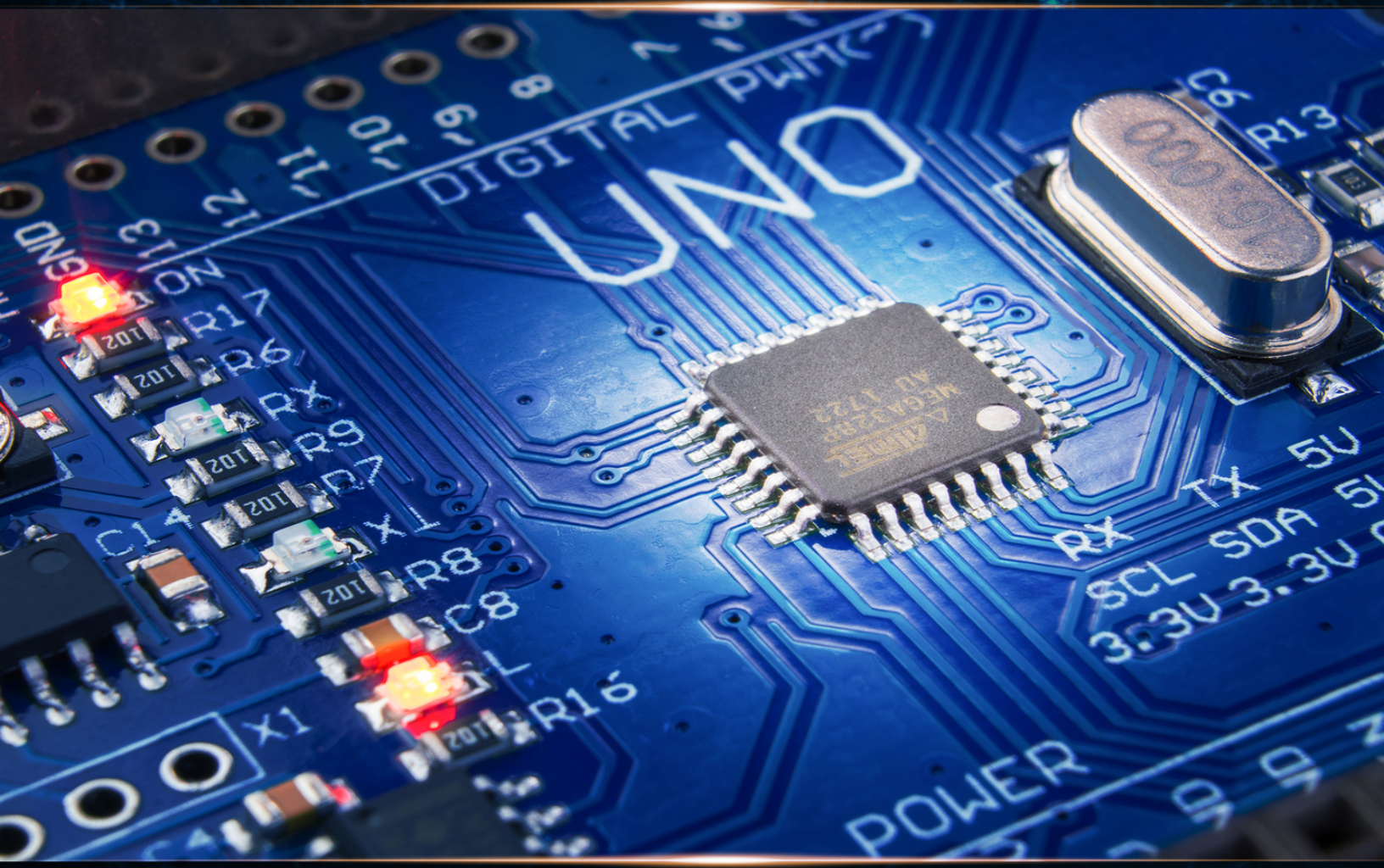
ARDUINO PROGRAMMING

ADVANCED METHODS AND STRATEGIES TO LEARN ARDUINO PROGRAMMING



STUART NICHOLAS


ARDU
PROGRA
ADVANCED METHODS
LEARN ARDUINO

Arduino Programming

Advanced Methods and Strategies to Learn Arduino Programming

Table of Contents

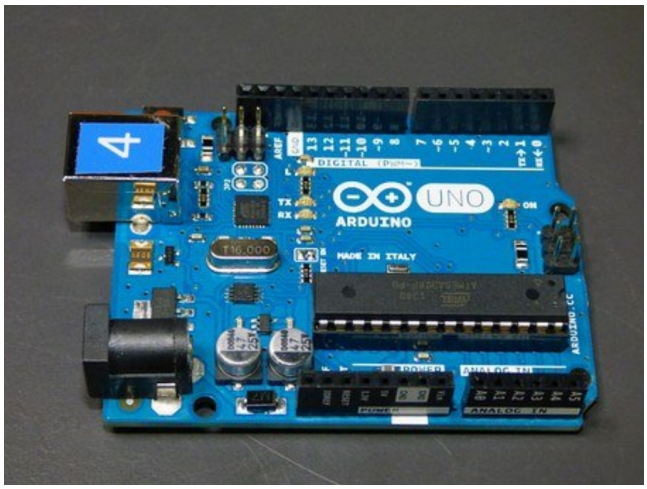
Introduction

The versatility, capability, and scope of Arduino are incredibly amazing. With hardware that is easily available in the market and its wide connectivity support, the possibilities of what users can do with this small computer are nearly endless. This is one of the major reasons why Arduino is being given more emphasis in the modern world. People are steadily discovering its hidden potential and realizing its true worth.

In this book, we will try to explore Arduino's capabilities to the furthest end and try to implement the knowledge we have learned into practicing building Arduino projects. Only talking about the theoretical possibilities of the techniques and methods is just not enough. Putting them into practice and seeing the results of your hard work provides a level of satisfaction that simply cannot be matched by anything else. The book will start by taking the readers through a brief tour exploring the Arduino environment's software and hardware aspects and then gradually proceeding towards learning how to use the Arduino’s integrated development environment. From here, we will move on to discussing advanced manipulation of elements in coding, such as variables, integers, strings, floating-points, loops, conditionals, and nests, etc. Once we have gone through initial important stuff, we will explore the higher levels of Arduino programming, where we will learn how to execute digital read and write operations, analog read and write operations, and manipulate both the board’s digital and analog pins through software and much more. Since we are focusing too much on coding anyway, we will also briefly explore some useful tips on making our Arduino programs work more efficiently without hogging too many resources by learning memory handling techniques. Finally, we will discuss a few Arduino projects which will involve circuitry, displays, and many more electronic components and hardware.

It is suggested to keep in mind that every chapter we go through has an important core concept that the reader has to learn to understand the later concepts. Even if the chapters' pace seems slow, be assured that after the initial half, the chapters will feel more concentrated and packed with advanced methods and techniques that you can employ and practice to polish your Arduino skills.

Chapter 1: Setting Up the Tools



Unlike other programming languages and their environments, Arduino is focused on allowing users who are not experienced in software and electronics programming to work on highly technical projects. Usually, such projects involve creating digital music instruments, working in robotics, games, and much more. The reason why the Arduino environment is preferred for fields such as mechatronics, robotics, electrical engineering, and even for those who are just tech enthusiasts is that users don’t need to be above-average regarding the software of their project and the electronic design as well. This makes even beginners exploring the world of technology and computing highly motivated when they find that Arduino allows them to transform their creative ideas into a tangible reality.

Due to the incredible functionality and usability of the Arduino environment, educational institutions worldwide use it for professionally training their students in their respective majors. For instance, people who are committed to creating designs and sketches of a product’s prototype will absolutely need to be aware of the product's technicalities before they can even proceed. However, with Arduino, this extra layer of complexity is shaved off from their work. They can easily focus on the design without having to worry about the sophistication under the hood of the product. In other words, Arduino itself is designed by people who are not specialists in robotics, electronics, or even engineering; this is why they have kept the

end-user as their primary focus and have given the best user-friendly experience to help facilitate the user.

Although Arduino's hardware capabilities are what makes it stand out, the reality is that without the proper software, the hardware is just a useless chunk of metal. Hence, Arduino collectively refers to both the software and hardware aspects of the environment.

The Arduino Software

Like any computer hardware, unless there’s software that is controlling it, it's nothing more than a useless pile of metal. The software allows a user to directly interact with a system’s embedded hardware, in turn utilizing its functionalities for specific purposes. This also applies to Arduino hardware as well. Since Arduino is used for specialized projects, the software needs to be tailored accordingly as well. These programs designed and developed specifically for Arduino hardware are known as ‘**sketches** .’ Just as an application is developed using the IDE of a particular programming language, sketches are also created using the Arduino IDE.

Arduino Hardware

The piece of hardware that the user will end up interacting with is the ‘**Arduino board** .’ The software that we created using the Arduino IDE directly interacts with this board, executing any instructions embedded within the program. Generally, people do not use just the Arduino board itself; on the contrary, there are extra components available that pop onto the Arduino board extending its usability. For instance, we can attach things like switches, thermal-sensors, gyroscopes, pressure sensors, LEDs, output displays, and even motors to the Arduino board, all depending on what the project requires.

When using the Arduino environment for a project, users can choose from a variety of Arduino hardware boards according to the needs of the project and the Arduino software. There are a variety of Arduino boards available in the community for users to pick up. There are official Arduino boards released by the Arduino team, as well as Arduino boards that are released by the members of the community. Suppose you are picking up a project from the community and modifying it according to your needs. In that case, it's recommended to use the tools as specified by the original project

designer or swap a few things in and out as you see fit. Arduino boards come in a variety of form-factors. You can find boards that are as small as credit cards and boards that are just large enough to design a project that features applications such as wearables. However, one thing to note is that different sized boards do not have the same specifications. Generally, smaller Arduino boards have fewer connection points for external peripherals and an underwhelming processor. In contrast, bigger boards feature a more robust processor and an increased number of connections (since the size is bigger, it can easily accommodate more ports). A quick run-through of the popular Arduino boards has been given below;

- Arduino Nano
- Arduino Bare Bones
- Boarduino
- Seeduino
- Arduino Teensy and Arduino Teensy plus

Setting Up The Arduino IDE

The Integrated Development Environment software for Arduino is available for the three most popular Operating Systems, i.e., Windows, macOS, and Linux. To download the IDE, you will have to visit Arduino's official website to do so, which has also been linked below;

http://arduino.cc/en/Main/Software

When you visit the website, select the Operating System for which you want to download the IDE application, and then follow the instructions. Once the Arduino IDE is installed, we can launch it by simply clicking on the '**Arduino.exe** ' file. However, installing the IDE is not enough because, at this point, the software (the IDE) and the hardware (the Arduino board) are not able to interact with each other. This is because we have not installed the necessary drivers for the Arduino board we are using. Luckily, this step is nothing complicated as well.

If you are using a PC running on Windows, connect the Arduino board to the computer using a USB cable. Once the board is connected, Windows will automatically find and install the necessary drivers. If the PC is unable to install the drivers properly for some reason, we can manually do this in

the device manager. The IDE installation folder also has a separate sub-folder for drivers of Arduino boards. There's a good chance that the driver for the board you are using will be in this folder as well. So open the device manager, look for the 'unrecognized' drivers, and then right-click them to open their properties. After this, all you need to do is simply click 'update driver' and choose the manual option and then specify the file directory which contains the drivers. In this way, you manually install your Arduino board's drivers.

On a Mac, the IDE application installer file will be a disk image, and inside this disk image, there is also an installation wizard for the drivers as well. This file is generally named 'FTDIUSBSerialDriver.' Once you install the IDE, then simply run this file to install the drivers as well.

The Arduino Board

In this section, we will talk about the proper way in which we can set up an Arduino board and see if it works correctly. Most of the available boards feature a small LED that lights up when connected to a power source. Check for the USB port on your Arduino board and then connect it to your PC via a USB cable. If the LED on the board lights up, it indicates that the board is working properly (powering up at least). There is another LED indicator on the board as well that can be found in the middle area. This LED lights up with an orange color and flickers to show there is no issue with the components.

Upon connecting the board to your PC, if the green LED does not light up, it means that the board has a power issue. There's a possibility that the USB port is faulty or the cable itself; however, if the peripherals are working, you should plug in the power adapter. If the LED still does not light up, then the problem is in the board's power supply. It's better to get a new one or a replacement in such cases.

The flashing orange LED light is controlled by a simple program loaded onto the board by the manufacturers themselves. If the orange LED light does not flicker, it means that either the sketch program has not been loaded on to the board's chip or is faulty. In such cases, you should manually flash the sketch program on the board. If it still doesn't work after doing that, then there's a problem with the Arduino board.

Using the Arduino's Integrated Development Environment

This section will be going over creating an Arduino sketch and then using it on the board.

Once we launch the IDE we downloaded on to our PC, we now can generate sketches, launch existing sketches, and even modify the sketches we have access to. These actions can be performed using the buttons present in the IDE's toolbar or by using assigned short-cut keys. However, we will not go into any further details of how to use the IDE's interface.

In the Arduino IDE, examples are also available to show the code for performing basic tasks on the Arduino board. To access these demonstrative sketches, you will have to go to the IDE toolbar and find the 'Files' option. In the 'Files' drop-down menu, go to 'Examples,' and you will find the examples section with a range of basic sketches for the Arduino board. For instance, the code for controlling the board's LED blinking can be found within this section as the '**Blink** ' sketch. It is easier for someone who does not have much experience with programming and coding to use sketch perfectly well because of these example codes. Apart from the basic codes, 17 example codes cover all the board's functionalities that can be controlled through sketch programs. Users can simply import the example code for the task they wish the board to execute and easily complete their projects. This convenience is very appreciated within the programming and professional community like mechatronics as well.

Once we have created a sketch we want to use on the Arduino board, we must compile it first. The board cannot use the sketch program unless it has been compiled before-hand. Once the sketch program has finished compiling, the user will see the '**Done Compiling** ' message in the IDE's console. However, one should always be wary of the sketch's size before using it with the board. This is because Arduino has limited storage support for the sketches being imported. If the sketch program exceeds the board's size limit, then the board's code execution will simply fail. Once you compile the sketch program, the sketch's size will be displayed in the IDE's console along with the size limit supported by the board itself. Different boards have different size limits, so check the maximum limit of the Arduino board you are using. An example of such a message has been shown below;

Binary sketch size: 1008 bytes (of a 32256 byte maximum)

If the instructions detailed in the sketch program exceeds the size that is supported by the Arduino board you are using, then the IDE will display the following error message;

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

If you encounter this error, you will have to shave down the sketch program to decrease its size until it can be accommodated within the board's memory. If making changes to the sketch program is not an option, then the only alternative is to simply get an Arduino board with larger memory space.

Like the Python and C programming language IDEs, the Arduino IDE compiler can also display errors made in the code of the sketch program. All the errors can be seen in the console after the compilation. However, once users import an example code and modify it by themselves, the IDE does not allow them to save these changes to the example file itself. This is to ensure that the integrity of the example codes remains intact. Once you use an example code and modify it, you will have to save it using the '**Save As** ' option, which can be found in the toolbar's 'Files' menu. Suppose you know that you'll frequently be making modifications and tweaking the source code. In that case, it is recommended to specify the sketch program's iteration to keep track of which changes were made to which iteration of the same sketch. Another important thing to be mindful of is that once a sketch program is uploaded to the Arduino board, then it cannot be downloaded back to the system from where it was originally exported. That's why you should always save your sketch programs on the system before uploading it to the board.

Exporting the Sketch Program and Executing it on an Arduino Board

This section will discuss how to export the sketch program we created and saved on our system to the Arduino board and see the code work its magic.

First and foremost, to copy the sketch from our system to the Arduino board, both need to be connected. So, we need a USB cable that has A-type

connection points on both ends. We then connect one end to the Arduino board and the second end to our system. Once the connection has been established, we are ready to export the sketch file to the board. Before we can do that, we must run the Arduino IDE and load a sketch program on to it. To keep things simple, we will export the blink sketch, which is available in the examples section in the IDE, to the connected Arduino board.

Once the IDE has loaded the sketch program we want to export, the next thing to do is to go to the IDE's toolbar and click the '**Tools** ' menu. From the drop-down list, look for the '**Boards** ' option. The Arduino board that is connected to your system will show up here; at this point, all you have to do is look for the name of the board you are using and select it. Once this is done, go to the '**Tools** ' menu again and this time, look for the '**Serial Port** ' option. Over here, you will find all the serial ports that are present on the system you are using. If your system is running the Windows OS, then you will either find a single entry or multiple entries in this section. In case you see only a single entry, then that's the port on which your board is connected to your system. If there are multiple entries, then the board will most likely be the last entry. In case you are using a Mac, then each entry shown in the '**Serial Port** ' option will have two iterations, and each iteration will have different ending values. Selecting either entry is fine.

Once you have selected the board and the corresponding serial port, all that's left to do is select the upload button on the main IDE interface or go to the '**Files** ' menu in the toolbar and select the '**Upload to I/O board** ' option from there.

Once these steps have been performed, the IDE will automatically compile the code in the sketch program. When the compilation is done, the program will be exported to the Arduino board. Once the export process is completed, the board will immediately start executing the instructions detailed in the sketch program exported to it. In this case, two LEDs will start blinking on the Arduino board as specified by the program. Once the sketch program's instruction set is completely executed, the LED's blinking will stop and the LED will be reset to its original state.

This is necessary because the PC needs to know which board the code needs to be sent to and which port connects it to the PC. Once the export process begins, the sketch currently being executed on the Arduino board is

suspended and until the new sketch program is loaded on to it. Once the process is completed, the execution of the new sketch begins. However, a thing to keep in mind is that whenever a sketch program is exported to an Arduino board, the sketch program running currently on the board is overwritten and lost. So, it is recommended to save all the sketches that you create because you might need it in the future.

There are some cases where the upload process might not be successful. Usually, the reason for this is that some old Arduino boards and some new versions have compatibility issues. In such scenarios, the IDE will display an error message to the user when the sketch program's export is unsuccessful. Most of the time, if you encounter this error, the issue is caused by an incorrect selection of the Arduino board or the serial port. Another case could be that the board is simply just not connected properly to the system. Selecting the wrong port is one of the most common causes of this issue. An easy workaround to this is to disconnect the Arduino board connected to the system and then check the ports being shown in the IDE's serial ports menu. Once you have taken a look at the available serial ports, connect the board back to the system and then check one which was not shown previously. The new port being displayed is the one through which your board is connected to the system.

Building a Sketch Program and Saving it

In this section, we will discuss the process of using the Arduino IDE to create a sketch program and then save it onto your system.

Just as how we can open Microsoft Word and choose the '**New** ' option to create an empty file and then fill it with content, in the same way, we can create a new sketch program by simply opening the IDE and choosing the '**New** ' option under the '**Files** ' menu. Once this is done, a new tab will be opened within the IDE to isolate the program you are working on from other sketches. In this tab, you can write as many code lines as necessary for your project (keeping in mind the size limit, obviously). Here's an example of a modified '**blink** ' code available in the IDE. The difference over here is that the duration in which the light stays illuminated before blinking is doubled.

```
const int ledPin = 13; // LED connected to digital pin 13
```

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

If we are using this code as a sketch program, we will have to compile it. So let's go over the process of exporting code on to Arduino boards. First, we will write this code (or copy it if you want to) in the IDE, and then we will compile it using the '**Compile** ' option in the IDE interface and export it to the board using the '**Upload** ' option either through the '**Files** ' menu or through the interface button. Once the file is uploaded, we then save it on our system as we might need it in the future. To do this, we use the '**Save** ' or '**Save As** ' option in the '**Files** ' menu.

When you save a sketch program, by default, the IDE software will set the location to a directory named '**Arduino** ' in the user's '**My Documents** .' This happens when we are using a Windows system; if it's a Mac, then the default file directory will be inside the '**Documents** ' folder. It is recommended to name your sketch programs such that they reflect their usage or the instructions; however, the name of the sketch cannot be the same as the example sketch files. This makes it easier when you are dealing with a lot of sketch programs and saves you from the trouble of having to look through each program and see what it does before you find the sketch you're looking for. Using the default save location of the IDE has its perks

as well. Sketch program files stored in this directory are automatically displayed within the '**Sketchbook menu** ' of the IDE, making it quicker and easier for the user to access.

One of the weaknesses of the Arduino IDE is its absence of sketch version management. This means that the IDE application does not have the functionality to revert the modifications made to a sketch over different iterations. So, if you find yourself frequently modifying your sketch files, then it is recommended that you keep each modified version of the program saved on your system. In this way, if you wish to use an older version of the same sketch program, you can do so.

If you are creating an Arduino IDE program from scratch or extending the example sketches, then it's a good idea to compile the program frequently. In this way, you can easily fix any errors that pop up as you finalize the code without having to deal with a whole bunch of errors at the end and getting overwhelmed with the necessary changes.

Before we move on, there's one final thing that needs to be discussed and it is the folder that contains the sketch file itself. In the Arduino IDE, the sketch programs must be stored within a folder that has a name matching the program's name.

Working with Arduino

Before we conclude this chapter, we will briefly demonstrate building a simple Arduino project using an Arduino board, some electrical components, and the Arduino IDE. This section will be brief and concise as practical projects have been discussed in detail in this book's upcoming chapters.

In this demonstration, we will be looking at a project which involves an LDR sensor connected to the Arduino board. This sensor will be responsible for controlling the frequency at which the LED on the board blinks and the duration as well. Since the core functionality is still the blinking of LED, we will be using the 'blink' sketch example and modify it according to the project's needs.

The idea here is that the LDR sensor will react to the board's LED illumination intensity. The LDR sensor will be connected to 'Pin 0' on the analog circuit and the board LED is accessed through 'pin 13' on the digital

section. Now that the project has been set up, all that's left to do is create a sketch program and then export it to the Arduino board connected to our system. The sketch which will be used is as follows;

```
const int ledPin = 13; // LED connected to digital pin 13

const int sensorPin = 0; // connect sensor to analog input 0

void setup()

{

pinMode(ledPin, OUTPUT); // enable output on the led pin

}

void loop()

{

int rate = analogRead(sensorPin); // read the analog input

Serial.println(rate);

rate = map(rate, 200,800,minDuration, maxDuration); // convert to blink rate

digitalWrite(ledPin, HIGH); // set the LED on

delay(rate); // wait duration dependent on light level

digitalWrite(ledPin, LOW); // set the LED off

delay(rate);

}
```

Before we can even leverage the sensor's functionality, it needs to be connected to the board so the two components can communicate with each other. To do this, we will have to create a simple circuit that will consist of the LDR sensor itself and a suitable resistor. A resistor with a rating between 1000 to 10,000 ohm will be work just fine. Now that we know how

the components need to be connected, let’s discuss the program demonstrated above. The core concept around which this program is designed is to define a certain value which corresponds to the intensity of the LED light. The brightness of the light emitted by the LED at pin ‘0’ will be interpreted by the LDR sensor connected to the board and its value will change accordingly. Based on this changing value, the sensor will increase or decrease the corresponding LED's voltage level accordingly. To read the value being pushed out by the sensor, we use a function known as ‘**analogRead()** .’ So, if the sensor detects that the light’s brightness is high, then the value will also be read as high by the function (in this case, the value will be ‘800’). Similarly, if the light's brightness is low, then the value given will be low as well (in this case, ‘200’). The voltage of the LED will be controlled by the LDR sensor accordingly.

We can specify the flickering rate of the LED by using the **‘map()’** function as well. The following code shown below demonstrates this;

```
const int ledPin = 13; // LED connected to digital pin 13

const int sensorPin = 0; // connect sensor to analog input 0

// the next two lines set the min and max delay between blinks

const int minDuration = 100; // minimum wait between blinks

const int maxDuration = 1000; // maximum wait between blinks

void setup()

{

pinMode(ledPin, OUTPUT); // enable output on the led pin

}

void loop()

{

int rate = analogRead(sensorPin); // read the analog input
```

```
// the next line scales the blink rate between the min and max values

rate = map(rate, 200,800,minDuration, maxDuration); // convert to blink rate

digitalWrite(ledPin, HIGH); // set the LED on

delay(rate); // wait duration dependent on light level

digitalWrite(ledPin, LOW); // set the LED off

delay(rate);

}
```

We can also use the Serial Monitor available in Arduino to take a look at various stuff to fine-tune the sketch program. For instance, we can observe the frequency at which the LED blinks at certain variable values. So, we can precisely know the blinking rate at a specific value and then set the value at which we want it to blink. This frequency is displayed in the IDE's Serial Monitor. The following lines of code demonstrate the usage of Serial Monitors;

```
const int ledPin = 13; // LED connected to digital pin 13

const int sensorPin = 0; // connect sensor to analog input 0

// the next two lines set the min and max delay between blinks

const int minDuration = 100; // minimum wait between blinks

const int maxDuration = 1000; // maximum wait between blinks

void setup()

{

pinMode(ledPin, OUTPUT); // enable output on the led pin

Serial.begin(9600); // initialize Serial
```

```
}

void loop()

{

int rate = analogRead(sensorPin); // read the analog input

// the next line scales the blink rate between the min and max values

rate = map(rate, 200,800,minDuration, maxDuration); // convert to blink rate

Serial.println(rate); // print rate to serial monitor

digitalWrite(ledPin, HIGH); // set the LED on

delay(rate); // wait duration dependent on light level

digitalWrite(ledPin, LOW); // set the LED off

delay(rate);

}
```

The LDR can also be used for purposes other than simply controlling the LED blink rate. For instance, we can wire a small speaker to ‘pin 9’ and its resistor to ground on the digital connection pins to manipulate the resulting sound’s volume.

Since we are working with sound this time, we will have to adjust the frequency at which the pin turns on and off such that it matches with a frequency in the sound spectrum. To do this, we can simply define a variable as ‘**rate** ’ and then divide it by a factor of 100. The following lines of code demonstrate this:

```
const int ledPin = 13; // LED connected to digital pin 13

const int sensorPin = 0; // connect sensor to analog input 0

const int minDuration = 100; // minimum wait between blinks
```

```
const int maxDuration = 1000; // maximum wait between blinks
void setup()
{
 pinMode(ledPin, OUTPUT); // enable output on the led pin
}
void loop()
{
int sensorReading = analogRead(sensorPin); // read the analog input
int rate = map(sensorReading, 200,800,minDuration, maxDuration);
rate = rate / 100; // add this line for audio frequency
digitalWrite(ledPin, HIGH); // set the LED on
delay(rate); // wait duration dependent on light level
digitalWrite(ledPin, LOW); // set the LED off
delay(rate);
}
```

Chapter 2: The Basics of Arduino Programming

So far, we've studied what an Arduino micro-controller is, and how the native coding environment that is provided alongside works. Let's now step up into learning about how actually to use all of this. This chapter will serve to create a basis for understanding and writing Arduino code, that we can interface into micro-controllers and create wonderful things as we so desire.

To be specific, this chapter will teach you about:

- Variables, constants, and their usage.
- The versatility of included math functions.
- Code comments.
- Coding in "Decisions" and reasoning behind it.
- Looping and reusing bits of code.

We've had some experience with Arduino IDE Client and its Web Editor and learned how to use setup() and loop(). This chapter, along with the next one, will deal with syntax, programming practices, and general how-to's

related to Arduino's programming language. We'll start by introducing the most fundamental components of basic code syntax.

Curly Bracket

The pair of curly brackets are used to tell the IDE where a block of code will start and where it will end; the left ({) for the former and the right (}) for the latter. You might have seen examples of this in prior sections, though we would like to mention these are used to make code blocks other than those of functions as well; this will be elaborated on further into this chapter.

Do note that full pairs of brackets must exist (an equal number of left and right brackets means that they're "balanced"); otherwise, errors such as crypt compiler errors will occur, in which case counting these pairs is a good idea.

Semicolons

A semicolon is used to close off and end a statement of code (i.e., put at the end, like a period for normal sentences) to distinguish it from other lines. Not doing so will result in a fairly easy-to-debug error during compilation, as the IDE will mention where this has occurred.

Another use for semicolons is separating elements in for loops, which will also be discussed later.

Comments

Comments are bits of written text tagged with a syntax (in our case, a mix of forwarding slashes and asterisks) so that the compiler doesn't read them and add them to the functionality of the program, allowing us to write whatever we want as comments to the actual code. There are two kinds of comments that the IDE has: *Block* comments and *Line* comments. Block comments are usually big blocks of text that describe the code around it, whereas line comments usually serve as a heading or a nametag for the code near it, telling the reader what it does.

A block comment uses ‘ /* ’ and ‘ */ ‘ to define where it starts and ends, respectively:

```
/* This is a block comment
```

```
This comment can span multiple lines
This type of comment is usually found outside function calls
*/
```

In contrast, a line comment begins with ‘ // ‘ and continues till the line ends. This can be put at the start of a line or somewhere in between; both of these work:

```
// This is a single line comment
Serial.println("Hello World"); // comment after statement
```

Using comments to describe your work is a very good habit for readers to be able to discern what you want to do.

Variables

Variables are the meat and potatoes of coding. These are objects that store information for further application by the functionality of the code. Every variable has a unique name used to identify it and thus the information it stores. A good habit for naming variables is using Camel Case (i.e., writing the first word in lowercase, and then the first letter of the second word in uppercase, like a camel’s humps, e.g., oldGreyDog, warmApplePie)

Another good habit is to initialize the variable by assigning it a value, which helps in errors where the variable is called without a value. To do so, we tell it what data type it will be, give it a name, use an equal sign, and then give it an initial value (usually zero).

```
int myInt = 0;
```

Datatype

What is ‘int’? It is a **datatype** , a kind of badge that says what set of values the variable can hold. Let’s examine some more built-in datatypes commonly used in coding with the Arduino language.

Boolean

This datatype is used for basic logical decisions at least as old as Shakespeare: true or false. This datatype‘s size is one bit. These are the only values this set has.

```
boolean myBool = true;
```

This code declares the variable myBool as being a boolean, with an initial value ‘true.’ As you might expect, this datatype is extremely common in Arduino, and all comparison operations produce (or ‘return’) a Boolean value.

Byte

A byte is eight bits, i.e., it is eight binary digits long, capable of referring to integer values from zero or 255. The keyword for this datatype is ‘byte.’

```
byte myByte = 128;
```

Integer

By far the most used data type, we use this to store integer values; the total number of unique values allowed is 2^16 or two bytes (sixteen binary digits long): from –32768 to 32767. We can also have it be unsigned, via adding the keyword ‘unsigned,’ to have this range go from 0 to 65,535. The keyword for this datatype is ‘int.’

```
int mySignedInt = 25;
unsigned int myUnsignedInt = 15;
```

Long

Similar to the previous datatypes, the long datatype can store four bytes (thirty-two binary digits) worth of integer numbers, from –2,147,483,648 to 2,147,483,647. ‘unsigned’ can be used as well for the same effect, going from 0 to 4,294,967,295. The keyword for this datatype is ‘long.’

```
```

```
long myLong = 123,456,789;
```

Do note that this consumes more memory (double that of integer), so take care to not use it unless it is necessary to store numbers that large.

Double and Float

These datatypes are ‘floating-point’ numbers, i.e., they allow us to go between integers and into decimal points. Both of these datatypes can store values ranging from -3.4028235x10^38 to 3.4028235x10^38.

Often (depending on what programming environment we’re using) these datatypes differ in the amount of decimal point accuracy, with float having an accuracy of six to seven decimal places as compared to double’s fifteen. However, for the Arduino platform, they both have the same accuracy of six to seven decimal places.

A few things to note: one, the precision inaccuracy may result in some weird values being returned to you, such as 6.0 divided by 3.0 giving a result of 1.9999999. two: floating-point/decimal computation is much more resource-intensive than the comparatively simpler integer math operations. For these reasons, do take to use these datatypes only when necessary.

The double datatype’s keyword is ‘double,’ while the keyword for the float datatype is ‘float.’

```
double myDouble = 1.25;
float myFloat = 1.5;
```

Character

Characters are generally referred to as the glyphs of alphabets and various other symbols, i.e., characters. However, on the back end, these characters are a numerical value that refers to the ASCII chart; we can store a character as either of these. The keyword for this datatype is ‘char’

```
char myChar = 'A';
char myChar = 65;
```

Both lines of the above code declare a character ‘myChar’ storing capital A. We can store a group of these characters using a method that will be discussed later, which will let us store words and sentences.

Arrays

An array is a storage structure that assigns an index value to some data and allows us to store multiple data entries into one structure, with the caveat that it must be of one datatype that is declared alongside the array during initialization. A few ways to define arrays are as follows:

```
int myInts[10];
int myInts[] = {1, 2, 3, 4};
int myInts[8] = {2, 4, 6, 8, 10};
```

Notice that the name has a pair of square brackets, which is how we tell the environment that we’re creating an array.

- The first line defines an uninitialized array that can store ten values in a row but doesn’t store any at the moment. Be careful of declaring uninitialized arrays, as its interactions with memory spaces can be quite weird.
- The second line initializes an array with an undefined size but with a defined number of values. In this case, the array matches the size of the row of values that we defined.
- The third line initializes an array with both a defined size and values. In this case, the defined values are assigned to the first spaces in the array, but the last three are not. Care must be taken in this case for the same reason as the first declaration, which will be demonstrated shortly.

Let’s look at how actually to use arrays:

We access a value in an array by writing the array name, the square brackets, and the value’s index inside the bracket, such as follows:

```
int myInts[] = {1, 2, 3, 4};
int myInt = myInts[1]; //counting from 0, myInts contains the second value in the array, ‘2.’
```

The above code notes that for most programming languages, we count indexes starting from zero. The following should serve as further clarification:

```
int myInts[] = {1, 2, 3, 4};
int myInt0 = myInts[0]; // contains 1
int myInt1 = myInts[1]; // contains 2
int myInt2 = myInts[2]; // contains 3
int myInt3 = myInts[3]; // contains 4
```

Now let's look at what happens when arrays aren't initialized. Run the following in the Arduino IDE, and you'll notice that the Serial Monitor displays a jumbled mess that is definitely not integers since they were never initialized.

```
int myInts[5];
Serial.println(myInts[0]);
Serial.println(myInts[1]);
Serial.println(myInts[2]);
Serial.println(myInts[3]);
Serial.println(myInts[4]);
```

Fortunately, providing values to uninitialized indexed array space is the same as providing a value to any variable, as follows:

```
int myInts[2];
myInts[0] = 0;
myInts[1] = 1;
```

Another possibility is multi-dimensional arrays, which are technically arrays stored within arrays. Two ways describing how we'd define one are as follows:

```
int myInts[3][4];
```

```
int myInts[][] = { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} };
```

They’re accessed in the same way as you’d expect:

```
int myInts[3][4];
int myInts[][] = { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} };
```

Now let’s use what we’ve learned about arrays and characters, and combine them into something that will let us write sentences:

Character Arrays

Character arrays are arrays that store characters. They’re initialized the same way as regular arrays:

```
char myStr[10];
char myStr[8] = {'A,’ 'r,’ 'd,’ 'u,’ 'i,’ 'n,’ 'o,’ '\0'};
```

Character arrays are often referred to as **Strings** of characters. The previous code describes how we’d make a string that says ‘Arduino.’ The ‘\0’ character represents a null which terminates the string by being at the end of it. This is referred to as **Null Termination** , and it helps the IDE and its functions determine where the string, stored in memory space, ends; otherwise, these functions would continue until some other null character is encountered and produce a bunch of garbage data meanwhile.

Simpler ways to declare strings are as follows:

```
char myStr[] = "Arduino";
char myStr[10] = "Arduino";
```

The first line makes an array that resizes to contain 9 characters - the word ‘Arduino’ and the null terminator, whereas the second line’s array leaves space for them to be put in their respective positions.

Note that a string object exists, which will be discussed outside of this chapter, but we’ll be using character arrays more often than not.

Let’s now look at something that allows us to lock a value in stone, providing some sort of bedrock for functions to refer to and work.

Constants

A constant remains constant, i.e., any value that it is initialized with does not change during runtime. There are two ways to declare constants; by either using the keyword ‘const’ for it or using ‘#define.’

The first method, using the keyword ‘const,’ changes the variable's behavior, similar to how ‘unsigned’ works. The ‘const’ keyword serves to make the variable read-only and cause an error if there is code that tries to change the variable during compilation. An example is as follows:

```
const float pi = 3.14;
```

We often use ‘const’ to declare constants, though we might prefer to use the other method if memory is a concern.

The second method, using ‘#define,’ allows the user to name the constant for use before the code is fully compiled, allowing you to reduce the amount of memory (by skipping allocation of the name) that is required. This trick proves useful in micro-controller models with smaller memory sizes, such as Arduino Nano.

An important thing to keep in mind while using the latter method to declare a constant is if the name we’re using for it is used elsewhere. For example, some other array or variable or even another constant, the original constant’s name will be replaced by its value instead. A good idea to counteract this is to have completely uppercase names for constants and regular Camel Case for other names.

‘#define’ also does not need to use a semicolon to terminate its line, as follows:

```
#define LED_PIN 8
```

We’ve defined objects and initialized them with values. Let’s now look at how we can use these values and work with them, using basic arithmetic.

Arithmetic Functions

The four basic arithmetic processes, summation, subtraction, multiplication, and division, are allowed for by most if not all programming languages. Arduino provides operators - characters that when used in code refer to these arithmetic operations. These processes only work between operands of the same datatype; however, an int will not add to a float. We can create an exception to this rule using casting, which will be discussed in a short while. For now, let's look at how to implement these basic functions:

```
z = x + y; // calculates the sum of x and y
z = x - y; // calculates the difference of x and y
z = x * y; // calculates the product of x and y
z = x / y; // calculates the quotient of x and y
```

Sometimes, we might only need the remainder of the division (in the case of integer operations), and we'll use another tool for this, called the modulo operator ('%'). An example of how this would work is as follows: when 5 is divided by 2, the result is 2.5. If we do not go into decimal places, this would instead return a remainder of 1. This value is the result of the modulo operation between these two numbers.

Another nifty syntax tool we can use is compound assignment operators, which combine the arithmetic operation with the assignment (or reassignment) operation of the variable. Here's a list of these operators:

```
x++; // increments x by 1 and assigns the result to x
x--; // decrements x by 1 and assigns the result to x
x += y; //increments x by y and assigns the result to x
x -= y; //decrement x by y and assigns the result to x
x *= y; //multiplies x and y and assigns the result to x
x /= y; //divides x and y and assigns the result to x
```

Alongside both of these, full-fledged functions exist to help with performing common math operations and calculations. Here's a list of those functions:

```
abs(x) // returns the absolute value of x
max(x,y) // returns the larger of the two values
min(x,y) //returns the smaller of the two values
pow(x,y) // returns the value of x raised to the power of y
sq(x) // returns the value of x squared
sqrt(x) // returns the square root of the value
```

Of particular note and usefulness, however, is the ability to compare values. Let's continue and learn about operators that allow us to do so:

Comparison Operators

The Arduino language contains operator characters that allow us to make comparisons between two objects and return a Boolean value of either one or zero to confirm true or false, respectively. Here's a list of those operators in use:

```
x == y // returns true if x is equal to y
x != y // returns true if x is not equal to y
x > y // returns true if x is greater than y
x < y // returns true if x is less than y
x >= y // returns true if x is greater or equal to y
x <= y // returns true if x is less than or equal to y
```

Let's now look at logical operators, i.e., the operators for AND, OR, and NOT logical functions:

Logical Operators

'AND,' 'OR,' and 'NOT' are three operators that refer to their respective logical operations. 'AND' checks whether both values (or the Boolean results of some comparison statement) is true and returns true, 'OR' checks for either value of the previous case and returns true, and 'NOT' returns the

opposite Boolean state of the value provided. Examples of these in action are provided below:

```
(x > 5 && x < 10) // true if x is greater than 5 and less than 10
(x > 5 || x < 1) // true if x is greater than 5 or less than 1
!(x == y) // returns true if x is not equal to y
```

Let's now look at what we referred to previously as casting.

Casting

The cast operator changes the datatype of a variable to a different one, allowing it to be used where there's a restriction of datatypes, like for arrays and arithmetic operations. Values converted via this operation are truncated to the relative decimal instead of being properly rounded up or down. For example, 8.7 being truncated to 8 instead of being rounded to 9. Thus, it is good to cast integers to float values instead of the other way around.

Casting is done by adding parentheses before the variable to be casted and writing the required datatype in it. An example:

```
int x = 5;
float y = 3.14;
float z = (float)x + y;
```

We need our program to make logical decisions as there's not a lot one can do without some sort of logic behind it. Since we've learned how to compare values and draw some logical conclusion, let's move on to using these tools and code in decision-making capabilities.

Decision Making

Decisions are often a matter of whether something will happen or not. Most programming languages, Arduino included, include an 'if' statement to that end. This function will check a statement inside a pair of parentheses, and if it is true(i.e., the Boolean statement returns a 'TRUE' Boolean value), it will run code placed inside curly brackets, as follows:

```
if (condition) {
// Code to execute
}
```

We can also use an ‘else’ statement when the conditional statement being checked returns false. The syntax is as follows:

```
if (condition) {
// Code to execute if condition is true
} else {
// Code to execute if condition is false
}
```

The else statement can also check a new statement and run if it returns a ‘TRUE’ Boolean value. Note that only the first block of code for which the respective statement returns true will execute, and the rest of the else statements will be ignored. The following code compares two variables, and the results of that comparison allows the ’if-else’ statements to function:

```
if (varA == varB) {
Serial.println("varA is equal to varB");
} else if (varA > varB) {
Serial.println("varA is greater than varB");
} else {
Serial.println("varB is greater than varA");
}
```

‘If-else' statements can get messy and expansive fairly fast, relative to how many statements we’d need to check. We use ‘switch/case' statements when we have multiple conditions (more than two or three).

‘switch/case’ statements take in a value inside parentheses (here ‘var’) and go down the list of cases, comparing the cases' value. If it finds a match, the statement inside the case runs. After its execution, we use ‘break’ to break the switch loop. If there is no match, a default case runs and then uses ‘break’ to end execution. The syntax for it is as follows:

```
switch (var) {
case match1:
// Code to execute if condition matches case
break;
case match2:
// Code to execute if condition matches case
break;
case match3:
// Code to execute if condition matches case
break;
default:
// Code to execute if condition matches case
}
```

We often need to have code repeating during runtime so that it functions continuously. We use loop structures for this. Let’s look at how to make loops now:

Looping

There are three kinds of loops: ‘for,’ ‘while’ and ‘do/while.’ These work in slightly different ways. Let’s start by looking at ‘for’ loops.

The **‘for’ loop** continuously repeats a block of code a specific number of times. This number is assigned to a variable that continuously changes per iteration of the loop which can also be used to access particular indexes of matrices.

There are three parts of a ‘for‘ loop’s statement: initialization, condition, and increment.

```
for (initialization; condition; change) { }
```

- The initialization creates a variable that the loop uses and increments (or decrements). We can initialize other variables here too, but we’d advise you to keep your code lean and only initialize variables that the loop uses.
- The condition checks the current state of the loop variable against some conditions. As long as it returns true, the code continues.
- The increment changes and updates the current state of the loop variable.

An example of a ‘for’ loop in action is as follows. The statement is set up such that the loop increments a newly initialized variable i, causing the loop to re-execute ten times:

```
for (int i = 0; i < 10; i++) {
// Code to execute
}
```

The **‘while’ loop** checks whether a statement is true or not and continues running *while* it returns a Boolean true. Care must be taken with this statement that the condition doesn’t infinitely return true, implying an infinitely repeating loop. The syntax for this loop is as follows:

```
while (condition)
{
// code to execute
}
```

An example of this loop in action would be as follows. Good practices implore you to have the conditional be a comparison statement:

```
```

```
int x = 0;
while (x < 200) {
// code to execute
x++;
}
```

The ‘while’ loop checks the condition before execution. If we want it to be executed at least once, we’ll use a **‘do/while’ loop** . This means that in the loop the condition is checked after the block of code is run. The syntax to follow is like so:

```
do {
// code to execute
} while (condition);
```

Similar to the previous, it would be a good idea to use a comparison statement as a condition. An example of this loop is as follows:

```
int x = 0;
do {
// code to execute
x++;
} while (x < 200);
```

Functions

Functions are blocks of code that perform a specific task. They are given unique names to identify them and called when needed. Programming libraries often have hundreds if not thousands of pre-defined functions, and we can create our own if we need to, as follows:

```
type name (parameters) { }
```

The declaration of a function requires us to start with defining what datatype it will return. If it will not return a value, this function has a ‘void’ datatype. Moving on to naming it, use a name that describes what it does or when it activates, such as ‘ledRedOn’ or ‘onRedCall,’ following Camel Case.

Next, a pair of parentheses encase the parameters that we need the function to require and use, to control its functionality. These parameters are passed to and used by the function within its logic to return an answer respective to the parameters. Multiple parameters can be added by separating them using commas.

A pair of curly brackets come next, and within them, we write the code block for the function to define its behavior.

A few examples of function syntaxes are as follows:

```
void myFunction() {
// Function code
}
void myFunction(int param) {
// Function code
}
int myFunction() {
// Function code
}
int myFunction(int param) {
// Function Code
}
```

- The first function does not return anything, and therefore of a void datatype, nor does it require. We use this kind of function when we do not need it to return a value or pass a parameter to it; it is entirely self-sufficient.

- The second function takes in an integer datatype parameter for its code block to use.
- The third function can return an integer datatype value but does not require a parameter to be inserted.
- The fourth combines the second and third kinds of functions and their respective functionalities.

To return a value from a function, we use the keyword ‘return.’ Let’s look at a simple example of a function:

```
int myFunction() {
var x = 1;
var y = 2;
return x + y;
}
```

Any variables that are initialized in the function are only usable in that domain, and cannot be called by anything outside of it (this also means that variables with the same names can exist uniquely inside different functions, outside of each other’s reach). The following code block demonstrates this:

```
int g = 1;
void function myFunction1() {
int x1 = 2;
}
void function myFunction2() {
int x2 = 3;
}
```

The variable ‘g’ is accessible globally, i.e., by both functions, but ‘x1’ and ‘x2’ can only be accessed by their respective functions, not outside of these domains.

Chapter 3: The Advanced Concepts of Arduino Programming

Let's now take the fundamental structures and syntaxes we've learned in the previous chapter and apply them to make some more complex Arduino-specific code structures. Do not be disheartened if coding doesn't come to you immediately; practice does make perfect, and you still have plenty of chapters to go through still.

This chapter will go through the following topics:

- Setting the pin mode of an Arduino digital pin.
- Sending and receiving values to and from an Arduino digital pin.
- Sending and receiving values to and from an Arduino digital pin.
- *Structures* , *unions* , and how to use them.
- Adding tabs to the IDE Client/Web Editor
- *Classes* , *objects* , and how to use those structures.

Setting Digital Pin Mode

In previous sections, we learned that an Arduino micro-controller board has a set of digital pins, that receive or send digital data (i.e., zeros and ones). These can only work when they're assigned the ability to either send or receive data. This is done by using the 'pinMode()' function. For basic examples like ours and smaller sketches, this function is called during 'setup()' execution, though do keep in mind that this is a function that can be called anywhere you wish.

The syntax for this function defines two parameters: 'pin' is the number of the pin being set and 'mode' being the setting of this pin, which can be given the argument 'INPUT' or 'OUTPUT' to read or send values from that pin.

```
pinMode(pin, mode);

pinMode( 11 , INPUT);
pinMode( 12 , OUTPUT);
```

A good habit is to never use a number directly as an argument to the ‘pin’ parameter and instead assign the value to a variable to be called for that parameter. This helps in keeping a value constant and stopping accidental errors. You may also use ‘#define’ to the same end.

An example of this function in action is as follows:

```
#define BUTTON_ONE 12
#define LED_ONE 11
void setup() {
pinMode(BUTTON_ONE, INPUT);
pinMode(LED_ONE, OUTPUT);
}
```

This code assigns ‘#define’constants BUTTON_ONE and LED_ONE to digital pins 12 and 11, respectively, and then sets the pins so that BUTTON_ONE receives an input and LED_ONE produces an output.

We can also use ‘pinMode()’ to change the pin’s pull-up resistor's functionality by passing the argument ‘INPUT_PULLUP’ to the mode parameter. This inverts whatever input it receives.

Let’s look at how to use these pins and their settings:

Digital Write

The function ‘digitalWrite()’ allows us to send a digital value onto the micro-controller’s digital pin, which then can be read by some other LED or sensor or electrical device. The syntax for it is as follows:

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

The ‘pin’ parameter defines the pin to be used, while the ‘value’ parameter is used to assign what value the pin should put out. This can be either HIGH or LOW (or Boolean and binary equivalents), such as follows:

```
digitalWrite(LED_ONE, HIGH);
delay(500);
```

```
digitalWrite(LED_ONE, LOW);
delay(500);
```

This code makes the pin (defined by the variable LED_ONE) put out a high voltage, wait a period of 500 milliseconds, change the pin so that it produces a low voltage, and waits a period of 500 milliseconds again (using the ‘delay()’ function). Let’s look at a fuller version of this code block:

```
#define LED_ONE 11
void setup() {
pinMode(LED_ONE, OUTPUT);
}
void loop() {
digitalWrite(LED_ONE, HIGH);
delay(500);
digitalWrite(LED_ONE, LOW);
delay(500);
}
```

This sets up pin 11 to send voltage from the board’s pin, as defined by the constant’s value and then continues looping the rest of the functionality discussed above.

Digital Read

The function ‘digitalRead()’ allows us to read what binary value is being provided to a particular digital pin. This function's syntax consists of only one parameter - the ‘pin,’ to define what pin to look at. Take heed of the fact that this function returns an integer value, instead of a Boolean one, though these values (0 and 1) are mutually interchangeable:

```
digitalRead(pin);
//a fuller example:
```

```
int val = digitalRead(BUTTON_ONE);
```

Let’s again look at a more practical example of this function in use, that reads the status of a button:

```
#define BUTTON_ONE 12
void setup() {
Serial.begin(9600);
pinMode(BUTTON_ONE, INPUT);
}
void loop() {
int val = digitalRead(BUTTON_ONE);
if (val == HIGH) {
Serial.println("Button HIGH");
} else {
Serial.println("Button LOW");
}
}
```

This code sets a ‘#define’ constant BUTTON_ONE to 12, which is used to point out the pin that is set to receive inputs via ‘pinMode().’ The loop reads the current value being sent to that pin via the button’s state and tells it to print out the respective statements for the respective conditions onto the Serial Monitor.

So far we’ve seen how to use digital pins to communicate digital data. This can easily prove to limit, so we can also use analog pins to communicate analog-form, continuous data. These pins work using Pulse Width Modulation (PWM), and the pins for these, for most Arduino boards, are pins 3, 5, 6, 9, 10, and 11. Let’s look at the functions for these analogous (no pun intended) to the digital read and write functions:

Analog Write

The function ‘analogWrite()’ allows us to write an analog value that is sent to a pin using Pulse Width Modulation, which, in short, produces a percentage of the maximum value that is given at digital 1. The syntax for this function is:

```
analogWrite(pin, value);
```

‘pin’ specifies what pin the analog value should be written on, and ‘value’ tells it the value that it is supposed to write. This value ranges from 0 to 255 (dividing the maximum value into 255 steps between the low and high states)

The following code describes how we’d use analogWrite() to make an LED glow gradually from being off to being brightly lit:

```
#define LED_ONE 11
int val = 0;
int change = 5;
void setup()
{
pinMode(LED_ONE, OUTPUT);
}
void loop()
{
val += change;
if (val > 250 || val < 5) {
change *= -1;
}
analogWrite(LED_ONE, val);
delay(100);
}
```

The ‘#define’ constant specifies the number 11, which is used to identify what pin is to be written on (and whether it’s to be written on) in ‘pinMode().’ Two variables are also declared globally, which are ‘val’ and ‘change.’ This stores the current value being sent to the pin and whether it should be changed to increment in the opposite direction according to the conditional in the sketch’s loop. A short delay is added before the loop is to repeat.

Analog Read

The function ‘analogRead()’ allows us to read a voltage in an analog way, such that every step between zero and five volts is accounted for in some manner. The value that this function returns is an integer between 0 and 1023, meaning 1023 steps divide 5V into increments of 0.0049V. Similar to the syntax of digitalRead(),’ ‘analogRead()’ only needs to be provided the number of the pin it’s supposed to read. The syntax for this function is as follows:

```
analogRead(pin);
```

Let’s look at an example of this being used to determine the temperature reported by a TMP36 temperature sensor:

```
#define TEMP_PIN 5
void setup() {
Serial.begin(9600);
}
void loop() {
int pinValue = analogRead(TEMP_PIN);
double voltage = pinValue * 0.0049;
double tempC = (voltage - .5) * 100.0;
double tempF = (tempC * 1.8) + 32;
Serial.print(tempC);
Serial.print(" - ");
```

```
Serial.println(tempF);
delay(2000);
}
```

After pin 5 is defined as the pin to be used and read, the loop reads the current value the pin is receiving. It converts it to the temperature being measured via arithmetic operations (in both Celsius and Fahrenheit), and prints this to the Serial Monitor. A two-second delay is added between loop iterations.

These functions will be mainstays in our coding examples and practice. Let's now look at code constructs to store multiple objects in a defined way. We call these constructs *Structures* .

Structures

A structure is a composite, user-defined datatype that can store multiple variables of varying datatypes. The syntax for defining this construction is as follows:

```
struct name {
variable list
.
.
};
```

The 'struct' keyword precedes the name of the structure. A pair of curly brackets stores the list of variables that we'll keep in the structure.

In the previous section's sketch, we used three different variables (of the same datatype 'double') to define different things. Let's try grouping them together using a structure, and name it 'temp_reading.' Note again that these datatypes do not need to be the same:

```
struct temp_reading {
double voltage;
```

```
double tempC;
double tempF;
};
```

Creating and using a structure object is simple, though it is a step up from defining integer values to variables. We need to use the ‘struct’ keyword and the structure's name as the datatype for initializing the value. The syntax for our particular example is as follows:

```
struct tmp36_reading temp;
```

Assigning or reading values stored in the structure’s list of variables is done as follows:

```
temp.voltage = pinValue * 0.0049;
temp.tempC = (temp.voltage - .5) * 100.0;
temp.tempF = (temp.tempC * 1.8) + 32;
```

Let’s now modify our original code so that it uses a structure and define a function that takes this function as an argument parameter:

```
#define TEMP_PIN 5
struct tmp36_reading {
double voltage;
double tempC;
double tempF;
};
void setup() {
Serial.begin(9600);
}
void loop() {
```

```
struct tmp36_reading temp;
int pinValue = analogRead(TEMP_PIN);
temp.voltage = pinValue * 0.0049;
temp.tempC = (temp.voltage - .5) * 100.0;
temp.tempF = (temp.tempC * 1.8) + 32;
showTemp(temp);
delay(2000);
}
void showTemp(struct tmp36_reading temp) {
Serial.print(temp.tempC);
Serial.print(" - ");
Serial.println(temp.tempF);
}
```

Note the inclusion of the structure globally and a function at the end that prints the variables inside the structure passed to it.

A structure is a good way to cut down clutter and group data, and it is a good idea to use them to store a group of variables that contribute to closely related actions. Another example of these groupings is as follows: the difference between these being that only one of its variables can store a value at any given time. These are called *Unions* .

Unions

This datatype allows us to store different variables of different datatypes, similar to how structures function. However, where they diverge is in the fact that only one of these variables is allowed to store data at a given instance. The syntax for this construction is as follows. The only difference between the structure's syntax and union is the keyword:

```
union name {
variable list
```

```
.
.
};
```

Let’s look at an example of a union. We’ll create a new union (with the name ‘some_data’) with a list of variables it can store:

```
union some_data {
int i;
double d;
char s[20];
};
```

Now, we’ll have a look at how its unique functionality works:

```
union some_data {
int i;
double d;
char s[20];
};
void setup() {
Serial.begin(9600);
union some_data myData;
myData.i = 42;
myData.d = 3.14;
strcpy( myData.s, "Arduino");
Serial.println(myData.s);
Serial.println(myData.d);
Serial.println(myData.i);
```

```
}
```

This code creates a union (the same as before) and initializes an object, named myData. It passes data into its variables, though printing these out onto the Serial Monitor yields the realization that only the last variable in the union declared is allowed to show its actual value.

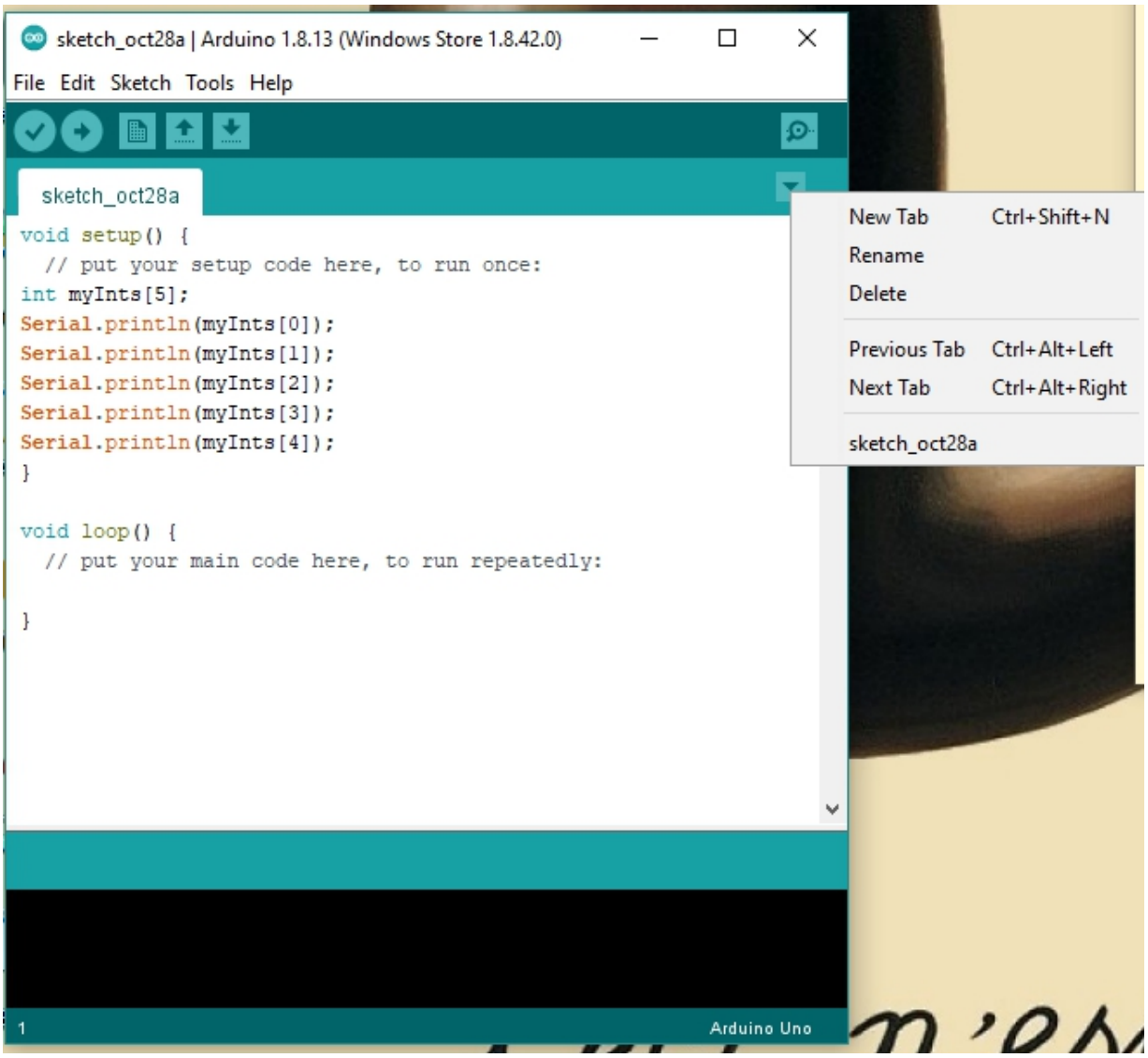
‘some_data’ will only store one value at a time, shifting from the 20 of int ‘i,’ to the 3.14 of the double ‘d,’ to finally the character array storing the value ‘Arduino’ (and its null terminator).

Before we proceed, it would be prudent to familiarize ourselves with a useful feature of the IDE Client/Web Editor; tabs.

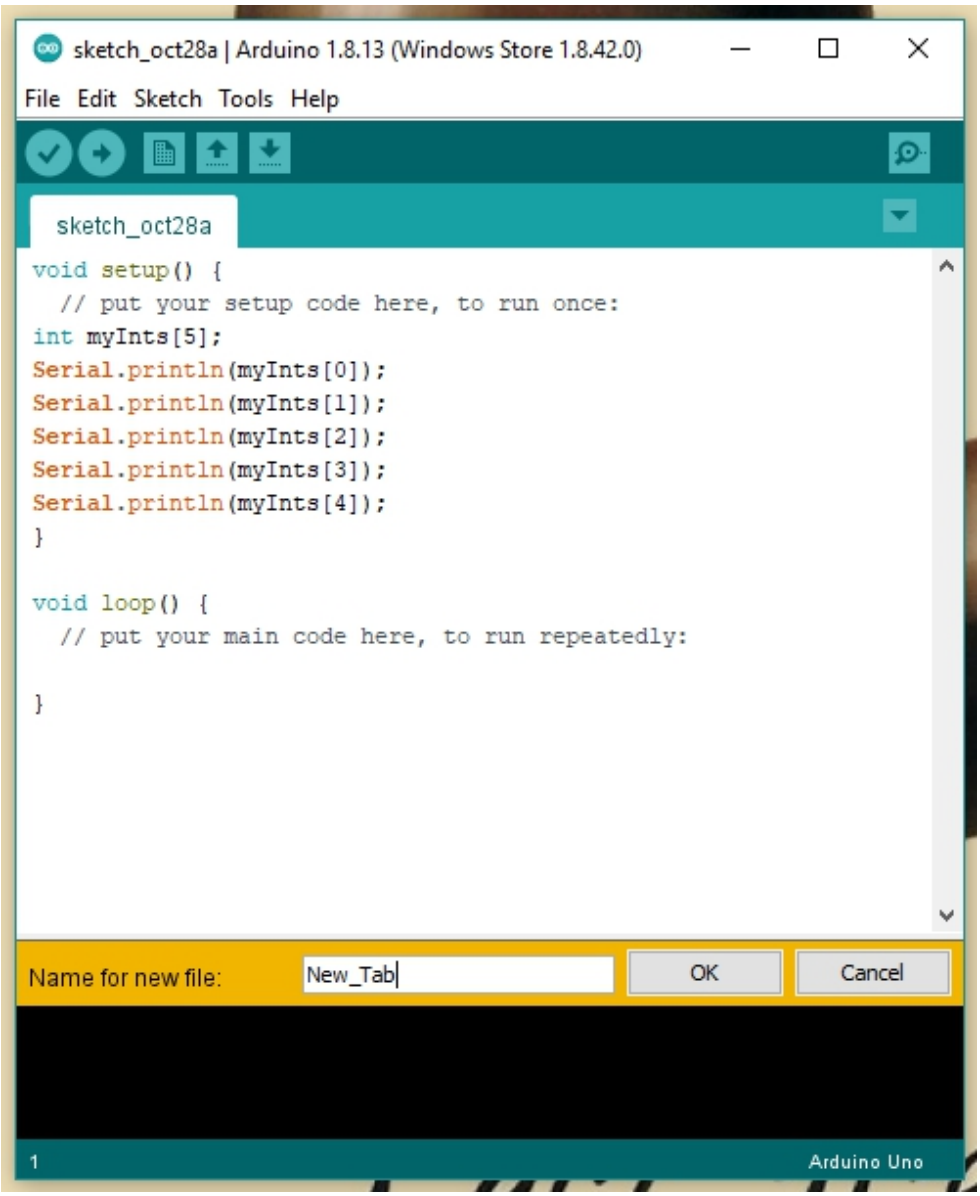
Adding Tabs

Tabs allow you to structure your workplace into different sections, allowing you to divide and conquer the task at hand and also reducing your frame of focus. Let's look at how we might add tabs:

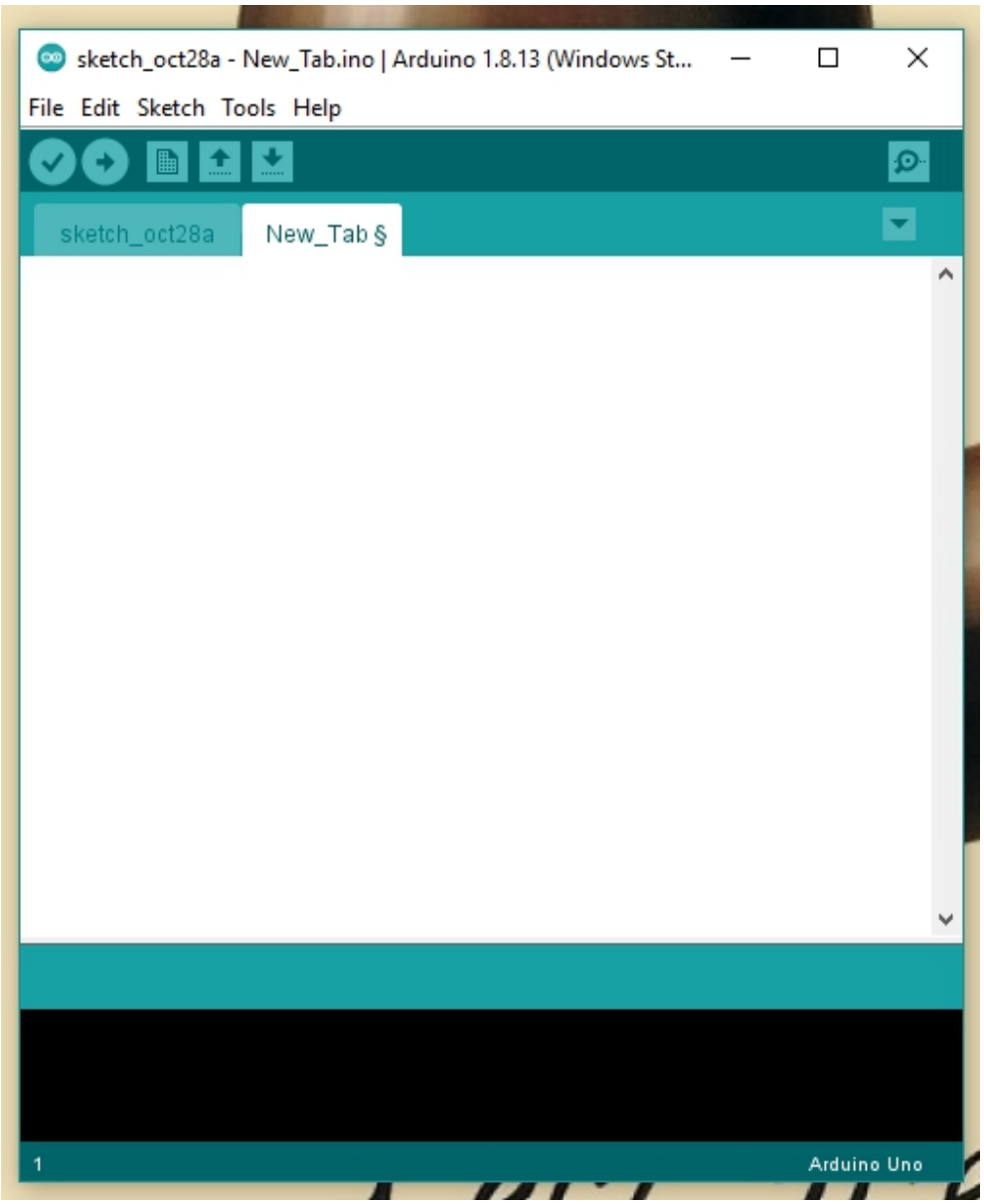
Click on the button at the IDE's top-right, with an upside-down triangle icon, like so:



In the drop-down menu that appears, press the *New Tab* option, and an orange bar will show up allowing you to name this new tab. Press the *OK* key to continue with this name and creating the tab.



After you press *OK* , the new tab appears:



Adding new tabs in the Web Editor is pretty much the same procedure. Click the upside-down triangle to get a drop-down menu, click *New Tab* , put your preferred name for the tab into the space in the yellow bar, press *OK* , and Voila! Your tab has been created.

A short note about how one might actually divide the code to distribute them into tabs. For large programs, keep the main ‘setup()’ and ‘loop()’ functions in one tab, and have related functions and objects grouped into one tab’s worth of functional table-space. Add constants globally, allowing them to be used by multiple tabs at the same time, by including them in a

header file, which contains declarations and definitions to be used outside it. Let's continue.

Working with Tabs

When we make a new tab to work with (or want to do so), the first decision is to figure out what to do with it. Let's continue our example code and create new tabs - 'led' to store functionality and 'led.h' to store constants.

Put the following block of code into the 'led.h' tab:

```
#ifndef LED_H
#define LED_H
#define LED_ONE 3
#define LED_TWO 11
#endif
```

This code defines 'LED_ONE' and 'LED_TWO,' and ensures this data is transferred to each tab only once. This is done via '#ifndef' and '#endif'. The former looks at whether 'LED_H' is defined and allows the rest of the code (before '#endif') to run if it is not.

Once we are done dealing with the preprocessors, we can now proceed to insert actual instructions to be executed by the compiler. These lines of code are to be inserted in the tab named '**led** .'

```
void blink_led(int led) {
digitalWrite(led, HIGH);
delay(500);
digitalWrite(led, LOW);
delay(500);
}
```

If we analyze this small portion of code, we will see that the '**void** ' function includes a series of parameters that control the LED's blinking by

defining the intervals (each interval has a 500ms delay).

In the main tab, write the following to include the header file that we defined into the sketch, using '#include'. Trying to refer to our constants without using this line produces an error: "the constant was not declared in this scope", meaning that the compiler could not found the constant.

```
#include "led.h"
```

Header files that we write from the sketch are surrounded by double-quotes, while header files from some other library are surrounded by less-than and greater-than signs. This will become useful later.

Finally, we use the function that we defined in the '**led** ' tab previously and use it in our program's loop. Also, you will see that in this demonstration, we did not use the '#include' preprocessor, this is because the tab we are using does not have any header in the first place.

```
#include "led.h"
void setup() {
// put your setup code here, to run once:
pinMode(LED_ONE, OUTPUT);
pinMode(LED_TWO, OUTPUT);
}
void loop() {
// put your main code here, to run repeatedly:
blink_led(LED_ONE);
delay(1000);
blink_led(LED_TWO);
}
```

At this point, we are now ready to take this program and execute it on our Arduino project and see the results.

Object-Oriented Programming

Object-Oriented Programming, or OOP for short, is a methodology of coding practices that aims to break down a program into modular component objects that interact with one another to create its functionality. An example would be an LED object, containing variables and such to define how we want it to work. But we need a structure or blueprint of sorts to add these variables to, which is where **classes** come in.

Create two tabs and name them ‘led.cpp’ and ‘led.h.’ The former will contain the main body of code, while the latter will contain definitions and initializations, but also importantly the definition of our class. Add the following to ‘led.h’:

```
#ifndef LED_H
#define LED_H
#define LED_ONE 3
#define LED_TWO 11
class Led{
int ledPin;
long onTime;
long offTime;
public:
Led(int pin, long on, long off);
void blinkLed();
void turnOn();
void turnOff();
};
#endif
```

This code is similar to what we used in the previous section’s header file. The key difference is the addition of a class definition, containing three

variables (or ‘properties’) inside it. In order of appearance, these variables define what pin to send data to, how long to keep the LED on, and how long it should stay off.

Next, we use a constructor to make a class instance, and after that three functions (or methods) are initialized.

Now, add the following block of code to ‘led.cpp,’ to give it functionality:

```
#include "led.h"
#include "Arduino.h"
Led::Led(int pin, long on, long off) {
ledPin = pin;
pinMode(ledPin, OUTPUT);
onTime = on;
offTime = off;
}
void Led::turnOn() {
digitalWrite(ledPin, HIGH);
}
void Led::turnOff(){
digitalWrite(ledPin, LOW);
}
void Led::blinkLed() {
this->turnOn();
delay(onTime);
this->turnOff();
delay(offTime);
}
```

We import two header files: ‘Arduino.h’ and our own ‘led.h’ into the code's main body. ‘Arduino.h’ contains all the libraries and their functions that Arduino has. It is usually automatically called, but this is not the case for different tabs, and we need to re-initialize this header file for our new environment.

Next up is the class constructor which is required to create a method for initializing a class. The syntax for it is to, one, name the constructor the same as the class, and two, separate these names via a pair of colons (::). This class contains the pin and a call to the pinMode function.

Led::turnOn() and Led::turnOff() turn the LED on or off using digitalWrite(), and we’ll use these to define how blink_Led() works. Note that we call functions and methods of a class using ‘->’ operators, and use the keyword ‘this’ to specify the current instance.

Let’s now go back to the main tab and put all of this together:

Add the ‘#include’ line to include our header’s functionality, after which you should create a globally accessible ‘Led’ class object, whose variables we’re passing into the class constructor:

```
#include "led.h"
Led led(LED_ONE, 1000, 500);
```

Use the class method to blink the LED (syntaxed as ‘Led.blinkLed()’) in the main tab as follows:

```
#include "led.h"
Led led(LED_ONE, 1000, 500);
void setup() {
}
void loop() {
led.blinkLed();
}
```

Uploading this code onto our previous prototype will make our LED blink.

Let's now discuss strings, objects made of character arrays, and Arduino's libraries for using them:

String Library

One of Arduino's basic libraries, this library allows us to work with character arrays in an extremely user-friendly manner. Strings are more flexible than these arrays, and allow us to work with text in a much easier and more intuitive fashion, though it does take more memory than them.

We can create a string object in a variety of ways:

```
String str1 = "Arduino";
String str2 = String("Arduino");
String str3 = String('B');
String str4 = String(str2 + " is Cool");
```

The first and second lines simply create strings that contain the word 'Arduino.' The third line creates a string with one character 'B,' and uses a pair of single colons to do so. The fourth line *concatenates* two strings, combining them at their ends into one new string.

We can also create strings from numbers, which is allowed for by pre-built constructors. A few examples of this are as follows:

```
String strNum1 = String(42); \\ return s a string '42'
String strNum2 = String(42, HEX); \\ returns a string '2a'
String strNum3 = String(42, BIN); \\ returns a string '101010'
```

Other functionalities in the string library include:

- concat(string), concatenates one string to the prior string's end
- endsWith(string): A Boolean conditional that returns true if the first string's end is the same as the argument string.
- equals(): returns true if both strings are the same.

- equalsIgnoreCase(): returns true if both strings are the same, ignoring lower- or upper-case differences
- length(): provides an integer value of the number of characters inside the string (without the null character).
- replace(substring1, substring2): replaces all instances of the first argument with the second argument.
- startsWith(string): A Boolean conditional that returns true if the first string's beginning is the same as the argument string.
- toUpperCase(): converts a string fully to uppercase, where possible.
- toLowerCase(): converts a string fully to lowercase, where possible.

Strings can be used in place of character arrays when needed, but be mindful of the fact that it requires more memory and time to execute. More often than not, this constraint is the only reason why character arrays are used for storing text instead of strings.

Arduino Motion Sensor Project

We shall learn, in this section, the process, and techniques for integrating a motion sensor with Arduino hardware. To do so, we shall use a pre-built motion sensor named HC-SR501. Owing to its simplicity of use, reasonable price range, and practical, flexible purpose, it is often one of the first tools that people use with Arduino (and the program) to test and develop an understanding of micro-controllers. It is often included in Arduino starter packages. We will learn how to:

- Attach the sensor to the Arduino.
- Read the output of the sensor.
- Create and read a Fritzing diagram of a completed build.

Introduction

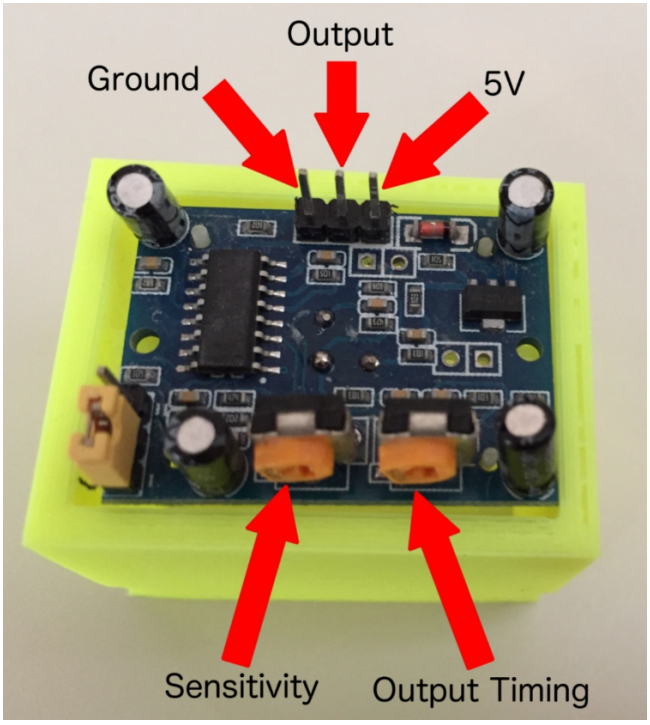
Most common motion sensors will use infrared light to track whether something’s passed in their vicinity. These sensors, termed **Passive Infrared Sensors** (PIR Sensors), detect motion in a small range around them (usually up to two meters for simpler sensors, but industrial and military models with a larger range and higher accuracy exist). These are

made of pyroelectric sensors which can read radiation (emitted by every object with a temperature higher than absolute zero) in the infrared range passively. This means that it can only read this information and not generate anything that other devices can work off of.

The pyroelectric sensor is divided into two halves. If there is no motion, both halves report the same amount of radiation and nothing happens, but if there is motion (within the sensor's range) the halves detect different amounts of radiation, and this dissonance causes the sensor to go off.

These sensors are available in a wide variety of constructions, shapes, sizes, and purpose-built configurations, due to their small size and power draw, and inexpensive nature. A few examples of products that use PIR sensors include automatic lights, holiday decorations, pest control setups, burglar alarms, and tripwires.

As we previously mentioned, we shall be using a motion sensor IC called HC-SR501. The following image shows the various adjustment screws and connectors on the body of the IC.



We'll explain what these screws and pins do, briefly. Screws act as analog switches, able to change values in a continuous range manually from the IC body itself, and different screws change different function variables. Pins

allow current to flow to and from the IC, allowing communication with the Arduino motherboard. Let's elaborate on the labeled screws and pins and what they do.

- **Sensitivity:** Changes the range at which the sensor is effective, from 3 meters to 7 meters. This value goes down with the clockwise rotation of the screw.
- **Output Timing:** Changes the delay or period wherein the sensor puts out a Boolean true state (a 'high' or '1' state), from 5 seconds to 300 seconds (5 minutes). This value goes up from the lower end with the clockwise rotation of the screw.
- **Ground:** Connect to the Ground of your connection (breadboard ground rail, Arduino Ground pin, source ground, etc.)
- **5V:** Connect to the Live of your power source, preferably a 5V voltage (breadboard power rail, Arduino 5V pin, source power, etc.)
- **Output:** Connect this to the relevant Arduino pin of your choosing. This sends out the IC output to the other ends of the wire (i.e., the micro-controller) for the period specified via the Output Timing screw.

Let's continue onwards and build a circuit using this sensor.

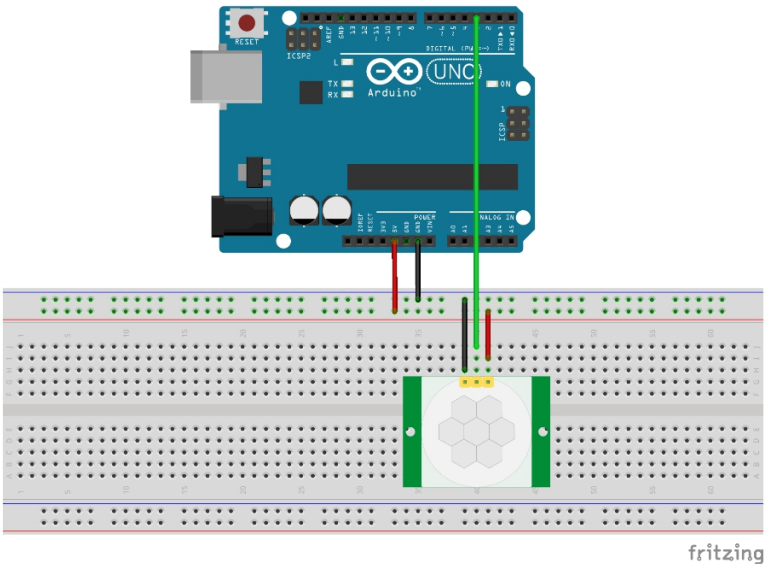
Components

For our circuit we shall require the components listed:

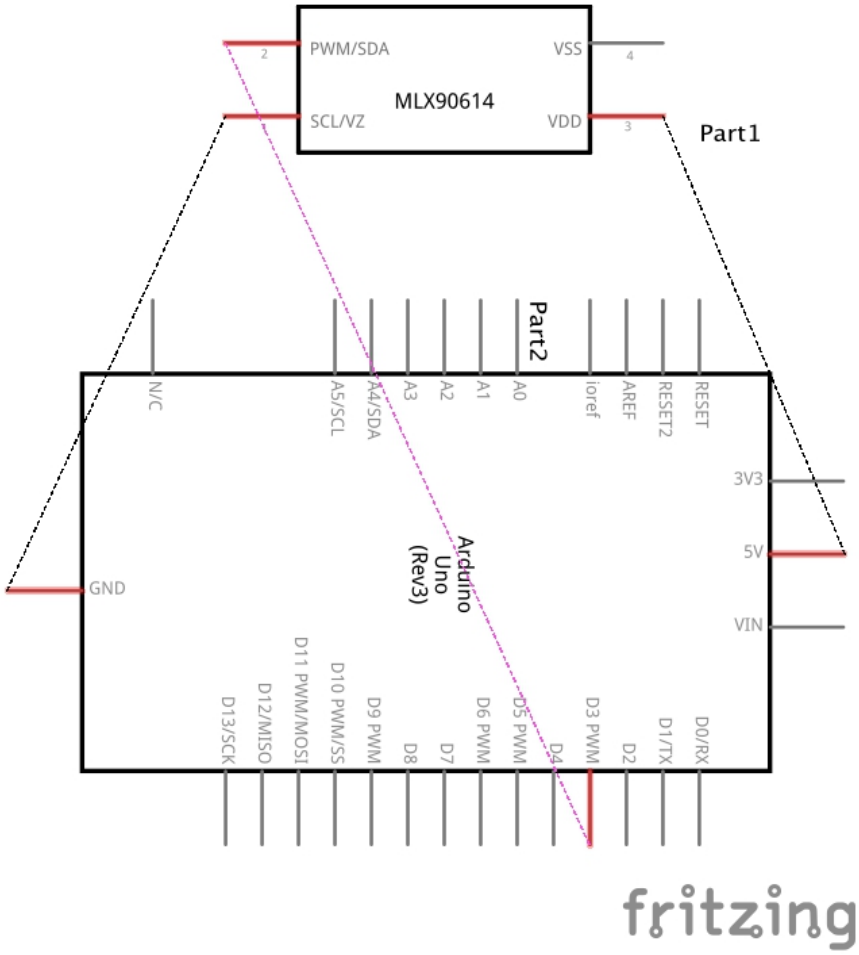
- Arduino Uno (or similar)
- HC-SR501 Motion Sensor
- Jumper Wires
- **Optional:** LED
- **Optional:** Breadboard

Circuit Diagram

The Fritzing diagram for our build-to-be is as follows:



Notice the connections with the HC-SR501 IC’s power pins. The Ground pin is connected to the breadboard’s ground rail, the 5V ping is connected to the power rail, and the output pin is connected to a digital input pin on the Arduino, whose respective Ground and 5V pins are also connected to the breadboard as described. The circuit schematic for this setup is as follows:



Code

Using the HC-SR501 sensor with Arduino is fairly simple; we only need to read its digital state. A HIGH state indicates the sensor has ‘seen’ something (this signal, again, lasts for however long we specified via the analog screw, which we’ll usually keep within the range of a few seconds), and a LOW state indicates that it hasn’t seen anything yet. We’ll send forth this status to the serial console, as follows:

```
#define MOTION_SENSOR 3

void setup() {

pinMode(MOTION_SENSOR, INPUT);

Serial.begin(9600);

}
```

```
void loop() {
int sensorValue = digitalRead(MOTION_SENSOR);
if (sensorValue == HIGH) {
Serial.println("Motion Detected");
}
delay(500);
}
```

We use #define to create a MOTION_SENSOR variable that will read the state of pin 3 (set via pinmode()). We'll also set up the serial console via setup().

The main loop (i.e., loop()) operates as follows: digitalRead() is called to read the sensor output on pin 3, and sensorValue takes up this value. When it is HIGH, a message ('Motion Detected') is printed in the serial console. When it is LOW, nothing happens. A 0.5 second delay later, the loop restarts.

Running the Project

Have the console open while the code runs to see the output, and wave something in front of the sensor which would then make the Serial Console print out the message.

Challenge

Try adding an LED that lights up when the sensor goes HIGH. Remember to add a resistor to the circuit. The following code lights up an LED connected to pin 5 when the condition is met:

```
#define MOTION_SENSOR 3
#define LED 5
void setup() {
pinMode(MOTION_SENSOR, INPUT);
pinMode(LED, OUTPUT);
```

```
digitalWrite(LED, LOW);
Serial.begin(9600);
}
void loop() {
int sensorValue = digitalRead(MOTION_SENSOR);
if (sensorValue == HIGH) {
Serial.println("Motion Detected");
}
digitalWrite(LED, sensorValue);
delay(500);
}
```

Arduino LCD Display Project

Components required

- 1 Arduino Uno or compatible board One Nokia 5110 LCD
- 1 K **Ω** resistor (one)
- 10 K **Ω** resistors (four)
- One breadboard
- Jumper wires

Circuit Diagrams

The diagram shown below demonstrates the circuit diagram needed for this topic's project:

Instead of the 5V that we have used in the prior projects, the Nokia 5110 LCD should utilize the Arduino's 3.3V power. The 3.3V input lines on the LCD are protected by utilizing the inline resistors. A 1K **Ω** (ohm) resistor is utilized by the CE lines and the 10K **Ω** resistor by the rest.

The table below tells that what pins on the Nokia 5110 LCD module are joined to what pins on Arduino:

Arduino	5110 LCD
3	RST
4	CE
5	DC
11	DIN
13	CLK
3.3V out	VCC
GND	BL
GND	GND

To turn it off, the back-light is set to the ground. The 3.3V power out that was utilized for the VCC pin, you can join the pin to it if you want to utilize the back-light.

Now let’s have a look at how we can show things on the LCD:

Code

We should install two Adafruit libraries, to begin with. The two libraries are as follows:

- The Adafruit GFX Library
- The Adafruit PCD8544 Nokia 5110 LCD Library.

The Above mentioned libraries are installed because we need them and the SPI library. To do this, the below-written include statements are added at the start of the sketch:

```
#include <SPI.h>

#include < Adafruit_GFX.h>

#include < Adafruit_PCD8544.h>
```

We won’t wish to begin an instance of the type i.e., the Adafruit_PCD8544 by utilizing the code written below:

```
Adafruit_PCD8544 display = Adafruit_PCD8544(13, 11, 5, 4, 3);
```

The CLK, DIN, DC, CE, and RST pins respectively are connected to the Arduino pin numbers that are basically the parameters.

To set up the Adafruit_PCD8544 instance, we should add the below-written code in the setup ( ) function:

```
Serial.begin(9600);

display.begin();

display.setContrast(40);
```

Now the remaining piece of code can be written in the setup ( ) function or in the loop ( ) for the test purposes. Let us begin with lighting up only one pixel on the display. We can do this by using the function drawPixel ( ) as shown in the piece of code below:

```
display.clearDisplay();

display.drawPixel(10, 10, BLACK);

display.display();
```

We need to clear the buffer and the display before we draw something on the screen. This is done by using the function clearDisplay ( ). After this, to light up only one pixel that is located at X coordinate 10 and Y coordinate 10, we use the function drawPixel ( ). As mentioned in the previous code we should run the display ( ) function before anything is shown on the LCD. This is very important to keep in mind to run the function clearDisplay ( ) prior to drawing anything on the LCD, and once we are done drawing everything on the LCD screen to display it, we should run the display ( ) function.

Drawing a line

Instead of calling the drawPixel ( ) function several times to draw a line, it would be much simpler to call the function drawLine ( ) as shown in the code below:

```
// draw a line

display.drawLine(3,3,30,30, BLACK);
```

```
display.display();
```

The function drawLine ( ) accepts 5 parameters. The first and second parameters are the X and Y coordinates for the beginning point of the line. The third and fourth parameters are the X and Y coordinates for the ending point of the line. The last and fifth parameter shows the color with which the line will be drawn. In this case, only two colors i.e., Black and White are available as the display of Nokia 5110 LCD is monochromatic.

Displaying Text

The Adafruit library additionally makes it a lot simpler to show the text on Nokia 5110 LCD. We can display the text by using the following code:

```
// Display text

display.setTextSize(1);

display.setTextColor(BLACK);

display.setCursor(0,0);

display.println("Hello, world!");

// Display Reverse Text

display.setTextColor(WHITE, BLACK);

display.println(3.14);

// Display Larger Text

display.setTextSize(2);

display.setTextColor(BLACK);

display.print("This is larger text");

display.display();
```

To set the size of the text, the function setTextSize ( ) is used. We can see in the first example that the size of the text is set to ‘1.’ The color of the text is set by using the setTextColor ( ) function. As we know that the display of

Nokia 5110 LCD has a monochromatic so only two colors i.e., Black and White are available. To write the text on the screen, the position of the cursor is set to the position on the screen by using the setCursor ( ) function. For this example, the position of the cursor is set to the upper left corner of the screen. At last, the message i.e., Hello World! is displayed on the screen by using the function println ( ).

In the next example of the above code, the foreground color is set to WHITE and the background color is set to BLACK to reverse the text by using the setTextColor ( ) function. Afterward, to display the value of PI on the screen the println ( ) function is used. As the setTextSize ( ) function is not called so the text size stays the same i.e., ‘1.’

In the third example of the code above, the text's size is set to ‘2’ and the color is set to ‘BLACK.’

Rotating Text

The text can also be rotated. This can be done by the following code:

```
display.setRotation(1);

display.setTextSize(1);

display.setTextColor(BLACK);

display.setCursor(0,0);

display.println("Hello, world!");

display.display();
```

The text can be rotated counterclockwise using the function setRotation ( ). To rotate the text 90 degrees counterclockwise we can use the value ‘1.’ Similarly, to rotate the text to 180 and 270 degrees counterclockwise we can use the values ‘2’ and ‘3,’ respectively. The following image display the text when the code written above is run:

Note that if the text's length is longer than the limit of what can be displayed on a single line of the screen, the text will move to the next line.

Basic Shapes

The Adafruit library additionally makes it possible for us to make basic shapes on the LCD. These shapes are rectangles, rounded rectangles, and circles. There also exist different functions that allow us to make and fill these shapes. The below-written code and images tell us the best way to utilize the functions of a circle:

```
display.drawCircle(display.width()/2, display.height()/2, 6, BLACK);
```

Filled Shape

```
display.fillCircle(display.width()/2, display.height()/2, 6, BLACK);
```

Four parameters are required in the circle function. The first and second parameters, the X and Y coordinates, are for the circle's center. In the above-written codes, the center of the screen is considered as the center of the circle. The radius of the circle is the third parameter. And the last and the fourth parameter of the circle function is the color of the circle. This parameter is also considered as the color to fill the circle for the function fillCircle ( ).

Rectangle

In this section, we will look at codes that correspond to displaying different types of rectangles. For instance, we will see how to display a rectangle that is internally filled with pixels, a rectangle whose corners are smoothly rounded off instead of being sharp and pointed, and finally, a rounded corner rectangle that is internally filled with pixels.

To display a standard rectangle, we use the following code

```
display.drawRect(15,15,30,15,BLACK);
```

Filled Rectangle

To display a rectangle that is filled on the inside, we use the following code.

```
display.fillRect(15,15,30,15,BLACK);
```

Rounded Rectangle

We use the ‘drawRoundRect()’ function from the ‘display’ class and pass it a total of five parameters for drawing a rounded rectangle. The first and second parameters are the X and Y coordinates of the rectangle's upper left corner. The third and fourth parameters are the X and Y coordinates of the

rectangle's lower right corner. The fifth parameter shows the color to draw the rectangle. This last parameter is also used to fill the rectangle with the respective color by using the fillRect ( ) function.

To display a rectangle that has rounded corners, we use the following code.

```
display.drawRoundRect(15,15,30,15,4,BLACK);
```

Filled Rounded Rectangles

```
display.fillRoundRect(15,15,30,15,8,BLACK);
```

In this +case, there are a few minor changes, we just need to include an extra parameter to tell the code to fill in the internals of the rounded rectangle and use the '**fillRoundRect()** ' function instead of the '**drawRoundRect()** ' function. The initial four parameters are similar to those of the normal rectangle functions, the two of which are the upper left corner coordinates. The other two are the coordinates of the lower right corners of the rectangle. The fifth parameter tells how much to round the rectangle's corners and the last and sixth parameter shows the color to draw and fill the rounded rectangle.

The examples of this chapter show that with the Nokia 5110 LCD we can do much more than text. And the Arduino library makes it simple to utilize.

We learned to add the monochromatic display of the Nokia 5110 LCD to put projects in this topic. These displays can extraordinarily improve the client experience of practically any project since we can mention to the clients what's going on with the project and tell them if there is an issue.

Chapter 4: Structuring and Arduino Programming

This chapter will focus a bit more on Arduino programming and structuring the sketches we build.

Structuring and Arduino Program

Arduino programs are normally alluded to as sketches, to underline the agile nature of advancement. The words program and sketch are compatible. Sketches have code – the guidelines the board will complete. Code required to run just a single time (for example, to set up the board for your application) should be kept in the setup function. While the code which has to be run consistently after the first step has completed goes into the loop function. Below shown is a regular sketch:

```
const int ledPin = 13; // LED connected to digital pin 13

// The setup() method runs once when the sketch starts

void setup()

{ pinMode(ledPin, OUTPUT); // initialize the digital pin as an output

}

// the loop() method runs over and over again,

void loop()

{

digitalWrite(ledPin, HIGH); // turn the LED on

delay(1000); // wait a second

digitalWrite(ledPin, LOW); // turn the LED off

delay(1000); // wait a second
```

```
}
```

Once the sketch program's transfer from the PC to the Arduino board is completed, the board begins from the start of the sketch and completes the directions consecutively. It also does the same when the board has the code and is turned on. It executes the code once in the setup function and afterward executes it in a loop. When it gets to the end limit of the loop (set apart by the closing bracket,}) it returns to the start of the loop.

This example persistently flashes a LED by composing HIGH and LOW yields to a pin. Refer to Chapter 5 to become familiar with utilizing Arduino pins. When the sketch starts, the code in setup sets the pin mode (so it’s equipped for lighting a LED). When the code's execution in setup is finished, the code in the loop is consistently executed to flash the LED until the Arduino board is switched on.

You don’t have to realize this to compose Arduino sketches, yet expert C and C++ developers may ponder where the expected main ( ) function (which is the entry point) has gone. Actually, it's present there, yet it's covered under the covers by the Arduino build environment. A halfway document is made by the build process that has the sketch code along with some other explanations:

```
int main(void)
{
        init();
    setup();
    for (;;)
        loop();
        return 0;
}
```

In the above sketch we see that the Arduino Hardware is initialized as at the start of the code the function init ( ) is called. After the init ( ) function the next function called is the setup ( ). After both these functions the last and

final function is called, that is the loop ( ). It is called repeatedly. And at last we see that the statement ‘return’ does not run. The reason behind this is that the for loop does not end.

Using Standard Variable Types

Arduino has various kinds of variables to represent the values more efficiently. You need to understand how to choose and utilize these data types of Arduino.

In the Arduino applications we see that the most basic data type used is int. int is an abbreviation for integer and is a 16- bit value. The following table highlights the numerous data types used in Arduino and their explanations, originally given by Michael Margolis in his book ‘**Arduino Cookbook** ’ which is a very good read.

Numeric types	**Bytes**	**Range**	**Use**
int	2	-32768 to 32767	Represents positive and negative integer values.
unsigned int	2	0 to 65535	Represents only positive values; otherwise, similar to int
Long	4	-2147483648 to 2147483647	Represents a very large range of positive and negative values.
unsigned long	4	4294967295	Represents a very large range of positive values.
Float	4	3.4028235E+38 to -3.4028235E+38	Represents numbers with fractions; use to approximate real world measurements.
double	4	Same as float	In Arduino, double is just another name for float.
boolean	1	false (0) or true (1)	Represents true and false values.
char	1	-128 to 127	Represents a single character. Can also represent a signed value between -128 and 127.
byte	1	0 to 255	Similar to char, but for unsigned values.
Other types			
string	Represents arrays of chars (characters) typically used to contain text.		
void	Used only in function declarations where no value is returned.		

(Reference: Arduino Cookbook by Michael Margolis)

If the values do not surpass the range and you do not have to deal with fractional numbers, then the variables proclaimed utilizing int will be appropriate. This is only possible for circumstances in which efficient memory or most of execution is not required. In the official example codes of Arduino, we can see that int data type is preferred for numeric values. But in some cases, you are required to select a different data type.

One might encounter different scenarios where they might either require a positive value or a negative value. So, signed and unsigned are the two numeric types that can be used to represent both positive and negative values respectively, variables are used without the keyword ‘unsigned’ in front of them. Positive values are always considered as unsigned. A signed variable has half the range of an unsigned variable so one main cause to utilize unsigned variables is the point at which the range of signed values won’t be in the limit of the variable. One other cause is that the developers prefer unsigned values because they want to show the individuals that they will not get a negative value.

True or false are the two potential outcomes of a Boolean data type. These kinds of data types are normally utilized to see the switch’s condition. Instead of using the words true or false, you can also utilize the words HIGH or LOW, respectively. For example, instead of using digitalWrite (pin, true) or digitalWrite (pin, 1), you can write digitalWrite (pin, HIGH) as it feels like a more suitable approach to turn on a LED. However, these all are considered identical when the code really performs the execution.

Floating- Point Numbers

The numbers having decimal points are represented with the help of Floating-point values. Fractional numbers can also be expressed using this approach. Now you are required to write a code that will help you to compute and analyze these numbers.

The code shown below tells you how to express floating-point variables, demonstrates issues you can come across while the comparison floating-point values, and also explains how to solve these issues:

```
/*

* Floating-point example
```

```
* This sketch initialized a float value to 1.1
* It repeatedly reduces the value by 0.1 until the value is 0
*/
float value = 1.1;
void setup()
{
Serial.begin(9600);
}
void loop()
{
value = value - 0.1; // reduce value by 0.1 each time through the loop
if( value == 0)
Serial.println("The value is exactly zero");
else if(fabs(value) < .0001) // function to take the absolute value of a float
Serial.println("The value is close enough to zero");
else
Serial.println(value);
delay(100);
}
```

The calculation performed by the Floating-point numbers isn’t precise. We can face some errors in these calculations. As we know that the numbers expressed internally just hold estimation so these mistakes happen in the light of the fact that a vast range of numbers falls in the category of

floating-point numbers. These mistakes bound you to check whether the values are in a **‘range of resilience’** or not.

The following is the output from this sketch:

```
1.00

0.90

0.80

0.70

0.60

0.50

0.40

0.30

0.20

0.10

The value is close enough to zero

-0.10

-0.20

The output keeps on giving negative numbers.
```

This might be possible that you assume the loop to terminate as its value becomes equal to ‘0.1.’ Afterward, from this value ‘0.1’ is deducted. In any case, this value never becomes sufficient enough to hold true for the if statement i.e., value == 0. This is because the only approach is that floating-point numbers can accommodate the vast range by saving an estimation of that respective value.

This problem can be solved by checking if a variable is near to the ideal value, as appeared in the code in the current solution

```
else if(fabs(value) < .0001) // function to take the absolute value of a float

Serial.println("The value is close enough to zero");
```

This code checks if the variable's value is within 0.0001 of the ideal value and if the case is true, it prints a message. The function known as ‘fabs,’ an abbreviation for floating-point absolute value, gives the absolute value of a floating-point variable. The magnitude of the value is returned through this function, if the value is within 0.0001 of 0, then the code will print a message that ‘the value is close enough to zero.’

Working with Groups of Values

Arrays are called the group of values. Basically, you are required to understand their creation and utilization. These arrays can have more than one dimension. Learn to access the array’s elements and find its size..

The following piece of codes shows the creation of 2 arrays. The first array in the following code consists of integers that denote the pins joined to switches. Similarly, the second array consists of integers that denote the pins joined to LEDs.

```
/*

array sketch

an array of switches controls an array of LEDs

*/

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

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

void setup()

{

for(int index = 0; index < 4; index++)

{
```

```
pinMode(ledPins[index], OUTPUT); // declare LED as output
pinMode(inputPins[index], INPUT); // declare pushbutton as input
digitalWrite(inputPins[index],HIGH); // enable pull-up resistors
//(see Recipe 5.2)
}
}
void loop(){
for(int index = 0; index < 4; index++)
{
int val = digitalRead(inputPins[i]); // read input value
if (val == LOW) // check if the switch is pressed
{
digitalWrite(ledPins[index], HIGH); // turn LED on if switch is pressed
}
else
{
digitalWrite(ledPins[i], LOW); // turn LED off
}
}
}
```

The assortments of successive variables having the similar data types are called arrays. An independent variable in that assortment is called an

element and the quantity of these independent variables determines the array’s dimension.

In the example mentioned above we see how to store an assortment of pins. This is a basic utilization of array in the Arduino code. It shows that the pins are joined to LEDs and switches. The most significant things we learn here are that how an array is created and how its elements are accessed.

Below given is a code that creates an array of data type integer. It has four elements and the values are initialized. The value of the first element is 2, the next element has value 3, etc:

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

You can create an array as shown below if you do not want the arrays' values to be initialized. Maybe these values will be accessible only when the code is being executed.

```
int array[4];
```

This declares an array having 4 elements and the value of each element is initially set to 0 (zero). The number written inside the square [] bracket shows the array's dimension and sets that number of elements of an array. The above array cannot have more than four integer values as its dimension is four. If the array declaration has initializers, then the array's dimension can be excluded as shown in the primary example. This is because the compiler sorts out how huge to make the array by getting the total number of initializers.

element [0] is the first element of the array:

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

If we look at the previous demonstration, we can find that the array's concluding component is 3 although we are working with a four-dimensional array. This is because it is a common trend that whatever the array's dimension is, the value of its last component will always be ‘n-1’ where ‘n’ is the total dimensions of the array.

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

It might appear to be odd that the last element of an array having dimension four is accessed through array [3], but since the first element of the array is

array [0], the four elements will be:

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

In the preceding sketch, for is used to access the four elements of the array:

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

{

//get the pin number by accessing each element in the pin arrays

pinMode(ledPins[index], OUTPUT); // declare LED as output

pinMode(inputPins[index], INPUT); // declare pushbutton as input

}
```

With values beginning at ‘0’ and finishing at ‘3,’ the loop will journey over the index variable. This is arguably one of the most common mistakes where sometimes an element that does not lie in the array’s dimension is sent an access request by the user. This is an error that can have various side effects. An approach to avoid such errors is to use a constant to set the array’s dimension. Consider the code written below:

```
const int PIN_COUNT = 4; // define a constant for the number of elements

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

for(int index = 0; index < PIN_COUNT; index++)

pinMode(inputPins[index], INPUT);
```

Arrays are also utilized to store text characters that are basically known as strings. These are known as strings or character strings in the Arduino code. At least one character is required to make character string. This is then followed by the null character i.e., 0, to demonstrate that the string ends here.

Using Strings in Arduino

You need to manage text. You have to copy the text, add bits together, and find the number of characters.

Character arrays are used to store text. They are normally known as strings. Arduino has an additional ability for utilizing a character array known as String that can store and manage text strings.

This topic explains how to use Arduino strings.

First load the sketch was written below and then to view the results open your Serial Monitor:

```
/*

Basic_Strings sketch

*/

String text1 = "This string";

String text2 = " has more text";

String text3; // to be assigned within the sketch

void setup()

{

Serial.begin(9600);

Serial.print( text1);

Serial.print(" is ");

Serial.print(text1.length());

Serial.println(" characters long.");

Serial.print("text2 is ");

Serial.print(text2.length());

Serial.println(" characters long.");
```

```
text1.concat(text2);

Serial.println("text1 now contains: ");

Serial.println(text1); }

void loop()

{

}
```

The above sketch makes two variables of type String. One is called message and the other is called ‘anotherMessage.’ Type String variables have built-in abilities that allow them to manipulate text. Message.length ( ) provides (returns) the value of the number of characters (length) in the string message.

To combine the contents of a string message.concat (anotherMessage) will be used; in this situation, it will combine the contents of anotherMessage to the end of the message. Here, concat is an abbreviation for concatenation.

The following will be shown on the Serial Monitor:

```
This string is 11 characters long.

text2 is 14 characters long.

text1 now contains:

This string has more text
```

There is another way to combine the strings i.e., by using the string addition operator. Write the following two lines of code at the end of setup code:

```
text3 = text1 + " and more";

Serial.println(text3);
```

The new code will bring about the Serial Monitor adding the line shown below to the end of the display:

```
This is a string with more text and more
```

To find the instance of a specific character in a string, the two functions indexOf and lastIndexOf can be used.

On the off chance that you see a line as shown below:

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

C-style character arrays are used in the code. And if you see a line of code like the following:

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

then Arduino Strings are used in the code. If you want a C-style character array to be converted into Arduino String, then assign its contents to the String object:

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

String newsString = oldString;
```

The Arduino distribution provides string example sketches.

Using Strings of C Programming Language

You need to learn that how you can use raw character strings: you need to understand how to declare or create a string, how to find its length, and how to copy, compare, or append strings. The C language does not support the Arduino style String ability, that’s why you need to get code written to work with primitive character arrays.

Character arrays are at times known as character strings or basically strings for short. This topic explains the functions that work on character strings.

String declaration is done as follows:

```
char StringA[8]; // declare a string of up to 7 chars plus terminating null

char StringB[8] = "Arduino"; // as above and init(ialize) the string to "Arduino";

char StringC[16] = "Arduino"; // as above, but string has room to grow

char StringD[ ] = "Arduino"; // the compiler inits the string and calculates size
```

To find the number of characters in an array before the null, use strlen i.e., abbreviation for sting length.

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

StringA will have length ‘0’ and other strings shown in the above code will have length 7. Strlen does not count the null that shows the end of the string.

Now, to copy one string to another string strcpy i.e., abbreviation for string copy is used.

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

To restrict the number of characters to copy use strncpy. This is useful as it prevents copying more characters than the destination strings range.

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

To append one string to the end of another string use strcat i.e., abbreviation for string concatenation:

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

To compare two strings use strcmp i.e., abbreviation for string compare.

```
if(strcmp(str, "Arduino") == 0)

// do something if the variable str is equal to "Arduino"
```

An array of characters is used to display the text in the Arduino environment. A string comprises of various characters followed by a null (it has a value equal to ‘0’). The null isn’t shown, however it is expected to show the end of a string.

Splitting Comma- Separated Text into Groups

You are given a string with at least two bits of data separated by commas or any other separator. You have to divide (split) the string so you can utilize each part of the string individually.

The following sketch displays the text present between ever comma:

```
/*
* SplitSplit sketch
* split a comma-separated string
*/
String message= "Peter,Paul,Mary"; // an example string
int commaPosition; // the position of the next comma in the string
void setup()
{
Serial.begin(9600);
}
void loop()
{
Serial.println(message); // show the source string
do
{
commaPosition = message.indexOf(,'');
if(commaPosition != -1)
{
Serial.println( message.substring(0,commaPosition));
message = message.substring(commaPosition+1, message.length());
}
else
```

```
{ // here after the last comma is found
if(message.length() > 0)
Serial.println(message); // if there is text after the last comma, print it
}
}
while(commaPosition >=0);
delay(5000);
}
```

The Serial Monitor will display the following text:

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

This above sketch utilizes String functions to get the text from between the commas. The line of code written below:

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

finds the position (index) of the first comma used in the String named message and sets the value of the variable commaPosition equivalent to it. If there is no comma found in the string, the value of commaPosition will be set to -1. If a comma is found in the string then to display the text from the start of the string to the comma's position, the substring function is used. The text that was displayed is taken out from the string message with the following code:

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

substring results in a string having text beginning from commaPosition + 1 i.e., the next position after the comma up to the end of the string message. This outcomes in that message having only the text after the first comma.

This is done repeatedly until no more commas are left in the string i.e., commaIndex will become – 1.

The low-level functions that belong to the standard C library can also be used if you are an expert programmer. The sketch written below has the same functionality as the previous one using Arduino strings:

```
/*

* SplitSplit sketch

* split a comma-separated string

*/

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

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

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

void setup()

{

Serial.begin(9600);

}

void loop()

{

char *str;

char *p;

strncpy(stringBuffer, stringList, MAX_STRING_LEN); // copy source string

Serial.println(stringBuffer); // show the source string
```

```
for( str = strtok_r(stringBuffer, ",", &p); // split using comma

str; // loop while str is not null

str = strtok_r(NULL, ",", &p) // get subsequent tokens

)

{

Serial.println(str);

if(strcmp(str, "Paul") == 0)

Serial.println("found Paul");

}

delay(5000);

}
```

The fundamental functionality originates from the function known as strtok_r i.e., the version's name of strtok that accompanies the Arduino compiler. You pass the string you need to tokenize i.e., separate into individual pieces, to the function strtok_r when you call it for the first time. However, whenever the function strtok_r finds a new token it overwrites the characters in this string, so as shown in the example above it's ideal to pass a copy of the string. Every call that follows utilizes a NULL to inform the function that it should go to the following token. In the above example, every token is compared to a target string i.e., ("Paul"), and is printed to the serial port.

Converting a Number to a String

You have to convert a given number to a string, possibly to represent the number on an LCD or some other display.

The variable String will change numbers to character strings spontaneously. You can utilize the contents of the variable or literal values. For instance, the code written below will work:

```
String myNumber = 1234;

So will this code:

int value = 127

String myReadout = "The reading was ";

myReadout.concat(value);

Or the following code:

int value = 127;

String myReadout = "The reading was ";

myReadout += value;
```

If and only if you are changing a number to show on a serial device or an LCD as a text, the easiest approach is to utilize Serial libraries and LCD's built-in conversion ability. However, maybe you are utilizing a device that does not have built-in assistance or want to change the number to a string.

When the numerical values are assigned to a String variable they are automatically converted by the class called Arduino String class. You can use the addition (+) operator or the concat function if you want to join the numeric values at the end of some string.

The piece of code written below gives a number having a value equal to 13:

```
int number = 12;

number += 1;

When we use the data type String, we get the following result:

String textNumber = 12

textNumber += 1;
```

In the above code textNumber is a string having text “121”.

When the String class was not introduced, the function itoa or ltoa was usually used to determine the Arduino code. The name ‘itoa’ originates from “integer to ASCII” and the name ‘ltoa’ originates from “long to ASCII”. The String version explained before is simpler to use, however, if you need to know the code that uses ltoa or itoa, the following will help you understand it:

The following functions require three parameters. First, the value to be converted. Second, the buffer that will store the resultant string. And third, the number base i.e., 2 for binary, 10 for decimal, and 16 for hex.

The conversion of numeric values using ltoa is explained in the code written below:

```
/*
* NumberToString
* Creates a string from a given number
*/
void setup()
{
Serial.begin(9600);
}
char buffer[12]; // long data type has 11 characters (including the
        // minus sign) and a terminating null
void loop()
{
long value = 12345;
ltoa(value, buffer, 10);
Serial.print( value);
```

```
Serial.print(" has ");
Serial.print(strlen(buffer));
Serial.println(" digits");
value = 123456789;
ltoa(value, buffer, 10);
Serial.print( value);
Serial.print(" has ");
Serial.print(strlen(buffer));
Serial.println(" digits");
delay(1000);
}
```

It is very important for the buffer being used in the sketch program to be big enough such that it's capable of handling every character which is being used in the string. Seven character buffer i.e., five digits, a '-' sign, and an ending zero, for a 16 – bit integer. Twelve character buffer i.e., ten digits, a '-' sign, and an ending zero, for a 32 – bit long integer. If you exceed the limit of buffer no warning is given; and this is such a bug that can cause a wide range of abnormal indications, as the overflow will manipulate some other pieces of memory that might be utilized by your program. So the least demanding approach to deal with this is to use a twelve character buffer and always prefer to in light of the fact that it can handle both 16 – bit and 32 – bit values.

Converting a String to a Number

You have to change a string to a number. Maybe you have got a value in the form of a string over a communication connection and you have to utilize this as a floating-point or integer value.

There are various approaches to understand and solve this. As each character is received it can be converted on the fly if the string is received as serial data. Refer to the preceding topics for an example of the most effective method to do this utilizing the serial port.

One more way to convert the text strings that represent numbers is to utilize C language's conversion functions. For int variable use the function atoi and for long variable use the function atol.

The following piece of code ends the approaching digits on any character that isn't a digit or if there is no space left in the buffer:

```
int blinkRate; // blink rate stored in this variable

char strValue[6]; // must be big enough to hold all the digits and the
        // 0 that terminates the string

int index = 0; // the index into the array storing the received digits

void loop()

{

if( Serial.available())

{

char ch = Serial.read();

if(index < 5 && ch >= '0' && ch <= '9'){

strValue[index++] = ch; // add the ASCII character to the string;

}

else

{

// here when buffer full or on the first non-digit

strValue[index] = 0; // terminate the string with a 0
```

```
blinkRate = atoi(strValue); // use atoi to convert the string to an int

index = 0;

}

}

blink();

}
```

The indistinctly named functions atoi i.e., for ASCII to int and atoll i.e., for ASCII to long are used to change a string into integer or long. If you want to use them then before calling the conversion function, you have to get and then store the whole string in a character array. The above code will make a character array i.e., strValue that can store five digits. As there must be space to store the terminating NULL, so it is declared as char strValue [6]. Digits are stored in the array from Serial.read till it receives a character that is an invalid digit. The array is ended with a NULL and the character array is converted into the variable blinkRate by calling the atoi function.

The value stored in blinkRate is used by a function named blink. The previous sections show the function of ‘blink.’

This example creates a character array, as opposed to utilizing the class String. At the time of this composition, the Arduino String library didn’t serve the purpose of converting strings into numbers.

Transforming the Lines of Code into Blocks

You should understand how you will be able to write more functions in your code. And also understand the right measure of functionality that you need to go into your functions. Additionally, you should be aware of designing the general structure of a code.

The activities performed by your sketch are sort out by functions into the functional blocks. Functions pack features into clearly described inputs i.e., data provided to a function and outputs i.e., data given as a result by a function that keeps it simpler to maintain, structure, and reuse the code. setup and loop are two functions in every Arduino sketch that are already

well known by you. To create a function you declare its return type i.e., the data it gives as a result, the name of the function, and any parameters i.e., values that will be passed in the function when it is called. However, parameters are optional. Below given is a basic function that causes the LED to blink. The following code has no parameters and it returns nothing. The keyword void, which was written before the function, shows that it will not return anything.

```
// blink an LED once

void blink1()

{

digitalWrite(13,HIGH); // turn the LED on

delay(500); // wait 500 milliseconds

digitalWrite(13,LOW); // turn the LED off

delay(500); // wait 500 milliseconds

}
```

The piece of code written below has a parameter i.e., int count that shows how frequently the LED will blink:

```
// blink an LED the number of times given in the count parameter

void blink2(int count)

{

while(count > 0 ) // repeat until count is no longer greater than zero

{

digitalWrite(13,HIGH);

delay(500);

digitalWrite(13,LOW);
```

```
delay(500);

count = count -1; // decrement count

}

}
```

This block of code's main function is to double-check the counter's value to see if it's '0' or some other value and take appropriate actions. For instance, in the scenario where the value is something other than '0,' the program blinks the LED on the Arduino board and, consequently, reduces the value on the counter by '1.' This process is repeated until the counter's value is brought to '0.'

The following is a code that accepts a parameter. It also gives some value as a result. How many times the LED blinks i.e., it is turned on and off is determined by this parameter. The LED keeps on blinking till we press a button. This value is then returned:

```
/*

As soon as the program starts executing, the LED begins blinking.

When a 'switch' is paired with 'pin 2,' the blinking will immediately stop after this switch is triggered.

The final result is that the program prints out the total number of blinks the LED went through until the switch was pressed.

*/

const int ledPin = 13;     // This is the LED's output pin

const int inputPin = 2;   // This is the Input pin which is connected to the switch

void setup()

{

pinMode(ledPin, OUTPUT);
```

```
pinMode(inputPin, INPUT);
digitalWrite(inputPin,HIGH); // use internal pull-up resistor (Recipe 5.2)
Serial.begin(9600);
}
void loop()
{
Serial.println("Press and hold the switch to stop blinking");
int count = blink3(250); // blink the LED 250ms on and 250ms off
Serial.print("The number of times the switch blinked was ");
Serial.println(count);
}
// blink an LED using the given delay period
// return the number of times the LED flashed
int blink3(int period)
{
int result = 0;
int switchVal = HIGH; //with pull-ups, this will be high when switch is up
while(switchVal == HIGH) // repeat this loop until switch is pressed
        // (it will go low when pressed)
{
digitalWrite(13,HIGH);
```

```
delay(period);

digitalWrite(13,LOW);

delay(period);

result = result + 1; // increment the count

switchVal = digitalRead(inputPin); // read input value

}

// here when switchVal is no longer HIGH because the switch is pressed

return result; // this value will be returned

}
```

Discussion:

The code in this problem's solution shows the three types of a function call that you'll encounter.

The following function blink1 does not have any parameter or return value.

```
void blink1()

{

// implementation code goes here...

}
```

The next function blink2 has one parameter yet does not have any return value:

```
void blink2(int count)

{

// implementation code goes here...

}
```

The third function blink3 has both i.e., a parameter and a return value as shown below:

```
int blink3(int period)

{

// implementation code goes here...

}
```

If void is written before the name of a function it shows that the respective function has no return type. Other than this all the functions have some return types that they mentioned before their name. When you declare a function a semicolon is not required at the end of the parenthesis. However, the semicolon plays an important role when you call a specific function.

Most of the functions that we demonstrate are of similar datatypes. For instance, consider the following example;

```
int sensorPercent(int pin)

{

int percent;

val = analogRead(pin); // read the sensor (ranges from 0 to 1023)

percent = map(val,0,1023,0,100); // percent will range from 0 to 100.

return percent;

}
```

The name of the function is sensorPercent. A pin number is passed as a parameter in this function and then returns a value i.e., percent. The int written before the name of the function indicates that the function will return an integer. While declaring functions keep in mind to select the proper return type according to what action the function has to perform. For the above-mentioned function, int data type is appropriate as it returns a number between 0 and 100.

The pin is a parameter of sensorPercent. The value that is passed to the function is assigned to the pin when the function is called.

The code that is written inside the parenthesis (the body of the function) executes the task you need, it reads the value from an analog input pin and afterward depicts the value to a percentage. In the previous example, the variable named percent temporarily holds the percentage. The value stored in the temporary variable i.e., percent is returned to the calling function using the following statement:

```
return percent;
```

We can also get similar functionality without the utilization of a temporary variable:

```
int sensorPercent(int pin)

{

val = analogRead(pin); // read the sensor (ranges from 0 to 1023)

return map(val,0,1023,0,100); // percent will ranges from 0 to 100.

}
```

The following code shows that how to call a function:

```
// print the percent value of 6 analog pins

for(int sensorPin = 0; sensorPin < 6; sensorPin++)

{

Serial.print("Percent of sensor on pin ");

Serial.print(sensorPin);

Serial.print(" is ");

int val = sensorPercent(sensorPin);

Serial.print(val);
```

```
}
```

Returning More Than One Value from a Function

You have to return at least two values from a given function. One of these forms of the function returns just one value or no value at all. However, at times you are required to alter or return more values.

There are different approaches to solve this problem. The simplest to comprehend is to have the function alter some global variables and to not really return any value from the function:

```
/*

swap sketch

demonstrates changing two values using global variables

*/

int x; // x and y are global variables

int y;

void setup()

{

Serial.begin(9600);

}

void loop()

{

x = random(10); // pick some random numbers

y = random(10);

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

Serial.print(x); Serial.print(","); Serial.println(y);
```

```
swap();
Serial.print("The value of x and y after swapping are: ");
Serial.print(x); Serial.print(","); Serial.println(y);Serial.println();
delay(1000);
}
// swap the two global values
void swap()
{
int temp;
temp = x;
x = y;
y = temp;
}
```

By utilizing the global variables, the swap function changes the two values. Global variables are available throughout the program and can be changed by anything. Moreover, they are very easy to understand. However, they are not used by expert programmers since it's very easy to modify the value of these variables or have a function quit working because you altered the type or name of this variable somewhere else in the sketch.

A more secure and more exquisite approach is to pass a reference to those values that you want to change and let the function modify the values by using the references. As shown below:

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

```
*/
void swap(int &value1, int &value2); // functions with references must be
declared before use
void setup() {
Serial.begin(9600);
}
void loop(){
int x = random(10); // pick some random numbers
int y = random(10);
Serial.print("The value of x and y before swapping are: ");
Serial.print(x); Serial.print(","); Serial.println(y);
swap(x,y);
Serial.print("The value of x and y after swapping are: ");
Serial.print(x); Serial.print(","); Serial.println(y);Serial.println();
delay(1000);
}
// swap the two given values
void swap(int &value1, int &value2)
{
int temp;
temp = value1;
value1 = value2;
```

```
value2 = temp;

}
```

The functions having parameters that are described in the preceding sections and the swap function both are similar. However, the symbol ‘&’ (ampersand) shows that the parameters are references. This means that the value of the variable that is given when the function is called is changed when the values inside the function change. This happens by first executing the code given in the above Solution and then checking that the parameters are swapped. Afterward, the code is modified by eliminating all the four ampersands. Two of them are present at the top of the declaration and the other two at the end.

The two lines that are changed should be as follows:

```
void swap(int value1, int value2); // functions with references must be declared

before use

...

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

By running the code, we see that the values are not swapped. The modifications that are done inside the function are local and when the function returns, these changes are lost.

Problem

You need to execute a piece of code if and only if a specific condition is valid. For instance, if a switch is pressed you might need to turn on a LED or if the value of analog is larger than the threshold.

Solution

The piece of code written below utilizes the wiring shown in the preceding sections:

```
/*

Pushbutton sketch
```

```
a switch connected to pin 2 lights the LED on pin 13
*/
const int ledPin = 13; // choose the pin for the LED
const int inputPin = 2; // choose the input pin (for a pushbutton)
void setup() {
pinMode(ledPin, OUTPUT); // declare LED pin as output
pinMode(inputPin, INPUT); // declare pushbutton pin as input
}
void loop(){
int val = digitalRead(inputPin); // read input value
if (val == HIGH) // check if the input is HIGH
{
digitalWrite(ledPin, HIGH); // turn LED on if switch is pressed
}
}
```

Discussion

To test the value of digitalRead, the if statement is used. The if statement should have a test written between the parenthesis that must be true or false. In this given problem's Solution, its val == High, and if the expression is true only then the if statement is executed. A block of code comprises of all the code written inside the parenthesis and if the parenthesis is not used, then the block comprises of the following executable statement ended by a semicolon.

Use the if… else statement, if you have to do a certain thing if the statement is true and another if the statement is false:

```
/*
Pushbutton sketch
a switch connected to pin 2 lights the LED on pin 13
*/
const int ledPin = 13; // choose the pin for the LED
const int inputPin = 2; // choose the input pin (for a pushbutton)
void setup() {
pinMode(ledPin, OUTPUT); // declare LED pin as output
pinMode(inputPin, INPUT); // declare pushbutton pin as input
}
void loop(){
int val = digitalRead(inputPin); // read input value
if (val == HIGH) // check if the input is HIGH
{
// do this if val is HIGH
digitalWrite(ledPin, HIGH); // turn LED on if switch is pressed
}
else
{
// else do this if val is not HIGH
digitalWrite(ledPin, LOW); // turn LED off
}
```

```
}
```

Problem

You want a piece of code to run until a given condition is true.

Solution

Until a condition is true, a while loops execute the code repeatedly.

```
while(analogRead(sensorPin) > 100)

{

flashLED(); // call a function to turn an LED on and off

}
```

The above code is written within the parenthesis of while the loop will execute until the given condition is true i.e., value of analogRead is greater than 100. When a value surpasses a threshold, it could blink a LED as an obvious warning. When the analogRead is greater than 100, the led blinks continuously. Otherwise, the LED is off if the value is less or equal to 100.

Parentheses fix the limit of a code block that is to be executed within a loop. Only the first line of the code in executed repeatedly if the parenthesis are not used.

```
while(analogRead(sensorPin) > 100)

flashLED(); // line immediately following the loop expression is executed
Serial.print(analogRead(sensorPin)); // this is not executed until after
        // the while loop finishes!!!
```

The ‘do… while’ loop is just like the while loop, however, the statements of the code are run once before checking the condition. You can use do… while loop when you want at least a single execution of the code, regardless of whether the condition is false:

```
do

{
```

```
flashLED(); // call a function to turn an LED on and off

}

while (analogRead(sensorPin) > 100);
```

The above code will cause the LED to blink at least a single time even if the condition is false. And the LED will keep blinking until the value from analogRead is greater than 100. The LED will blink only once if the value is not greater than 100. A battery – circuit could be made using this code. The LED's persistent blinking will show that the battery is fully charged and if it blinks every other second, this will show that the circuit is active.

You need to execute at least one statement a specific number of times. The ‘while’ loop and ‘for’ loop are almost the same. However, in the ‘for’ loop, you have more command over the beginning and terminating conditions.

In the code lines demonstrated below, the program tells the user the ‘i’ variable’s value as it went through the ‘for’ loop.

```
/*

ForLoop sketch

demonstrates for loop

*/

void setup() {

Serial.begin(9600);}

void loop(){

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

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

{

Serial.println(i);

}
```

```
}
```

We get the following result from the above sketch:

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

0

1

2

3
```

Discussion

Generally, the **‘for’** loop consists of three portions. First is the initialization, second is the conditional test and third is iteration i.e., an expression that is executed toward the finish of each pass through the loop. A semicolon distinguishes all these three parts. In the above solution's code, variable i is initialized to ‘0’ i.e., int i = 0. The condition i.e., i < 4 checks if the variable i is less than 4. And the iteration i.e., i ++ increments the value of the variable i by 1 after a single iteration.

A for loop can either create a new variable that is restricted within the for loop or it can use an existing variable. The following sketch shows the use of an already created variable:

```
int j;

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

for(j=0; j < 4; j++ )

{

Serial.println(j);

}
```

This code is nearly similar to the previous code. However, the only difference is in the initialization part i.e., the variable j does not have int

keyword written before it, because it is already defined outside the loop. Bothe these codes get the same outputs as shown below:

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

0

1

2

3
```

If you want to eliminate the initialization part totally you can just utilize a variable defined before. The following code begins the loop where j has a value equal to 1.

```
int j = 1;

Serial.println("for( ; j < 4; j++ )");

for( ; j < 4; j++ )

{

Serial.println(j);

}

The above code displays the following output:

for( ; j < 4; j++)

1

2

3
```

You control the loop termination in the conditional test. In the preceding example, when the variable's value exceeds 4 the loop ends because the condition is no longer true.

The code written below checks the value of the variable whether it is equal to 4 or less than 4. It displays the numbers from 0 to 4:

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

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

{

Serial.println(i);

}
```

The iteration statement that is the third part of the for loop gets executed toward the finish of each pass through the loop. This can be any legitimate statement of C or C++. The below given code increments the value of variable i by 2 after each iteration:

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

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

{

Serial.println(i);

}
```

This code prints only two values i.e., 0 and 2.

The iteration statement can be utilized to get the values from high to low, for example in the following code, from 3 to 0:

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

for(int i=3; i > = 0 ; i--)

{

Serial.println(i);

}
```

Just like the other parts, the iteration statement can also be left empty. But in mind to put the semicolons in between to separate the three parts of a ‘for’ loop.

The following piece of code shows that the for loop does not increment or decrement the value of variable i until an input pin is high and this is dine in the if statement after Serial.print:

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

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

{

Serial.println(i);

if(digitalRead(inPin) == HIGH);

i++; // only increment the value if the input is high

}
```

Problem

Based on some conditions that you are testing you need to end a loop early.

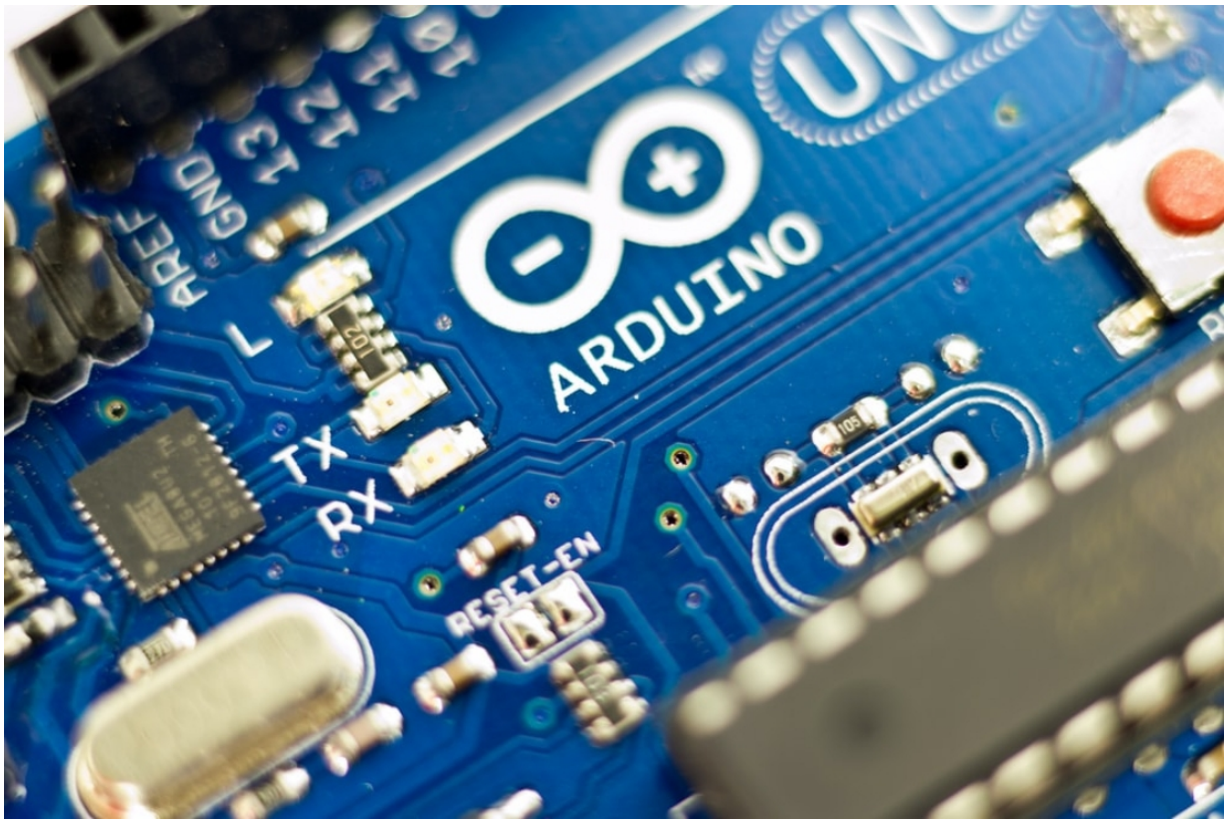
You will use the code written below:

```
while(analogRead(sensorPin) > 100)

{

if(digitalRead(switchPin) == HIGH)

{

break; //exit the loop if the switch is pressed

}

flashLED(); // call a function to turn an LED on and off

}
```

Discussion

This code is like the one that uses while loops, however, this code uses a break statement if the loop needs to be terminated if the digital pin is high. For instance, the LED will stop blinking and the loop will terminate regardless of whether the condition in the while loop is true, if the switch is connected to the pins.

Chapter 5: Coding and Memory Handling



Whenever we talk about programming, an essential part is to understand the tools available for us to use within the language we are working with. Every language features their own set of tools which make the preferable for a certain niche of programming. For instance, for statistical and data analysis, people prefer the ‘R programming language.’ For machine learning and working with neural networks, people prefer ‘Python,’ and in some cases, both Python and C++ can be used to create the same instruction set. However, it all boils down to which language the programmer is comfortable using. The reason why programmers have a preference when it comes to programming languages is because of the tools they are packing. These tools are essentially the libraries that feature classes, objects, and functions that enable the programmer to create a useful application. In short, libraries extend the usefulness of a programming language as a whole and Arduino is no exception (since it uses C++ as the base programming language).

In addition, coding requires that the user also accounts for any complications and other aspects which arise when software interacts with hardware. One of the most important aspects to be wary of when writing any program is its memory and error handling capabilities. In Arduino, error handling isn't emphasized as much as memory handling because the programs have fewer chances of encountering an error as compared to applications designed for PCs. A good memory handling program can execute more efficiently, consuming less RAM all whilst running smoothly and fast. This becomes even more important when we are working on a resource-intensive project and is required to deliver a level of performance that is well over the capabilities of the Arduino's chipset itself. On the Arduino board, there a total of three memory types which have been listed below;

- Random Access Memory
- Program Memory
- EEPROM

This chapter will discuss how to use libraries, create our own, and even modify existing libraries. In the latter half, we will cover different memory handling techniques to make sketch programs more efficient without relying heavily on RAM.

Using the Libraries

The Arduino distribution comes with a collection of libraries for users to implement in their sketch programs from the get-go. Using a library that is by default included in the IDE is very simple. All we have to do is go to the '**Sketch** ' menu and in the '**Import Library** ' section, we will find all of the libraries currently available in the Arduino IDE. Apart from the ones that come with the distribution, we can install additional libraries as well. These libraries will show up on the same menu and will be separated from the default ones by a line.

To use a library, we simply go to the '**Import Library** ' option and choose the one we want to use for the current sketch program. If we know the name of the library, we can also directly add it to our sketch program by using the **'include'** header as shown below;

```
#include <nameOfTheLibrarySelected.h>
```

In this way, all of the functions, methods, classes, objects, and commands specified within this library become available for use in the current coding session. Here's a list of the libraries that you'll find come with the Arduino distribution.

- ***EEPROM*** ; this library contains functions and classes that allow the user to save data within the EEPROM memory of the board and read it. The data stored within this memory is not lost even when the power is turned off.
- ***Ethernet*** ; this library features elements that allow the user to code a sketch program capable of interacting with the Arduino board's Ethernet shield.
- ***Firmata*** ; this library contains protocols to facilitate serial communication. It also includes protocols that help the program to control the functioning of the board as well.
- ***LiquidCrystal*** ; this library allows the sketch program to be able to control the LCD display connected to the Arduino board.
- ***Matrix*** ; this library is primarily used to manipulate LED displays being used with the Arduino board.
- ***SD*** ; this library features functions that allow the program to perform read and write operations on an SD card.
- ***Servo*** ; this library is used when the Arduino board is connected to a 'Servo motor' so that it can control its functioning.
- ***SoftwareSerial*** ; this library is used to enable additional serial ports on the Arduino board.
- ***SPI*** ; this library is primarily used for controlling the Ethernet as well as any SPI hardware connected to the board.
- ***Sprite*** ; this library allows the program to use 'sprites' on a matrix LED connected to the Arduino board.

Many more libraries are available with the Arduino distribution which unlock its full potential when used, making the hardware more versatile and compatible with many projects.

Installing Additional Libraries

As we discussed before, we are not limited to only using the libraries that come with the Arduino distribution, we can install additional libraries as well. To install a library in the IDE software, we first need to download it.

Suppose the downloaded file is in the form of a .'zip' file. In that case, we will first need to unzip and then relocate the folder containing the library files to the '**libraries** ' directory found within the '**Arduino Documents** ' folder. If you don't know where this root folder is, you can simply go to the Arduino IDE and under the '**Sketch** ' menu, go to the '**Show Sketch Folder** ' menu. This will bring us to Arduino's sketch folder and from here, we simply go to the root directory and this will lead us to the '**Arduino Documents** ' folder. There's a good chance that this will be the first time you are adding a library to the IDE, if that's the case, then a '**libraries** ' folder might not even exist. In this scenario, just create a folder named '**libraries** ' by yourself and then put all of the libraries you want to add to this folder.

Once you add a library, it will show up when the IDE is booted. This is because the Arduino IDE scans its directory and checks if anything was added when it is being booted. However, if the IDE is running and the libraries are added, then they will not show up unless the IDE is closed and launched again.

Modifying Libraries

Users have the option to make changes to libraries as well if they want to add certain functionalities needed for their projects. For instance, if we download a library named '**TimeAlarms** ,' we can use its methods and classes to program an alarm on the Arduino board. However, this library has a limitation: it only allows a total of 6 alarms to be set at any time. Let's say that we need more than six alarms for our project. In such a scenario, it is much better and easier to modify the library itself rather than looking for one that supports more than 6 alarms.

The following sketch program is using the '**TimeAlarms** ' library. In this demonstration, we are specifying 7 alarms instead of 6 to show the error it will trigger.

```
/*

multiple_alarms sketch

has more timer repeats than the library supports out of the box -
```

```
you will need to edit the header file to enable more than 6 alarms
*/
#include <Time.h>
#include <TimeAlarms.h>
int currentSeconds = 0;
void setup()
{
Serial.begin(9600);
// create 7 alarm tasks
 Alarm.timerRepeat(1, repeatTask1);
Alarm.timerRepeat(2, repeatTask2);
Alarm.timerRepeat(3, repeatTask3);
Alarm.timerRepeat(4, repeatTask4);
Alarm.timerRepeat(5, repeatTask5);
Alarm.timerRepeat(6, repeatTask6);
Alarm.timerRepeat(7, repeatTask7); //7th timer repeat
}
void repeatTask1()
{
Serial.print("task 1 ");
}
void repeatTask2()
```

```
{
Serial.print("task 2 ");
}
void repeatTask3()
{
Serial.print("task 3 ");
}
void repeatTask4()
{
Serial.print("task 4 ");
}
void repeatTask5()
{
 Serial.print("task 5 ");
}
void repeatTask6()
{
Serial.print("task 6 ");
}
void repeatTask7()
{
```

```
Serial.print("task 7 ");

}

void loop()

{

if(second() != currentSeconds)

{

// print the time for each new second

// the task numbers will be printed when the alarm for that task is triggered

Serial.println();

Serial.print(second());

Serial.print("->");

currentSeconds = second();

Alarm.delay(1); //Alarm.delay must be called to service the alarms

}

}
```

Once this sketch program is uploaded to the Arduino board, it will start executing. But we won't do that just yet because we know it will not work. Upon compiling and running the sketch program, the Serial Monitor will display each task's output for a total of 9 seconds. Each second, different tasks will be performed. The result is as shown below;

```
1->task 1

2->task 1    task 2

3->task 1    task 3
```

```
4->task 1    task 2 task 4

5->task 1    task 5

6->task 1    task 2 task 3 task 6

7->task 1

8->task 1    task 2 task 4

9->task 1    task 3
```

If we carefully analyze this result, then we can see that the task scheduled at the 7th second did not execute. This is because we used a 7th timer object even though the library only has 6 timer objects. To make this sketch work, we will have to open the library to modify it. To do this, we simply open the library we want to modify using a text editor. There's no specific text editor application which is mandatory to install, even the default text editors that come with the Operating System are more than enough. Find the library you want to modify in the '**libraries** ' folder and then open it with the text editor. If you are running Windows, you can use 'Notepad' and if you are using Mac, you can use the 'TextEdit' application. Once in the libraries folder, we need to find the '**TimeAlarms.h** ' file and open it using the desired text editor.

Once we open the header file using the text editor, the very first lines will be like this;

```
#ifndef TimeAlarms_h

#define TimeAlarms_h


#include <inttypes.h>

#include "Time.h"

#define dtNBR_ALARMS 6
```

If we look at the very last line, we will see the number of timer objects defined. To add support for 7 alarms instead of 6, we simply need to change

the value assigned to **‘dtNBR_ALARMS’** accordingly. Since we only want 7 alarms, we will assign it a value of ‘7.’

```
#define dtNMBR_ALARMS 7
```

Once the modifications are done, we need to save the file to make the changes permanent and close it. Now, we run the lines of code in the Arduino IDE again and this time, instead of the 7th -second task being missed, it will be properly executed this time.

```
1->task 1

2->task 1 task 2

3->task 1 task 3

4->task 1 task 2 task 4

5->task 1 task 5

6->task 1 task 2 task 3 task 6

7->task 1 task 7

8->task 1 task 2 task 4

9->task 1 task 3
```

The result displayed by the Serial Output Monitor this time clearly shows us that the task scheduled at the 7th second succeeded this time whereas it failed when we used the original library.

However, modifications made to the library are not for free, on the contrary, they come at a cost that can be sometimes quite hefty to pay for the board’s resources. For instance, the type of change we made to this library will ultimately consume more system resources and this consumption will affect the resources available for the rest of the program. But not everything is doom and gloom. In fact, we can use this to our advantage as well. By carefully structuring the sketch program, we can identify the portion that needs more system memory (RAM) and the portion that does not need it. Since the requirement is imbalanced, we can modify certain elements of the sketch program accordingly. For instance, we can selectively decrease the

memory allocated to the ‘serial library’ being used in the program, this will increase the memory resource available to the rest of the program. Similarly, we can increase the memory resource available to a library being used in the program if the other code lines don’t require as much RAM. Hence, when creating a sketch program, one should always be wary of system requirements to take appropriate measures. Otherwise, the code execution will have problems.

Creating a Library

Generally, the reason why users would want even to create libraries is not to invent new functions and capabilities within the programming language, instead, this method is a pretty convenient way of sharing a block of code or reusing it in other programs.

This section will create a library from a sketch program example, which will be done without using any classes.

The sketch we will be using for this demonstration is given below;

```
/*

* blinkLibTest

*/

const int firstLedPin = 3; // choose the pin for each of the LEDs

const int secondLedPin = 5;

const int thirdLedPin = 6;
void setup()

{

pinMode(firstLedPin, OUTPUT); // declare LED pins as output

pinMode(secondLedPin, OUTPUT); // declare LED pins as output

pinMode(thirdLedPin, OUTPUT); // declare LED pins as output

}
```

```
void loop()
{
// flash each of the LEDs for 1000 milliseconds (1 second)
blinkLED(firstLedPin, 1000);
blinkLED(secondLedPin, 1000);
blinkLED(thirdLedPin, 1000);
}
```

We will now take out the '**blinkLED()** ' function from the sketch program and copy it over to a different file and name it '**blinkLED.cpp** .'

```
/* blinkLED.cpp
* simple library to light an LED for a duration given in milliseconds
*/
#include <WProgram.h> // Arduino includes
#include "blinkLED.h"
// blink the LED on the given pin for the duration in milliseconds
void blinkLED(int pin, int duration)
{
digitalWrite(pin, HIGH); // turn LED on
delay(duration);
digitalWrite(pin, LOW); // turn LED off
delay(duration);
}
```

All that's left to do is create a header file with the same name as the .'cpp' file, in this case, the name of the header file would be '**blinkLED.h** .' The contents of this header file are shown below;

```
/*

* blinkLED.h

* Library header file for BlinkLED library

*/

void blinkLED(int pin, int duration); // function prototype
```

Once done, we put the 'blinkLED.ccp' file and the 'blinkLED.h' into a folder of the same name and place it in the '**libraries** ' folder. Now we are ready to use the functions defined within this code for any sketch programs.

We can even modify a library that we create to extend its functionalities further as well. In this case, as soon as we use the library in a blank sketch program, all three of the LEDs on the Arduino board will start flickering. But by adding in a few things, we can change how the library works as well. For instance, we can add a quality of life improvement in this library by including constants that define each blink's delays. This will allow users to refer to the constant value when changing the delay between each blink of the LED light instead of having to work with values in milliseconds. The first option is more convenient and easy to deal with. To make this modification, we first need to open the header file and inside the header file, we need to add the following lines;

```
/*

* blinkLED.h

* Library header file for BlinkLED library

*/

void blinkLED(int pin, int duration); // function prototype
```

After that, we open the 'blinkLED.ccp' file and change the lines of code present in the '**void loop()** ' section as shown below;

```
void loop()

{

blinkLED(firstLedPin, BLINK_SHORT);

blinkLED(secondLedPin, BLINK_MEDIUM);

blinkLED(thirdLedPin, BLINK_LONG);

}
```

In this way, we assigned the constant values a variable that effectively describes the effect of the value upon the delay in the LED's blinking lights. When we export the sketch file to the Arduino board, we will see that the light will blink really fast at the start, then the intervals between each blink will be longer than before and in the final blink, the delay between the on and off intervals will be the longest.

We can also add different functions to this library as well which were not previously present. For instance, we can specify the number of times an LED should blink when the code is executed. This has been demonstrated in the following lines of code;

```
void loop()

{

blinkLED(firstLedPin,BLINK_SHORT,5 ); // blink 5 times

blinkLED(secondLedPin,BLINK_MEDIUM,3 ); // blink 3 times

blinkLED(thirdLedPin, BLINK_LONG); // blink once

}
```

If we want the library to include such functionality, we will have to make changes to both the ‘blinkLED.h’ and ‘blinkLED.cpp’ files. First things first, we open the header file using a text editor and include the function prototype in it;

```
/*
```

```
* BlinkLED.h

* Header file for BlinkLED library

*/

const int BLINK_SHORT = 250;

const int BLINK_MEDIUM = 500;

const int BLINK_LONG = 1000;

void blinkLED(int pin, int duration);

// new function for repeat count

void blinkLED(int pin, int duration, int repeats);
```

Next, we open the ‘blinkLED.ccp’ file and include the function corresponding to the prototype added in the header file.

```
/*

* BlinkLED.cpp

*/

#include <WProgram.h> // Arduino includes

#include "BlinkLED.h"

// blink the LED on the given pin for the duration in milliseconds

void blinkLED(int pin, int duration)

{

digitalWrite(pin, HIGH); // turn LED on

delay(duration);

digitalWrite(pin, LOW); // turn LED off
```

```
delay(duration);
}
/* function to repeat blinking
void blinkLED(int pin, int duration, int repeats)
{
while(repeats)
{
blinkLED(pin, duration);
repeats = repeats -1;
}
}
```

Creating Libraries from Other Libraries

While this may sound confusing when read out loud, but it's not that complicated at all. Just as we used a chunk of code to create a library, we can also use existing libraries as a part of another library. In other words, we can take an entire library, or one portion of it and then extend it to create a different yet similar library. This can be done with multiple libraries as well, just as how you would make a cocktail. One possible use of this technique would be to build a library to help us in debugging. We will be using this library to dispatch a print command and the result would be displayed on another Arduino board. This is possible through a library known as '**Wire Library** .'

The sketch program we will be using to demonstrate this task will be using two main components;

- The wire library
- The Arduino print commands

However, for this demonstration to work, we need two Arduino boards and these boards must be connected to each other through an '**I2C** ' connection. We will discuss I2C serial port connections some other time, for now, let's focus on the main concept, which is to use the wire library to create another library (which will have a bit of salvaged code as well). We will be using appropriate code lines for handling the I2C connection and include it in the new library we will be creating.

Let's name the folder where we will be placing the library as '**i2cDebug** ' (reminder, every library folder which has a header and .ccp files need to be placed within the root directory of '**libraries** '). First, let's build a header file for the corresponding library;

```
/*

* i2cDebug.h

*/

#include <WProgram.h>

#include <Print.h> // the Arduino print class

class I2CDebugClass : public Print

{

private:

int I2CAddress;

byte count;

void write(byte c);

public:

I2CDebugClass();

boolean begin(int id);

};
```

```
extern I2CDebugClass i2cDebug; // the i2c debug object
```

Now that we have a header file, we now need a .'**ccp** ' file to place it alongside the header file in the '**libraries** ' directory inside the '**i2cDebug** ' folder. The contents of the .'**ccp** ' file will be as follows;

```
/*

* i2cDebug.cpp

*/

#include <i2cDebug.h>

#include <Wire.h> // the Arduino I2C library

I2CDebugClass::I2CDebugClass()

{

}

 boolean I2CDebugClass::begin(int id)

{

I2CAddress = id; // save the slave's address

Wire.begin(); // join I2C bus (address optional for master)

return true;

}

void I2CDebugClass::write(byte c)

{

if( count == 0)

{

// here if the first char in the transmission
```

```
Wire.beginTransmission(I2CAddress); // transmit to device
}
Wire.send(c);
// if the I2C buffer is full or an end of line is reached, send the data
// BUFFER_LENGTH is defined in the Wire library
if(++count >= BUFFER_LENGTH || c == '\n')
{
// send data if buffer full or newline character
Wire.endTransmission();
count = 0;
}
}
I2CDebugClass i2cDebug; // Create an I2C debug object
```

Once we have created both the .'h' and .'ccp' files, we simply place them in their appropriate folder inside the '**libraries** ' directory. We now have created an entirely new library from the '**Wire library** ' and some lines of code (to handle I2C connections). Let's now load the following sketch program which will be using the '**i2cDebug library** ' to perform its core functions.

```
/*
* i2cDebug
* example sketch for i2cDebug library
*/
#include <Wire.h> // the Arduino I2C library
```

```
#include <i2cDebug.h>
const int address = 4; //the address to be used by the communicating devices

const int sensorPin = 0; // select the analog input pin for the sensor

int val; // variable to store the sensor value

void setup()

{

Serial.begin(9600);

i2cDebug.begin(address);

}

void loop()

{

// read the voltage on the pot(val ranges from 0 to 1023)

 val = analogRead(sensorPin);

Serial.println(val);

 i2cDebug.println(val);

}
```

Memory Handling

This section will discuss how to implement memory handling techniques to ensure that our sketch program will never have to work with insufficient RAM and sacrifice performance. Before we can even determine how much memory we have to handle carefully, we first need to know how much memory is even allocated to the sketch program in the first place. This can be done by using the '**memoryFree()** ' function as shown in the following demonstration;

```
void setup()
{
Serial.begin(9600);
}
void loop()
{
Serial.print(memoryFree()); // print the free memory
Serial.print(' '); // print a space
delay(1000);
}
// variables created by the build process when compiling the sketch
extern int __bss_end;
extern void *__brkval;
// function to return the amount of free RAM
int memoryFree()
{
int freeValue;
if((int)__brkval == 0)
freeValue = ((int)&freeValue) - ((int)&__bss_end);
else
freeValue = ((int)&freeValue) - ((int)__brkval);
return freeValue;
```

```
}
```

This program uses the function ‘memoryFree()’ to validate the total system memory available for use. In turn, the function performs this task by utilizing a special type of variable known as ‘**system variables** .’ System variables are not generally used in programs because the purpose of these variables is to facilitate the IDE’s compiler in managing the internal resources. As the program is undergoing the execution process, the number of bytes it stores in the system memory constantly changes. Thus, it is important to make sure that the program only uses as much memory as is available and not more than that.

The most common elements that drain memory have been listed below;

- During the initialization of constants. For example;

```
#define ERROR_MESSAGE "an error has occurred"
```

- During the declaration of global variables. For example;

```
char myMessage[] = "Hello World";
```

- When a function is called. For example;

```
void myFunction(int value)

{

int result;

result = value * 2;

return result;

}
```

- During memory allocation. For example;

```
String stringOne = "Arduino String";
```

Using lines of code that increasingly incorporate more of the elements listed above will steadily and surely drain the system memory to its last drop. In addition, it is also recommended to be wary of the global variables because these are declared most commonly in libraries. Setting a parameter on which we base the declaration of variables can also backfire because the number of variables might increase to the point where the majority of the system memory is used up during code execution.

Saving and Fetching Numeric Values from Program Memory

It is not possible to save everything in the system memory of the Arduino board. Moreover, we want to have the maximum amount of free RAM available whenever possible. In such cases, we can use the board's flash memory (also known as the program memory) to do this. This section will discuss how we can store numeric values in the system's program memory.

In the sketch, we will demonstrate functions in a way such that it makes adjustments to the fading away of the LED light according to our non-linear sensitivity perception. The sketch program will be using a total of 256 values and these values are in the form of a table. Moreover, this entire bundle of data is stored within the system's program memory instead of the RAM.

When we execute this sketch, the light transitioning from bright to completely fading away is extremely smooth. This sketch works on the LED connected to the pin 5 terminal and this animation can be compared to the LED connected to the pin 3 terminal where the latter has abrupt and hasty fade animations.

The sketch for storing 256 constant numeric values in the system's program has been shown below;

```
/* ProgmemCurve sketch

 * uses table in Progmem to convert linear to exponential output


*/
```

```
#include <avr/pgmspace.h> // needed for PROGMEM

// table of exponential values

// generated for values of i from 0 to 255 -> x=round( pow( 2.0, i/32.0) - 1);

const byte table[]PROGMEM = {

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2,

2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3,

3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5,

5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7,

7, 7, 7, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 10, 10, 10,

10, 11, 11, 11, 11, 12, 12, 12, 12, 13, 13, 13, 14, 14, 14, 15,

15, 15, 16, 16, 16, 17, 17, 18, 18, 18, 19, 19, 20, 20, 21, 21,

22, 22, 23, 23, 24, 24, 25, 25, 26, 26, 27, 28, 28, 29, 30, 30,

31, 32, 32, 33, 34, 35, 35, 36, 37, 38, 39, 40, 40, 41, 42, 43,

44, 45, 46, 47, 48, 49, 51, 52, 53, 54, 55, 56, 58, 59, 60, 62,

63, 64, 66, 67, 69, 70, 72, 73, 75, 77, 78, 80, 82, 84, 86, 88,

90, 91, 94, 96, 98, 100, 102, 104, 107, 109, 111, 114, 116, 119, 122, 124,

127, 130, 133, 136, 139, 142, 145, 148, 151, 155, 158, 161, 165, 169, 172, 176,

180, 184, 188, 192, 196, 201, 205, 210, 214, 219, 224, 229, 234, 239, 244, 250
```

```
};
const int rawLedPin = 3; // this LED is fed with raw values
const int adjustedLedPin = 5; // this LED is driven from table
int brightness = 0;
int increment = 1;
void setup()
{
// pins driven by analogWrite do not need to be declared as outputs
}
void loop()
{
if(brightness > 255)
{
increment = -1; // count down after reaching 255
}
else if(brightness < 1)
{
increment = 1; // count up after dropping back down to 0
}
brightness = brightness + increment; // increment (or decrement sign is minus)
// write the brightness value to the LEDs
```

```
analogWrite(rawLedPin, brightness); // this is the raw value

int adjustedBrightness = pgm_read_byte(&table[brightness]); // adjusted value

analogWrite(adjustedLedPin, adjustedBrightness);

delay(10); // 10ms for each step change means 2.55 secs to fade up or down

}
```

When working with a large expression of values that iterate over a regular interval, it is better to simply calculate these values before-hand and store them in the form of an array. In this way, when the code is executed, it doesn't have to calculate each value repeatedly. This is not only time-efficient, making the program to run faster, but also memory efficient as well because if the values were to be calculated multiple times, then the result would be stored in the system's RAM. Instead of doing that, we calculated the values before-hand and stored the resulting array in the program memory, hence saving Arduino's RAM resource from being completely drained by this program. At the beginning of the sketch program, we first and foremost define the table used to store the values. This table is created through the following line;

```
const byte table[ ]PROGMEM = {

0, . . .
```

The '**PROGMEM** ' expression is used to inform the compiler that the values written henceforth are to be saved in the board's program memory rather than the board's system memory. The values are then defined in the form of an array. To use the 'PROGMEM' argument, the sketch program needs the corresponding definitions as well, otherwise, the compiler won't be able to understand this instruction. For this purpose, we include a header file with the name of '**pgmspace.h** ' in the libraries folder and then import it into the program using the '**#include** ' argument.

The overall brightness of the LED light is controlled using the following lines of code in the sketch program shown above to ensure a smooth fading

transition;

```
int adjustedBrightness = pgm_read_byte(&table[brightness]);

analogWrite(adjustedLedPin, adjustedBrightness);
```

This entire expression can also be written in the following way as well;

```
pgm_read_byte(table + brightness);
```

Let’s see another demonstration as well.

```
/* ir-distance_Progmem sketch

* prints distance & changes LED flash rate depending on distance from IR sensor

* uses progmem for table

*/

#include <avr/pgmspace.h> // needed when using Progmem

// table entries are distances in steps of 250 millivolts

const int TABLE_ENTRIES = 12;

const int firstElement = 250; // first entry is 250 mV

const int interval = 250; // millivolts between each element

// the following is the definition of the table in Program Memory

const int distanceP[TABLE_ENTRIES] PROGMEM = { 150,140,130,100,60,50,

40,35,30,25,20,15 };

// This function reads from Program Memory at the given index

int getTableEntry(int index)

{
```

```
int value = pgm_read_word(&distanceP[index]);

return value;

}
```

If you recall the beginning of this book, you'll find that we discussed a similar experiment by using the LDR sensor instead of an Infrared Sensor. However, the difference is that instead of measuring the light intensity being emitted by the LED, we measure the distance of the light coming from the LED and reaching the Infrared sensor. In the case of the LDR experiment, we assigned different values that correspond to the different intensities of the LED light. Similarly, we will assign values in this program as well but these values will be stored in the board's flash memory instead of RAM because all of these values are constant (meaning, the values themselves won't change). To do this, we use the function '**PROGMEM** .'

The distance is measured and interpreted by using the function '**getDistance()** ' and the '**getTableEntry()** function is used by the program to fetch the corresponding values from the program memory and not from the RAM.

```
int getDistance(int mV)

{

if( mV > interval * TABLE_ENTRIES )

return getTableEntry(TABLE_ENTRIES-1); // the minimum distance

else

{

int index = mV / interval;

float frac = (mV % 250) / (float)interval;

return getTableEntry(index) - ((getTableEntry(index) -

getTableEntry(index+1)) * frac);
```

```
}

}
```

Saving and Fetching Strings Using the Program Memory

Another element in coding that eats up RAM greedily is strings. To conserve memory space, we will move the string constants being used in the sketch program to be stored in the system's program memory.

The sketch shown below generates a string directly in the flash memory (program memory) of the Arduino board. When the code is executed, the strings are fetched from the program memory and the resulting text is displayed in the IDE's serial monitor. In addition, the sketch also has code that tells the free RAM available in the system for use;

```
#include <avr/pgmspace.h> // for progmem

//create a string of 20 characters in progmem

const prog_uchar myText[] = "arduino duemilanove ";

void setup()

{

Serial.begin(9600);

}

void loop()

{

Serial.print(memoryFree()); // print the free memory

Serial.print(' '); // print a space

printP(myText); // print the string

delay(1000);
```

```
}
// function to print a PROGMEM string
void printP(const prog_uchar *str)
{
char c;
while((c = pgm_read_byte(str++)))
Serial.print(c,BYTE);
}
// variables created by the build process when compiling the sketch
extern int __bss_end;
extern void *__brkval;
// function to return the amount of free RAM
int memoryFree()
{
int freeValue;
if((int)__brkval == 0)
freeValue = ((int)&freeValue) - ((int)&__bss_end);
else
freeValue = ((int)&freeValue) - ((int)__brkval);
return freeValue;
}
```

Each character in a string takes up about one byte of memory space and thus, a large number of strings can easily gobble up a system's RAM very easily. We used the same '**PROGMEM** ' expression to move the string constants to the system's program memory just as how we used it to move numeric constants in the previous section. To use this function, we need to include the header file which includes the corresponding definitions.

```
#include <avr/pgmspace.h> // for progmem
```

Strings that are being declared in the system's program memory have a slightly different syntax which is as follows;

```
const prog_uchar myText[] = "arduino duemilanove "; //a string of 20 characters

in progmem
```

We can also leverage the functionality of preprocessors when declaring the strings. This not only creates all of the strings before the code in the program even starts executing. Moreover, the syntax used for declaring strings by using preprocessors is easier and simpler as shown below;

```
#define P(name) const prog_uchar name[] PROGMEM // declare a PROGMEM string
```

Once the declaration has been done using the syntax shown above, all we have to do is use a short-expression '**P(name)** ' and it will be automatically be replaced by the entire string it is linked to.

```
P(myTextP) = "arduino duemilanove "; //a string of 20 characters in progmem
```

Now let's compare this approach to using the standard way of declaring and storing strings in the system RAM.

```
char myText[] = "arduino duemilanove "; //a string of 20 characters

void setup()

{

Serial.begin(9600);
```

```
}

void loop()

{

Serial.print(memoryFree()); // print the free memory

Serial.print(' '); // print a space

Serial.print(myText); // print the string

delay(1000);

}
```

Upon executing this block of code, the ‘memoryFree()’ function will tell us that we only have a total of 20 bytes of free RAM space.

Optimizing Constant Values

This section will discuss an optimization technique that will help reduce the memory usage of integer values. This is done by utilizing the ‘**#define** ’ and ‘**const** ’ expressions to tell the compiler the values are to be interpreted as constants. Once the values are interpreted as constants, they can be declared in multiple ways choosing the one that fits the situation the best.

Here's an example showing how to declare integer values as constant by using the ‘**const** ’ argument.

```
const int ledPin=13;
```

If we were to declare the value traditionally, it would be like this;

```
int ledPin=13;
```

Generally, it is recommended to use constant values wherever possible instead of just using standard integer values. Moreover, using references is also a really good practice because you might need to change the value of the integer at some point in program development. Then you will have to arduously look through the entire source code and figure out the values that also need to be adjusted when making such a change.

Three distinct approaches can define a constant integer value and each approach has been demonstrated below;

```
int ledPin = 13; // a variable, but this wastes RAM

const int ledPin = 13; // a const does not use RAM

#define ledPin 13 // using a #define

// the preprocessor replaces ledPin with 13

pinMode(ledPin, OUTPUT);
```

By looking at this block, you will notice that the first two approaches are quite similar at first glance but that's not the case at all. In the first method, we are declaring the integer as a standard variable. Since this variable can be changed at any point in the code execution, a piece of memory is reserved in the RAM for it. In the second method, we are using the **'const** ' argument to declare the integer as a constant variable. In this case, since the variable's value is constant, this means that the value will not change during code execution, thus the compiler doesn't need to reserve space for this variable in the system RAM.

Memory Handling through Conditional Compilation

There are cases when programmers keep multiple versions of the same source code for different purposes. For instance, the source code used for debugging will be a different version as compared to the source code being executed as a program on the Arduino board. Similarly, some programmers even create different versions of the same sketch program to ensure that their program is compatible with different Arduino boards (since each Arduino board has slightly different specifications, like system memory, processor speed, etc.). This is an advanced practice that helps developers a lot throughout their program's developmental stages and debugging stages as well.

To make this possible, we use conditionals during the compilation process. These conditionals mainly act on the compiler's preprocessor and directly interfere with how the compiler builds the sketch program. Thus, we can manipulate the way the program is built.

Let’s take a look at a sketch example. The example you are about to see is strictly compatible with only Arduino boards that have been released after version ‘0018.’ In other words, boards of versions ‘0018’ and older will not support this sketch.

```
#if ARDUINO > 18

#include <SPI.h> // needed for Arduino versions later than 0018

#endif
```

Let’s now look at the sketch example which will use the ‘if’ conditional statement to determine if we want to use the program for debugging purposes. If the conditional is true, then the ‘**DEBUG** ’ function defined in the header file will be executed.

```
/*

Pot_Debug sketch

blink an LED at a rate set by the position of a potentiometer

Uses Serial port for debug if DEBUG is defined

*/

const int potPin = 0; // select the input pin for the potentiometer

const int ledPin = 13; // select the pin for the LED

int val = 0; // variable to store the value coming from the sensor

#define DEBUG

void setup()

{

Serial.begin(9600);

pinMode(ledPin, OUTPUT); // declare the ledPin as an OUTPUT

}
```

```
void loop() {
val = analogRead(potPin); // read the voltage on the pot
digitalWrite(ledPin, HIGH); // turn the ledPin on
delay(val); // blink rate set by pot value
digitalWrite(ledPin, LOW); // turn the ledPin off
delay(val); // turn LED off for same period as it was turned on
#if defined DEBUG
Serial.println(val);
#endif
}
```

Conclusion

We have now finally concluded our journey of exploring the world of Arduino. Along the way, we have discovered many important methods and techniques that advanced programmers use as their daily practice. It is very important to polish your skills and use what you learned to understand the concepts even more intricately. Just reading alone is not enough. Just as how we demonstrated a practical example in every important chapter, similarly, the reader is encouraged to practice whatever technique they have learned along the way. The book is designed so that each chapter builds upon the concepts explained in the preceding chapters. Hence, if the reader does not properly pick up the crucial concepts in the beginning chapters, the upcoming chapters will be very hard to grasp, especially the practical portions.

We have gone through many coding techniques, methods, and practices that are both advance and easy to use. We have learned how to effectively manipulate strings in Arduino, use LCDs and Motion sensors, use components like Infrared Sensors to detect the intensity of the light, and control the blinking accordingly and this is just the tip of the iceberg. This book is jam-packed with coding concepts and practical illustrations. We hope that the reader found this book useful and learned a lot from it.

References:

1). Mastering Arduino by author **Jon Hoffman**

2). Arduino Cookbook by author **Michael Margolis;** ISBN: 978-0-596-80247-9