with sets of characters like words, sentences, or anything of that nature. The idea of strings presents you with the opportunity to do this.

Strings are basically just a set of character values that are linked together as an array. Therefore, they’re contiguous in memory, and the computer sees them as one large and interconnected unit. Working with strings means learning to manipulate these units to the best of your ability.

It is really simple in and of itself. Strings are essentially just character arrays. This means that we’re technically working with what are called C-style Strings, which are basically strings that have a very low level of abstraction. For example, in a lot of more modern and higher-level languages, strings aren’t revealed in their character as a character array; they’re rather treated as a more abstract object, even if they are a character array at their core.

A string is composed of n+1 characters, where n is the number of letters within the string in a general sense. So, for example, the size of a string for

the word “hello” would be six characters. The reason it is n+1 is that the string ends in a null terminating character, \0, which indicates that the end of the array has been reached and properly terminates it.

You can define a string in the same way that you would an array. You can also make them bigger than the string that they’re going to contain. When you define an array, you may give it a value right off the bat, but you can also just define their size and expand them at a later point. This also makes strings, in one manner or another, dynamic and able to be changed at a later point in the program by rewriting the data within the string.

This information is of great use to you as a programmer because strings are a fundamental part of any sort of program that handles information, especially those which handle file input and output.

We’ve already spent a bit of time rehashing information from the book prior, but just for the sake of clarity, we’re going to go ahead and define a string:

```
char myString[6] = “hello”;
```

You can then refer to this entire string at a later point by the name of the character. Most of the data that is worked with by the Arduino will be worked with in terms of bytes, and most actual textual data will be worked with in terms of C strings because characters are tremendously easy to parse.

It is important that we cover all of this so that we can actually develop an idea of how to treat strings in the context of Arduino programming alongside everything that we’re going to be working on through this book.

Serial

While you can't necessarily implement the stream class itself, you can implement its derivatives, and this is where you start to find a whole lot of utility. The serial class is an extension of the stream class that allows the Arduino board to communicate with other devices such as a computer.

Serial is enacted through both the Serial port on the Arduino as well as the USB link to the computer. In this section, we’re going to be outlining all of the different functions which make up the Serial class so that you can make the absolute most of this invaluable resource.

```
if (Serial)
```

Serial.begin(rate)

You're already familiar with this function. It allows you to start the serial transmission of data. You can specify the specific rate of data transmission in bits per second.

Serial.end()

This allows you to end serial communication. You can later restart the serial communication by calling the Serial.begin() function if you wish. While the serial communication is disabled, you can use the serial pins for generalized entry and exit of data.

Serial.find(string)

This will search for the given string within the data provided by the Serial. If the string is found, the method will return true. If the string is not found, the method will return false.

Serial.findUntil(string, OPTIONAL endString)

This will look for the string within the serial buffer until either the string is found or a specified terminating string is found. If the target string is found, then the method will return true. If the terminating string is found or if the method times out, it will return false.

Serial.flush()

This will allow you to halt processes until all data being sent to the serial has been sent. Straightforward!

Serial.parseFloat()

This will return the first floating point number to be provided by the serial stream. It will be brought to an end by any character that isn't a floating point.

Serial.parseInt()

This will return the first integer number to be provided by the serial stream. It will be brought to an end by the first character that isn't a digit.

Serial.peek()

This will return the very next character to be imported by the serial buffer. However, it will not remove the character from the buffer. This makes it fundamentally different from the Serial.read() method we'll be getting to momentarily. This means that you can simply see what character is coming next.

Serial.print(value, OPTIONAL format)

You can specify the format, optionally. Otherwise, integers will print as decimals by default; floats will print to two decimal places by default, and so forth.

You can send characters or strings as is to the print statement and it will print them without any issue.

Serial.println(value, OPTIONAL format)

This will allow you to print out values just like you would with the normal print method

Serial.read()

This will read in the data which is coming in through the serial port. Simple enough! It is added to an incoming stream of serial data called the serial buffer. When you read from this buffer, the information is destroyed, so be sure to save the data to a variable if you need to reuse it at some point.

Serial.readBytes(serialBuffer, numberOfBytes)

This will read in characters from the serial port to a buffer. You can determine the number of bytes that are to be read. Your buffer must be either a char array or a byte array.

Serial.readBytesUntil(terminatorCharacter, serialBuffer, numberOfBytes)

This will read in characters from the serial either until the given number of bytes has been read or until a given terminating character has been read. In either case, the method will terminate.

Serial.write()

This will write data to the serial port; however, this particular method only sends binary data to the serial port. If you need to send ASCII data, you

should use the print method instead.

Serial.serialEvent()

Whenever data comes to be available for use by the serial port, this function will be called. You can then use the Serial.read() function in order to read in data from the serial port.

With that, we've covered a lot of the particular functions related to the serial class and how it pertains to programming with the Arduino API. The next thing that we're going to need to work with is the Ethernet class.

User Defined Functions

One of the ways you can help keep your code neat, organized, and modular (reusable) is to use functions in your code. Additionally, they help make your code smaller by making certain sections reusable. Functions are tools that were created to serve a particular function, as the name suggests.

While we have already encountered a few user-defined functions, we will cover them in greater detail now and explain some of the features we may have glossed over when we encountered them last time. Let's look at the declaration of a function now:

```
float employeeEarnings (float hoursWorked, float payRate) {
float result; // this will be the value we return when this function is called. It should match the datatype before our function name.
result = hoursWorked * payRate
return result//  return tells the function to send a value
back once to where it was called
}
```

This function clearly takes two arguments, hoursWorked and payRate, both of which are ‘floats.’ It does some simple math on them and then returns a float as a value. Return means to terminate the function and send back whatever value is placed after the word return, usually a variable, as the result of some calculations.

Let's see us call this function now to get an employee's earnings:

```
void loop () {
floathoursWorked = 37.5;
float payRate= 18.50;
float result = employeeEarnings (hoursWorked, payRate)
// result will be 693.75
```

First, the function must be declared outside of any other functions. This means you need to write the code for the function you are creating outside of either setup() or loop(), or any other user-defined function.

Let's see another example sample sketch that could be used to smooth sensor readings:

```
int sensorSmoothing (analogPin) {
int sensorValue = 0;
for (int index = 0; index < 5, index++)
digitalWrite(LED_BUILTIN, HIGH);  //Turn on LED
for                                                                    smoothing
sensorValue = sensorValue + analogRead(analogPin)
delay(100)// 100 millisecond delay between samples
}
digitalWrite(LED_BUILTIN, LOW);//turn off LED
sensorValue = sensorValue / 5// average the values over five samples
return sensorValue;
}
```

This kind of function can be used for smoothing the data input of many sensors if they are prone to jittery inputs. This will average the samples to give a more consistent flow of data. You can see that this code is very similar to our last example:

```
void loop () {
int sensorPin = 0;// analog pin 0
int sensorValue = sensorSmoothing (sensorPin);
}
```

Here, when we try to initialize our sensorValue variable, it will call the sensorSmoothing() function on analog pin 0, and return the average result over five samples.

Functions do not always need to have parameters or return variables either. Sometimes functions can return no value and have no parameters. All they do is execute a few lines of code and then terminate, bringing the compiler back to place in the code they were called.

With all said and done, I now name you an Arduino Programmer. You now have all the knowledge you need to get started and create some incredible projects. The world of Arduino awaits!

CONCLUSION

Arduino is fun, make no mistake about that. It has limitless potential, despite its limited power and apparent size. The world continues to use these tiny pieces of technology to create staggering designs, projects, and so much more. The challenge, however, lies in learning how to use Arduino.

Throughout this book, I have explained many concepts, theories, and procedures, and explained how each of these worked. Of course, I do not expect you to grasp all of the knowledge in a single go because the fact is that it takes time, patience, consistency, and perseverance to fully learn any programming language, let alone Arduino.

Whether you are trying to enter the world of programming as a professional or simply to take it up as a hobby, Arduino is the perfect way to get started. In this book, I have given you some examples of how to create simple circuits. However, I know that there will be many who will want to create even more complex designs. Don’t worry, I am not saying goodbye without sharing a few great ideas you can get started on. If you need help, you can easily browse YouTube, Google, or your favorite search engine to gain more inspiration and have your technical queries answered.

Here are 12 brilliant projects you can work on right away:

1. Making a buzz wire game – You effectively create a simple circuit connected to a wire. The object is to transfer a looped wire across the buzz wire itself without making contact. It is fun, and it is quite challenging as well.
2. Creating an Arduino MIDI controller – For music lovers, you may be thrilled to know that you can use Arduino to create MIDI controllers to create music. Of course, it may not be your full 88-key setup, but it is a start!

3. Arduino game controller – You will need Unity 3D for this project. The game engine is available for free, and you do not necessarily need to know C# programming to create simple games. You can always download games made by other creators and play these using your own designed controller.
4. RFID smart lock – Security that looks good, feels good, and certainly brings a sense of joy when the lock opens.
5. Simple Arduino Alarm – You can create a simple alarm system that will be triggered using ping sensors. You can install the system within your premises or in your vehicle.
6. Traffic light controller – This project involves the use of three LEDs, a breadboard, some resistors and wires. You will need to use logic and conditional statements to make this project work, but it is worth the effort.
7. Mood lamp – Yes! You can also create mood lamps using Arduino.
8. Temperature controller – You created a temperature gauging device. It is now time to take things up a notch and create a temperature controlling device.
9. Recreate the classic “pong” game – This project is exciting and is endless fun.
10. Prank remote – Not all Arduino projects need to be serious in nature. If you are up for some pranks, this is a perfect way to get started!
11. Laser turret – Do not worry, as this one won’t harm anyone, but it does certainly make the project look more futuristic. It uses servo motors and some intelligent coding to create the project.
12. LED cube – There are thousands of videos out there, each creating unique designs. These cubes are perfect to use within your home to add a bit of light.

With that said, I bid you farewell and hope that this book has helped you learn a lot about the amazing world of possibilities that is Arduino.

REFERENCES

ircuitsToday.com. (n.d.). *Invention Story and History of Development of Arduino*. Electronic Circuits and Diagrams-Electronic Projects and Design. https://www.circuitstoday.com/story-and-history-of-development-of-arduino