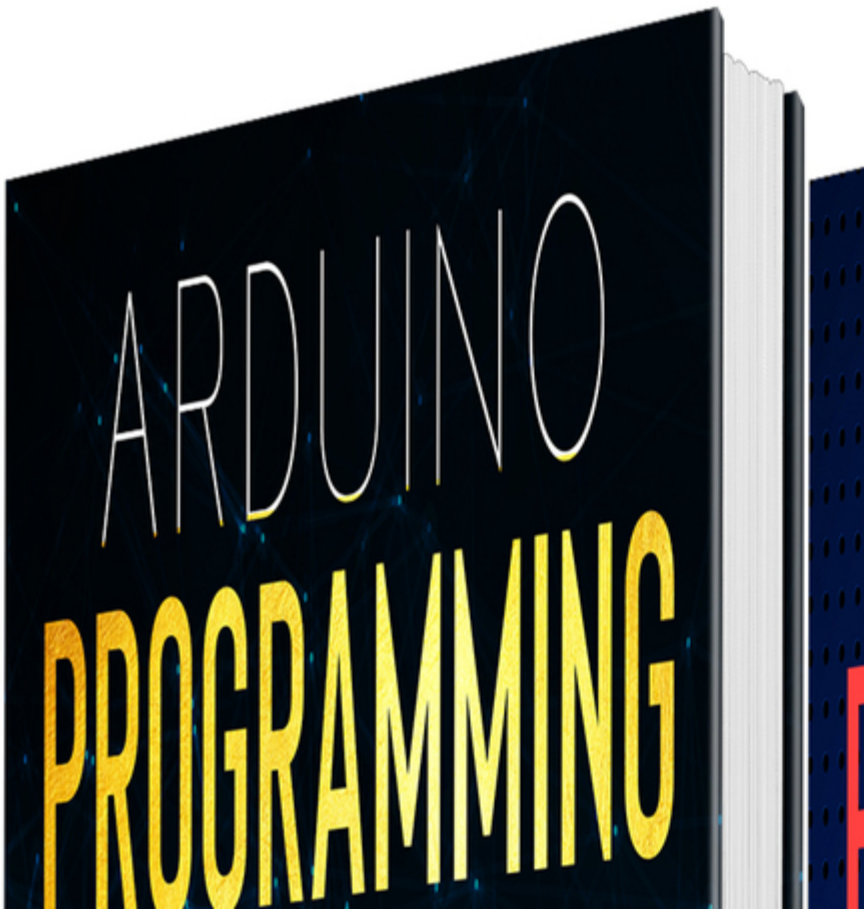

ARDUINO
PROGRAMMING
A COMPREHENSIVE BEGINNER'S GUIDE TO LEARN THE
REALMS OF ARDUINO FROM A-Z
STUART NICHOLAS
ARDUINO
PROGRAMMING
TIP AND TRICKS TO LEARN
ARDUINO PROGRAMMING EFFICIENTLY
STUART NICHOLAS
ARDUINO
PROGRAMMING
ADVANCED METHODS AND STRATEGIES TO
LEARN ARDUINO PROGRAMMING
STUART NICHOLAS


ARDUINO
PROGRAMMING

ARDUINO PROGRAMMING

STUART NICHOLAS

© Copyright 2021 - All rights reserved.

This document is geared towards providing exact and reliable information in regards to the topic and issue covered. The publication is sold with the idea that the publisher is not required to render accounting, officially permitted or otherwise qualified services. If advice is necessary, legal or professional, a practiced individual in the profession should be ordered.

- From a Declaration of Principles which was accepted and approved equally by a Committee of the American Bar Association and a Committee of Publishers and Associations.

In no way is it legal to reproduce, duplicate, or transmit any part of this document in either electronic means or printed format. Recording of this publication is strictly prohibited, and any storage of this document is not allowed unless with written permission from the publisher. All rights reserved.

The information provided herein is stated to be truthful and consistent, in that any liability, in terms of inattention or otherwise, by any usage or abuse of any policies, processes, or directions contained within is the solitary and utter responsibility of the recipient reader. Under no circumstances will any legal responsibility or blame be held against the publisher for any reparation, damages, or monetary loss due to the information herein, either directly or indirectly.

Respective authors own all copyrights not held by the publisher.

The information herein is offered for informational purposes solely and is universal as so. The presentation of the information is without a contract or any type of guarantee assurance.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are owned by the owners themselves, not affiliated with this document.

Table of Contents

ARDUINO PROGRAMMING
A Comprehensive Beginner's Guide
to learn the Realms of Arduino from A-Z

Introduction
- Where Do I Find a Microcontroller?
- Arduino
- Know the Hardware

Chapter 1 : Basic Concepts of Arduino
- Physical Computing
- Open Source Hardware
- Free Software
- Arduino
- Processing
- Fritzing

Chapter 2 : Electronics
- Concept of Electronics
- Voltage
- Electric Current
- Direct Current
- Alternate Current
- Resistance
- Ohm's Law
- Electronic Systems
- Summary of Electronic Systems
- Electronic Signals
- Voltage Divider
- Analog to Digital Converter
- PWM Pulse Width Modulation
- Serial Communication

Chapter 3 : Electronic Components
- Microcontroller
- Protoboard
- Resistor
- Thermistor
- Diode
- Transistor
- Capacitor
- LED
- RGB LED
- LCD
- Button
- Reed Switch
- Potentiometer
- Photocell
- Piezoelectric Transducer
- DC Motor
- Relay

Chapter 4 : Arduino Programming
- The Arduino Project
- Arduino IDE for Windows
- Arduino IDE for Linux
- Programming
- Programming Language
- Algorithm
- Arduino Software
- Sketches
- Serial Monitor
- Arduino Library
- Programming Arduino
- Digital Functions
- Analog Functions

Chapter 5 : Starter Kit with Arduino Uno R3
- Lessons
- Button
- Serial Reading of a Digital Input
- Serial Reading of an Analog Input
- Command with Serial Communication
- Fade
- Loop
- LDR Sensor
- Thermistor
- DC Motor
- LCD
- Typing into the LCD
- Piezo as Analog Output
- Scroll Bar with Processing
- Arduino - Thermostat

Chapter 6 : If Statement with Arduino, Control the Execution of your Code
- If with Arduino - Necessary Material
- Syntax of the if Statement with Arduino
- Comparison Operators in if Statements with Arduino
- Example if Statement with Arduino
- Boolean Operators if with Arduino
- if-else with Arduino
- Example if-else statement with Arduino
- else if with Arduino
- Example else if statement with Arduino

Chapter 7 : Battery and Battery Charge Meter with Arduino
- The Objective of the Battery and Battery Charge Meter with the Arduino Board
- Arduino Components that We Will Use
- Riding the Circuit with Arduino
- Programming the Battery and Battery Charge Meter with Arduino

Arduino Native Code

Chapter 8 : ADS1115 ADC Digital-Analog Converter for Arduino and ESP8266

What is a Digital-Analog Converter?
What is an ADS1115?
Install Library for ADS1115
Single-End Mode Programming
Differential Mode
Differential Mode Programming
Comparator Mode

Chapter 9 : Arduino Interruptions

Practical Example: Controlling the Speed of Fantastic Car Lights

Conclusion

References

ARDUINO PROGRAMMING

Tip and Tricks to Learn Arduino Programming Efficiently

Introduction

What is Arduino?
Advantages of Arduino
Arduino - Choosing a Hardware Platform
Equipment Arduino UNO R3
Clones, or Arduino (un) Original
Materials you need to program Arduino
Arduino IDE Installation
What happens when you install the wrong IDE?
If you already have Arduino

Chapter 1 : Basic Concepts of Programming

Conversions

Decimal <-> Binary
Binary <-> Hexadecimal
Decimal <-> Hexadecimal
QUICK Method
The Software
Installation
Downloading
Windows
Mac Os
Under Linux
Software Interface Software Launch
Correspondence
Approach and Use of the Software
The File Menu
Buttons
Equipment
Presentation of the Card
Constitution of the Card
Materials
Visualization
Connectivity
Installation
Windows
Test your Equipment
The Arduino Language (1/2)
Language syntax
The Minimum Code
Function
The Instructions
The Braces
Comments
Accents
The Variables
What is a Variable?

A Variable is a Number
The Name of a Variable
Define a Variable
Code: C
The Arduino Language (2/2)
What is a Loop?
The do... while loop
Concatenation
The Functions
What is a Function?
Make a Function
But what is "Void"?
Empty Functions
The "Typed" Functions
Functions with Parameters
What is going on?
The Tables
What's the Point?
The Maximum Score
Average Calculation

Chapter 2 : Input / Output Management

Our First Program
But how do you calculate this resistance?
Where do we start?
Create A New Project
What does "HIGH Value or LOW Value" Mean?
Use the Command
The Program
Millis () Function
Limits of the Delay () Function

Chapter 3 : Programming With Arduino

The Basic Structure of a Sketch

1. Variables to Name
2. Setup (essential in the program)
3. Loop (essential in the program)
Instructions
A Flashing LED
Circuit
The Change Indicators
Sketch:
An LED Pulsing Leave
Sketch:
Simultaneous Lighting and Sound
Sketch:
Activate a LED Touch of a Button
Sketch:
An RGB LED Drive
Sketch 1:
Sketch 2:
The Motion
Sketch:
Read Brightness Sensor
Read Voltages
The "Serial Monitor"
Sketch:
Knobs
Sketch:
Measure Temperatures
Sketch:
Measure Distance
Code:
Enhancements: Reversing Warning
Infrared Remote Control for Controlling Arduino Microcontrollers
Code:
Servo Drive
Code:

LCD Display
Code:
Relay Card
Code:
Stepper Motor
Cabling
Code
Code:
Drop Sensor
Code:
RFID Kit
Read an RFID tag with Arduino and Process the Data
Preparation - the first sketch with the RFID READER
RFID Sketch 2
Sketch 3

Chapter 4 : Accessories for Arduino

Keypad Shield
Code:
Instructions for LCD Display with I2C Port
Program
Sketch:
Example of use:
Two I²C Displays
Arduino RTC
Cabling:
Extension:
Keypad
The Sketch:
Keypad - Castle
Color Sensor
Cabling:
Generate Tones
Code:

Create Melodies
Arduino Tilt Sensor SW-520D
The Sketch:
Temperature Measured with the LM35
Expansion of the Program:
Code:
Measure the Voltage
Cabling:
We now come to the code:
Four-Digit 7-Segment Display
Programming
Code:

Conclusion

References

ARDUINO PROGRAMMING
Advanced Methods and Strategies
to Learn Arduino Programming

Introduction

Chapter 1 : Setting Up the Tools
The Arduino Software
Arduino Hardware
Setting Up The Arduino IDE
Working with Arduino

Chapter 2 : The Basics of Arduino Programming
Curly Bracket
Semicolons
Comments
Variables
Datatype

Boolean
Byte
Integer
Long
Double and Float
Character
Arrays
Character Arrays
Constants
Arithmetic Functions
Comparison Operators
Logical Operators
Casting
Decision Making
Looping
Functions

Chapter 3 : The Advanced Concepts of Arduino Programming
Setting Digital Pin Mode
Digital Write
Digital Read
Analog Read
Structures
Unions
Adding Tabs
Working with Tabs
Object-Oriented Programming
String Library
Arduino Motion Sensor Project
Arduino LCD Display Project

Chapter 4 : Structuring and Arduino Programming
Structuring and Arduino Program
Using Standard Variable Types

Floating- Point Numbers
Working with Groups of Values
Using Strings in Arduino
Using Strings of C Programming Language
Splitting Comma- Separated Text into Groups
Converting a Number to a String
Converting a String to a Number
Transforming the Lines of Code into Blocks
Returning More Than One Value from a Function

Chapter 5 : Coding and Memory Handling
Using the Libraries
Installing Additional Libraries
Modifying Libraries
Creating a Library
Creating Libraries from Other Libraries
Memory Handling
Saving and Fetching Numeric Values from Program Memory
Saving and Fetching Strings Using the Program Memory

Conclusion

References

ARDUINO PROGRAMMING

A Comprehensive Beginner's Guide to learn the Realms of Arduino from A-Z

STUART NICHOLAS

Introduction

The Internet of Things concept is fashionable, which refers to the ability to interact with physical devices, obtaining information/metrics (e.g., temperature, humidity, etc.) and sending commands/actions (e.g., opening the door, turning on air conditioning, etc.).

The concept, which is not new, implies collaboration between electronics professionals, programmers, and even DBAs. This book is suitable for developers and DBAs who have little knowledge of electronics, giving them an introduction to using Arduino, one of the most well-known electronics platforms used in this area.

In this section, we will make a brief introduction to the Arduino, which is basically a board with a programmable microcontroller, cheap, and easy to use.

The first thing we have to be clear about is that it is a microcontroller and how it differs from the microprocessor. We have all ever come into contact with both concepts, that is, we all have a computer at home, whether desktop, portable, etc.

Well, the core of our computers is a microprocessor, a chip in charge of performing complex operations from some instructions (which we will call program) and some input data obtaining some output data. To process and store this data, we need to connect the microprocessor to RAM and other I / O devices (Input / Output), which are connected through the motherboard.

Defined the microprocessor in a concise way, and taking into account that we had said that the microprocessor needs to be connected to the memory through the motherboard, in the microcontroller, we have both the memory where we store the program, and the memory where it is stored — data, in the same assembly (on the same chip).

Sorry if I have taken licenses to define it, I know there are several more differences, such as the location of the data memory and program in reference to the microprocessor.

Where Do I Find a Microcontroller?

We find it in most electronic devices that we use every day, such as remote controls, watches, televisions, cars, and much more. The importance of knowing how they work and how they are programmed opens up many doors.

Arduino

A few years ago, a free project, called Arduino , appeared, which facilitated access to this class of devices for students since it is an open hardware-based board (its design is free and can be reproduced by anyone).

Initially, the board was connected through a USB port to program it (it usually is done based on its IDE that can be found here). Arduino programming was not done at a low level with assembler like many microcontrollers (from now on we will call them PICs), but it is done with a language more understandable by most of us, C / C ++ (the basic reference to the language we found here and examples on this route).

With these elements, a programmer who does not know about PICs would be able to program the Arduino in a short time.

The UNO version board consisted of 14 digital I / O pins, of which 2 had a serial connection. These pins are useful for most basic sensors, or for relays, actuators, etc., which only have two states, on or off (or with the Arduino constants HIGH and LOW). It also has six analog pins, capable of reading up to 1024 voltage levels, called analog port resolution. These are used to read sensors that give us different voltage ranges depending on their state, such as heat resistance, variable resistance, etc.

Basically, with the structure of the Arduino UNO, anyone can enter the world of PIC programming. But, for larger projects, more power is needed, with which the Arduino boys were creating plates and improved versions of the Arduino UNO. I expose a few:

Arduino Leonardo

It is the first evolution of the Arduino Mega, in fact, it has the same ports, but the PIC is different, with greater capacity and higher working frequency. Another difference is the USB port, which at the same time of

being used to upload the program, can use the USB as a host; that is, we can use it as a keyboard, etc ... Another inclusion is the SDA and SCL ports, used for devices that communicate through the I2C protocol.

Arduino Mega 2560

For those who fall short, the Leonardo designed the Arduino Mega 2560, with much more speed in the microcontroller and many more digital (54 pins) and analog (16 pins) ports. We have the pins for I2C like Leonardo (although they change position).

This design is also the Arduino Mega ADK, which is a modified board of the Mega 2560, but a USB host port is added. Google has designed several projects with this board and its flagship, Android, an example you have on this page .

Another evolution is the Arduino DUE, with the same design as the Mega 2560 but with a 32-bit microprocessor. It is much faster than the Mega 2560.

Arduino YUN

Basically, this board has the same pin and processor characteristics as the Arduino Leonardo, but a microcomputer has been incorporated where a small Unix resides, which allows us to mount fundamental web servers. This board is mostly used for projects in which the sensors have to report through an Ethernet network quickly and easily.

More developments will come in the future, merging the power of Arduino boards with computer motherboards and offering us the possibility of creating a multitude of projects, with unlimited power and scalability.

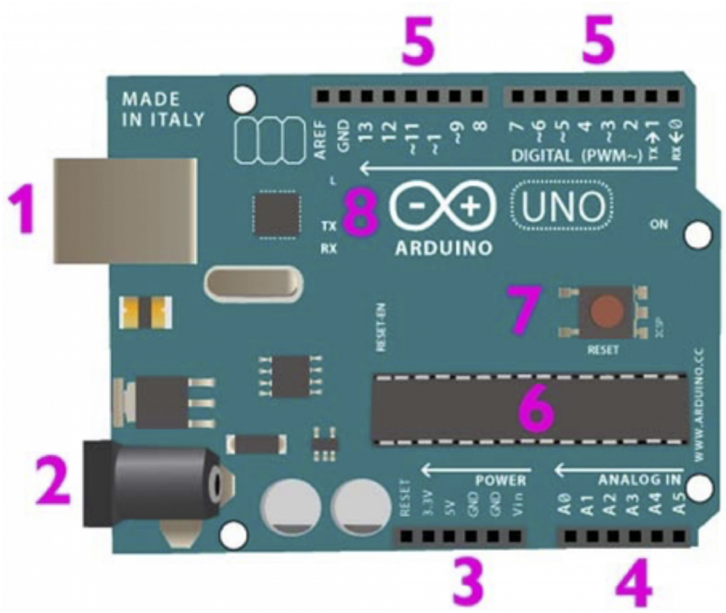
Know the Hardware

There are several models of Arduino. Here we will just work with the Uno and Mega models.

The biggest differences between them are the speed, the amount of memory, and the number and type of pins. Some are connected via USB via a Type B USB connection, and others via a Type B Micro-USB connection. The positions of the Reset button and LEDs also vary by model and version.

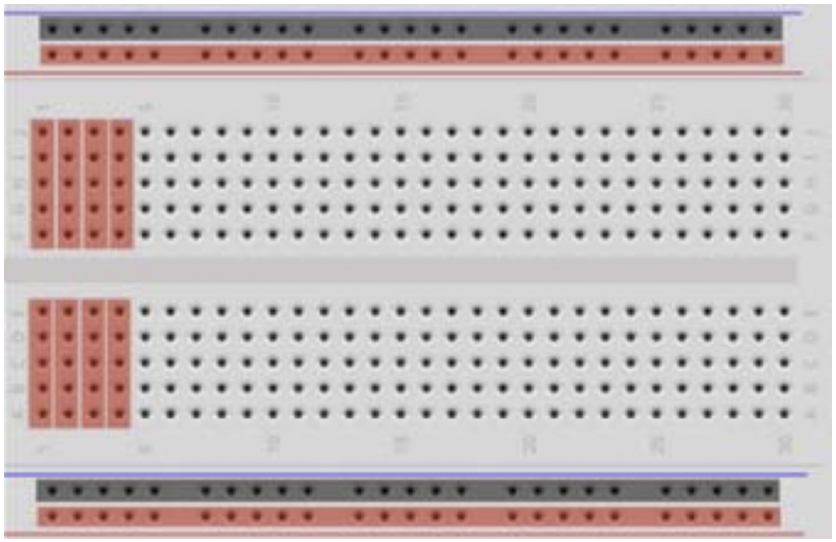
For the purposes of this workshop, we will use pins and features that are common to everyone.

Arduino Uno



1. USB for power and communication;
2. Alternative power (from 9V to 12V, center pin is +);
3. Power pins (outputs and inputs);
4. Analog input pins (can be used as digital);
5. Digital input / output pins (~ means PWM digital output)
6. Processor;
7. Reset button
8. LEDs: RX and TX indicate serial communication (USB port), L shows pin 13 value.

Breadboards



Hello World with Integrated LED

After installing the Arduino IDE (https://Arduino.CC/en/Main/Software) and connecting it via the USB cable, you need to configure it. We start by choosing the serial port where it is connected (usually there will only be one).

Next, you must choose the Arduino model.

In some cases (such as Arduino Mega), it is also necessary to choose the version.

The Arduino IDE comes with some examples. Let's start by using the Blink example for flashing the integrated LED on board (L), which is connected to pin 13.

The example shows two functions:

Setup()

In this function, we initialize the hardware we are using. This function runs once when Arduino is powered.

Loop()

In this function, we put the program that performs the desired function. At the end of this function, it is called again (repeats indefinitely).

After creating the program, we must validate it. For this, we use the first toolbar button (√).

To send the program to Arduino, we use the second button (→). During upload, we will see several messages at the bottom of the IDE as well as the flashing RX and TX LEDs on the board. Once the upload is complete, the program starts automatically. In this case, the LED L on the board will flash.

You can adjust the delay()to change the duration of each state (on or off) of the LED. The delay()uses milliseconds. We start by implementing the following scheme, with a button and a high resistance (e.g., 10 kΩ).

This scheme, which, while the button is not pressed, is sending 0V to the input, is called a pull-down. There is also a pull-up, but by default, the input gets + 5V (or + 3.3V, depending on the type of card).

Then we load the Button example. After upload, the button will work to activate the LED. Using the scheme from the previous lab, we will send the status of the button over the USB / serial connection to the computer. Let's open the DigitalReadSerial example.

After sending the program to Arduino, let's see what is sent to the USB / serial port using Serial Monitor.

Note: Before turning on the Serial Monitor, in some, the serial output buffer fills up quickly, then the LED starts flashing randomly, and the program hangs on the Serial.println().

We can also send custom to debug text. For example, we add the line (always respect case).

```
[...] Serial. println ( buttonState );
if ( buttonState ) Serial . println ( "ON" ); else Serial . println ( "OFF" );
delay ( 1 ); // delay in between reads for stability [...]
```

Building on the scheme of previous labs, let's add a relay (which acts as a switch that can control external equipment).

We add to the end of the function setup():

```
pinMode ( 13 , OUTPUT );
pinMode ( 12 , OUTPUT );
```

We add to the end of the function loop():

```
digitalWrite ( 13 , buttonState );
digitalWrite ( 12 , ! buttonState );
```

Why is the pin 12 (relay) value reversed? The relays we are going to use will be active if their signal pin receives 0V, and they are inactive if they receive 5V (or not turned on). Not all relays are like this, so try testing before using it.

Using the scheme of previous labs, we will add an analog temperature or light sensor to pin A0.

Next, we load the example AnalogReadSerial.

We open the Serial Monitor again to see the sensor values. Analog pin reading (A0, A1, etc.) returns values between 0 and 1023 (i.e., linearly between 0V and 5V). However, likely, the sensor chosen will only show values in part of this range. Record the minimum and maximum sensor values for use in the following laboratory.

Using the scheme of previous laboratories, we will define rules for activating the relay. For example, if we use a light sensor, when we no longer have light, we want to enable the relay to turn on a lamp.

We add to the end of the function setup():

```
pinMode ( 13 , OUTPUT );
pinMode ( 12 , OUTPUT );
pinMode ( 2 , INPUT );
```

We add to the end of the function loop():

```
if ( sensorValue > 500 || digitalRead ( 2 )) { // Relay on
  digitalWrite ( 13 , HIGH );
  digitalWrite ( 12 , LOW ); } else { // Rele off
  digitalWrite ( 13 , LOW );
  digitalWrite ( 12 , HIGH ); }
```

The exact value (here 500) and the operator (>or <) may vary by sensor. Use the values obtained in the previous laboratory to calibrate.

Chapter 1

Basic Concepts of Arduino

Physical Computing

Physical computing means building interactive physical systems through the use of software and hardware that integrate, can feel and respond to the analog world. Although this definition is broad, in a more general sense, physical computing is a creative framework for understanding the relationship between humans and the digital world. In practice, this term often describes design drawings DIY or objects using sensors and microcontrollers for translate analog inputs to software-based systems or control electromechanical devices such as motors, servos, lighting, or other hardware.

Other physical computing implementations work with voice recognition, which captures and interpret sound waves through microphones or other devices, audio wave detection devices, also computer vision, which applies algorithms to videos detected by some type of camera. Tactile interfaces are also an example of physical computing.

Prototyping (creating quick assemblies with the help of a protoboard and basic electronics components) plays an important role in physical computing. Tools such as Arduino and Fritzing are useful for designers, artists, students, and hobbyists because they help to make prototypes quickly.

Open Source Hardware

Free hardware is called hardware devices whose specifications and schematic diagrams are publicly accessible, either under some kind of payment or free of charge. The philosophy of free software (ideas about freedom of knowledge) is applicable to that of free hardware. It must be remembered at all times that free is not synonymous with free. Free hardware is part of free culture.

Open Source Hardware consists of physical technology devices designed and offered by the open design movement. Both free software and open-source hardware are created under the open-source culture movement and apply this concept to a variety of components. The term usually means that the information on the hardware is easily recognized.

The design on the hardware (i.e., mechanical drawings, schematics, material list, PCB layout data, source code, and integrated circuit layout data), plus of the free software that drives the hardware, are all released with the free and open-source approach. Every year the Open Source Hardware Association organizes the Open-Hardware Summit conference, which is the world's first comprehensive conference on open hardware, a space to discuss and draw attention to this rapidly growing movement.

Since the hardware has direct variable costs associated with it, no definition of free software can be applied directly without modification. In contrast, the term free hardware has been used primarily to reflect the use of free software with hardware and the free release of information regarding hardware, often including the version of schematic diagrams, designs, and assemblies.

Free Software

Free software is software that is distributed along with its source code and is released under terms that guarantee users the freedom to study, adapt/modify, and distribute the software. Free software is often developed in collaboration between volunteer programmers as part of an open-source software development project.

The Free Software Foundation considers software as free when it meets the four types of freedom for users:

- Freedom 0: The freedom to run the program, for any purpose;
- Freedom 1: The freedom to study the software;
- Freedom 2: The freedom to redistribute copies of the program so that you can help your neighbor;
- Freedom 3: Freedom to modify the program and distribute these modifications so that the entire community benefits.

The users of this type of software are free because they do not need to ask permission and are not bound by restrictive proprietary licenses. The Open Source Initiative (OSI) is an organization dedicated to promoting open-source or free software. It was created to encourage a rapprochement of commercial entities with free software. Its main performance is to certify which licenses fit as free software licenses, software, and its technological and economic advantages.

Free software, although this name is also sometimes confused with "free" because of the ambiguity of the term "free," so "free software" and "logical free" are also used) It is the name of the software that respects the freedom of the users on their acquired product and, therefore, once obtained, it can be used, copied, studied, modified, and redistributed freely. According to the Free Software Foundation, free software refers to the freedom of users to execute, copy, distribute, study, modify the software, and distribute it modified.

OSI, like many community members, believes that software is first and foremost a tool and that the merit of this tool should be judged based on technical criteria. For them, free software, in the long run, is economically more efficient and of better quality and should, therefore, be encouraged. In addition, the participation of companies in the free software ecosystem is considered fundamental because they are the companies that enable the increase in the development, implementation, and use of free software.

Arduino

Arduino is an open electronics platform for prototyping based on free, flexible, and easy-to-use software and hardware. It was developed for artists, designers, hobbyists, and anyone interested in creating objects or interactive environments. The Arduino can acquire environmental information through its input pins, for which a complete range of sensors can be used. On the other hand, Arduino can act in the environment by controlling lights, motors, or other actuators.

The fields of action for the control of systems are immense, being able to have applications in the area of 3D printing, robotics, transport engineering, agronomic engineering, musical engineering, fashion, and many others. The microcontroller of the Arduino board is programmed using the Arduino

programming language, based on Wiring, and the development environment (IDE) is based on Processing.

The projects developed with Arduino can be run even without the need to be connected to a computer, although they can also be done by communicating with different types of software (such as Flash, Processing, or MaxMSP). Plates can be handmade or purchased assembled from the factory.

You can download the software that can be done for free, and the board designs are available under an open license, so you are also free to adapt it to your needs.

Processing

Processing is an open-source programming language, and integrated development environment (IDE) built for the electronic arts and communities of visual projects to teach basic notions of computer programming in a visual context. The project was initiated in 2001 by Casey Reas and Ben Fry, both former members of the MIT Media Lab Computing Group. One of the objectives of Processing is to act as a tool for nonprogrammers initiated with programming, through immediate satisfaction with visual feedback.

Fritzing

Fritzing is an open-source electronic design automation program designed to help designers and artists move from prototypes (using, for example, test boards) to final products. Fritzing was created under the principles of Processing and Arduino and allows designers, artists, researchers, and amateurs to document your Arduino-based prototype and create printed circuit diagrams for later manufacturing. Besides.., it has a complementary website that helps share and discuss projects, experiences, and reduce manufacturing costs.

Chapter 2

Electronics

From the beginning of this chapter, you will find out about the technical terms that electronics have and that at the end of this one, you will most likely manage expertly.

Concept of Electronics

In a broader definition, we can say that electronics is the branch of science that studies the use of circuits formed by electrical and electronic components, with the main objective of representing, storing, transmitting, or processing information beyond the control of processes and servomechanisms.

From this point of view, it can also be said that the internal circuits of the computers, telecommunications systems, the various types of sensors and transducers are all within the area of interest of electronics. It is divided into analog and digital because its work coordinates choose to obey.

These two forms of presentation of the electrical signals to be treated. It is also considered to be a branch of electricity, which, in turn, is a branch of physics where the phenomena of the charges are studied - elementary electrical power, the properties, and the behavior of the electron, photons, elementary particles, electromagnetic waves, etc.

It studies and uses systems whose operation is based on the conduction and control of the flow of electrons or other electrically charged particles. The design and large construction of electronic circuits to solve practical problems are part of electronics and the fields of electronic, electromechanical, and computer engineering in the design of software for their control.

Electronics currently develops a wide variety of tasks. The main uses of electronic circuits are control, processing, distribution of information, conversion, and distribution of electrical energy. These two uses involve the creation or detection of electromagnetic fields and electric currents. Look

around the radio, TV, PC, mobile phone, washing machine, all of them have electronics.

Voltage

Electric voltage, also known as the potential difference (DDP) or voltage, is the difference in electrical potential between two points or the difference in potential electrical energy per electric charge unit between two points. Its unit of measure is the volt or joules by coulomb. The potential difference is equal to the work that must be done per unit of load against an electric field to move any load.

It is a physical quantity that drives electrons along a conductor (for example, a cable) in a closed electrical circuit, causing the flow of an electric current. Its unit is Volt (V). The instrument used to measure the voltage is known as a voltmeter.

A voltmeter can be used to measure the potential difference between two points in a system, and usually, a common reference point is earth. Fields can cause electrical voltage by an electric current under the action of a magnetic field by a variant magnetic field or a combination of the three.

Electric Current

Electric current is the orderly flow of electrically charged particles, or also the displacement of loads within a conductor when there is an electrical potential difference between the ends. The standard unit in the International System of Units for measuring current intensity is ampere. To measure the current, you can use an ammeter. An electric current, since it is electric, this is a phenomenon that can be used as an electromagnet, being this the principle of operation of a motor.

The instrument used to measure the intensity of the electric current is the galvanometer, which, calibrated in amps, is called an ammeter placed in series with the conductor whose intensity you want to measure.

Direct Current

Direct current, direct current, galvanic current, or direct current (DC) is the orderly flow of electrons in one direction at a time. This type of current is

generated by car or motorcycle batteries (6, 12 or 24V), small batteries (usually 9V), batteries (1.2V and 1.5V), dynamos, solar cells and power supplies of various technologies, which rectify the alternating current to produce direct current.

It is the continuous flow of electrons through a conductor between two points of different potential. In the direct current, the electric charges always circulate in the same direction. It is continuing the current still maintains the same polarity. In the systematic American standard, the color black corresponds to the negative and the red to the positive or is symbolized for the positive with VCC, +, VSS, and for the negative with 0V, -, GND.

It has usually used for powering electronic devices (between 1.2 V and 24V) and digital circuits of computer equipment (computers, modems, hubs, etc.). This type of circuit has a negative pole and a positive pole (is polarized), whose intensity is maintained. More correctly, the intensity increases at the beginning to a maximum point, remaining continuous, that is, without changing. When turned off, it decreases to zero and extinguishes.

Many devices need direct current to operate on all those that carry electronics (audiovisual equipment, computers, etc.). For this purpose, power supplies are used. You can find it in the batteries, batteries, output of the computer chargers.

Alternate Current

Alternating current (AC) is an electrical current whose direction varies in time, as opposed to the following of the direct current whose direction remains constant over time. The usual waveform in an AC power is sinusoidal for to be the most efficient form of energy transmission. However, in specific applications, different waveforms are used, such as triangular or square waves. While the direct current source consists of the positive and negative poles of an alternating current is composed of phases (and often neutral wire).

It is the electric current in which the magnitude and direction vary cyclically. The most commonly used alternating current waveform is that of

a sine wave.

The AC voltage is what reaches the electrical outlets of homes and businesses, and it is very common to find it in the sockets where our appliances are connected. However, audio and radio signals transmitted by electric cables are also examples of alternating current. In these uses, the most important purpose is usually the transmission and retrieval of the information encoded (or modulated) on the AC signal.

Resistance

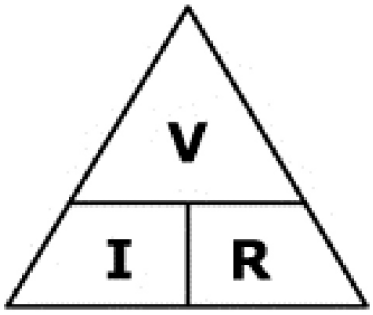
Electrical resistance is the capacity of a body to oppose the passage of electric current even when there is a difference of applied potential. It is measured in ohms (Ω).

Resistors are components whose purpose is to offer opposition to the passage of electric current through its material. We call this opposition electrical resistance. They cause a voltage drop in the power supply to a power supply unit may be limited to some part of an electrical circuit, but it will never cause an electric current drop, even though the current is limited.

This means that the electrical current entering one resistor terminal will be exactly the same as that of leaving the other terminal, but there is a voltage drop. Using this, it is possible to use the resistors to control the voltage on the desired components. You can find resistance in electric heaters, electronic cards, stoves are very useful to limit the passage of current.

Ohm's Law

Ohm's Law states that the current (I) circulating through a given circuit is directly proportional to the applied voltage (V) and inversely proportional to its resistance (R).



The pyramid on the side is very useful to know this formula. For example, cover with one finger the letter V (voltage), then the voltage will be equal to the current (I) times the resistance (R). Or, to calculate the resistance, divide the voltage (V) by the current (I).

Electronic Systems

An electronic system is a set of circuits that interact with each other to obtain a result. One way of understanding electronic systems is to divide them into inputs, outputs, and signal processing.

Inputs

Inputs are electronic or mechanical sensors that take signals (in the form of temperature, pressure, humidity, contact, light, motion, pH, etc.) from the physical world and convert them into current or voltage signals. Examples of inputs are gas sensors, temperature sensors, pulsators, photocells, potentiometers, motion sensors, and many more.

Outputs

Outputs are actuators or other devices that convert current or voltage signals into physically useful signals such as motion, light, sound, force, or rotation, among others. Examples of outputs are motors, LEDs, or light systems that switch on automatically when darkens or a buzzer that generates several tones.

Signal Processing

Signal processing is carried out using circuits known as microcontrollers. These are integrated circuits built to manipulate, interpret, and transform voltage and current signals coming from sensors (inputs) and to activate certain actions in the outputs.

Summary of Electronic Systems

As an example, we imagine a TV set. The input is a signal received by an antenna or a cable. The integrated circuits inside the device extract information about brightness, color, and sound from this signal. The devices of output are the LCD screen, which converts the electronic signals into visible images, and the speakers, which emit the sound.

Another example might be a circuit that controls the temperature of an environment. A temperature sensor and an integrated circuit are responsible for converting an input signal to an appropriate voltage level. If the registered ambient temperature is too high, this circuit will send the information to a motor to turn on a fan that will cool the room.

Electronic Signals

The inputs and outputs of an electronic system will be considered as variable signals. In electronics, we work with variables that are taken in the form of voltage or current, which can simply be called signals. The signals can be of two types: digital or analog.

Digital Variable

Also called discrete variables are characterized by having two different states and, therefore, can also be called binary (in logic would be True (V) and False (F) values, or could be 1 or 0 respectively). An example of a digital signal is the bell switch of your house because it has only two states, pulsed and pulseless.

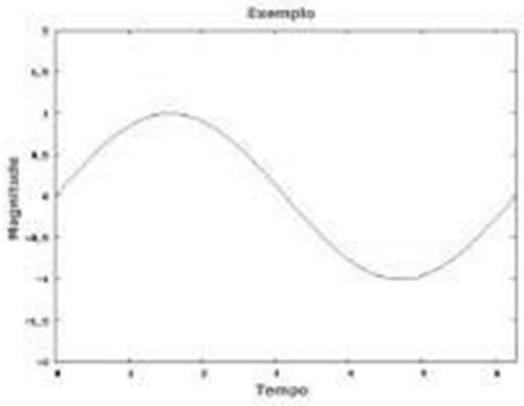
Analog Variable

They are those that can take an infinite number of values comprised between two limits. Most real-life phenomena are signals of this type (sound, temperature, brightness, etc.).

An example of an analog electronic system is that of a speaker, who is concerned with amplifying the sound of his voice so that a large audience can hear it. The sound waves that are analog in their origin are captured by a microphone and converted into a small analog voltage variation, called an audio signal.

Voltage Divider

In electronics, the voltage divider rule is a design technique used to create a voltage (Volt) that is proportional to the other (Vin). In this way, the voltage of a source is distributed among one or more resistors connected in series. On a circuit like this, two resistors are connected in series as in the following diagram:



The output voltage, Volt, is given by the formula:

$$V_{out} = \frac{R_2}{R_1 + R_2} \cdot V_{in}$$

This way, we can obtain any fraction between 0 and 1 of the $_{Vin}$ voltage.

Analog to Digital Converter

An analog-to-digital (ADC) converter is an electronic device capable of generating a digital representation from an analog magnitude by converting an analog input at a binary value. It is used in electronic equipment such as computers, sound and video recorders, and telecommunications equipment.

These converters are very useful in the interface between digital devices and analog devices and are used in applications such as sensor reading, audio, video, etc. scanning.

PWM Pulse Width Modulation

Pulse Width Modulation (PWM) - better known by its acronym in English PWM (Pulse-Width Modulation) - of a signal or power supplies involves modulating its duty cycle to carry any information about a communication channel or to control the amount of energy being sent in a load.

For example, if we apply PWM to an LED, we can vary the brightness intensity, and if we use PWM to a DC motor, we can achieve its speed with the characteristic of keeping its strength constant.

Serial Communication

It is a digital data communication interface where information is sent one bit at a time, sequentially. It's different from parallel communication, where everyone the bits of each symbol are sent together. The serial interface is used in all long-range communication and most cases, in computer networks.

One of its uses is to monitor the state of a connected peripheral through the computer screen. For example, when pulsing the letter, A on the keyboard should light an LED that is remotely connected to the computer.

Chapter 3

Electronic Components

Various electronic components join forces to achieve fantastic applications such as your home TV or computer, inside you go and find cards with resistors, capacitors, integrated circuits, transistors, among others.

Microcontroller

A microcontroller is a programmable integrated circuit capable of to execute the orders recorded in his memory. A microcontroller has inside it three main functional units: central processing unit, memory, input, and output peripherals.

The microcontrollers are different from the processors because, besides the logical components and usual arithmetic of a general-purpose microprocessor, the microcontroller integrates additional elements into its internal structure, such as memory read and write for data storage, read-only memory for storage of programs, EEPROMs for permanent data storage, peripheral devices such as analog / digital converters (ADC), digital-to-analog (DAC) converters in some cases; and, data input and output interfaces.

They are generally used in automation and control of products and peripherals, such as automotive engine control systems, remote controls, office and residential machines, toys, supervisory systems, etc. By reducing the size, cost, and energy consumption, and when compared to the way of the use of conventional microprocessors, allied to the ease of application design, along with its low price, the microcontrollers are an efficient alternative for controlling many processes and applications.

Protoboard

It is a reusable board used to build prototypes of seamless electronic circuits. A protoboard is made of perforated plastic blocks and several thin sheets of copper, tin, and phosphorus alloy.

Resistor

It is a component formed by carbon and other resistant elements used to limit the electric current in a circuit. It opposes the passage of the current. The maximum current in a resistor is conditioned by the maximum power that can dissipate your body. This power can be visually identified from the diameter without another indication being necessary. The most common values are 0.25 W, 0.5 W, and 1 W.

Due to their very small size, it is not possible to print their respective resistors on the resistors. The color-coding was then chosen, which consists of colored stripes on the resistor body indicated as a, b, c, and % tolerance. The first three ranges are used to indicate the nominal value of its resistance and the last range, the percentage by which the resistance can vary its nominal value, according to the following equation:

$$R = (10a + b) \times 10c \pm \% \text{ of tolerance}$$

The value of the electrical resistance is obtained by reading the figures as a one, two, or three-digit number; it is multiplied by the multiplier, and the result is obtained in Ohms (Ω).

Thermistor

The NTC (Negative Temperature Coefficient) thermistor is a temperature-sensitive semiconductor electronic component used for control, measurement, or polarization of electronic circuits. It has a coefficient of resistance variation that varies negatively as the temperature increases, i.e., its electrical resistance decreases with increasing temperature.

Diode

It is the simplest type of semiconductor electronic component. It is a component that allows the chain to cross only in one direction.

Transistor

It is mainly used as an amplifier, an electrical signal switch, and an electric rectifier in a circuit. The term comes from the English transfer resistor (resistor/ transfer resistance), as its inventors knew it.

The process of resistance transfer in the case of an analog circuit, it means that the characteristic impedance of the component varies upwards or downwards from the pre-assigned polarization.

Thanks to this function, the electrical current passing between collector (C), base (B), and emitter (E) of the transistor varies within specific pre-set parameters and processes the signal amplification.

The term "amplify" means the procedure for making a weaker electrical signal on a stronger one. A low-intensity electrical signal, such as the signal generated by a microphone, is injected into an electronic circuit (transistorized, for example), whose main function is to transform this weak signal generated by the microphone into electrical signals with them.

This whole process is called signal gain. Nowadays, transistors are found in all appliances for household and everyday use: radios, televisions, recorders, stereos, microwaves, washing machines, cars, calculators, printers, mobile phones, etc.

Capacitor

The capacitor is an electrical device capable of storing electrical charges. In electronic circuits, some components require DC power, while the power supply is connected to AC power. Solving this problem is one example of the usefulness of a capacitor.

This element is capable of storing electrical potential energy during a time interval and is constructed using a uniform electrical field. A capacitor consists of two conductive parts, called armatures and insulating material with specific properties called the dielectric.

LED

The LED (Light Emitting Diode) is a diode that emits light when energized. LEDs have many advantages over incandescent light sources such as lower energy consumption, longer lifetime, smaller size, exceptional durability, and reliability. The LED has a polarity, an order of connection. Connecting it upside down will not work correctly. Check the drawings for a match between the negative and the positive.

They are mainly used in microelectronic products, such as a warning signal. It is also widely used in panels, curtains, and led tracks. They can be found in larger sizes, such as some models of traffic lights or displays.

RGB LED

An RGB LED is an LED that incorporates in a single package three LEDs, one red, one green, and one red, blue. In this way, it is possible to form thousands of colors by adjusting each color individually. The three LED's are joined by a negative or cathode.

LCD

A liquid crystal display, or LCD (liquid crystal display), is a thin panel used to display information by electronic means, such as text, images, and videos. An LCD consists of an electrically controlled light polarizing liquid that is compressed into cells between two transparent polarizing blades. Its main features are lightness and portability. Its low power consumption allows it to be used in portable equipment, powered by electronic battery.

An LCD can vary the number of lines and characters per line, the color of the characters, and the background color, as well as having or not backlighting. The models with backlight have better visualization.

Button

A button, or push button, is used to activate some function. The buttons are usually activated by pressing them. A button on an electronic device often functions as an electrical switch. There are two contacts inside, and if it is a normally closed or normally open device, pressing the button will activate the opposite function to the one currently being performed.

Reed Switch

It is an electrical switch activated by a magnetic field, for example, with a magnet. When the contacts are open, they close in the presence of a magnetic field. When they're closed, they open.

It is commonly used in the door and window sensors for anti-theft alarms. The magnet is attached to the door and the reed switch to the stop.

Potentiometer

A potentiometer is a resistance whose value is variable. In this way, indirectly, you can control the current intensity that flows through a circuit if it is connected in parallel, or control the voltage by connecting it in series. They are suitable for use as a control element in electronic devices. You can activate it to vary the normal operating parameters. An example is the volume button of a radio.

Photocell

The LDR (Light Dependent Resistor) is a resistance whose value in ohms varies according to the incident light. A photocell has a low resistance value in the presence of light and a high value in its absence.

It can be found in various consumer goods, such as cameras, light meters, radio clocks, security alarms, or public lighting systems.

Piezoelectric Transducer

A piezoelectric transducer is efficient for detecting vibrations or blows. It can be used as a sensor by reading the output voltage. This electroacoustic transducer can also be used as a small buzzer to produce a continuous or intermittent sound or buzz.

DC Motor

The direct current (DC) motor is a machine that converts electrical energy into mechanical energy, causing a rotary motion. This direct current machine is one of the most versatile. Its easy control of position, pause, and speed makes it one of the best options in process control and automation applications. For example, you can find yourself in the traction of battery-powered toy cars or on the wheels of a robot.

Relay

It is an electromechanical switch used to turn devices on or off. When a current circulates through the internal coil, it creates a magnetic field that attracts one or a series of contacts by the closing or opening circuits. When the coil current ceases, the magnetic field also ceases, causing the connections to return to their original position.

Chapter 4

Arduino Programming

The Arduino Project

The Arduino project began in 2005 to create a device for students that would offer integrated control of design and interaction projects, and that would be more cost-effective than the prototyping systems available to date.

What we now call Arduino (the microcontroller) was born in the Italian city of Ivrea. In this same city in the 10th and 11th centuries, there was another Arduino (a nobleman) who proclaimed himself king of all Italy; obviously, the thing didn't work and, as was usual at the time, he was killed by his rivals. The fact is that in his hometown he is still very remembered, the main avenue of the city is called "Via Arduino" as well as many local shops.

While they lived there, the members of the team that created Arduino (the microcontroller), after work, would have a beer. Where? At the Arduino Bar. So the name of Arduino (the microcontroller) is a tribute to Arduino (the bar), which in turn was a tribute to the other Arduino (the nobleman).

The Arduino project was developed by Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, and David Mellis. It is based on an open-source platform called Wiring, created by Colombian artist Hernando Barragan during a master's thesis at the Interaction Design Institute Ivrea . On the other hand, Wiring is based on Processing, and Casey Reas and Ben Fry created its integrated development environment.

"I don't believe that Arduino would exist without Wiring, and I don't believe that Wiring would exist without Processing. And Processing would definitely not exist without Design By Numbers and John Maeda".
- Interview with Casey Reas and Ben Fry , Shiffman Daniel (Sep/2009)

Arduino Uno R3

1. The USB connector for AB-type cable
2. Reset button

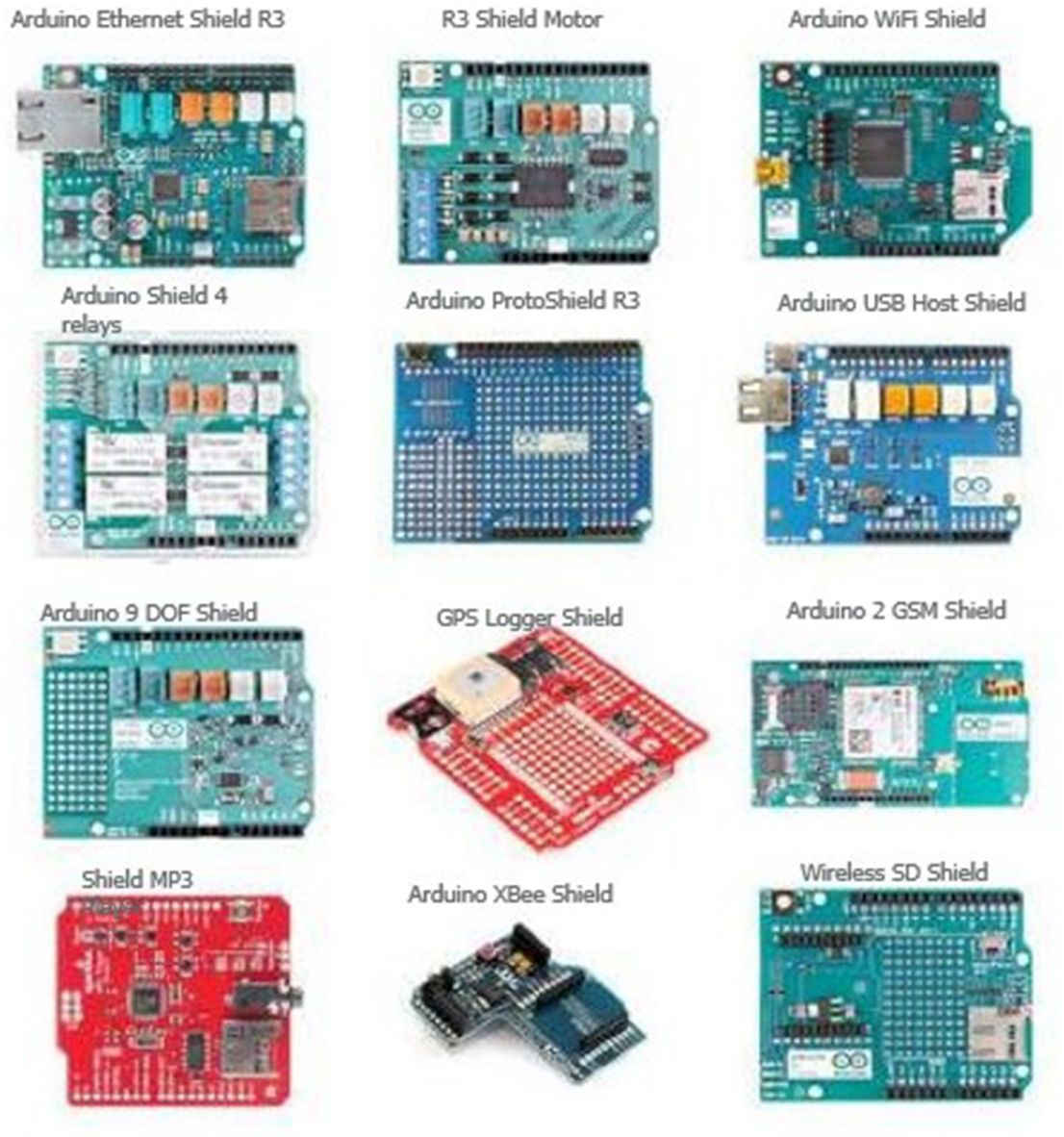
3. Digital input and output pins and PWM
4. Green LED board on
5. Orange LED connected to pin13
6. ATmega in charge of communication with the computer
7. LED TX (transmitter) and RX (receiver) of serial communication
8. ICSP port for serial programming
9. ATmega 328 Microcontroller, Arduino's brain
10. Crystal quartz 16Mhz
11. Voltage regulator
12. 2.1mm female connector with positive center
13. Voltage and ground pins
14. Analog inputs

Arduino Family

Over the years, the Arduino line has been growing more and more and bringing solutions to the most diverse projects.

Shields for Arduino

A shield is a board that allows you to expand the original features of Arduino. Some examples:



Arduino IDE for Windows

Arduino board and a USB AB cable

This lesson will help you install Arduino Uno, Arduino Duemilanove, Nano, Arduino Mega 2560, or Diecimila boards. For other boards of the Arduino line, you should look for the corresponding lesson. You will also need a USB AB cable.

Download Arduino Software

Download the latest version of the Arduino software. When finished, unzip the file and maintain the folder and subfolder structure. If you want to save this folder in the C: drive of your computer. Inside this folder, there is a file called arduino.exe, which is the entry point of the Arduino program, the IDE (Integrated Development Environment).

Connecting Arduino

The isolated Arduino Uno uses power from the computer via the USB connection; no external power is required. Connect the Arduino card to the computer using the USB AB cable. The green power LED (PWR) should light up.

Installing drivers

Drivers for Arduino Uno or Arduino Mega 2560 with Windows 7, Vista or XP:

- Connect the card to the computer and wait for Windows to start the driver installation process. After a few moments, the process will fail. Click finish and dismiss the wizard's help.
- Click on the Main Menu and open the Control Panel.
- Within the Control Panel, navigate to System and Security. Then click System, select Hardware, and then click Device Manager.
- Search for Ports (COM & LPT), where you should see an Arduino UNO option (COMxx).
- Right-click on Arduino UNO (COMxx) and choose the Update Driver option.
- Then choose the Install option from a specific list or location (Advanced) and click move forward.
- Finally, browse and choose the Arduino.inf driver located in the Drivers folder of the Arduino software you downloaded.
- Windows will finish installing the driver from this point.

Opening the Arduino Program

Double-click the Arduino application, the arduino.exe file. If the program loads with the language that is not your preference, you can change it in the preferences section of the program.

Example Blinking

Open the example Blink: File > Examples > 01.Basics > Blink

Select your Card

You must select your Arduino card: Tools > Card > Arduino Uno.

Select the Port

Now select the serial port that will connect Arduino: Tools > Serial Port. You must choose the same port you used to confirm the system in step 4.

Upload the Program

Now simply click the Load button in the program window. Wait a few seconds. You should see the RX and TX LEDs on the card blink. If the process has been executed, you will generally see a "Transfer complete" message.

After a few seconds, you will see the LED of pin 13 blink orange. In this case, congratulations! Your Arduino is ready and installed.

If you have installation problems, you can access the official Arduino page with some solutions.

Arduino IDE for Linux

It will be necessary to install some programs to use Arduino on Linux. The form of the procedure depends on the distribution.

First, download the latest version of Arduino for Linux from the official page.

For more details, select your distribution:

- ArchLinux
- Debian
- Fedora
- Gentoo
- MEPIS
- Mint
- openSUSE

- Puppy
- Pussy
- Slackware
- Ubuntu
- Xandros (Debian derivative) on Asus Eee PC
- CentOS 6

Programming

Programming is a great resource that allows us to create several sequences of logical steps to meet our needs and those of our systems. Programming is an art that requires excellent logical skill and concentration on the part of the programmer.

Programming Concept

It is the process of designing, writing, proving, debugging, and maintaining the source code of computer programs. The source code is written in a programming language. The purpose of programming is to create programs that perform the desired behavior.

The process of writing code often requires knowledge in several different areas, in addition to the domain of the language to be used, specialized algorithms, and formal logic. Scheduling encompasses areas such as application analysis and design.

To create a program that the computer interprets and executes the written instructions, a programming language must be used. In the beginning, computers interpreted only instructions in a specific language, a low-level programming language known as machine code, which was too complicated to program. It consists exclusively of number strings 1 and 0 (binary system).

To facilitate the programming work, the first scientists who worked in the area decided to replace the instructions, sequences of one and zero, by English words or letters, encoding and creating a higher level language known as Assembly. For example, to add up, you use the English letter A add. Actually, writing in assembly language is basically the same as with

machine language, but letters and words are easier to remember and understand than sequences of binary numbers.

As the complexity of the tasks performed by computers increased, it was necessary to develop a simpler method of programming. Then the high-level languages were created. While a task as simple as multiplying two numbers requires a set of assembly language instructions, in a high-level language, one is sufficient.

Programming Language

A programming language is an artificial language developed to express operations that can be performed by machines such as computers. They can be used to create programs that control the physical and logical behavior of a device, to express algorithms accurately, or as a mode of communication between people.

It consists of a set of symbols and syntactic and semantic rules that define its structure and the meaning of its elements and expressions.

The process by which the source code of a computer program is written, tested, debugged, compiled, and maintained is called programming.

Machine Language

A code system directly interpretable by a micro programmable circuit, such as a computer microprocessor or a microcontroller. A machine code program consists of a sequence of numbers that mean a sequence of instructions to be executed.

The machine language works with two voltage levels. Such levels, by abstraction, are symbolized by zero (0) and one (1), so the machine language only uses these signs.

Computer programs are rarely created in machine language but must be translated (by compilers) to be run directly by the computer.

There is currently a fashionable option of not running them directly, but using an interpreter, this one running directly on machine code and previously compiled.

Assembly Language

It is a low-level programming language for computers, microcontrollers, and other programmable integrated circuits. Machine language, which is a mere bit pattern, becomes readable by replacing the raw values with symbols called mnemonics. These symbols are generally defined by the hardware manufacturer and are based on codes that symbolize the processing steps (the instructions).

An assembly language is therefore specific to each computer architecture and can only be used by a specific microprocessor. This contrasts with most high-level programming languages that are ideally portable, which means that a program can run on a variety of computers.

High-Level Language

The high-level programming language is what is called, in Computer Science of programming languages, a language with a relatively high level of abstraction.

Far from the machine code and closer to human language, high-level languages are not directly related to computer architecture.

The programmer of a high-level language does not need to know processor features such as instructions and registers. These characteristics are abstracted in a high-level language.

For these languages, a certain amount of programming knowledge is required to perform sequences of logical instructions. High-level languages were created so that the average user could solve a data processing problem more easily and quickly.

Algorithm

An algorithm is a finite sequence of well-defined and unambiguous instructions, each of which can be performed mechanically over a finite period and with a finite amount of effort.

The concept of algorithm is often illustrated by the example of a cooking recipe, although many algorithms are more complex. They may repeat steps (making iterations) or need decisions (such as comparisons or logic) until the task is completed. An algorithm does not necessarily represent a computer program, but the steps necessary to perform a task. Its

implementation can be done by a computer, by another type of automaton, or even by a human being.

Arduino Software

To run the program, we enter the Arduino folder stored on the computer and look for the icon. Double-click to open the program.

The Arduino program is also known as the Arduino IDE (Integrated Development Environment) because besides the programming environment, it also consists of a code editor, a compiler, and a debugger.

Sketches

Software written using Arduino is called Sketches. These Sketches are written in the text editor of the Arduino IDE and are saved with the .ino file extension. This editor has cut/paste and fetch/replace text features. The message area gives feedback when saving and exporting files and also displays error information when compiling Sketches. The lower right corner of the window displays the current card and the serial port. The toolbar buttons let you check, load, create, open, and save Sketches or open the serial monitor.

Note: In IDE versions before 1.0 Sketches are saved with the .pde extension. It is possible to open these files with version 1.0, but you will be asked to save Sketch with the .ino extension.

Serial Monitor

Displays serial data being sent from the Arduino card to the computer. To send data to the card, enter the text and click the "send" button or press enter.

Communication between the Arduino board and your computer can take place at several predefined standard speeds. For this to happen, it is important to set the same speed in both Sketch and Serial Monitor.

In Sketch, this choice is made through the Serial. Begin function. And in the Serial Monitor via the drop-down menu in the lower right corner.

Note that on Mac or Linux, the Arduino card will reset (re-run your Sketch from the beginning) when you open the serial monitor.

Serial communication with the Arduino board can also be done through other programming languages such as Processing, Flash, Python, MaxMSP, and many others.

Arduino Library

The Arduino environment can be extended through the use of libraries, as can most programming platforms. Libraries provide extra functionality for use in sketches. For example, to work with hardware or data manipulation.

Some libraries are already installed with the Arduino IDE, but you can also download or create your own.

To use a library in a sketch, select in your Arduino IDE: Sketch>Import Library.

Included in the programming is the functionality of an existing library from the command:

```
#include < LiquidCrystal.h>
```

Programming Arduino

Arduino is programmed in a high-level language similar to C/C++ and usually has the following components to elaborate the algorithm:

- Structures
- Variables
- Boolean, comparison and arithmetic operators
- Control Structure
- Digital and analog functions

For more details, visit the see the extended reference to more advanced features of the Arduino language and the libraries page for interaction with specific types of hardware on the official Arduino website.

Structures

These are two main functions that every program in Arduino must-have.

The setup() function is called when a program starts running. Use this function to initialize your variables, pin modes, declare the use of bookstores, etc. This function will be performed once after the Arduino board is turned on or reset.

```
setup(){
}
```

After creating a setup() function that declares the initial values, the loop() function does exactly what its name suggests, goes into looping (it always executes the same block of code), allowing your program to make changes and respond. Use this function to control the Arduino board actively.

```
loop(){
}
```

Variables

Variables are expressions that you can use in programs to store values such as reading from a sensor on an analog pin. Here we highlight some:

- Boolean Variables

Boolean variables, so-called in honor of George Boole, can have only two values: true and false.

```
boolean running = false;
```

- Int

An integer is the main type of data for numeric storage capable of storing 2-byte numbers. This covers the range of -32,768 to 32,767 (minimum value of -2^15 and maximum value of (2^15) -1).

```
int ledPin = 13;
```

- Char

A data type that occupies 1 byte of memory and stores the value of an ASCII character. Literal characters are written in quotation marks.

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

Boolean Operators

These operators can be used within the condition in an if sentence.

- && ("and" logical)

True only if the two operands are true, that is, the first condition and the second condition are true. Example

```
if (digitalRead(2) == 1&&      digitalRead(3) == 1) { // read two switches
    // ...
    }
```

is only true if both switches are closed.

- || ("or" logical)

True if any of the operands is true, that is if the first or second condition is true. Example

```
if (x > 0 || y > 0) {
// ...
}
```

is true only if x or y is greater than 0.

- ! (denial)

True only if the operation is false. Example

```
if (!x) {
// ...
}
```

is true only if x is false (i.e., if x is equal to 0).2

Comparison Operators

if, which is used in conjunction with a comparison operator, checks when a condition is met, such as input above a certain value. The format for an if a check is:

```
if (algumVariavel > 50)
{
// do something
```

```
}
```

The program checks if any Variable (putting accents on variable names is not a good idea) is greater than 50. If it is, the program performs a specific action. Put another way, if the sentence inside the parentheses is true, the code inside the wheel keys; otherwise, the program skips this block of code.

Keys can be omitted after an if sentence if there is only a single line of code.

(defined by a semicolon) that will be executed conditionally:

```
if (x > 120)     digitalWrite(LEDpin, HIGH);
if (x > 120) digitalWrite(LEDpin, HIGH);
if (x > 120) {digitalWrite(LEDpin, HIGH);}//        all are correct
```

The sentence being verified requires the use of at least one of the following operators comparison:

x == y (x equals y)
x != y (x is not equal to y) x < y (x is less than y) x > y (x is greater than y)
x <= y (x is less than or equal to y) x >= y (x is greater than or equal to y)

Arithmetic Operators

They apply when using variables.

- = (equality)
- + (addition)
- - (subtraction)
- * (multiplication)
- / (business area)
- % (rest of the division)

Control Structures

These are instructions that allow you to decide and perform several repetitions according to certain parameters. Among the most important we

can highlight:

Switch/Case

Like if sentences, switch/case-control the flow of programs. Switch/case allows the programmer to construct a list of "cases" within a key-delimited block. The program checks each case with the test variable and executes the code if it finds an identical value.

```
switch (var) { case 1:
//do something when var == 1 case 2:
//do something when var == 2 default:
// if no value is identical, make the default
// default is optional
}
```

While

While it will cause the code block between keys to repeat continuously and indefinitely until the expression between parentheses () becomes false. Something has to cause a change in the value of the variable being checked, or the code will always be turning around inside the while. This could be the increment of a variable or an external condition, such as testing a sensor.

```
var = 0;
while(var < 200){
// some code that repeats 200 times var++;
}
```

For

The sentence is used to repeat a key-delimited code block. An increment counter is normally used to control and finalize the loop. The for sentence is useful for any repetitive operation, and is often used with arrays to operate on data sets or pins.

```
// Increase the brightness of an LED using an int PWM pin PWMpin =
13; // an LED on          pin 13
void setup()
{
// no setup required
}
```

```
void loop()
{
for (int i=0; i <= 255; i++){ analogWrite(PWMpin, i); delay(10);
}
}
```

Digital Functions

Oriented to review the status and configuration of digital inputs and outputs.

pinMode()

Sets the specified pin to behave either as an input or output.

Syntax:

```
pinMode(pin, mode)
pinMode(9, OUTPUT);      // determines digital pin 9 as an output.
```

digitalRead()

Reads the value of a specified digital pin, HIGH or LOW. Syntax:

```
digitalRead(pin)
buttonState = digitalRead(9);   // Reading the status of a button on pin 9.
```

digitalWrite()

Write a HIGH or LOW value on a digital pin. Syntax:

```
digitalWrite(pin, value)
digitalWrite(9, HIGH);      // Puts pin 9 into HIGH state.
```

Analog Functions

Ideal for reading or writing analog values.

analogRead()

It reads the value of a specified analog pin. The Arduino board contains a 10-bit analog-to-digital converter with 6 channels. With this, it can map input voltages between 0 and 5 volts to integer values between 0 and 1023.

This allows a resolution between readings of 5 volts / 1024 units or 0.0049 volts (4.9 mV) per unit.

Syntax:

```
analogRead(pin)
int a = analogRead (A0); // Reads the value of analog pin A0 and stores
//this value in variable "a".
```

analogWrite()

Write an analog value (PWM wave) on a pin. It can be used to light an LED by varying the brightness or rotate a variable speed motor.

Syntax:

```
analogWrite(pin, value)
analogWrite (9.134);        // Sends the analog value 134 to pin 9.
```

Chapter 5

Starter Kit with Arduino Uno R3

In this beginner Kit with Arduino Uno R3, you will explore lots of lessons. It has everything you need to perform all the lessons in this chapter and to start developing your own projects with the Arduino platform, without the need for welding.

Lessons

Lessons developed as an example to use all the components of your Beginner Kit with Arduino Uno R3. In each lesson, you identify the materials needed for its execution, the previous knowledge needed, and what you will learn, the assembly diagram, the programming code, tips, and extra exercises.

Hello World - Blinking

This example shows the most uncomplicated experience you can do with an Arduino to check a physical output: blink an LED.

When you're learning how to program, in most programming languages, the first code you write says "Hello World" on your computer screen. Since the Arduino board does not have a display, we will replace this function by flashing an LED.

What Will I Learn?

- Activate a digital output
- Lighting an ON/OFF LED
- Timing an output signal
- Syntax of an Arduino program

Previous Knowledge

- Digital Signal
- DigitalWrite() function
- The polarity of an LED (page 39)

- Connecting the Arduino board to the computer

Required Materials

- 1 Arduino Uno1 LED
- 1 AB USB cable

This code already comes with the Arduino IDE. You can access in: Archive > Examples > 01. Basics > Blink

We only reproduce here with explanations and comments.

In the following program, the first command is to initialize pin 13 as an output through the line.

```
pinMode(13, OUTPUT);
```

In the main loop of the code, you turn on the LED with this command line:

```
digitalWrite(13, HIGH);
```

This command directs 5 volts to pin 13 and lights it up. You switch off the LED with the following command:

```
digitalWrite(13, LOW);
```

This command removes the 5 volts from pin 13, returning to 0 and turning off the LED. Between power off and on you need enough time for one person to see the difference, so the delay() command tells Arduino not to do anything for 1000 milliseconds, or one second. When you use the delay() command, nothing else happens in this period. Once you have understood the basic examples.

Source Code

```
/ *
Blinking
Illuminates an LED for one second, and then goes out for the same time repeatedly.
* /

// Sets a name for pin 13: int led = 13;
```

```
// It is executed each time Arduino starts: void setup () {
// Initializes the digital pin as output. pinMode (led, OUTPUT);
}

// The loop () function continues to perform while Arduino is powered,
// or until the reset button is pressed.
void loop () {
digitalWrite (led, HIGH); // Lights up the LED
delay (1000); // Wait one second (1s = 1000ms) digitalWrite (led, LOW); // Turn off the LED
delay (1000); // Wait one second (1s = 1000ms)
}
```

Tips

1. In the Arduino // language it is used to add comments to the line of code, being very useful to explain a syntax or leave a reminder. An example of its use:

```
digitalWrite(13,LOW);      //Deletes the LED
```

2. Digital signals (ON and OFF) are present in many sensors. Meet some of them:

- Infrared motion sensor
- Sharp GP2Y0A41SK0F distance sensor - 4 to 30cm

Exercise 1

From the source code presented in this lesson, make the necessary modifications to make the LED stay:

- 3 seconds on and 3 seconds off
- 200 milliseconds on and 500 milliseconds off

Exercise 2

From the same source code, make a new assembly of this lesson and make the necessary modifications to the source code so that the LED is placed on Pin 5 and is on for 2 seconds and off for 1 second.

Note that for any pin other than 13, it is necessary to place a resistor in series with O LED, In this case, a resistor from 330Ω is sufficient.

Button

The button is a component that connects two points of the circuit when it is pressed. In this example, when the button is pressed, the LED lights up.

What Will I Learn?

- Wiring a circuit
- Conditional if/else
- Status of a button
- Read a digital input and write a digital output

Previous Knowledge

- Digital Signal
- DigitalWrite() and digitalRead() function
- Voltage divider
- Conditional, Boolean and comparison operators

Required Materials

- 1 Arduino Uno
- 1 Button
- 1 LED
- 1 Resistor 10kΩ
- 1 AB USB cable Jumpers
- 1 Protoboard

Source Code

```
/ *
Button
Turns on and off an LED connected to digital pin 13 when a button
connected to pin 2 is pressed.

The Circuit:
* LED connected to pin 13 and ground
* button connected to pin 2 from 5V
* 10K resistor connected to pin 2 from ground
* /
```

```
// constants are not changed.
// These are used here to define the pin numbers: const int buttonPin =
2; // the pin number of the const int ledPin button = 13; // the pin
number of the LED pin
// variables that must change:
int buttonState = 0; // variable to read the state of the button
void setup () {
// initializes the LED pin as output: pinMode (ledPin, OUTPUT);
// initializes the button pin as input: pinMode (buttonPin, INPUT);
}
void loop () {
// reads the button value: buttonState = digitalRead (buttonPin);
// check if the button is pressed.
// if yes, buttonState and HIGH: if (buttonState == HIGH) {
// turn on the LED: digitalWrite (ledPin, HIGH);
}
else {
// turns off the LED: digitalWrite (ledPin, LOW);
}
}
```

Tips

1. When you are programming with the Arduino software, many of the words you write are reserved for the language. These words are placed in a different color, and it is a hint to check if they are written correctly. As in the example:

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

2. In a project with the use of several buttons with different functionalities, it can be useful to work with parts like these:

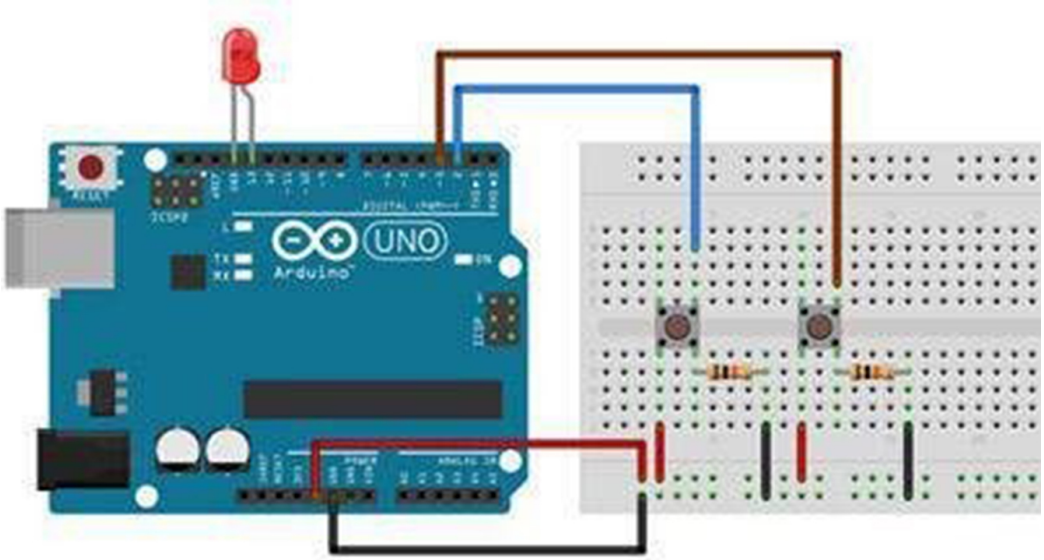


Colorful touch button set.

Exercise 1

To prevent accidents in the workplace, a safety rule in several industrial types of equipment is to require a user to press two buttons, one with each hand, to drive a machine. This is the case with the cutting machine used in paper mills.

With the following assembly, we can simulate this situation. The LED only illuminates if both circuit buttons are pressed:



Exercise 2

Make one more change to the source code of exercise 1 so that you can light the LED on pin 13 by pressing either button 1 or button 2. When you stop pressing, the LED goes out.

Serial Reading of a Digital Input

This example shows how to monitor the status of a switch by establishing serial communication between your Arduino and the computer via USB.

What will you learn?

- Controlling a digital input
- View data by computer
- Serial Monitor
- Read a digital input

Previous Knowledge

- Digital Signal
- DigitalRead() and Serial.print function
- DigitalWrite() function and comparison operators

Required Materials

- 1 Arduino Uno

- 1 Button
- 1 LED
- 1 Resistor 10kΩ
- 1 AB USB cable Jumpers
- 1 Protoboard

Source Code

In this program, the first thing you will do in the configuration function is to start the serial communication at 9600 data bits per second between Arduino and his computer: Serial.begin (9600);

Remember to place the value 9600 also in the Serial Monitor, as explained earlier.

Then initialize the digital pin 2, the pin that will scan the button as a digital input:

```
int pushButton = 2;
```

When the button is pressed, 5 volts will flow freely through its circuit, and when not pressed, the input pin will be grounded. This is a digital input, which means that the key can only have one state (seen by your Arduino as "1", or HIGH) or an off state (seen by your Arduino as a "0", or LOW), with nothing in between.

Now when you open your Serial Monitor in the Arduino environment you will see a stream of "0". if your key is open, or "1" if your key is closed.

```
/ *
DigitalReadSerial
Read the digital input on pin 2 and print the result on the serial monitor. This example is a public domain.
* /
int pushButton = 2; // pin 2 has a button connected to it. int ledPin = 13; // LED input on pin 13.
void setup () {
// Initialize serial communication at 9600 bits per second: Serial.begin (9600);
```

```
pinMode (pushButton, INPUT); // define the button as an input.
pinMode (ledPin, OUTPUT); // set the LED as an output.
}
void loop () {
// reads the input pin:
int buttonState = digitalRead (pushButton); if (buttonState == 1) {
digitalWrite (ledPin, HIGH);
} else {digitalWrite (ledPin, LOW);
}
// prints the state of the button: Serial.println (buttonState);
delay (1); // Delay between readings (in milliseconds)
}
```

Tips

1. The binary system is a positional numbering system in which all quantities are represented based on two numbers, i.e., zero and one (0 and 1). Computers work internally with two voltage levels, so their natural numbering system is the binary system (on, off).

The binary system is the basis for Boolean algebra that allows logical and arithmetic operations using only two digits or two states (yes and no, false and true, all or nothing, 1 or 0, on and off). All digital electronics and computing are based on this binary system and Boole's logic, which allows representing by digital electronic circuits (logical gates) the numbers, characters, perform logical and arithmetic operations. Computer programs are encoded in binary form and stored in media (memories, disks, etc.) under this format.

2. To remember:

- To read a digital signal use: digitalRead(numeroPin).
- To write a digital signal use: digitalWrite(numeroPin, value).
- A digital output or input is always HIGH or LOW.

Exercise 1

Here we will do one more exercise using the same assembly of this Lesson.

Once you have the button working, you often want to do some action based on the number of times the button is pressed. To do this, you need to know

when the button changes from off to on, and count how many times this change of state happens. This is called state change detection. Every 4 pulses, the LED will turn on.

```
/ *
Pulse counter (edge detection)
created on 9/27/2005, modified on 8/30/2011 by Tom Igoe. This example is from the public domain. http://arduino.cc/en/Lesson/ButtonStateChange
* /

// constants are not changed:
const int buttonPin = 2; // the pin number of the const int ledPin button = 13; // the number of the LED pin

// variables that must change:
int buttonPushCounter = 0; // counter for the number of prints of the int button buttonState = 0; // current status of the button
int lastButtonState = 0; // previous button state

void setup () {
pinMode (buttonPin, INPUT); // initialize the button pin as input pinMode (ledPin, OUTPUT); // initialize the digital pin as output Serial.begin (9600); // initialize the serial communication
}

void loop () {
// reads the button value: buttonState = digitalRead (buttonPin);

// compare the current state of the button with its previous state if (buttonState! = lastButtonState) {
// if the button status has been changed, increment the if counter (buttonState == HIGH) {
buttonPushCounter ++; Serial.print ("number of pulses:"); Serial.println (buttonPushCounter);
}
}
// saves the current state of the button as the last state to start the
// next loop
```

```
lastButtonState = buttonState;

// Turn on the LED every 4 pulses by checking the if button counter
module (buttonPushCounter% 4 == 0) {
digitalWrite (ledPin, HIGH);
It is {digitalWrite (ledPin, LOW);
}
}
```

Serial Reading of an Analog Input

This example shows how to read a pin from an analog input, map the result to a range of 0 to 255, and use that result to set the PWM modulation of an output pin to turn an LED on and off as a dimmer.

What Will I Learn?

- Control an analog input
- View data by computer
- Multiple states of a potentiometer
- Read an analog input

Previous Knowledge

- Analog signal
- Function analogRead() and Serial.print

Required Materials

- 1 Arduino Uno
- 1 LED
- 1 Resistor 330Ω
- 1 Potentiometer
- 1 AB USB cable Jumpers
- 1 Protoboard

Source Code

```
/ *
Analog Input, Analog Output, Serial Output
```

Read the analog input pin, map the result to a range between 0 and 255, and use the result to establish the PWM pulse of the output pin.
It is also possible to follow the result through the Serial Monitor.

The circuit:
- The central pin of the potentiometer connected to the analog pin 0. The side pins of the Potentiometer connected to ground and 5V.
- LED connected to digital pin 9 and ground.

Created on 29/12/2008, Modified on 04/04/2012 by Tom Igoe
This example is a public domain.
* /

```
// constants are not changed:
const int analogInPin = A0; // Potentiometer analog input
const int analogOutPin = 9; // Analog output where the LED is connected
int sensorValue = 0; // potentiometer reading
int outputValue = 0; // PWM output reading (analog)

void setup () {
// Initialize serial communication: Serial.begin (9600);
}

void loop () {
// sensorValue = analogRead (analogInPin);

// maps the result of the analog input within the range of 0 to 255: outputValue = map (sensorValue, 0, 1023, 0, 255);

// changes the value of the analog output: analogWrite (analogOutPin, outputValue);

// prints the result on the serial monitor: Serial.print ("sensor ="); Serial.print (sensorValue); Serial.print ("\ t output ="); Serial.println (outputValue);

// Wait 2 milliseconds before the next loop: delay (2);
}
```

Tips

1. Note that the analog inputs of Arduino have a resolution of 10 bits (values from 0 to 1023), but the analog outputs by PWM have a resolution of 8 bits (values from 0 to 1023).

255). This is why you need the 'map' function to 'map' the values so that they remain proportional.

2. Other elements that are also potentiometers:

- Force-sensitive resistor 0.5".
- Square force sensitive resistor.

Command with Serial Communication

Through this lesson, you will control the activation of a relay and an LED from the Serial Monitor of your computer.

What Will I Learn?

- Execute command via Serial Communication
- Controlling the activation of a relay and LED via the computer
- Char variable

Previous Knowledge

- Boolean Variables
- Serial.print

Required Materials

- 1 Arduino Uno
- 2 LEDs
- 1 Resistor 330Ω
- 1 Relay
- 1 AB USB cable Jumpers
- 1 Protoboard

Source Code

```
// ************************************************
**************************
```

```
// * Code for testing Arduino Multilogic triggering relay kit,
// * connected to digital output 2 and GND, monitored by LED 13
// * this code has public domain
// ***********************************************
**************************

// initialize a char type variable that uses 1 byte to store
// 1 character char input = 0; int rele = 2; int led = 13;
boolean y = true; // initializes a boolean type variable

void setup () {pinMode (relay, OUTPUT); pinMode (led, OUTPUT);

Serial.Begin (9600); Serial.println ();
Serial.print ("** Code to trigger relay connected to Arduino pin 2");
Serial.println ("via serial monitor **");
Serial.println ("");
Serial.println ("Press 1 and then ENTER to invert the relay state
again");
Serial.println ("Awaiting command:");
}
void loop () {
if (Serial.available ()> 0) {input = Serial.read ();

if (input == '1') {
Serial.print ("The re-read is now");
if (y) {
digitalWrite (relay, HIGH); digitalWrite (led, HIGH); Serial.println
("on");
}
else {
digitalWrite (relay, LOW); digitalWrite (led, LOW); Serial.println
("off");
}
y =! y; // alters or value of y, le and e equal to not
}
else {
Serial.println ("Invalid command");
}
```

```
}
}
```

Fade

This example demonstrates the use of the analogWrite() function to turn off a fade LED. AnalogWrite uses a PWM pulse, switching the digital pin on and off quickly, creating a fade effect.

What Will You Learn?

- Switching a faded LED on and off
- Intensify the concept of PWM

Previous Knowledge

- PWM
- AnalogWrite() function
- The polarity of an LED
- Increase and manipulate variables

Required Materials

- 1 Arduino Uno
- 1 LED
- 1 Resistor 330Ω
- 1 AB USB cable Jumpers
- 1 Protoboard

Source Code

```
/*
Fade
This example shows how to perform a fade on an LED on pin 9 using
the analogWrite() function.
This example is a public domain
* /
int led = 9; // LED pin
int brightness = 0; // LED brightness intensity
```

```
int fadeAmount = 5; // on how many points to apply the fade on the LED

void setup () {
// sets pin 9 as output:
pinMode (led, OUTPUT);
}

// the loop runs in sequence continuously: void loop () {
// sets the brightness of pin 9:
analogWrite (led, brightness);
// changes the brightness for the next loop: brightness = brightness + fadeAmount;

// reverses the direction of the fade at the end of the fade:
if (brightness == 0 || brightness == 255) {fadeAmount = -fadeAmount;
}
// wait 3
0 milliseconds to see the dimmer effect: delay (30);
}
```

Tips

Connect the most extended leg of the LED to the digital pin 9 of your Arduino via a

330Ω. Connect the shortest leg of the LED directly to the ground.

After defining that pin 9 will be your ledPin, nothing more should be done in the setup() function of the code.

The analogWrite() function that you will use in the main loop of the code requires two arguments: one telling the function which pin to trigger and the other indicating which PWM value to use.

To run the fade on the LED, gradually increase the PWM value from 0 (entirely off) to 255 (fully on) and then decrease again to 0 to complete the cycle. In the code below, the PWM value is set using a variable called brightness. Each time the loop rotates, it increases the value of the variable according to the fadeAmount.

If brightness is set between the extreme values (0 or 255), then fadeAmount changes to its negative. For example, if fadeAmount is 5, then it is set to -5. If it is -5, then it would be set to 5. The next time you rotate the loop, this change causes the brightness increment also to change direction.

analogWrite() can change the PWM value very quickly, so the delay at the end of the code controls the fade speed. Try to modify the delay value and see how this changes the program.

Loop

Many times you want to repeat an action on a series of pins and do something different for each one. In this case, the example flashes 6 LEDs using the for() loop function to circulate back and forth between pins 2 and 7. The LEDs turn on and off in sequence, using both the digitalWrite() and delay() functions.

We can call this example "Super Machine" by recalling the television series of the 80s in which the famous actor David Hasselhoff directed his Pontiac with artificial intelligence. The car was turbocharged with various LEDs of various possible sizes to reproduce bright effects.

We thought it would be interesting to use this metaphor of the "Super Machine" to learn more about sequential programming and good programming techniques for the I/O information on the board.

What Will You Learn?

- for() loop function
- digitalWrite()
- delay()

Required Materials

- 1 Arduino Uno6 LEDs
- 1 ABJumpers USB cable
- 6 Resistors 330Ω

Source Code

The code below starts using the for() loop function to designate digital pins 2 to 7 as outputs of the 6 LEDs used. In the main loop of the code, two for() loops are used to increase the loop by going through the LEDs, one by one, from pin 2 to pin 7. Once pin 7 is lit, the process reverses, going back through each LED.

For more information on the for() function, see page 82.

```
/ *
Loop

Demonstrates the use of the for () loop function.
It lights up several LEDs in sequence, and then the coating.

The circuit:
* LEDs between pins 2 to 7 and ground

Created in 2006 by David A. Mellis
Modified on August 30, 2011, by Tom Igoe

This code is a public domain.
http://www.arduino.cc/en/Lesson/ForLoop
* /
int timer = 100; // The higher the value, the slower the sequence of LEDs. void setup () {
// Use for loop to initialize each pin as output: for (int thisPin = 2; thisPin <8; thisPin ++) {
pinMode (thisPin, OUTPUT);
}
}

void loop () {
// loop from lowest to highest pin: for (int thisPin = 2; thisPin <8; thisPin ++) {
// Connect this pin: digitalWrite (thisPin, HIGH); delay (timer);
// turns off this pin: digitalWrite (thisPin, LOW);
}

// loop from highest to lowest pin: for (int thisPin = 7; thisPin> = 2; thisPin--) {
```

```
// Connect this pin: digitalWrite (thisPin, HIGH); delay (timer);
// turns off this pin: digitalWrite (thisPin, LOW);
}
}
```

LDR Sensor

In this lesson, we will use an LDR (Light Dependent Resistor) to simulate a light compensation of 5 levels, that is, depending on whether there is more or less light focusing on the sensor, the system turns on or off a series of LEDs.

This program could be used in a lighting system with five lines of light that

they light up as the sun goes down, progressively compensating for the lack of light.

In addition, a potentiometer adjusts the minimum critical light level from which the circuit will activate.

What Will You Learn?

- Serial reading of an analog sensor
- Using an analog reading
- Arduino AREF pin

Previous Knowledge

- DigitalWrite() function
- Conditional if/else

Required Materials

- 1 Arduino Uno
- 1 LDR
- 5 LEDs
- 1 AB USB
- 5 Resistors 330Ω

Source Code

```
/ *
```

LDR sensor
Connect an LDR to an analog input to control five outputs depending on ambient light.
* /
// Save the data collected by the LDR sensor: intLDR value = 0;
// Set the input pins of the LEDs:
int Ledpin1 = 12; int Ledpin2 = 11; int Ledpin3 = 10; int Ledpin4 = 9; int Ledpin5 = 8;

// Set LDR sensor input pin
int pinLDR = 0;

void setup ()
{
Serial.Begin (9600);

// Set the LED output pins: pinMode (Ledpin1, OUTPUT); pinMode (Ledpin2, OUTPUT); pinMode (Ledpin3, OUTPUT);

pinMode (Ledpin4, OUTPUT); pinMode (Ledpin5, OUTPUT);

// Defined the use of an external reference:
analogReference (EXTERNAL);

}
void loop ()
{
// Save the reading value of a variable: LDR value = analogRead (pinLDR); Serial.println (LDR value);

// Definition of the control pattern of the LEDs:
if (LDR value> = 1023)
{
digitalWrite (Ledpin1, LOW); digitalWrite (Ledpin2, LOW); digitalWrite (Ledpin3, LOW); digitalWrite (Ledpin4, LOW); digitalWrite (Ledpin5, LOW);
}
else if ((LDR value> = 823) & (LDR value <1023)))
{

```
digitalWrite (Ledpin1, HIGH); digitalWrite (Ledpin2, LOW);
digitalWrite (Ledpin3, LOW); digitalWrite (Ledpin4, LOW);
digitalWrite (Ledpin5, LOW);
}

else if ((LDR value> = 623) & (LDR value <823)))
{
digitalWrite (Ledpin1, HIGH); digitalWrite (Ledpin2, HIGH);
digitalWrite (Ledpin3, LOW);

digitalWrite (Ledpin4, LOW); digitalWrite (Ledpin5, LOW);
}
else if ((LDR value> = 423) & (LDR value <623)))
{
digitalWrite (Ledpin1, HIGH); digitalWrite (Ledpin2, HIGH);
digitalWrite (Ledpin3, HIGH); digitalWrite (Ledpin4, LOW);
digitalWrite (Ledpin5, LOW);
}
else if ((LDR value> = 223) & (LDR value <423)))
{
digitalWrite (Ledpin1, HIGH); digitalWrite (Ledpin2, HIGH);
digitalWrite (Ledpin3, HIGH); digitalWrite (Ledpin4, HIGH);
digitalWrite (Ledpin5, LOW);
}
else
{
digitalWrite (Ledpin1, HIGH); digitalWrite (Ledpin2, HIGH);
digitalWrite (Ledpin3, HIGH); digitalWrite (Ledpin4, HIGH);
digitalWrite (Ledpin5, HIGH);
}
}
```

Tips

When the Arduino receives an analog signal, it converts it to digital in 1024 parts. This operation is standard since Arduino thinks that the signal he will receive varies between 0v and 5v, which gives us a value for each part of approximately 4.88 mV.

But we can say no, that really the system will work between 0v and 3v, thus getting 1024 parts distributed between 0v and 3v, which gives us a value for each part of 2.9 mV, i.e., a much higher resolution. The distribution of these values we will divide equally in our program to make a progressive activation of the lighting lines.

If we set the reference too low, the LEDs start working with less ambient light than if we set a higher signal, remember:

More light = less resistance = more Volt less light = more resistance = less Volt less

This control will be done via a potentiometer, where we can calibrate the system through the ambient light.

pinMode(EXTERNAL);

With this instruction, we are telling our Arduino not to use the reference voltage (+5V) but to apply it through the AREF pin.

Thermistor

In this lesson, you have to use a Thermistor (Temperature Dependent Resistor) to do a temperature reading.

The result, in degrees Celsius, we will see through the Serial Monitor of Arduino's IDE.

What Will I Learn?

- Serial reading of an analog sensor (Thermistor)
- Using an analog reading
- Float variable

Previous Knowledge

- analogRead function
- Serial.print

Required Materials

- 1 Arduino Uno
- 1 Thermistor

- 1 Resistor 1KΩ
- 1 AB USB cable Jumpers
- 1 Protoboard

Source Code

Note that it is not an accurate thermometer, just an approximate example based on empirical data.
Connect 1k resistor from A0 to ground and + 5V thermistor to A0 * /

```
#define pin_termistor A0
void setup (void) {Serial.begin (9600);
}
void loop (void) {float reading; float reading1;
reading = analogRead (pin_termistor); Serial.print ("Read pin A0 =");
Serial.println (read);
reading1 = (reading * 0.2027) -82; Serial.print ("Temperature approx.
Celsius ="); Serial.println (reading1);
Serial.println (""); delay (2500);
}
```

Tips

There are two types of thermistors:

- NTC (Negative Temperature Coefficient) - thermistors whose coefficient of resistance variation with temperature is negative: resistance decreases with temperature increase.
- PTC (Positive Temperature Coefficient) - thermistors whose coefficient of resistance varies with the temperature is positive: the resistance increases with the increase of the temperature according to the characteristic curve/table of the thermistor, its resistance value can decrease or increase in a greater or lesser degree in a certain temperature range.

Thus, some can serve as overheat protection by limiting the electrical current when a certain temperature is exceeded. Another application is the measurement of temperature (in motors, for example) because we can, with

the thermistor, obtain a variation of electrical resistance in the function of the variation of temperature.

DC Motor

In this lesson, we will control a DC motor through Arduino. The push of the button will start our engine.

What Will I Learn?

- Digital one-button reading
- Controlling a DC motor with Arduino

Previous Knowledge

- DigitalWrite() function
- DigitalRead() function
- Conditional if/else

Required Materials

- 1 Arduino Uno
- 1 DC motor
- 1 Resistor 330Ω
- 1 Resistor 15Ω
- 1 AB USB cable

Source Code

```
// Connect motor on pin 2 in series with a 15 ohm resistor
// to limit the current to 40mA just not to overload the Arduino
const int motorPin = 2; const int buttonPin = 7; int buttonState = 0;

void setup () {pinMode (buttonPin, INPUT); pinMode (motorPin, OUTPUT);
}

void loop () {
buttonState = digitalRead (buttonPin); if (buttonState == HIGH) {
digitalWrite (motorPin, HIGH);
}
```

```
else {
digitalWrite (motorPin, LOW);
}
}
```

Tip

We can change the direction of rotation of a DC motor by simply reversing the direction of the current. With the same assembly of this lesson, the test reverses the connections of the motor and verify that it will start to rotate in the opposite direction.

LCD

The LCD is an important part of projects where you need to visualize the reading of a sensor or even to transmit information to the user.

In this exercise, you will learn how to connect the 2x16 LCD Display of your Kit, which already comes with the welded pins.

What Will You Learn?

- Connect your LCD to Arduino Uno
- Programming phrases to appear on the LCD
- Adjust the brightness of the display with a potentiometer
- Know the functions of the LiquidCrystal.h library
- Use the functions:
    - LCD.print
    - LCD.setCursor
    - scrollDisplayLeft()
    - scrollDisplayRight()

Required Materials

- 1 Arduino Uno
- 1 LCD display
- 1 Potentiometer
- 1 AB USB cable Jumpers
- 1 Protoboard

Source Code

```
/ *
LiquidCrystal Multilogic Code Library
This library works with all displays compatible with the
Hitachi HD44780 driver.
Circuit:
* LCD RS pin on digital pin 12
* LCD pin Enable on digital pin 11
* LCD pin D4 pin on digital pin 5
* LCD pin D5 pin on digital pin 4
* LCD pin D6 pin on digital pin 3
* LCD pin D7 pin on digital pin 2
* LCD pin R / W on ground
* 10K Trimpot:
+ 5V on + 5V
* Earth on earth
* wiper to LCD VO pin (pin 3)

Public domain code based on the original lesson:
http://www.arduino.cc/en/Lesson/LiquidCrystal
* /

// Includes library code: #include <LiquidCrystal.h>


// Initialize the library and define the pins used
LiquidCrystal lcd (12, 11, 5, 4, 3, 2);

void setup () {
// define the number of columns and lines of the display:
lcd begin (16, 2);
// Send the message to the display. lcd print ("Multilogic");
lcd setCursor (0.1); // Positions the cursor on the first column (0) and
on the second line (1) of the Display.
lcd print ("shop");
}

void loop () {
```

}

Tips

If your project needs more space to view information or a different LCD, get to know these other options:

- 2x40 LCD - white on blue
- 2x16 LCD display - RGB background

Exercise 1

The Arduino website offers several other projects with the LiquidCrystal.h Library. Here we will do one more exercise using the same assembly of this Lesson.

In this exercise, you can also modify the original text and control how long your text stays fixed and how long it scrolls to the right or left.

```
/ *
LiquidCrystal Library - scrollDisplayLeft () and scrollDisplayRight ()

LiquidCrystal Multilogic Code Library
This library works with all displays compatible with the
Hitachi HD44780 driver

This code writes "Multilogic Shop" on the LCD and uses scrollDisplayLeft () and scrollDisplayRight () to pass the text.

Circuit:
* LCD RS pin on digital pin 12
* LCD pin Enable on digital pin 11
* LCD pin D4 pin on digital pin 5
* LCD pin D5 pin on digital pin 4
* LCD pin D6 pin on digital pin 3
* LCD pin D7 pin on digital pin 2
* LCD pin R / W on ground
* 10K Trimpot:
+ 5V on + 5V
* Earth on earth
```

* wiper to LCD VO pin (pin 3)

Library originally added 18 Apr 2008 by David A. Mellis
library modified 5 Jul 2009 by Limor Fried (http://www.ladyada.net)
example added 9 Jul 2009 by Tom Igoe
modified 22 Nov 2010 by Tom Igoe

Public domain code based on the original lesson:
http://arduino.cc/en/Lesson/LiquidCrystalScroll
* /

// Includes library code:
#include <LiquidCrystal.h>

// Initialize the library and define the pins used
LiquidCrystal lcd (12, 11, 5, 4, 3, 2);

void setup () {
// define the number of columns and rows:
lcd begin (16, 2);
// Send the message to the display. lcd print ("Desired Text"); delay (2000);
}

void loop () {
// walks 16 positions for the text to exit the display on the left:
for (int positionCounter = 0; positionCounter <16; positionCounter ++)
{
// walk a position to the left: lcd. scrollDisplayLeft ();
// Wait a moment: delay (250);
}

// walks 32 positions for the text to exit the display on the right:
for (int positionCounter = 0; positionCounter <32; positionCounter ++)
{
// walk a position to the right: lcd. scrollDisplayRight ();

// Wait a moment: delay (250);
}

```
// walks 16 positions to the left to move back to the center: for (int positionCounter = 0; positionCounter <16; positionCounter ++) {
// walk a position to the left: lcd. scrollDisplayLeft ();
// Wait a moment: delay (250);
}

// delay at the end of the full loop:
delay (2000);

}
```

Typing into the LCD

Transmitting information or custom commands through the LCD can be useful and necessary in many projects or activities.

In this lesson, you will be able to control the text that appears on display through the Serial Monitor of the Arduino IDE in a fast and fun way.

What Will You Learn?

- Connect your LCD to Arduino Uno
- Customize phrases to appear in the LCD
- Using the Serial Monitor to enter texts

Required Materials

- 1 Arduino Uno
- 1 LCD display
- 1 Potentiometer
- 1 AB USB cable Jumpers
- 1 Protoboard

Source Code

```
/ *
This code sends a text entered via USB to the Circuit display:
* LCD RS pin on digital pin 12
* LCD pin R / W on pin 11
* LCD pin Enable on digital pin 10
* LCD pin D4 pin on digital pin 5
```

```
* LCD pin D5 pin on digital pin 4
* LCD pin D6 pin on digital pin 3
* LCD pin D7 pin on digital pin 2
* 10K Trimpot:
+ 5V on + 5V
* Earth on earth
* wiper to LCD VO pin (pin 3)
* /
#include <LiquidCrystal.h>
LiquidCrystal lcd (12, 11, 10, 5, 4, 3, 2); void setup ()
{
Serial. begin (9600); lcd begin (2, 20);
lcd clear (); lcd setCursor (0,0); lcd print ("Multilogic"); lcd setCursor
(0.1); lcd print ("shop");


}
void loop ()
{
if (Serial.available ())
{
// f reads through the serial monitor // char cr = Serial. read ();
// determine a character to clear the screen /// if (cr == '%')
{
lcd clear ();
}
// determine a character to jump to the bottom // else line if (cr = '>')
{
lcd.setCursor (0.1);
}
else
{
// if the typed character doesn't clear or skip line goes to display ///
// does not accept accent // lcd. write (cr);
}
}
}
```

Piezo as Analog Output

In this lesson, we will use a piezo as analog output. We use the possibility of processors to produce PWM signals to play music or emit sounds.

What Will I Learn?

- Connect your electric piezo to Arduino Uno.

Required Materials

- 1 Arduino Uno
- 1 Electric piezo
- 1 AB USB cable Jumpers
- 1 Protoboard

Tip

Do you want to amplify the sound of the piezo?

Place the back of the piezo (the whole golden part) on the bottom of a can of soda or chocolate can, and you will hear the loudest sound.

Use your creativity and test that other elements can amplify the sound of the piezo.

Scroll Bar with Processing

In this lesson, we will use Arduino's serial communication to interact with a Processing program running on the computer.

What Will You Learn?

From a graphical interface in Processing running on the computer, perform interactions with Arduino.

Previous Knowledge

Have the latest version of Processing installed and running on your computer.

Required Materials

- 1 Arduino Uno R3
- 5 LEDs

- 5 330Ω Resistor
- 1 Protoboard
- 1 USB - AB Cable and premium Jumpers. Or you can use the Arduino Uno R3 Starter Kit components.

Source Code

```
/ *
Scroll Bar with Processing
* /
char val; // variable to save the value received by the serial interface
char val_old; // variable to store the previous value received by the
serial interface

// setup function runs only once when program execution starts void
setup () {
// we initialize the pins from 2 to 6 as pinMode (2, OUTPUT) outputs;
pinMode (3, OUTPUT);

pinMode (4, OUTPUT); pinMode (5, OUTPUT); pinMode (6,
OUTPUT);

// we initialize the serial communication with a speed of 115200 baud
Serial. begin (115200);
}

// the loop function is executed indefinitely
void loop () {
if (Serial. available ()) {// checks if data is available for reading
val = Serial. read (); // read the data and store it in val
}
if (val! = val_old) {// acts only if there is a change in the value of val
alloff (); // perform the alloff function that erases all LEDs for (int i =
2; i <(int (val) +2); i ++) {
// causes the value of i to vary from 2 to val + 2
// note that val stores a character and we need
// of the int (val) function to convert to a digitalWrite (i, HIGH)
numerical value; // turns on the LED on pin i
}
```

```
val_old = val; // saves the value of val in val_old
}
}

void alloff () {// function to turn off all LEDs
for (int i = 2; i <7; i ++) {// causes i to vary from 2 to 6 digitalWrite (i,
LOW); // turns off the LED on pin i
}
}
```

Arduino - Thermostat

In this lesson, we will set up a simple experimental thermostat with Arduino. This mounting causes a relay and an LED to turn on when the temperature drops below a set minimum value or turn off when it exceeds a set maximum temperature.

Note that temperature measurement is empirical and should not be used as an accurate instrument.

What You Will learn?

- Use your Arduino to control devices from the ambient temperature reading.
- For this experiment, you can vary the temperature of the sensor just by holding it with your fingertips.
- Note that just as we are using a relay to light an LED, you could drive any other device (a motor, for example) within the relay's power range.

Necessary Materials

- 1 Arduino Uno R3
- 2 LEDs
- 1 Relay
- 1 Resistor 1kΩ
- 1 Resistor 300Ω
- 1 Thermistor
- 1 Protoboard

- 1 USB - AB Cable and Premium Jumpers. Or you can use the Arduino Uno R3 Starter Kit components.

Source Code

```
// Example of a simple experimental Arduino thermostat
// This setup makes the relay and a led
// turn on when the temperature drops below a minimum
// and turn off when it exceeds a maximum
// The temperature measurement is an empirical approximation
// Do not use an accurate instrument, but rather as a
didactic // thermostat model.

// relay connected to digital pin 2
// Led connected to digital pin 13
// resistive divider of thermistor connected to pin A0 analog

// declaration of variables:
#define pin_termistor A0
int rele = 2;
int led = 13;
float reading;
float read1;

// set acting mode:
void setup (void) {
  pinMode (relay, OUTPUT);
  pinMode (led, OUTPUT);
  Serial.Begin (9600);
}

// infinite loop:
void loop (void) {
  read = analogRead (pin_termistor);
  Serial.print ("Read pin A0 =");
  Serial.println (read);
  read1 = (read * 0.2027) -85; // Calculate the temp. approximate
  if (read1 <32) // set trigger temperature
    {
```

```
      digitalWrite (relay, HIGH); // relay
      digitalWrite (led, HIGH); // turn on Led
     }
  if (read1> 35) // set cutoff point
     {
      digitalWrite (relay, LOW); // turn off
      digitalWrite relay (led, LOW); // turn off Led
     }
  Serial.print ("Temperature approx. Celsius ="); // send and temp. for
serial monitor
  Serial.println (read1) ;
  Serial.println ("");
  delay (2500);
}
```

Chapter 6

If Statement with Arduino, Control the Execution of your Code

If conditional statements with Arduino are the most used in programming Maker projects. They serve to make your program do one thing or another, depending on each situation.

Imagine the loop () function of an Arduino code that is repeated indefinitely within our program. It is where things really happen. It repeats very fast, and the speed depends directly on the clock of the microcontroller CPU.

If we talk about Arduino UNO, its speed is 20 MHz, that is, 20,000,000 instructions per second. And on an ESP8266, the clock speed is 80 MHz (80,000,000 instructions per second).

Surely you are wondering if the same code that is inside the loop () function is repeated over and over again, why do we get different results?

Well, it is largely due to the conditional statements if with Arduino. In this lesson, you will learn the following concepts:

- Syntax of the if-else and else if statement
- Comparison operators (>,> =, <=, ==,! =)
- Boolean operators (||, &&,!)

And of course, you will learn how to use the if statement with Arduino through a case study.

Take the plate out of the box, and let's start with this chapter.

What will you learn in this chapter?

- if with Arduino necessary material
- Syntax of the if statement with Arduino
- Comparison operators in if statements with Arduino
- Example if statement with Arduino

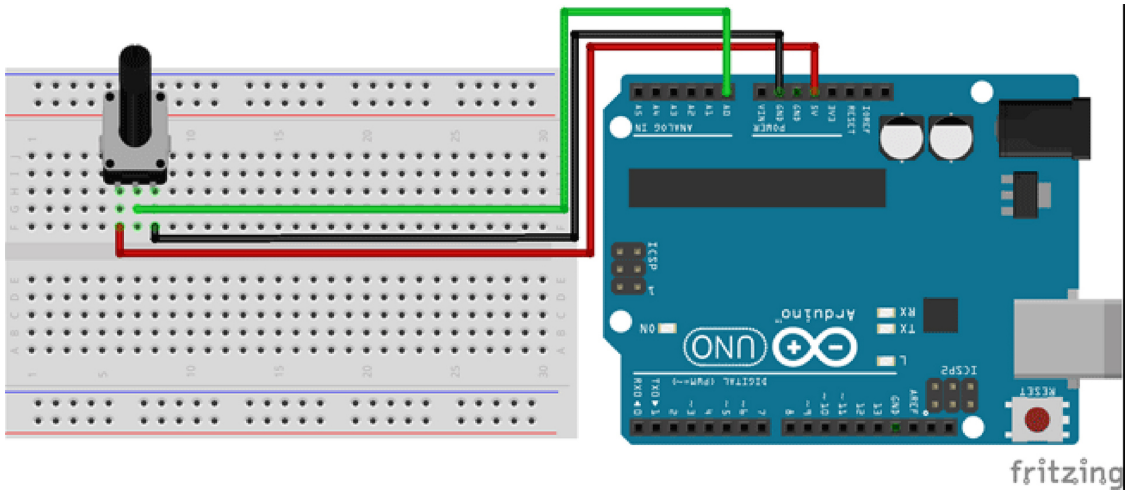
- Boolean operators if with Arduino
- if-else with Arduino
- Example if-else statement with Arduino
- else if with Arduino
- Example else if statement with Arduino

If with Arduino - Necessary Material

Although we will see all this later, in this lesson with the if-else and Arduino statements, you will need the following material.

- An Arduino UNO or derivative. Any compatible board can be worth even an ESP8266.
- A protoboard to connect the components.
- A potentiometer does not matter the value of the resistance.
- And cables to connect everything.

The circuit we will use is very simple. It only requires a potentiometer connected to an analog pin of the Arduino board. The electrical scheme would be as follows.



Let's now see the syntax that is hidden behind the if-else statement with Arduino.

Syntax of the if Statement with Arduino

The syntax of the if statement with Arduino is very simple. We begin by writing the reserved word if (it is translated as conditional). Then in brackets, we put the condition and finally open and close the keys.

```
if (Condition) {
// Everything we put here will be executed
  // only if the condition is met
}
```

The condition is what will cause the code between the keys to be executed. If it is true, the Arduino code flow will enter the keys and execute everything inside. But of course, what is a condition?

Imagine you tell your child, "If you pass all the subjects, I will give you a bicycle." That would be a condition. Within the parentheses will be the condition or conditional sentence. It is something that can be evaluated as true or false (in English, it is true or false).

If your child passes all subjects, it is true (true) and therefore has a bicycle. If your child does not pass all subjects is false (false) and therefore runs out of bicycle.

```
If you approve_all then
    buy_bikes ()
End yes
```

These conditions are called Boolean due to the great English mathematician George Boole, creator of Boolean logic. With this logic, there are only two possible states 0 or 1 that are equivalent to false (false) and true (true).

Ok, we are clear that an if statement with Arduino allows us to execute code depending on a condition. Now let's see how we can make conditional with the comparison operators.

Comparison Operators in if Statements with Arduino

Really, for something to be true or false, we need to compare it with something. Typically, the conditional statement if with Arduino with numbers is used, but other data can be used.

We can compare temperature, atmospheric pressure, ultraviolet radiation, time, intensity, voltage, water level, and any magnitude that can be measured.

Something that I think we all understand is temperature. For example, we can write a code where, depending on the temperature, a servo motor that activates a fan moves.

```
if (temperature > 25) {
    Servo.move (30);
}
```

In this case, we are saying that whenever the temperature is greater than 25, move a servomotor. But we could also have programmed if the temperature is less than 30 or if it is greater than or equal to 25.

More or less I think you're getting where I want to go

All of the above are comparisons. All we do is compare two values and evaluate whether it is true or false. To compare, comparison operators are used.

With the if statement with Arduino, you can use six comparison operators:

- Greater than
- <Less than
- = Greater than or equal to
- <= Less than or equal to
- == Equal to
- ! = Different from

I am sure that more than one comparison operator sounds to you. They are often used to express mathematical formulas.

The first 4 (greater than, less than, greater than or equal to or less than or equal to) are quite explicit. However, the last two (equal to and different from) require a little explanation.

Let's see some examples of each of them.

Comparison operator greater than (>)

As long as the number on the left is greater than the number on the right, the condition of the if statement with Arduino is true. Otherwise, it is false.

- 50> 10 -> true
- 15> 80 -> false

Comparison operator less than (<)

As long as the number on the left is less than the number on the right, the condition of the if statement with Arduino is true. Otherwise, it is false.

- 10 <50 -> true
- 80 <15 -> false

Comparison operator greater than or equal to (> =)

As long as the number on the left is greater than or equal to the number on the right, the condition of the if statement with Arduino is true. Otherwise, it is false.

- 50> = 50 -> true
- 50> = 49 -> true
- 15> = 90 -> false

Comparison operator less than or equal to (<=)

As long as the number on the left is less than or equal to the number on the right, the condition of the if statement with Arduino is true. Otherwise, it is false.

- 50 <= 50 -> true
- 49 <= 50 -> true
- 90 <= 15 -> false

Comparison operator equal to (==)

As long as the number on the left is the same as the number on the right, the condition of the if statement with Arduino is true. Otherwise, it is false.

- 20 == 20 -> true
- 15 == 20 -> false

Look at the operator symbol equal to (==), why do you use a double equal sign? Do you need to verify twice that they are equal? The answer is simpler than it seems.

The equal sign already has a function, that of the assignment. When we assign a value to a variable, for example. The double equal sign is used as an equality comparator so as not to confuse the compiler since otherwise, I would not know if we are comparing or assigning a value.

Comparison operator other than (! =)

As long as the number on the left is different than the number on the right, the condition of the if statement with Arduino is true. Otherwise, it is false.

- 20! = 10 -> true
- 10! = 10 -> false

This comparison operator is the opposite of equal to.

Although in the examples I am using numbers, in the conditions of the if statements with Arduino, you can also use variables as we will see in the examples that follow.

With this, we have everything we need to start using the if statement with Arduino. Let's see a very simple example.

Example if Statement with Arduino

In this example, we are going to use the if statement with Arduino using the electric scheme of the potentiometer that we have seen at the beginning of this chapter. The idea is to be able to show by the serial monitor when the analog value exceeds a value.

I remind you that the analog pin has a range of values that goes from 0 to 1023 in an Arduino UNO. When we have 0V at the input, analogRead returns 0. When we have 5V at the input, analogRead returns 1023.

The code must be able to display text on the serial monitor when it exceeds 300.

```
// Variable with analog pin
const byte Analogpin = A0;
```

```
void setup () {
   // Start serial communication
   Serial.begin (9600);
}

void loop () {
   // Read analog pin
   int value = analogRead (Analogpin);

   // Show analog pin value
   Serial.println (value);

   // If statement with Arduino
   if (value> 300) {
     // Show text by serial monitor
     Serial.println ("Value exceeded");
   }
   // 100 ms delay
   delay (100);
}
```

The code begins by declaring a constant variable of the type byte called Analogicpin. This variable contains the analog pin where the potentiometer is connected.

In the setup () function, we initiate serial communication with Serial.begin ().

In the loop () function, the first thing to do is read the analog pin with the analogRead function. The result is stored in the value variable of type int. This value is shown by the serial monitor with the Serial.println () function.

Then begins the statement of the if statement with Arduino. As we have seen, you put the reserved word if and then open and close parentheses. Within the parentheses goes the condition.

If we want to know if the value of the analog pin is greater than 300, we use the comparison operator greater than (>). On the left, you have the value you want to evaluate, and on the right, the threshold value that determines the condition.

Then open and close keys and inside is the code that will be executed if the if condition with Arduino is true. In this case, use the Serial.println () statement to display a text on the serial monitor (no matter what).

Finally, put a delay of 100 milliseconds so that we can appreciate it when you write on the serial monitor and when not.

Now, if you load the code to the board, you will see how the analog pin value appears, and whenever it is above 300, it will show the text Value exceeded on the serial monitor.

It has been simple, right? Try different comparison operators and see when the text appears on the serial monitor.

Now think about how you would do it if you want the code to run when it is between two values. For example, if the value is between 300 and 500, it shows text on the serial monitor.

With only the if statement with Arduino, you won't be able to do it. For this, you need Boolean operators.

Boolean Operators if with Arduino

A Boolean operator allows us to make combinations between conditions within an if in Arduino. This programming tool is very powerful and is used in many cases.

There are 3 Boolean operators

- || or (0)
- && and (y)
- ! not (no)

As I said, they serve to combine conditions. For example, you can know if the value obtained from analogRead is between 300 and 500.

It's something like when you press two keys on your keyboard. If you press the c key, write that letter in a document, for example. But if you press the control key (Ctrl) and the c key, what you do is copy what you have selected. More or less, it is something similar.

We are going to see each of these Boolean operators with examples so that you understand it better.

Boolean Operator or (||)

It can be translated as "or this or that." It is represented with two vertical lines (||) that you find on key 1 of the keyboard. To use it, you must first press Alt Gr and then the 1 key.

This operator makes the if condition with Arduino true if either of the two or more conditions are met. For example, if we have this code:

```
if (value> 300 II value <200) {
    // Show text by serial monitor
    Serial.println ("Value exceeded");
}
```

We are saying that whenever the value is greater than 300 or that the value is less than 200, display a text on the serial monitor. This means that whenever the value is between 201 and 299, it will not show any text.

In the rest of the cases, it will show the text on the serial monitor.

Boolean Operator and (&&)

It can be translated by "this and that." It is represented by two ampersands (&&) symbols found on key 6 of the keyboard. Pressing capital letters and 6, you can use it in your codes.

This operator makes the if condition with Arduino true only if both or several conditions are met. For example, if we have the code:

```
if (value> 300 && value <500) {
    // Show text by serial monitor
      Serial.println ("Value exceeded");
}
```

It means that it will only be true if the value is greater than 300, and the value is less than 500. This gives us a range of values between 301 and 499 (don't forget that the symbols> and <exclude the numbers on the right).

Therefore, only when you are in that range will you show the text on the serial monitor.

Boolean Operator not (!)

It translates as "no." It is represented by the final exclamation mark (!) Located on the 1 key. You can use it by pressing the uppercase key plus the 1 key.

What this operator does is something strange. If an expression is true, it returns false, and if it is false, it returns true.

To understand the following example, you also have to understand what is true and what is false. They are still two states, like the binary code of which I have spoken to you before. We can have a 0 (false) or a 1 (true).

But the reality is that there are more numbers. The question is, is 2 true or false? Boolean data types are always false as long as it is 0. Otherwise, it is true. Therefore 2 and any other numbers are true if evaluated in a condition. This is only a reminder.

In a digital pin, we also have two values, LOW and HIGH. If you enter the Arduino core code, you can verify that these two reserved words are actually two constants whose value is 0 for LOW and 1 for HIGH.

Therefore, if we have a digital pin connected to a button, and the input is 0V, the status will be LOW. If we store that value in a variable, we could evaluate its value if it is false or true in an if conditional statement with Arduino.

For example, it can help us to know if a button is not pressed. In normal use, when pressed, it will have a HIGH state that, as we have seen, is true. While when not pressed, it has a LOW status, false.

```
boolean button = digitalRead (Buttonpin),
// If the button is false (equal to 0) everything is true
if (! button1) {
    // Turn on the LED
    digitalWrite (led Warning, HIGH);
}
```

This would be the way we would detect if a button is not pressed. In this case, it will pass inside the if with Arduino and execute the code inside.

Other Considerations of Boolean Operators with an Arduino if

Finally, I want to add that you can join as many conditions as we want to an if with Arduino. It is convenient, as far as possible, not to add many conditionals as this would make our code unreadable.

Also, note the use of the not (!) Operator with other comparison operators. For example, the same expression can be expressed in different ways thanks to the not operator. If you want to know if a number is between 300 and 500, you can use these two conditions:

- (number> = 300 && number <= 500)
- ! (number <300 && number> 500)

These two expressions return the same value for the same number.

Now we will continue to see how you can execute different pieces of code depending on different conditions with if-else in Arduino.

if-else with Arduino

So far, what we have seen is that if it meets a condition, it executes the code inside an if with Arduino. But what if we have more than one condition? For example, if the temperature is greater than 25, it moves the servo motor 135º if it is less than 25, it moves the servo motor 45º.

With what we know so far, we could do two if with Arduino.

```
if (temperature> 25) {
    Servo.move (135);
}
if (temperature <= 25) {
    Servo.move (45);
}
```

It would not be bad, but there is a more optimal way to do it, using the else conditional statement with Arduino.

Else can be translated as "yes no." It is always linked to a conditional statement if Arduino can never go alone. Therefore if-else would be translated as if the condition is met, execute the code if you do not execute this other code.

It is logical to think that when we are evaluating the same variable, it cannot have two values. In this case, the temperature is either greater than 25 or less than or equal to 25, and there is no other.

The biggest advantage of using if-else with Arduino is that we will make our code more efficient. In the first case (with two if), the two conditions will always be evaluated even if one of them is true.

If, for example, the temperature is 27º when it reaches the first if, since the temperature is greater than 25, it will execute the code inside. When finished, it will exit the if and go to the next if. Check if the temperature is less than or equal to 25, something absurd if nothing has changed within the first if.

This is where, if-else, comes into play.

```
if (temperature> 25) {
    Servo.move (135);
} else {
    Servo.move (45);
}
```

The second if is replaced with the condition by an else. This conditional statement does not have any condition. That is, if the previous condition or conditions are not met, it will always enter through the else.

The advantage is that if the first condition is met, you will stop checking the rest of the conditions. This programming structure is called nested if.

Be very careful with these types of structures since they can lead us to what is known as the spaghetti code. It is such an obfuscated code that it looks like a plate of spaghetti.

Let's see it with an example.

Example if-else statement with Arduino

We will continue with the same electric scheme of the potentiometer. The idea with this practical exercise is to turn on the LED integrated with the Arduino board (it is connected to pin 13) when the analog value exceeds 500. Otherwise, turn off the LED.

The code would be as follows.

```
// Variable with analog pin
const byte Analogicpin = A0;
// Variable LED pin
const byte Ledpin = 13;

void setup () {
  // Start serial communication
  Serial.begin (9600);

  // Pin mode
  pinMode (Ledpin, OUTPUT);
}

void loop () {
  // Read analog pin
  int value = analogRead (Analogicpin);

  // Show analog pin value
  Serial.println (value);

  // If statement with Arduino
  if (value> 500) {
    // Turn on LED
    digitalWrite (Ledpin, HIGH);
  } else {
    // Turn off LED
    digitalWrite (Ledpin, LOW);
  }

  // 100 ms delay
  delay (100);
}
```

The first part of the code is the declaration of variables. We need two to store the number of pins. They are constants of the byte type.

The Analogicpin variable stores the analog pin where the potentiometer is connected, pin A0.

The Ledpin variable stores the digital pin where the LED is connected, pin 13.

In the setup () function, we started the serial communication with Serial.begin () and put the LED pin in OUTPUT mode.

Within the loop () function, we read the analog pin with analogRead () and store it in the variable of the int value type. Then we show it by the serial monitor.

On the next line begins the conditional statement if-else with Arduino. As always, we put the word if and in brackets of the condition. In this case, it is if value (it is the value obtained from the analog pin) is greater than 500, then it enters inside the if.

Between braces, we put the code that we want to be executed when the condition is met. In this case, the LED is lit with the digitalWrite () statement.

Then we put the else and between braces the code that we want to be executed. In this case, the LED goes out with the digitalWrite () statement.

Finally, a delay of 100 ms so that we can follow what is shown by the serial monitor.

If you load the code to the board and play with the potentiometer, you will see how it turns on and off when you go over 500 or lower than 500.

The last thing that remains to be seen is if we want to have three or more conditional sentences if-else nested. We can do that with else-if, another way of writing an if with Arduino.

else if with Arduino

The else if conditional statement with Arduino will allow us to have two or more conditions within a nested if block.

The else conditional statement is very useful in cases where we only have two possibilities, but what happens if, for example, you want the LED to flash when the value of the potentiometer is greater than 800. When it is

between 501 and 800, it stays on, and when it is less than or equal to 500, it turns off.

We can choose to put everything full of if without nesting, i.e., separate if statements. We have already seen that this is not a good idea when we are working with the same variable.

The else statement falls short because it does not allow a condition to be set.

But there is the else if statement which, as with the else statement, is always linked to an if with Arduino. It could be translated as if you have not fulfilled the above, check if this other is fulfilled.

It really is as if it were another if, but by putting the reserved word else in front, we are joining all the if. The syntax is very similar.

```
Else if (condition) {
    // code to execute
}
```

First, we put the else, then if and in brackets the condition. Between the keys, the code we want to be executed when the condition is met.

For example, this would be the code if we want to control 3 ranges on an analog pin.

```
// If statement with Arduino
if (value> 800) {
    // Flashing LED
    digitalWrite (Ledpin, HIGH);
    delay (500);
    digitalWrite (Ledpin, LOW);
} else if (value <= 800 && value> 500) {
    // Turn on LED
    digitalWrite (Ledpin, HIGH);
} else {
    // Turn off LED
    digitalWrite (Ledpin, LOW);
}
```

The first if checks if the value of the analog pin is greater than 800. Since it only has the symbol greater than, it will start at 801. From 801 to 1023 it will execute the code inside, that is, it will flash the LED.

The second condition begins with else if and then in brackets of the condition. Whenever the analog value is less than or equal to 800 or greater than 500 (will start at 501), it will turn on the LED and leave it fixed.

Finally, if the two previous conditions are not met, it will enter the else and turn off the LED.

I repeat that when several if they are nested if one of them meets the condition, the rest of the if will not be executed. This will make our code more efficient.

Example else if statement with Arduino

To finish, we go with an example. You just have to replace the if we have seen in the previous example with this series of nested if.

```
// If statement with Arduino
   if (value> 800) {
     // Flashing LED
     digitalWrite (Ledpin, HIGH);
     delay (500);
     digitalWrite (Ledpin, LOW);
   } else if (value <= 800 && value> 500) {
     // Turn on LED
     digitalWrite (Ledpin, HIGH);
   } else {
     // Turn off LED
     digitalWrite (Ledpin, LOW);
   }
```

The complete code would be as follows.

```
// Variable with analog pin
const byte Analogicpin = A0;
// Variable LED pin
const byte Ledpin = 13;
```

```
void setup () {
   // Start serial communication
   Serial.begin (9600);

   // Pin mode
   pinMode (Ledpin, OUTPUT);
}

void loop () {
   // Read analog pin
   int value = analogRead (Analogicpin);

   // Motivate analog pin value
   Serial.println (value);

   // If statement with Arduino
   if (value> 800) {
     // Flashing LED
     digitalWrite (Ledpin, HIGH);
     delay (500);
     digitalWrite (Ledpin, LOW) ;
   } else if (value <= 800 && value> 500) {
     // Turn on LED
     digitalWrite (Ledpin, HIGH);
   } else {
     // Turn off LED
     digitalWrite (Ledpin, LOW);
   }

   // 100 ms delay
   delay (100);
}
```

Try charging and playing with the potentiometer. You will see how it goes from being off to on and finally flashing.

In this chapter, we have seen one of the most powerful tools that programming languages have, the else if-else statements with Arduino. You have been able to verify that, although the loop () function is repeated over

and over again very quickly and indefinitely, we can change the result using the if-else conditional statements with Arduino.

It is another tool in programming, but it is not the only one. This can help you control the temperature or water level with Arduino. In addition, it works in the same way on either another Arduino board or on an ESP8266. Moreover, in all programming languages, there are the if, if-else, and else if statements. By mastering the conditional sentences in Arduino, you will master the execution of your program.

Chapter 7

Battery and Battery Charge Meter with Arduino

In this chapter, I will explain how we can make a battery and battery charge meter. We will do it through the assembly of a circuit with Arduino.

It is usually very typical that we have at home different electrical devices that use batteries. Sometimes, we doubt whether the device is broken or the batteries have run out.

Thanks to this project, this will no longer be a problem. The battery and battery charge meter will give you the solution.

What will you learn in this chapter?

- The objective of the battery and battery charge meter with the Arduino board
- Arduino components that we will use
- Riding the circuit with Arduino
- Programming the battery and battery charge meter with Arduino
- Arduino native code

The Objective of the Battery and Battery Charge Meter with the Arduino Board

We will use Arduino to read the voltage supplied by a battery through an analog input. Depending on this voltage, we will light an LED of one color. If the battery is new, a green LED will light. If the battery is not new, but part of its energy has been consumed, we will light a yellow LED. Finally, if the battery is worn out or does not supply enough voltage, we will turn on a red LED.

We must be very careful with the type of battery and battery we are going to measure. It is very dangerous to supply more than 5V to the analog pins of

Arduino. If what we want is to measure batteries, the most typical, we can only do it with AA, AAA, C, and D batteries. They are the ones used in television controls, toys, and even to feed Arduino.

Much eye with a 9V battery, the squares. As I said before, these batteries far exceed the 5V limit. We must also be careful with the batteries as it will depend on the voltage they supply. Check before connecting that it really is less than or equal to 5V.

Measuring more than 5V for some analog input can damage the board.

Arduino Components that We Will Use

Let's see what kind of components we need for this circuit.

- Arduino UNO or any Arduino board
- Protoboard where we will connect the components
- Cables for connection between components and board
- 3 220 Ω resistors
- 1 10 kΩ resistor
- 1 red 5mm LED
- 1 yellow 5 mm LED
- 1 5mm green LED

As you can see, it is a very simple circuit. With three LEDs and four resistors, it is enough to build a battery or battery charge meter with Arduino.

Riding the Circuit with Arduino

Once all the information is collected, we go to the assembly. In the following image, I show you how you should connect the different components. Pay attention, especially in the resistances.

Let's see how the components have been connected. The first is the LEDs. Each one is connected in series with a resistance of 220 Ω to extend their useful life. The green LED is connected to pin 2, the yellow LED is connected to pin 3, and the red LED is connected to pin 4. This is important to remember when we see the programming.

To measure the battery, have placed a resistance pull-down. What this type of resistance does is maintain a low logical state, that is, at 0V. It is important to use this type of resistor since, and when we do not have the battery or battery connected to measure, we have an undetermined state at the analog pin input, which causes it to oscillate and maybe until some LED is lit. You can try to remove this resistance, and you will see the result.

The positive pole of the battery is connected to the pull-down resistor and to the analog input A0. The other end of the earth's resistance. Finally, the negative pole of the battery must be connected to the Arduino earth.

Beware of reversing polarities that is, positive with negative or vice versa.

It is more than advisable that all components have the same reference to ground. All must be connected to the same Arduino GND pin.

Programming the Battery and Battery Charge Meter with Arduino

Now play the logic and programming part. The first thing to do is to raise the problem or algorithm we want to achieve. Once we are clear, we can start programming. As I always say, an algorithm is a sequence of ordered steps that we must follow to achieve an objective. In this case, our goal is to measure the charge of a battery or battery with Arduino.

Algorithm

In this section, I will detail the steps that we must follow without writing a line of code; we will do that later when we are clear about what we have to do.

1. Read the analog pin where the battery is connected
2. We calculate the voltage for the value you have given us
3. We evaluate the voltage

    a) If it is greater than or equal to the maximum threshold
        I. We turn the green LED

    b) If it is less than the maximum threshold and greater than the average threshold

I. We light a yellow LED

c) If it is less than the average threshold and greater than the minimum threshold
   I. We turn on the red LED

d) The rest of the cases
   I. Does not light any LED

4. We turn off all LEDs

Analyzing the algorithm that we are going to implement, we conclude that we are going to use three thresholds:

- The maximum threshold: will indicate that the battery is fully charged.
- Average threshold: from this threshold to the maximum threshold, the battery has been used but still has power.
- Minimum threshold: from this threshold to the average threshold, the battery does not supply enough power. Below this threshold, we interpret that there is no battery connected.

Arduino Native Code

We are going to see the code of the battery charge meter with Arduino in parts.

Variables and Constants

We will use the constants to store the pins where we will connect the LEDs and the analog pin where we will connect the battery. The thresholds will be variable, although we could also use constants. Finally, we will declare three variables to store the value returned by the analog pin, the voltage on that pin, and the waiting time for the LEDs to flash.

We declare the constants with the pins where we connect the LEDs and the analog pin.

```
// Pins for the LEDs
#define LEDGREEN 2
```

```
#define LEDYELLOW 3
#define LEDRED 4
#define ANALOGBATTERY 0
```

The next thing is to declare the variables and thresholds. The latter will depend on the type of battery. In this case, I am going to do the tests with an AA battery

```
// Variables
int analogValue = 0;
float voltage = 0;
int ledDelay = 800;

// Thresholds
maximum float = 1.6;
medium float = 1.4;
minimum float = 0.3;
```

Setup Function

In the setup function, we initialize the serial monitor and put the LED pins in output mode.

```
void setup () {
   // We start the serial monitor
   Serial.begin (9600) ;

   // LED pins in output mode
   pinMode (GREENLED, OUTPUT);
   pinMode (YELLOWLED, OUTPUT);
   pinMode (REDLED, OUTPUT);

}
```

Loop Function

We begin the loop function that will be repeated continuously. The first thing is to read the analog pin and store it in the variable Analog Value.

```
void loop () {
   // We read analog input value
   analogValue = analogRead (ANALOGBATTERY);
```

We calculate the voltage. It is a simple rule of 3. If 5V is 1024, with five dividing by 1024 and multiplying it by the value given by the analog pin, we already have the voltage. As simple as that. To verify that everything is fine, we show it by the serial monitor.

```
// We get the voltage
   voltage = 0.0048 * analogValue;
   Serial.print ("Voltage:");
   Serial.println (voltage);
```

In the next part we will decide which LEDs we should turn on. We do this through the conditional structures if.

```
// Depending on the voltage we show an LED or another
   if (voltage> = maximum)
   {
     digitalWrite (GREENLED, HIGH);
     delay (ledDelay);
     digitalWrite (GREENLED, LOW);
   }
   else if (voltage <maximum && voltage> medium )
   {
     digitalWrite (YELLOWLED, HIGH);
     delay (ledDelay);
     digitalWrite (YELLOWLED, LOW);
   }
   else if (voltage <medium && voltage> minimum)
   {
     digitalWrite (REDLED, HIGH);
     delay (ledDelay);
     digitalWrite (REDLED, LOW);
   }
```

Finally, we turn off all the LEDs. Thus we begin in the next iteration with the initial state all turned off. As we are talking about very little time, microseconds, this will not affect the operation.

```
// We turn off all LEDs
   digitalWrite (GREENLED, LOW);
```

```
    digitalWrite (YELLOWLED, LOW);
    digitalWrite (REDLED, LOW);
}
```

Full Code

Then I leave the complete code, so you do not have to copy part by part. It is not very good practice to copy and paste without reading the above. If you really want to learn, follow all the steps, and try to write the code yourself. Only then will you learn to program with Arduino?

```
// Pins for the LEDs
#define LED GREEN 2
#define LED YELLOW 3
#define LED RED 4
#define ANALOGBATTERY 0

// Variables
int analogValue = 0;
float voltage = 0;
int ledDelay = 800;

// Thresholds
maximum float = 1.6;
medium float = 1.4;
minimum float = 0.3;

void setup () {
  // We start the serial monitor
  Serial.begin (9600);

  // LED pins in output mode
  pinMode (GREENLED, OUTPUT);
  pinMode (YELLOWLED, OUTPUT);
  pinMode (REDLED, OUTPUT);

}
void loop () {
  // We read analog input value
  analogValue = analogRead (ANALOGBATTERY);
```

```
  // We get the voltage
  voltage = 0.0048 * analogValue;
  Serial.print ("Voltage:");
  Serial.println (voltage);

  // Depending on the voltage we show an LED or another
  if (voltage> = maximum)
  {
    digitalWrite (GREEN LED, HIGH);
    delay (ledDelay);
    digitalWrite (GREEN LED, LOW) ;
  }
  else if (voltage <maximum && voltage> medium)
  {
    digitalWrite (YELLOWLED, HIGH);
    delay (ledDelay);
    digitalWrite (YELLOWLED, LOW);
  }
  else if (voltage <medium && voltage> minimum)
  {
    digitalWrite (REDLED, HIGH);
    delay (ledDelay);
    digitalWrite (REDLED, LOW);
  }

  // We turn off all LEDs
  digitalWrite (GREENLED, LOW);
  digitalWrite (YELLOWLED, LOW);
  digitalWrite (REDLED, LOW);
}
```

Chapter 8

ADS1115 ADC Digital-Analog Converter for Arduino and ESP8266

The ADS1115 is a 16-bit ADC digital-analog converter that can be very useful for certain projects with Arduino and ESP8266. Although the Arduino boards usually have some built-in ADC digital-analog converter, the ADS1115 has more features and more resolution.

For example, the Arduino UNO incorporates six 10-bit ADCs; however, the ADS1115 has a 16-bit resolution. On the other hand, by using an external ADC digital-analog converter, you will be able to free the processor.

In the case of ESP8266, it only incorporates a 10-bit digital-analog converter that measures in the range of 0V and 1V. This makes the conversion depends on the resistances that are used to make the voltage divider.

Using the ADS1115, you will gain a lot in accuracy and conversion quality. Communication between the external digital-analog converter ADS1115 and Arduino or ESP8266 is through I2C.

All this is what you will see in this lesson, but first, let me show you what analog, digital, and the digital-analog converter is.

What will you learn in this chapter?

- What is a digital-analog converter?
- What is an ADS1115?
- Install library for ADS1115
- Single-end mode
- Differential mode
- Comparator mode

What is a Digital-Analog Converter?

The digital-analog converter allows you to convert from the analog world to the digital world. All the signals detected and processed by the human being are analog signals. The objective of converting an analog signal into a digital signal is that it can be treated and processed by a microcontroller for various purposes.

Machines only understand 1s and 0s. Televisions, computers, tablets, or mobiles, all have at least one digital-analog converter. There are different types of the digital-analog converter (ADC) that are used for different purposes. The three most common types are:

- Flash
- Successive approximations
- Sigma-delta

The most common and most economical digital-analog converter (ADC) is that of successive approximations. The two most important characteristics of a digital-analog converter (ADC) are the resolution and the sampling rate or frequency.

Resolution of a Digital-Analog Converter

The resolution of a digital-analog converter (ADC) is expressed in the number of bits. Sets the number of levels into which an analog input range can be divided. To calculate the resolution of an n-bit digital-analog converter (ADC), divide 1 by 2 raised to n.

$$\text{resolution} = \frac{1}{2^n}$$

For example, a 16-bit digital-analog converter (ADC) has a resolution of

$$\text{resolution} = \frac{1}{2^{16}} = \frac{1}{65536} = 0.0000152$$

If the input range is 5V,

$$\frac{5v}{2^{16}} = \frac{5}{65536} = 0.00007629V = 0.07629mV$$

What comes to say that we can measure variations of 0.0763mV in the analog input signal?

Variations below this voltage will not be detected.

In general, when a digital-analog converter has a higher resolution, it provides more details of the analog signal than a digital-analog converter with a lower resolution.

Is it better to have an ADC with a higher resolution?

It depends. If you are only interested in temperature variations from grade to grade at best with the Arduino digital-analog converter itself is more than enough.

Whenever possible, avoid adding layers that make the project more complex and expensive without any compelling reason.

Speed of a digital-analog converter

The other characters in which we must look at a digital-analog converter (ADC) is the speed or frequency of sampling. It is the number of times the digital-analog converter (ADC) samples the analog signal in 1 second. Its unit is expressed in hertz (Hz).

For example, sampling frequencies of 44 kHz, 22 kHz, and 11 kHz are mainly used for audio signals. If the first one is used, 44 kHz means that the digital-analog converter samples the signal 44000 times per second. The higher the sampling frequency, the better the conversion from analog to digital.

What is an ADS1115?

Now that we are clear about what a digital-analog converter is let's move on to see the ADS1115. The ADS1115 is an ideal external digital-analog converter when more resolution or more analog pins are required.

A typical case is when you want to measure different analog signals with an ESP8266. This SoC only incorporates an ADC. In these cases, it is essential to use an external digital-analog converter (ADC) such as the ADS1115.

As with many components of this type, the ADS1115 is the chip that names the electronic component. You will find it in different shapes and formats, according to the manufacturer.

One of the most famous is that of Adafruit, although you can find other similar ones from other brands. I am going to work with the GY-ADS1115 / ADS1015, whose price is about $ 5.

Everything you see in this lesson will also be useful for any other ADS1115, no matter the manufacturer. The most normal thing is to come with the solderless pins. So prepare the soldering iron and tin because you have to weld.

Next, we will see the main features of the ADS1115.

Main Features of the ADS1115

The first thing is to always go to the technical data sheet, where you will get the best and most reliable information on an electronic component.

On the website of the chip manufacturer, Texas Instruments, you will find it here.

Characteristic	**Value**
Operating voltage	from 2V to 5.5V
Current consumption	150 µA (continuous mode)
Programmable Sampling Rate	8 Hz to 860 Hz
Resolution	16-bit
Channels	4 input channels or 2 differentials
Communication interface	I2C (4 addresses)

We will be explaining all these features of the ADS1115 throughout this lesson.

Before we go to see the pins.

ADS1115 Pins

The number of pins depends a lot on the component, but, normally, we have access to 10 pins.

- The VDD and GND pins are the power pins.
- The SCL and SDA pins are the I2C clock and data pins.
- Pins A0, A1, A2, and A3 are the four analog input pins.
- The ADDR and ALERT pins are special, and we will see them in more detail.

ADDR Pin

The ADS1115 digital-analog converter can be configured with 4 I2C addresses. This means that you can connect up to 4 ADS1115 on the same I2C bus.

The ADDR pin is used to select an address from the four possible ones. Depending on where you connect, you will use one address or another.

The possible I2C addresses are 0x48, 0x49, 0x4A and 0x4B.

In the following table, I show the possible connections with their corresponding addresses.

ADDR connection	**Direction**
GND	0x48
5V	0x49
SDA	0x4A
SCL	0x4B

What this table means is that the ADDR pin has to be connected to the GND, 5V, SDA or SCL pin to get the corresponding address.

This type of addressing allows up to 4 ADS1115 modules to be connected to the same board.

ALERT Pin

The ADS1115 digital-analog converter is equipped with a customizable comparator that issues an alert on the ALERT / RDY pin. This feature is very useful since it can significantly reduce external circuits in many projects.

Imagine you have a sensor to detect floods. Thanks to a comparator like the ADS1115 you could do that when it exceeds the threshold, on the ALERT pin it emits a signal and wakes the microcontroller. This allows device consumption to be very low.

It can work in two ways.

In traditional comparator mode, the ALERT pin is activated when the signal exceeds a set high threshold (TH_H) and is deactivated when the threshold drops below the low threshold (TH_L).

In the window comparator mode, lets you know if the signal is between the limits of a window set by the high threshold (TH_H) and the low threshold (TH_L).

This means that it is activated when the high threshold is exceeded or when they fall below the low threshold.

These two modes can be configured at the software level.

ADS1115 Conversion Modes

The ADS1115 digital-analog converter can operate in two conversion modes: single-shot mode and continuous conversion mode.

In single-shot mode, the digital-analog converter (ADC) converts the input signal to demand and stores the result in an internal register.

By sending a signal to ADS1115, we can indicate when to convert.

Then the ADS1115 enters the energy-saving mode. This mode is used to save energy on systems that only require periodic conversions, or there are long periods of inactivity between conversions.

In continuous conversion mode, the digital-analog converter (ADC) automatically begins converting the input signal as soon as the previous

conversion is complete. That is, convert a value from analog to digital and when it finishes, take another sample. At what velocity?

The conversion rate, as we have seen before, is programmable.

The data can be read at any time and always reflect the most recent conversion.

Resolution of the ADS1115 Digital-Analog Converter

Although I have already commented that the resolution of the ADS1115 is 16-bit, the reality is that not all 16-bit bits are used to express the voltage value. The output you get from the ADS1115 is known as a signed integer, that is, one of the bits of the 16-bit word is used to set the sign, positive or negative.

Therefore, only 15-bit of the 16-bit is used, which means that there are 32,768 possible values ($\mathbf{2}^{\mathbf{15}}$).

The first value would be 0, and the last would be 32,767.

Programmable Gain Amplifier or PGA

Another of the peculiarities that give the ADS1115 even greater precision and reliability is the programmable gain amplifier or PGA that is incorporated.

Ok, but what is a programmable gain amplifier?

Basically, it is an operational amplifier to which we can modify the gain through the code, programming. The programmable gain amplifier sets the full scale, that is, indicates the reference value.

In Arduino, this value is determined by the reference voltage, which in the case of Arduino UNO is 5V and in the Arduino MKR range, and the ESP8266 is 3V3.

In ADS1115, it is established by the PGA. By default, this reference value is ± 6,144 V. This means that the value of 32,677 (maximum value with 15-bit) corresponds to 6,144 V.

In Arduino UNO, with a 10-bit ADC (1024 possible values), the value 1023 corresponds to 5 V.

If you do the conversion, you have to for the ADS1115

$$\text{scalefactor} = \frac{6.144\text{V}}{32.767} = 0.0001875\text{V} = 0.0001875\text{mV}$$

And for an Arduino UNO

$$\text{scalefactor} = \frac{5\text{V}}{1.023} = 0.0048875\text{V} = 0.0048875\text{mV}$$

This means that with the ADS1115, a significant improvement is achieved with respect to the ADC of the Arduino UNO, around 26 times better.

But the best thing is that the gain amplifier is programmable, and we can change the reference value to smaller values. The value of 6.144 V is the worst case.

In the following table, you have a summary of all possible PGA values.

PGA	**Reference (V)**	**Scale factor**
2/3	6.144 V	0.1875 mV
1	4,096 V	0.1250 mV
two	2,048 V	0.0625 mV
4	1,024 V	0.0312 mV
8	0.512 V	0.0156 mV
16	0.256 V	0.0078 mV

Note that by setting the PGA with the value 16, a scale factor of 0.0078 mV is achieved, that is, very small voltage variations can be detected.

But be careful, the reference in volts of the previous table does not mean that we can measure signals in that range of values.

The Maximum Range of Analog Inputs of ADS1115

The above table can be very misleading as it can lead to error, although the largest reference is 6.1444 V. The ADS1115 can only measure the power value (VDD) plus 0.3 V for its analog pins.

What does this mean? VDD is the supply voltage, and as we have seen, it can be between 2V and 5.5V.

Therefore, if we supply the ADS1115 with 5V, we can only measure voltages on the analog pins up to 5V + 0.3V, that is, 5.3V.

The same will happen if we feed with 3.3V. In this case, the maximum would be 3.6V.

Applying a voltage greater than VDD + 0.3V to an ADS1115 may damage the chip. Never exceed this voltage on an analog pin.

I repeat that the PGA reference is not the same (it is the scale factor) and the maximum value that the analog inputs support.

The last thing we are going to see before seeing the examples are the operating modes: single-ended mode, differential mode, and comparator mode.

ADS1115 Operating Modes

The single-ended mode makes the four analog inputs of the ADS1115 work like those of an Arduino. The digital-analog converter (ADC) reads the difference between each input and ground.

The differential mode, however, measures the difference between two of the analog inputs of the ADS1115. That is, a measurement is made in which neither side is in GND.

In this mode, the pins are paired two by two, A0 with A1 and A2 with A3. This means that you will only have two channels.

In channel 1, returns the difference between pins A0 and A1, and channel 2 returns the difference between pins A2 and A3.

In the comparator mode, the ALERT pin that we have seen before is used to launch a signal every time a threshold set within the ADS1115 is exceeded.

To test each mode, we will make some examples. The first thing we will do is install the library to be able to program the ADS1115.

Install Library for ADS1115

The first thing is to install the library to control the ADS1115. The most famous and popular is the Adafruit.

In the lesson on how to install a library with Arduino, I show you not only to install it but also how to select the most suitable for your project.

Just look for the Adafruit ADS1X15 library in the library manager.

You need to install it and go. Now you can access both the library and the code of the examples.

This library allows you to control both the ADS1115-based board and the ADS1015-based board. In this lesson, I will only talk about ADS1115.

Single-End Mode

The first mode we are going to see is the single-ended mode. This mode causes the ADS1115 to behave like a normal ADC.

It has 4 numbered analog inputs such as A0, A1, A2, and A3.

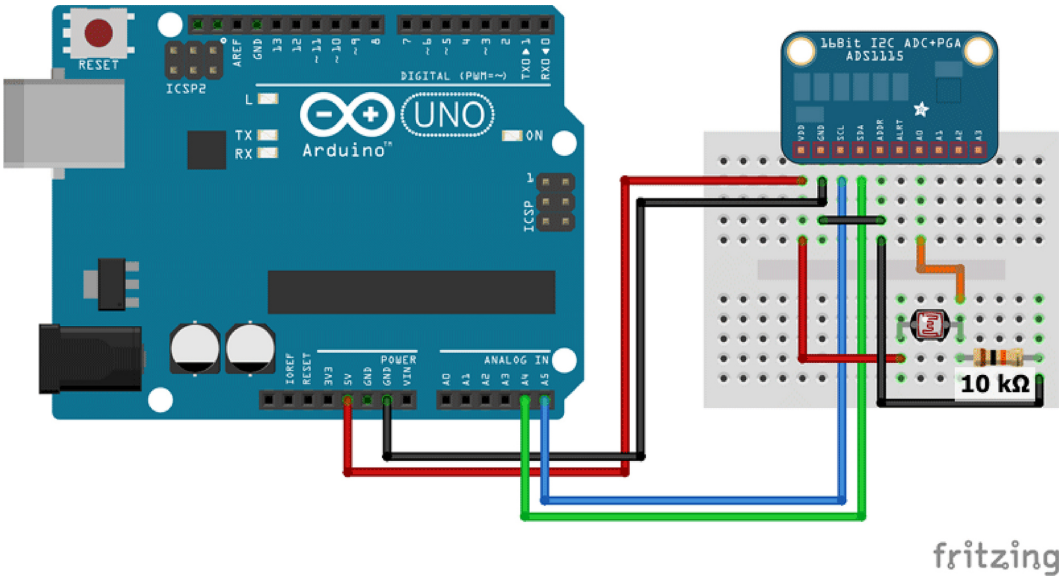
The idea is to connect an analog sensor to one of those inputs and measure its value.

In this example, you will use a photoresistor or LDR connected to the ADS1115.

Then I leave the electrical scheme to connect the ADS1115 to an Arduino UNO and a NodeMCU board based on the ESP8266.

ADS1115 Connection with Arduino UNO

The connection of the ADS1115 with the Arduino UNO is very simple. Follow the following scheme.



The SDA pin of the ADS1115 goes to the A4 pin of the Arduino UNO and the SCL pin to the A5 pin.

The I2C address is configured by connecting the ADDR pin of the ADS1115 to the GND pin.

In addition, I added a photoresistor or LDR to test the ADS1115 digital-analog converter (ADC).

The resistance is connected through a voltage divider with another 10kΩ resistor.

The analog pin of the ADS1115 that I am going to use is the A0.

ADS1115 Connection with ESP8266

The connection with the ESP8266 is very similar to that of Arduino UNO. In this case, I will use a NodeMCU board.

The SDA pin of the ADS1115 is connected to pin D2 of the NodeMCU ESP8266 board and pin SCL to pin D1.

The I2C address is configured by connecting the ADDR pin of the ADS1115 to the GND pin.

As in the Arduino UNO scheme, I have connected a photoresistor or LDR through a voltage divider with a 10kΩ resistor.

The LDR is connected to pin A0 of ADS1115.

Single-End Mode Programming

Once we have everything connected between the board and the ADS1115, we will see the programming.

We start with libraries and variables.

Libraries and Variables

```
#include <Wire.h>
#include <Adafruit_ADS1015.h>

// Create class object
Adafruit_ADS1115 ads;
```

The first thing is to import the Wire.h and Adafruit_ADS1015.h libraries. Although this last library is called ADS1015, it also contains the code to control the ADS1115.

Then create an object of the Adafruit_ADS1115 class that we call ads.

Setup () Function

```
void setup (void)
{
  Serial.begin (9600);
  delay (200);
  // Start ADS1115
  ads.begin ();
}
```

The setup () function begins by initiating serial communication to display messages through the serial monitor.

Then start the ADS1115 by calling the ads.begin () function that does not support any parameters.

Loop () Function

```
void loop (void)
{
  // Get data from A0 of ADS1115
```

```
    short adc0 = ads.readADC_SingleEnded (0);
    Serial.print ("A0:");
    Serial.println (adc0);

    delay (1000);
}
```

The loop () function is where you really have to read the analog pin.

The first thing is to declare a variable of type short called adc0. This variable will store the value of the digital pin A0.

It is of the short type because this type of stores signed 16-bit integers. Enough to store the value returned by an analog pin, a 15-bit number as we have seen before (from 0 to 32767).

The ads.readADC_SingleEnded (0) function is used to obtain the ADS1115 analog pin value. The parameter that is passed to this function is the analog pin number.

0 will return the value of analog pin A0, 1 that of pin A1, and so on.

Finally, the information obtained by the serial monitor is shown in the loop () function.

Now there is only one thing left, upload the code to the board and open the serial monitor.

Full Code

```
#include <Wire.h>
#include <Adafruit_ADS1015.h>

// Create class object
Adafruit_ADS1115 ads;

void setup (void)
{
    Serial.begin (9600);
    delay (200);

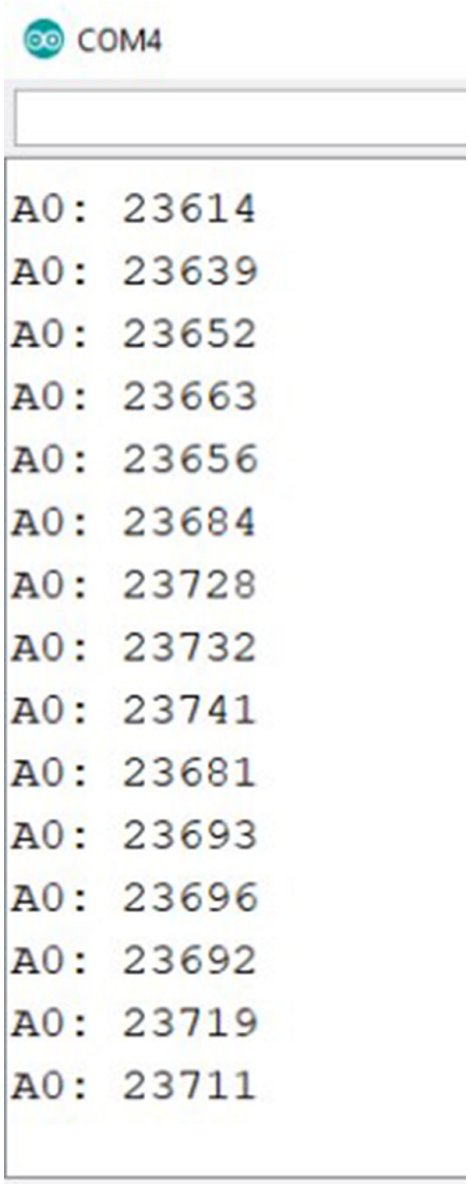
    // Start ADS1115
```

```
    ads.begin ();
}

void loop (void)
{
    // Get data from A0 of ADS1115
    short adc0 = ads.readADC_SingleEnded (0);
    Serial.print ("A0:");
    Serial.println (adc0);

    delay (1000);
}
```

You will get something similar to this.



If you put your hand on top of the photo resistance, the value will decrease, and if you put your hand on it, the value will increase.

But what would happen if I want more precision? As we have seen, the ADS1115 includes a programmable gain amplifier or PGA that allows us to have more precision with the same number of bits.

As it does? Changing the reference value.

By default, the ADS1115 uses the reference voltage of 6.1444V, which implies a scale factor of 0.1875mV.

If we change the reference voltage, for example, to 4,096V, a scale factor of 0.125mV is obtained.

To change the scale factor, you just have to add the call to the ads.setGain (GAIN_ONE) function just before starting the ADS1115 with the ads.begin () function on line 13.

The code would be as follows.

```
#include <Wire.h>
#include <Adafruit_ADS1015.h>

// Create class object
Adafruit_ADS1115 ads;

void setup (void)
{
  Serial.begin (9600);
  delay (200);

  // Change scale factor
  ads.setGain (GAIN_ONE);

  // Start ADS1115
  ads.begin ();
}

void loop (void)
{
  // Get data from A0 of ADS1115
  short adc0 = ads.readADC_SingleEnded (0);
  Serial.print ("A0:");
  Serial.println (adc0);

  delay (1000);
}
```

What happens when you change the voltage reference? Since the range is reduced, and therefore the scale factor decreases.

Thanks to this, smaller variations in brightness can now be measured.

In the following table, I show the different values that this function can take, the reference voltage, and the corresponding scale factor.

Parameter	Reference	Scale factor
GAIN_TWOTHIRDS	6.144V	0.1875mV
GAIN_ONE	4.096V	0.125mV
GAIN_TWO	2,048V	0.0625mV
GAIN_FOUR	1,024V	0.03125mV
GAIN_EIGHT	0.512V	0.015625mV
GAIN_SIXTEEN	0.256V	0.0078125mV

And that's how the ADS1115 works normally, like any other digital-analog converter (ADC).

Now let's see another mode of operation, the differential mode.

Differential Mode

As I said, the differential mode of the ADS1115 allows us to measure the difference between two analog pins of this module.

You can measure the voltage difference between A0 and A1 or between A2 and A3. That is why it is called differential.

But why can this type of measure be useful?

If you usually do projects with Arduino at some point, you will need to power your project with batteries.

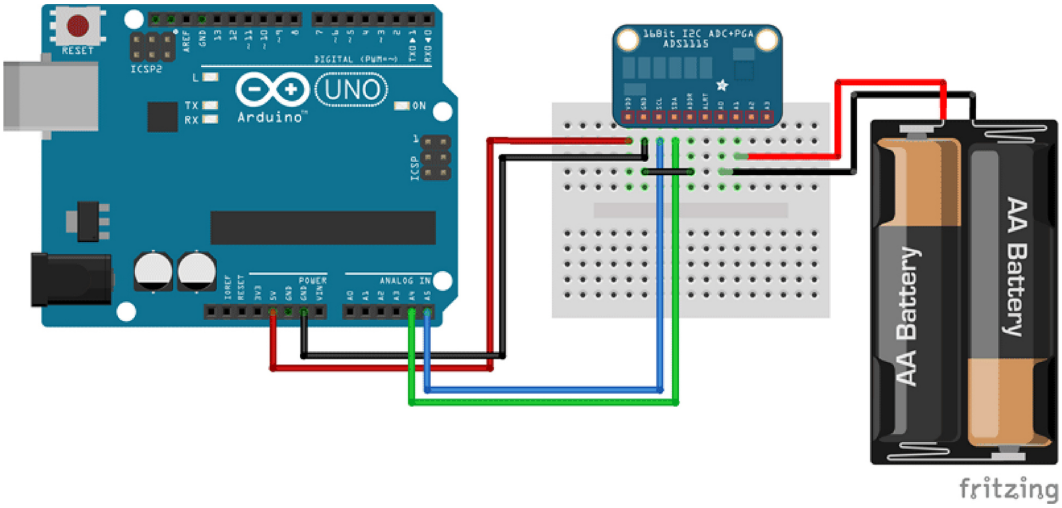
The differential mode will allow you to make a measurement in which none of the sides of the measurement is connected to GND, as is the case when measuring a battery or battery with Arduino.

In addition, another benefit of this mode is that you can perform negative voltage measurements.

In the following example, I will show you how you can do it.

ADS1115 Connection with Arduino UNO

Assemble the circuit starting from the following scheme.



I have used a battery holder with two AA batteries, but you can use another class of batteries, such as AAA or even a single 1.5V battery.

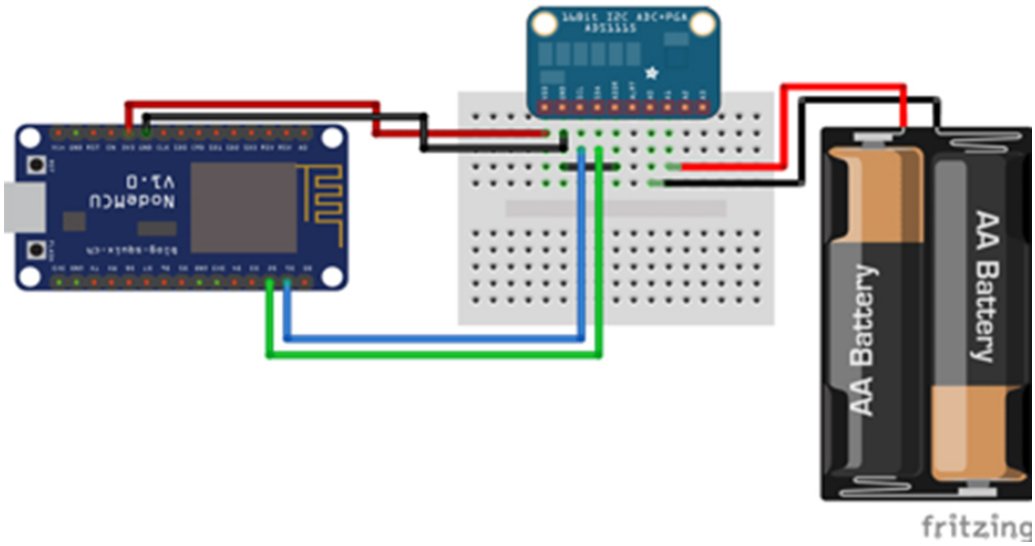
The positive end of the battery holder is connected to the A1, and the negative end is connected to the A0 of the ADS1115.

When the two batteries are connected in series, the voltages are added. Therefore 1.5 + 1.5 = 3V. This will be at best.

The programming will be the same: what will change will be the value obtained from the difference.

ADS1115 Connection with ESP8266

In the following scheme, I show you how to connect the ADS1115 with a NodeMCU ESP8266 board to make a differential measurement.



As in the scheme with the Arduino UNO, I used an AA battery holder.

The positive end of the battery holder to A1 and the negative end to A0.

Be very careful if you are going to measure a voltage higher than 3.3V with which the ADS1115 is powered. Remember that you cannot beat VDD + 0.3V, that is 3.3 + 0.3 = 3.6V

Differential Mode Programming

The ADS1115 differential mode programming is very simple and very similar to what we have seen in the single-ended mode.

Then I leave the code

```
#include <Wire.h>
#include <Adafruit_ADS1015.h>

// Create class object
Adafruit_ADS1115 ads;

// Scale factor. By default the reference value is 6.144V
// The scale factor of this reference is 0.1875mV
float scale factor = 0.1875F;

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

```
    // Scale factor
    ads.setGain (GAIN_TWOTHIRDS);

    // Start ADS1115
    ads.begin ();
}

void loop (void)
{
    // Get the differential
    short difference_0_1 = ads.readADC_Differential_0_1 ();
    // Convert to volts
    float volts = (difference_0_1 * scale factor) /1000.0;

    Serial.print ("Difference 0-1 =");
    Serial.println (difference_0_1);
    Serial.print ("Voltage =");
    Serial.println (volts, 4);
    Serial.println ();


    delay (2000);
}
```

The first thing the code does is import the libraries and create an instance of the Adafruit_ADS1115 class called ads.

Then I declare a float type variable to store the scale factor called the Scale factor.

The scale factor depends on the voltage reference that has been selected. By default, it is 6.144V, which corresponds to a scale factor of 0.1875mV.

If the voltage reference is modified, you will have to modify the scale factor and put the corresponding one according to the table we have seen in the previous section.

In the setup () function, serial communication is initiated, the scale factor is set with the ads.setGain function (GAIN_TWOTHIRDS), and the ADS1115 sensor is started with the ads.begin () function .

As I said, if you modify the scale factor with the function ads.setGain (...) will have to change the initial value of the variable scalefactor to the match.

In the loop () function, the first thing is to obtain the difference between pin A0 and A1 using the ads.readADC_Differential_0_1 () function.

This function returns the difference between A0 and A1; that is, the value in A0 subtracts the value from A1.

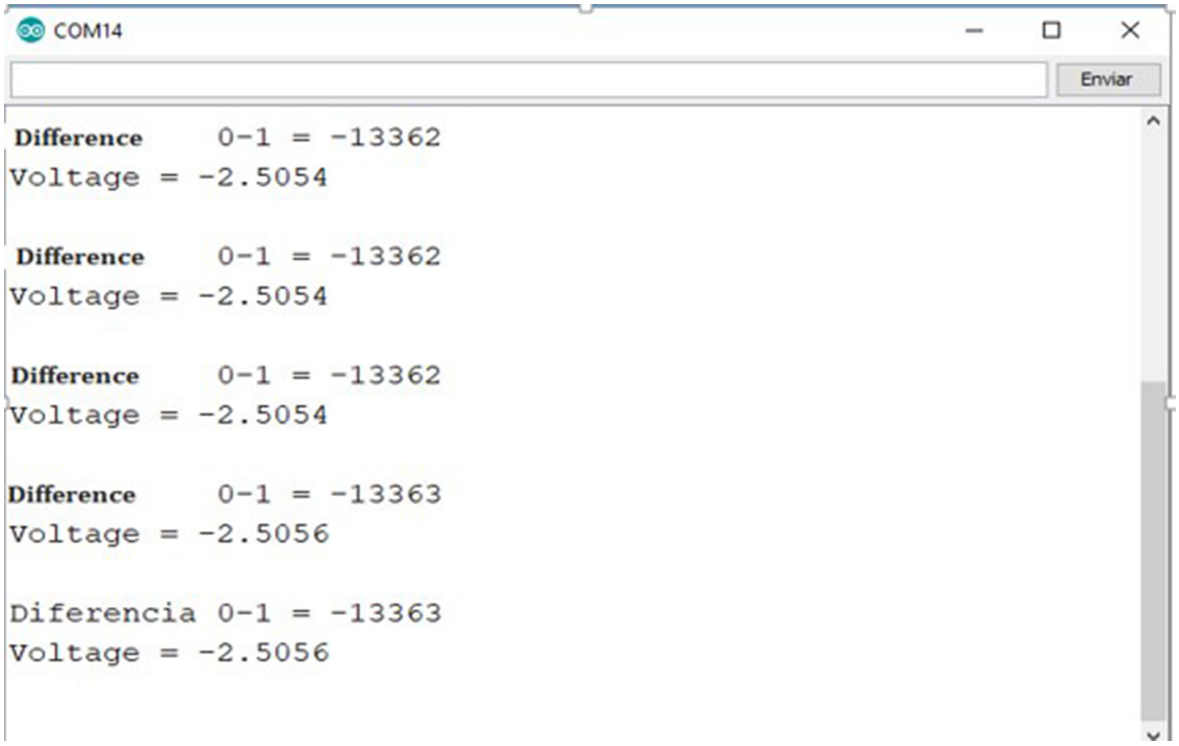
If you need to use the other two pins you can use the ads.readADC_Differential_2_3 () a function that will return the difference between A2 and A3.

This difference is expressed as a signed 16-bit number that is between 32,768 and 32,767. This value must be converted to volts.

To do this, multiply by a Scalefactor and get millivolts. To get volts, you have to divide by 1,000. The same is what is done on line 27 of the code.

Finally, the information is shown by the serial monitor.

Now you just have to load the code to the board and open the serial monitor. You will get something similar to this.



As you can see, negative numbers come out. This is because we are subtracting the voltage we have on pin A0 minus pin A1.

As we have connected the positive of the battery holder to A1, the result is negative.

You can try changing the cables (positive at A0 and negative at A1) and see how the positive voltage appears.

Finally, we will see how to make the ADS1115 launch an alert when it is running in comparator mode.

Comparator Mode

The comparator mode allows a comparison of the value obtained by one of the analog pins with a preset threshold.

As we have seen before, two types of comparisons can be configured: the traditional one or by a window.

If the traditional one is used, as long as the value exceeds the maximum threshold, the ALERT pin will be set to LOW. By default, it is always in the HIGH state.

If you use the type per window, it will change state whenever the high threshold is exceeded and whenever it falls below the low threshold.

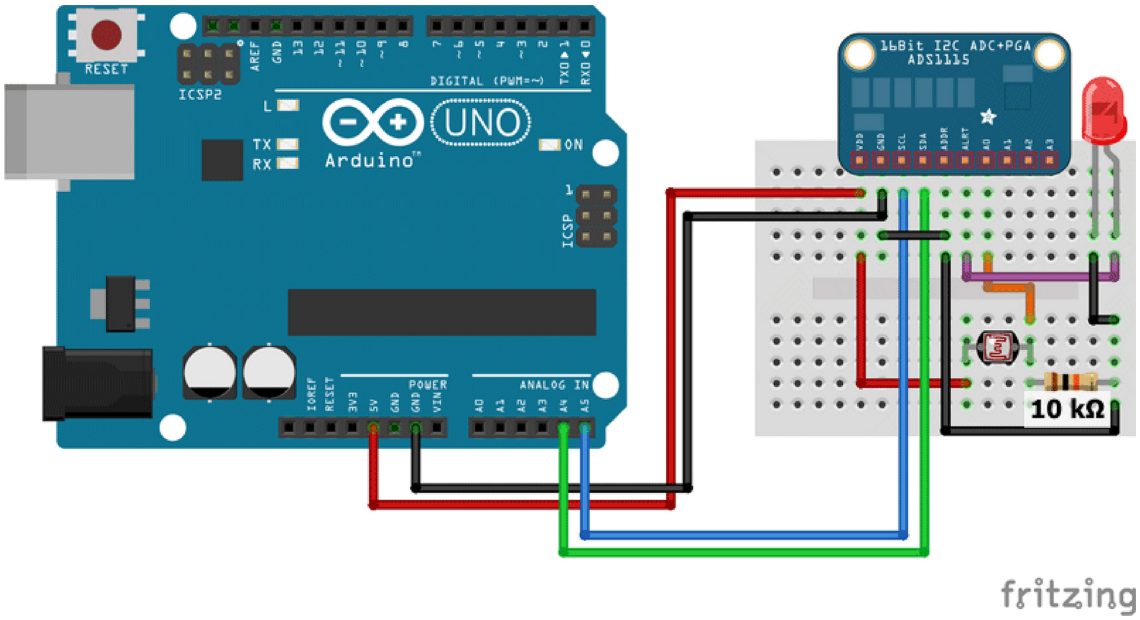
With the Adafruit library, you can only work with a threshold, high threshold, and traditional mode. It has not implemented the other mode.

This is precisely what we will see next.

ADS1115 and Arduino UNO Connection

The circuit we are going to use is the same as the one we have seen in a previous section.

Follow the following scheme to connect the components.

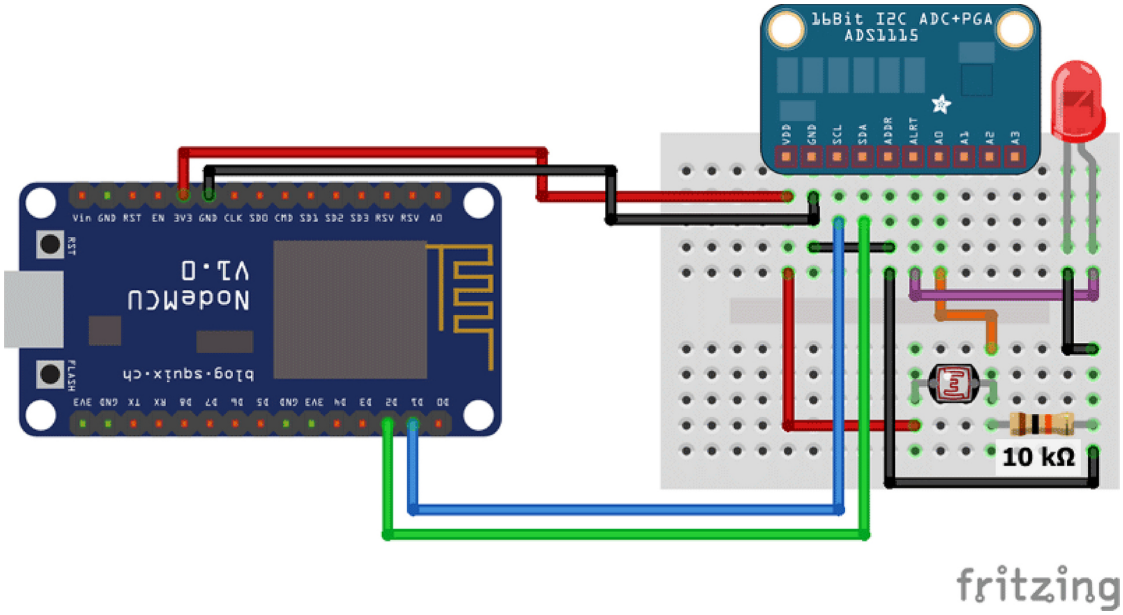


The photoresistor or LDR is connected to pin A0, and we also connect an LED to the ALERT pin.

The idea is to set a threshold in the ADS1115, and every time you lower the brightness of that threshold, turn on the LED.

ADS1115 and NodeMCU ESP8266 Connection

The connection with the NodeMCU ESP8266 board is practically the same. Follow the following scheme.



As with the Arduino board, the photoresistor or LDR is connected to pin A0, and the LED to pin ALERT.

Now only programming remains.

Programming in Comparator Mode

Below is the complete code with the comparator example.

```
#include <Wire.h>
#include <Adafruit_ADS1015.h>

// Create class object
Adafruit_ADS1115 ads;

void setup (void)
{
  Serial.begin (9600);
  delay (200);

  // Scale facto r
  ads.setGain (GAIN_TWOTHIRDS);

  // Start ADS1115
  ads.begin ();

  // Scale factor. By default the reference value is 6.144V
  // The scale factor of this reference is 0.1875mV
  float scale factor = 0.1875F;

  // Comparator threshold at 2.5V
  // To calculate, divide the (voltage / scale factor) * 1,000
  float threshold = (2.5 / Scale factor) * 1000.0F;
  Serial.print ("Threshold =");
  Serial.println (threshold);

  // Assign the threshold to analog pin A0
  ads.startComparator_SingleEnded (0, threshold);
}

void loop (void)
{
```

```
    // Only the comparison will be made after reading
    short readADC0 = ads.getLastConversionResults ();
    Serial.print ("ADC0:");
    Serial.println (read ADC0);

    delay (100);
}
```

The first part of the code is the same as the codes we have seen so far in this lesson, importing libraries and instance to the Adafruit_ADS1115 class.

In the setup () function, the serial communication starts, the scale factor is set, and the ADS1115 starts.

In the next part of the code, the high threshold of the comparator mode is set from line 20 to line 29.

The first thing is to declare a variable with the scale factor. Remember that by default, the scale factor is 0.1875mV, corresponding to 6.144V voltage reference.

The variable that stores this value is a scale factor of the float type.

The threshold must be set to a number that ranges from 0 to 32,767 (they are 15-bit resolution). The logical thing is to set a threshold voltage and then pass it to the number.

To do that conversion you have to apply the following formula:

$$\frac{\text{voltage}_{\text{threshold}}}{\text{scalefactor}} = 1.000$$

The threshold voltage must be divided by the scale factor. It must be multiplied by 1,000 since the scaling factor is in millivolts, in case it is in Volts it would not be necessary

To assign the threshold in the ADS1115, you must use the ads.startComparator_SingleEnded (ADC, threshold) function.

The first parameter of the function indicates the analog pin to be used, and the second parameter indicates the threshold to be set.

It's that simple.

If, for example, you want to set a threshold of 2.5V, the number to use is the following.

$$\frac{2.5}{0.1875} \times 1.000 = 13.333{,}33$$

In the loop () function, you must constantly check the value on the analog pin with the ads.getLastConversionResults () function so that the ADS1115 internally makes the comparison.

If the voltage on the A0 is greater than 2.5V, the ALERT pin of the ADS1115 will be in the LOW state, that is, the pin will be activated.

If the voltage at A0 is less than 2.5V, the ALERT pin will be in the HIGH state, that is, the pin will be deactivated.

By default, the ALERT pin works inverted.

Now there is only one thing left, upload the code to the board and try zooming in and out of light to the photoresistor or LDR.

You will see how, when the light hits the LED, the LED goes out, and when the light doesn't hit, the LED goes off.

With this, I conclude this lesson on the digital-analog converter.

The ADS1115 digital-analog converter can be a great ally when we need more analog pins or more precision in our projects.

It is very simple to use with an Arduino or an ESP8266 thanks to its I2C interface.

It allows working in different operating modes: single-ended mode, differential mode, or comparator mode.

Using one or the other will depend on the requirements of our project.

Thanks to the Adafruit library, its use with any of these two plates is very simple, and also the code is 100% compatible between them.

Chapter 9

Arduino Interruptions

Atmel microcontrollers incorporate a mechanism that is often unknown to all of us. The interruptions with Arduino allow us to react to external events to the plate in a quick way. When a signal is detected, an interruption interrupts the process that is running. This will allow us two things. On the one hand, quickly execute a piece of code, and on the other hand, stop the execution of the code that was being executed.

For example, imagine that you want to cook a pizza in the oven. Typically, heat the oven to a certain temperature, put the pizza in, and set the clock for 25 minutes. Meanwhile, you are going to watch TV.

When the clock ends the countdown, a bell rings, and you interrupt watching television to attend the oven. You take the pizza out of the oven, take it to the living room, and keep watching TV. This would be a real-life example of how interruptions work.

Interruptions are not simply to change and do something different. For example, we are riding a bicycle, we get home and keep the bike in the garage, or a button is pressed by the user and allows you to stop the DFRobot.

In these two cases, we do not interrupt an action; we go on to do something different. We finish doing one thing and do another. In the case of the button, the most we can do is remember the state of the pressed button. We must use the interruptions for what they are, interrupt what is being executed, and perform another task, then return to the same place where we had left.

Within the range of interruptions with Arduino are scheduled interruptions and external interruptions. The philosophy is the same, to be able to execute a code when a specific event happens. In the case of scheduled interruptions, the event will trigger every certain interval of time that we must configure.

This type of interruptions is not the object of this article. If you want to know more, you can see the reference of the TimerOne library that is responsible for configuring and managing this type of interruptions.

The other type of interruptions, the external ones, allows us to react to external events. An event may be that a button is pressed, or that a sensor captures certain information.

What will you learn in this chapter?

- Why use interrupts with Arduino
- How to use interrupts with Arduino
- Practical example: controlling the speed of fantastic car lights

Why Use Interrupts with Arduino?

Using interruptions will allow us to forget about controlling certain pins. This is very important because, within an application or program, we will not do a single thing. For example, we want an LED to turn on or off when we press a button. This is really simple, but when we also want another LED to flash, things get complicated.

The probability of capturing the event when the button is pressed decreases with the increase in flashing time. In these cases, and many others, we are interested in releasing the Arduino processor so that only when the button is pressed, it takes a specific action. So we will not have to be constantly checking the X pin if you have pressed the button.

Precisely this is the sense of interruptions, to be able to do other things as long as the event does not happen, but when that external event is present, to quickly execute the associated code.

Internally, Arduino (or rather the AtMega microcontroller) has certain configured interrupts that it throws according to the situation. For the transmission of data through the serial port, to reset the board before loading a program, I2C communication, etc ...

In this case, we will create our own interruptions associated with certain pins according to the type of board.

How to Use Interrupts with Arduino

Inside the Arduino board, we find different pins that allow us to have interruptions. It will depend on the type of plate.

LICENSE PLATE	**PINS INTERRUPTIONS**
One, Nano, Mini	2. 3
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21
Micro, Leonardo	0, 1, 2, 3, 7
Zero	All digital pins except 4
MKR1000	0, 1, 4, 5, 6, 7, 8, 9, A1, A2
Due, 101	All digital pins

In order to use interruptions with Arduino, we do not need to incorporate any type of library into our program. We only need to make references to certain functions or methods. Now we will see what they are.

attachInterrupt (pin, ISR, mode)

This function will allow us to define or configure one of the pins as an interrupt port. The three parameters it supports are:

- Pin: we must take special care with this parameter. Indicates which pin we will use as an interruption. We must not pass the pin number, and we must pass the ordinal. That is, if we work with Arduino UNO, we have two pins for interruptions 2 and 3. If we want to use the 2, we must put a 0 and if we want to use the 3, we must put a 1. This is very easily solved using another function, digitalPinToInterrupt (pin), which returns the ordinal of the pin we want to use. In this case, we must pass your number. For example, in the previous case, if we want to use pin 2, we would call the digital function PinToInterrupt (2). This will return a 0 and is the method recommended by Arduino to pass this parameter.
- ISR: is an abbreviation for Interrupt Service Routine and is nothing more than the function or method that is called when

the interruption occurs. It is of a particular type since it does not support parameters and does not return any value either.

- mode: defines when the interruption should be triggered. It can take four constant values depending on what we want to do:
    - LOW: the interrupt will be launched when the pin is in the low state.
    - CHANGE: The interrupt will be launched when the pin changes the value from high to low, or from low to high.
    - RISING: the interrupt will be launched when the pin changes state from low to high.
    - FALLING: the interrupt will be launched when the pin changes state from high to low.

There is the fifth state that only the Arduino Due, Zero and MKR1000 allow:

- HIGH: the interrupt will be launched when the pin is in a high state.

detachInterrupt (pin)

If attachInterrupt () allows us to configure a pin as an interrupt, the detachInterrupt () method overrides that setting. As a parameter, we pass the pin, and we can do it with the digital function PinToInterrupt (pin number) that will return the ordinal of the pin from which we want to cancel the configuration.

Practical Example: Controlling the Speed of Fantastic Car Lights

The best way to learn is to put into practice what is learned in theory. For this, what we are going to do is the typical sketch with the fantastic car lights. In this case, we will control the speed with which it moves through two buttons .

Electrical Circuit

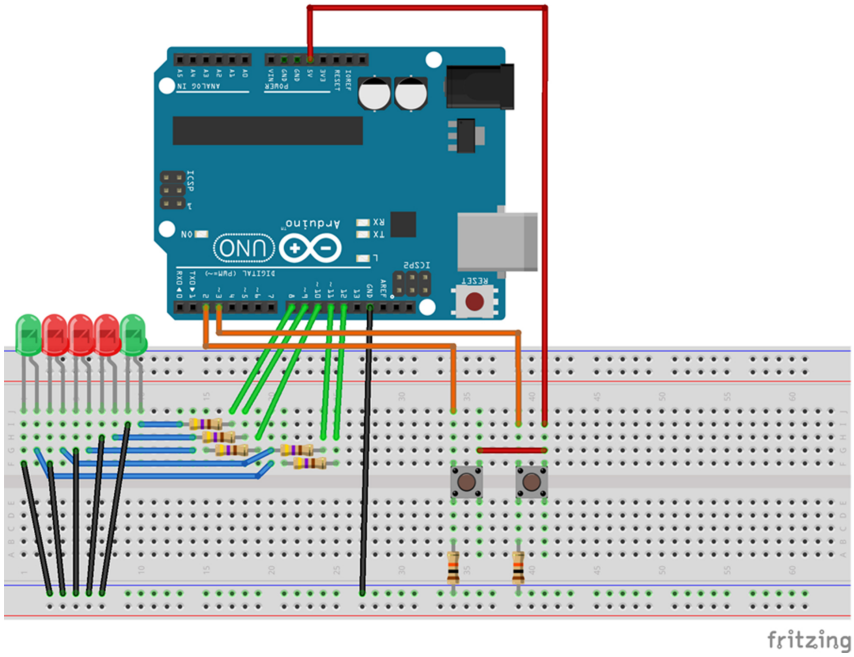
We will need the following material:

- 2 green LEDs
- 3 red LEDs
- 5 470 Ω resistors
- 2 push buttons
- 1 1 kΩ resistor
- Arduino UNO
- Protoboard
- Male / male cables

Each button will be connected to a pin and will have a pull-down resistor. This will make in a normal state, without pressing, the pin will have a low state. When the button is activated, we will have a high state. Therefore we must detect a change of state from low to high (RISING).

On the other hand, the LEDs will do a sequence of turning on and off from left to right and from right to left. Simulate the fantastic car lights. We will start with speed, and as we press the buttons, the speed will increase or decrease.

The electrical scheme is as follows.



Programming the Interruption Sketch with Arduino

The first part is the declaration of counters and variables. At this point, it is necessary to make a clarification. When we work with interruptions with Arduino, we must take special care with the ISR methods that will be executed when the event occurs.

In general, we must make ISR methods or functions as short and as fast as possible. A blockage between the different ISRs can occur. What happens in these cases is that only one is executed at a time, the rest of the interruptions being queued.

Within the interruptions, the delay () (delay) does not work; therefore, do not use it. Remember that it is about executing code as quickly as possible. If you want to make the process stop within an ISR, use the delayMicrosends () function since it works normally.

The variables that we will use both within the ISR methods and outside, either in the loop () or in any other function, we must declare them as volatile. This reserved word tells the compiler that these variables can change their value at any time and, therefore, the compiler must reload the variable every time reference is made to it. In contrast, other variables require the compiler to rely on a copy that it may have in a processor register.

Sketch Fantastic Car Lights with Interruptions

```
* /

// Global speed variable
volatile int speed = 20;
// Constants of maximum, minimum speed and how much increases
maximum int const = 1000;
minimum int const = 20;
const int increases = 20;

// Array with the pin numbers where they are connected
// the LEDs of the fantastic car lights
int leds [5] = {8, 9, 10, 11, 12};

void setup () {
  // Initialize the LED pins as output and at low state
  for (int i = 0; i <5; i ++)
  {
    pinMode (leds [i], OUTPUT);
    digitalWrite (leds [i], LOW);
  }

  // We assign the minimum speed
  speed = minimum;

  // Configure the interrupt pins so that
  // detect a change from low to high
  attachInterrupt (digitalPinToInterrupt (2), speedLess, RISING) ;
  attachInterrupt (digitalPinToInterrupt (3), speedMas, RISING);

}

void loop () {
  // This first loop travels the array from left to right
  for (int i = 0; i <5; i ++)
  {
    // Only for the second and consecutive pin, we turn off the previous
pin
    // In the case of 0, it is not necessary since by default it is off
```

```
    // Be careful that we go out of the range of the array 0-1 = -1 this
element does not exist
    if (i> 0)
    {
      // Turn off the LED on the left
      digitalWrite (leds [i - 1], LOW);
    }

    // We turn on the LED in which we are
    digitalWrite (leds [i], HIGH);

    // We wait for the time marked by speed
    delay (speed);
  }

  // Turn off the last LED on, element 5 of the array
  digitalWrite (leds [4], LOW);

  // We traverse the array in the reverse direction from right to left
  for (int i = 4; i> = 0; i--)
  {
    // In the first case as the LED is off, we do nothin g
    // Be careful that we go out of the range of the 4 + 1 = 5 array this
element does not exist
    if (i <4)
    {
      // Turn off the LED on the right
      digitalWrite (leds [i + 1], LOW);
    }
    // We turn on the LED in which we are
    digitalWrite (leds [i], HIGH);

    // We wait for the time marked by speed
    delay (speed);
  }

  // Turn off the last LED on, element 0 of the array
  digitalWrite (leds [0], LOW);
```

```
}

// ISR pin 2, slow down
void speed Less ()
{
  // Decrease the set value
  speed = speed - increases;

  // If we have reached the minimum speed we do not decrease more
  if (speed <minimum)
  {
    speed = minimum;
  }
}

// ISR pin 3, increase the speed
void speedMore ()
{
  // Increase the set value
  speed = speed + increases;

  // If we have reached the maximum speed we do not increase more
  if (speed> maximum)
  {
    speed = maximum;
  }
}
```

And finally, here in this chapter, we have seen the interruptions with Arduino. It is very important to know this technique very well since it will save us a lot of work on certain occasions.

Conclusion

Now that you have a little more skills and intimacy with electronics and programming, do you feel safe to create something? If you need more knowledge on Arduino, our Arduino series has several books ranging from super basic to advanced. Here are some projects that may be interesting for those like you who just had their first contact with Arduino.

Some components that we didn't work so hard on were the sensors, we saw only vibration and light, but there is a sensor for almost everything you can imagine! With them, you can carry out various projects, such as those listed below:

1. Temperature sensor
2. Proximity sensor, which measures distance using ultrasound
3. Touch sensor to play songs
4. Soil moisture sensor to know when your plant needs water
5. Flame sensor, be careful with this sensor testing

In addition to sensors, other cool components to learn to move are motors. In the book, we have seen how the servo motor works, but it is also interesting to know the simplest direct current motor or even the stepper motor.

To interact with Arduino, it is interesting to put a display on some projects. We saw the multi-segment display, which is quite simple. There are other models with more room to write, such as the LCD Display and OLED Display.

With Arduino, we can control things in the house too! We can go beyond the small electronic components using a component called relay, and we saw in this book how it works to light a lamp with a button. Remember that in a Project, we lit an LED according to the light? We also lit a lamp. Just be careful when working with the mains, turn everything off when

mounting, and turn on only after mounting everything. The damage can be huge, so if possible, do it with some follow-up.

Of course, from a sample project, you can add other sensors and apply different logic to the proposal, so your project will better meet your needs. If you want to learn more about Arduino programming, the rest of the books of this series will teach you how to work with programming logic fluently.

Did you know there are other development boards besides Arduino? Some have easy access to the internet, some have a lot of processing, while others are practically a computer, with the operating system and all.

We on other boards can say, for example, are Raspberry and Internet of things with electronics. With them, you can start learning about a new range of options in the world of electronics and programming like Arduino.

No doubt, Arduino is an amazing platform, and the amount of projects that can be done is virtually endless. We hope that throughout the development of the 15 kit projects, you have understood the first steps with Arduino, and the guide has been enlightening. The main goal is actually to pique your curiosity and make you create your own designs.

References

https://classroom.littlebits.com/projects/pokelight

https://fabacademy.org/archives/2012/students/di_vozzo.romain/week_7_fab_academy_electronics_design_feb_29.html

https://www.littlebits.com/projects/croq-cat

https://digitalmedia.risd.edu/pbadger/PhysComp/index.php?n=Devices.MomentaryCode

https://www.arduino.cc/en/Tutorial/ForLoopIteration

https://sites.google.com/site/hwcontwerpen/sketches-les-0-t-m-9/les1d-looplicht

http://fritzing.org/projects/brightness-indicator-photocell-with-tweak

https://www.arduino.cc/en/tutorial/loop

https://stackoverflow.com/questions/50251876/how-can-i-blink-a-led-differently-when-i-press-a-toggle-button/50259528

https://stackoverflow.com/questions/50251876/how-can-i-blink-a-led-differently-when-i-press-a-toggle-button/50259528

https://www.arduino.cc/en/Tutorial/AnalogInOutSerial

https://www.arduino.cc/reference/en/#structure

ARDUINO PROGRAMMING

Tip and Tricks to Learn Arduino Programming Efficiently

STUART NICHOLAS

Introduction

Have you ever wondered how to start the adventure with microcontrollers? Check Arduino. This open platform has been prepared for all DIY enthusiasts, including robot builders. The first part of the book discusses the basics required for further lessons.

If you are interested in electronics at least to a small degree and would like to start programming your own systems, then Arduino will be the perfect solution to start with. You won't have to waste time designing your own tiles, choosing the right programmer, and tedious environment configurations. Everything will work almost immediately.

This book has been planned for 11 parts (with possible continuation). Of book, he does not discuss the whole Arduino, and the subject is so extensive that you can write about it (several) books. The purpose of this book is to explain the basics in practice and to interest the reader in further exploring this platform.

What is Arduino?

For a beginner, Arduino is a ready "development kit" with the popular AVR microcontroller. Created according to the relevant assumptions, thanks to which:

1. Does not require an external programmer
2. Works well with a dedicated compiler
3. You can buy an "infinite" number of expansion boards (e.g., motor controllers, displays, executive modules)

However, the true power of Arduino lies in a dedicated programming language based on C / C ++. Starting the adventure with microcontrollers, it's worth learning at least the basics about their construction and how they work.

Fortunately, in the case of Arduino, to program your system, you do not need to know the microcontroller registers. Everything is based on friendly

libraries, thanks to which the creation of even a complicated program is within reach of a novice programmer.

In short, Arduino is modules with microcontrollers that can be easily programmed using generally available libraries in a language similar to C / C ++.

Advantages of Arduino

The project began to be developed in 2005 in Italy. Since then, he has gathered a mass of supporters and fanatical users. From the very beginning, Arduino was prepared for people who did not have much in common with programming microcontrollers.

Excellent environment, friendly syntax, and low prices have made Ardunio extremely popular. The community built around this project is huge. This brings many benefits. From a beginner's point of view, three are most important:

- A huge number of ready-made solutions. Various projects are created on Arduino. If you came up with something "new" and interesting, then 90% someone has already done it on Arduino earlier and posted a description of the project on the Internet.
- The popularity of the platform has meant that manufacturers have prepared countless tile varieties and extensions - you will find more about this later in the book.
- A large number of users make it easier to find help when they get stuck in an important point of the project.

Arduino - Choosing a Hardware Platform

Arduino is an Open Hardware platform. This means that all materials needed to create your own development kit operating in this standard are available. For this reason, you can find many different tiles compatible with Arduino.

At the moment, Arduino officially talks about 20 available models on its websites. In every good store, you will find at least a few different sets. For

the needs of the book, I decided to choose the most popular tile - Arduino UNO R3.

Arduino is a project that is constantly growing - both in programming and hardware. The UNO version 3 board is currently prompted by Arudino, like the one on which the latest libraries and expansion boards can be used for a long time.

Equipment Arduino UNO R3

The heart of the system is the popular 8-bit microcontroller from Atmel, AVR ATmega328, operating at a frequency of 16 MHz.

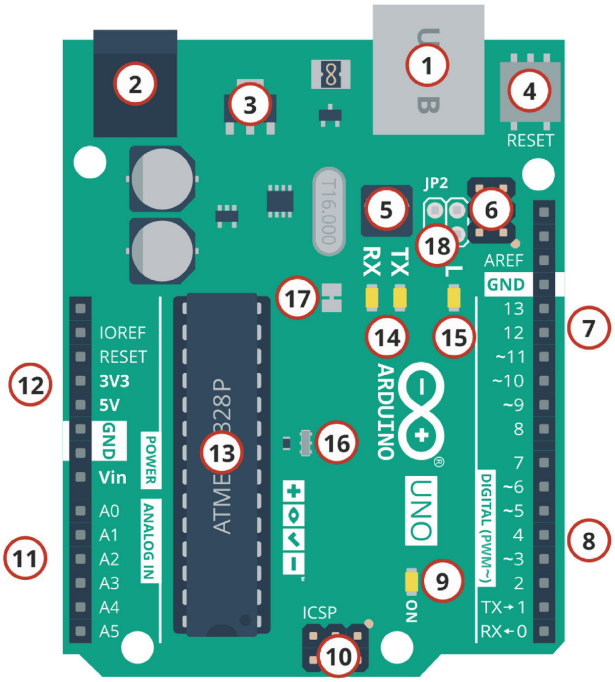
Working at such a frequency in a simplified way means that the microcontroller can perform 16 million operations per second. It's a lot.

Special connectors, placed characteristically on the sides of the board, are the outputs of the most important signals. There we will find 14 programmable digital inputs/outputs. Six of them can be used as PWM outputs (e.g., for motor control), and another six as analog inputs. We will also find there a reset signal and power supply.

Arduino can be powered in several ways. The most popular methods are:

1. Power supply via USB cable
2. Powered by a plug-in power supply (optimally 7V - 12V) or batteries

The most important elements are marked on the graphic below:



1. USB connector - used for power supply, programming, and communication with a computer
2. Power connector (optimally 7V - 12V)
3. Voltage stabilizer - the input voltage from connector 2 is reduced to 5V thanks to this system
4. Reset button - resets the Arduino board
5. Microcontroller responsible for communication with the computer via USB
6. Programming connector for the microcontroller from step 5.
7. Signal connector *
8. Signal connector *
9. LED diode signaling connection of voltage to Arduino
10. Programmer output for the microcontroller from point 13.

11. Signal connector *

12. Power connector *

13. Arduino heart, AVR ATmega328 main microcontroller

14. LEDs signaling transmission to/from the computer

15. LED at the user's disposal

16. Ceramic resonator clocking the microcontroller (point 13) with a frequency of 16MHz

17. A jumper that when cut off disables Arduino's automatic reset.

18. Solder pads with derived microcontroller signals from point 5, used extremely rarely in very specific and non-standard situations.

Clones, or Arduino (un) Original

As I mentioned, Arduino is an open-hardware platform. This means that anyone can make their own Arduino or design a tile per this standard. Sets from other companies operating, such as Arduino, are commonly called clones.

Clones can be divided into two types:

- Total fakes, imitating originals
- The tiles comply with the Arduino standard

For example, if you are looking for the Arduino UNO mentioned above and you find a plate identical to my photos at a price of less than $120, it will be 100% fake. You can take a chance and buy one. The choice is yours, whether you want to support companies that earn dishonestly or choose those that have put a little more effort into the production of their version.

Warning. The cheap imitation ones are often made of inferior quality elements that can damage the entire system.

Materials you need to program Arduino

In addition to the Arduino UNO mentioned above, a handful of additional elements will be used in the first seven parts of the book. A USB cable and connection cables will definitely be useful. Plus colored LEDs and buttons. We will present more complicated items on a 2x16 character text display.

In the further chapters of the book, we will also discuss controlling actuators such as servos and motors. Sensors will also be useful. We will use light sensors (photoresistors) and an ultrasonic distance sensor.

The set of necessary elements is visible in the photo below:



Now we will deal with the installation of the environment. I assume you don't have the equipment you need yet. However, Arduino will be necessary for further learning.

Arduino IDE Installation

The appropriate environment must be installed before programming. The latest Arduino IDE can be downloaded from the project's official website . The installer takes about 53 MB.

Warning. Update (14.04.2015)
Due to the conflict, two versions of the environment appeared within the Arduino team. Fortunately, this doesn't matter much to the end-user. However, if you purchased the original board, check the address on its back before installing the environment.

There you will find a reference to one of the following pages:

- http://arduino.org/
- http://arduino.cc/

If the first address occurs, download the environment from here . However, if you find a second address on the back of the tile, get it from here .

What happens when you install the wrong IDE?

Note - You will only be notified that the tile you have does not come from an authorized source. You can turn off this message with one click. Everything will work without problems.

Download the latest version.

Installation is standard. In the beginning, we accept the product license:

Then click a few times Next, and we become happy users of the Arduino IDE. Along the way, pay attention to the components installed:

If we choose to install the USB driver (which is recommended), we will see a system warning at the end of the installation.

Click " Install this driver software anyway, " you don't have to worry about anything. After the installation is complete, a new icon should appear on your desktop:

If the software has been installed correctly, we will see the welcome screen after starting it:

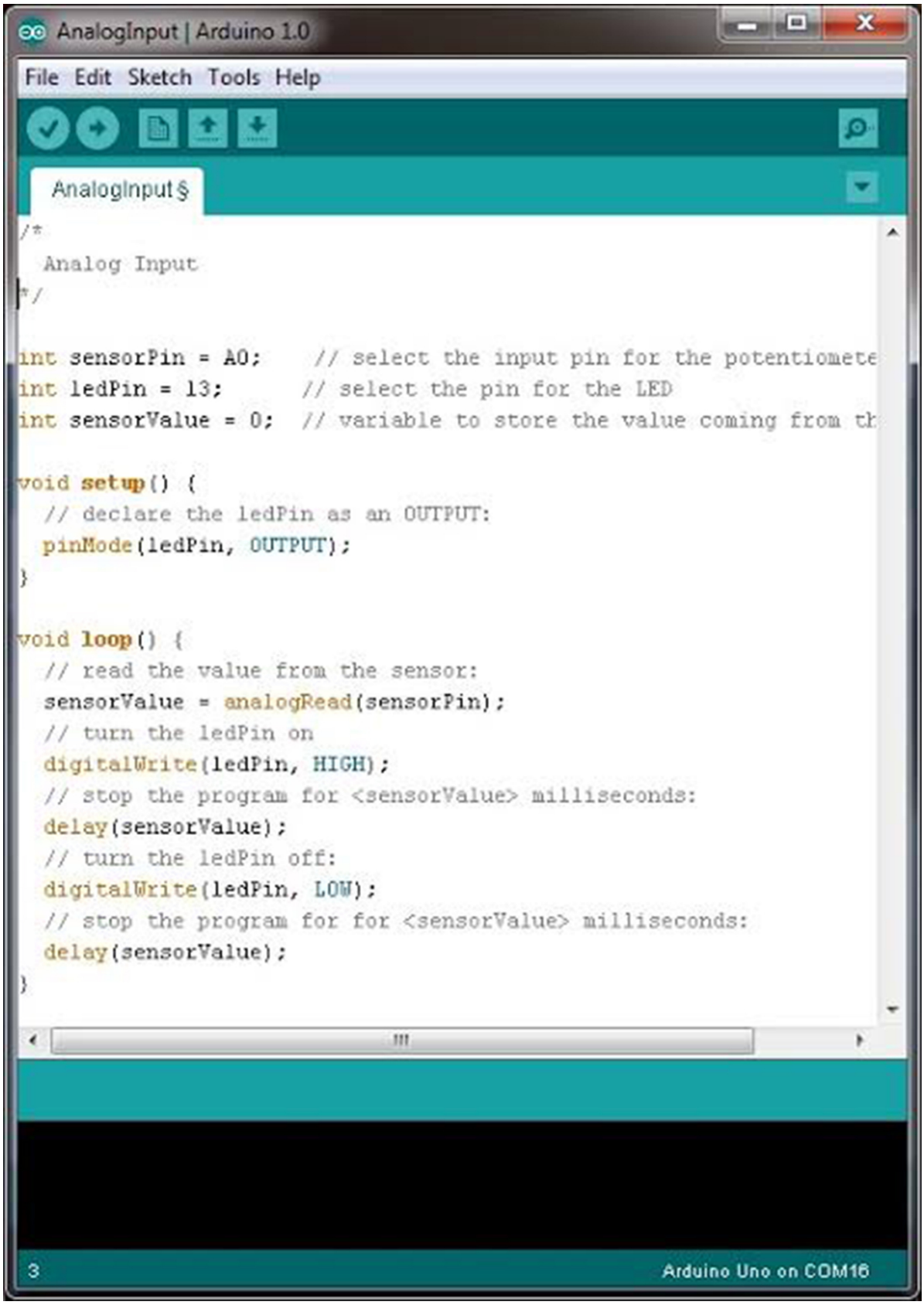
And after a while the editor:

If you already have Arduino

I wrote earlier that we would not program in this part. However, if you already have Arduino, you can do something very simple to check if everything works. First, start the Arduino IDE editor. Then from the menu select:

File -> Examples -> 01. Basics -> Blink

A separate window will open with the program code (do not temporarily penetrate its structure). It should look similar to the following:


AnalogInput | Arduino 1.0
File Edit Sketch Tools Help
AnalogInput §
/*
Analog Input
*/
int sensorPin = A0; // select the input pin for the potentiomete
int ledPin = 13; // select the pin for the LED
int sensorValue = 0; // variable to store the value coming from th
void setup() {
// declare the ledPin as an OUTPUT:
pinMode(ledPin, OUTPUT);
}
void loop() {
// read the value from the sensor:
sensorValue = analogRead(sensorPin);
// turn the ledPin on
digitalWrite(ledPin, HIGH);
// stop the program for <sensorValue> milliseconds:
delay(sensorValue);
// turn the ledPin off:
digitalWrite(ledPin, LOW);
// stop the program for for <sensorValue> milliseconds:
delay(sensorValue);
}
3
Arduino Uno on COM16

Now connect your Arduino to the computer. Use a USB cable for this, and you do not need to connect battery power. You only need this one cable. Now the computer will detect new hardware and install drivers.

When the equipment is ready for use, check which COM port has been assigned to your board. You can do this by going to the device manager:

Computer -> Properties -> Device Manager

Now we can go back to the Arduino IDE settings. Here we have to choose two options. First of all, we tell the compiler which board we use. Then indicate the previously checked COM port number.

The Arduino IDE only shows the available COM ports. For me it was COM1 and COM21, you can have many more. The main thing is to choose the right one.

Board connected and set up. Now we can proceed to upload the program. To do this, we need to choose two options:

- Verify
- Load

The first one is the equivalent of "Compiles" found in other environments. This process is responsible for checking the correctness of the code and its compilation, or conversion into a language understandable for programmed electronic devices.

The second command is responsible for sending the program to the Arduino UNO board. After clicking the Load button, the LEDs labeled TX and RX should flash on the board. This means that the data is transferred from/to the computer.

When the process is done correctly, you'll find the appropriate message at the bottom of the Arduino IDE. There will be information that the program has been sent and how much space has taken up in the memory of our microcontroller - in this case, it was 1 084 bytes.

The fact that the program has been sent correctly can also be seen on Arduino. As I mentioned earlier, we have 1 LED on the board. After

uploading the program, it should flash.

The effect is not thrilling, but we know that everything works. Next, we'll start writing programs ourselves. In the meantime, you can try to edit the code we have downloaded and test how the system will behave.

Now here it’s enough "long introduction" to the book. I hope that it explains the basic issues and encourages you to follow the next chapter.

Chapter 1

Basic Concepts of Programming

Conversions

Often, we need to convert the numbers to different bases. We will find two methods and it’s good to know both. The first will teach you how to do the conversions "by hand,” allowing you to understand things well. The second that of the calculator will allow you to make conversions without getting tired.

Decimal <-> Binary

To convert a decimal number (in base 10) to a binary number (in base 2, you follow that's good.), You need to know how to divide by 2. Will it be okay? Take your number, then divide it by 2. Then divide the quotient obtained by 2, then so on until you have a zero quotient. Then you have to read the remains from bottom to top to get your binary number.

Binary <-> Hexadecimal

Converting from binary to hexadecimal is the easiest to do.

First, start grouping the bits in blocks of four starting at the right. If there are not enough bits left to make the last group of four, we add zeros.

Take the number 42, which is written in binary, as we have seen, 101010, and we will get two groups of 4 bits, which will be 0010 1010. Then, calculate block by block to obtain a hexadecimal digit taking into account the value of each bit. The first bit of low weight (far right) will be worth, for example, A (: A in hexadecimal). Then the other block will simply be worth 2 (). So 42 in the decimal base is worth 2A in the hexadecimal base, which is also written to pass from hexadecimal to binary, it suffices to do the reverse operation with the help of the decimal base from time to time. The procedure to follow is as follows:

- I separate the numbers one by one (we get 2 and A)

- I "convert" their values to decimal (which makes us 2 and 10)
- I put these values in binary (and we, therefore, have 0010 1010)

Decimal <-> Hexadecimal

This case is more delicate to treat because it requires to know the multiplication table by 16. As you have followed the previous explanations, you understand how to do it here. As I am bad at math, I would advise you to go through the binary base to do the conversions.

QUICK Method

For this, go to Start / All programs / Accessories / Calculator. Who Said I Was Lazy?

You see at the top, that there are options to check to display the number entered in the base that we want. Currently, I am in base 10 (decimal - Dec button). If I click on Hex:

I see that my number 42 has been converted to 2A. And now if I click on Bin:

Yes, that's right. Why didn't we start by explaining that? Who knows.

Now that you have acquired the essential basics to continue the book, we will see how the material you bought looks and which we will need to follow this book.

The Software

To give you a little extra time to get your Arduino board, I will briefly show you how the Arduino software looks.

Installation

There is no need to install the Arduino software on your computer since it is a portable version. Let's look at the steps together to prepare your computer for using the Arduino board.

Downloading

To download the software, go to the download page of the Arduino.cc website. You have two categories:

Download: In this category, you can download the latest version of the software. Windows platforms,

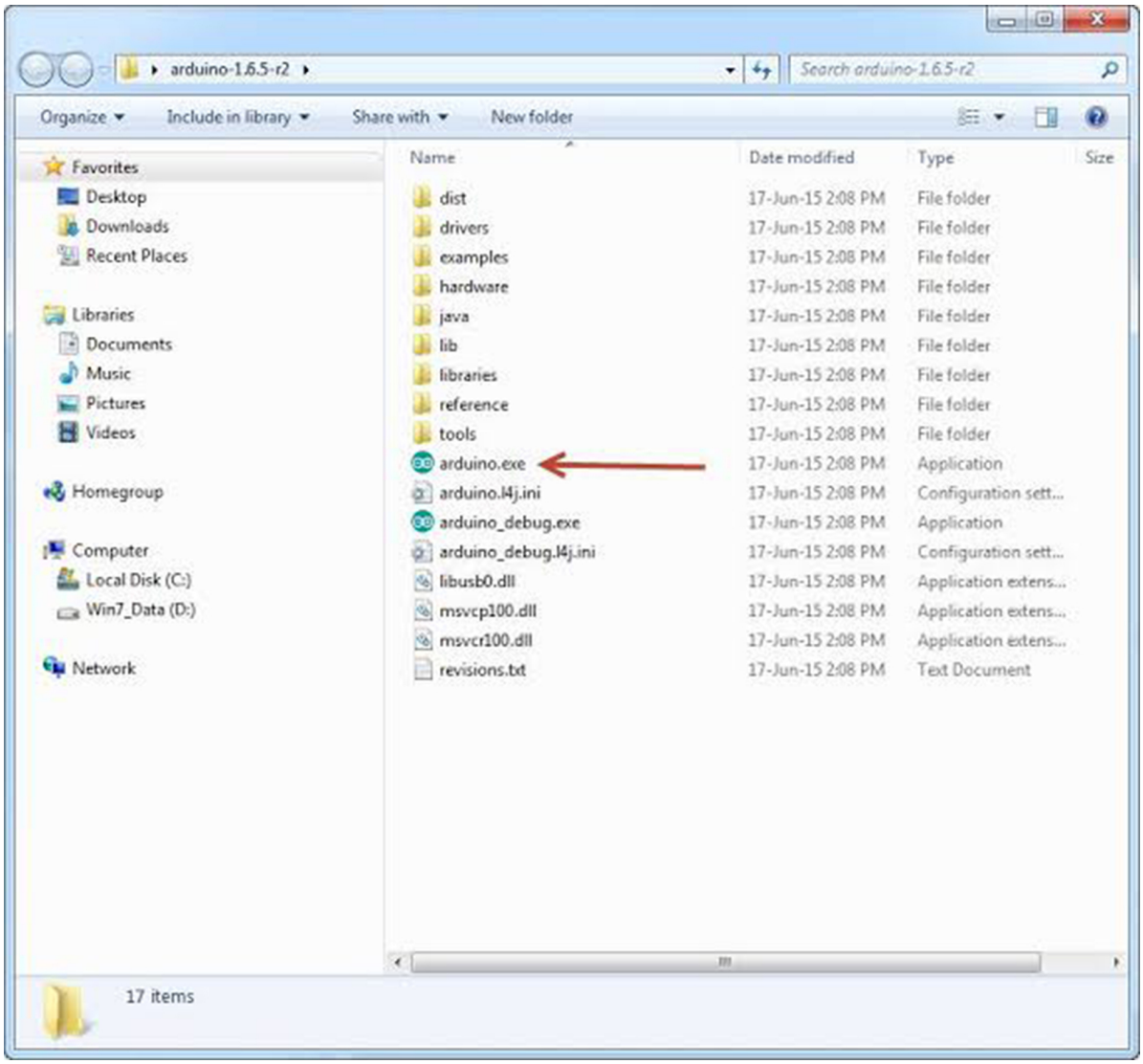
The software supports Linux and Mac. This is where you will download the software.

Previous IDE Versions: In this category, you have all the versions of the software, under the platforms previously mentioned, since the beginning of its creation.

Windows

For me, it will be under Windows. I click on the Windows link, and the file appears:

Once the download is complete, you have to decompress the file with a decompression utility (7-zip, WinRar, etc.). Inside the folder are a few files and the software executable:



Mac Os

Click on the Mac OS link. A.dmg file appears. Save it.

Double-click on the.dmg file: There is the Arduino application (.app), but also the driver to install (.mpkg). Proceed with the driver installation then install the application by dragging it into the shortcut of the "Applications" folder, which is normally present on your computer.

Under Linux

Nothing could be simpler, by going to the software library, look for the "Arduino" software. Several outbuildings will be installed at the same time.

Software Interface Software Launch

Launch the software by double-clicking on the icon with the "infinite" symbol in green. It is the executable of the software. What jumps out at first is the clarity of the presentation of the software. We immediately see its intuitive interface.

Correspondence

Frame number 1: these are the software configuration options

Frame number 2: it contains the buttons that will serve us when we are going to program our cards Frame number 3: this block will contain the program that we are going to create

Frame number 4: This is important because it will help us to correct the faults in our program. It's a debugger.

Approach and Use of the Software

Let's take a more serious look at the use of the software. The menu bar is surrounded in red and numbered by number 1.

The File Menu

It is mainly this menu that we will use the most. He has several things that are going to be very useful to us:

New: will allow you to create a new program. When this button is pressed, a new window, identical to this one, appears on the screen.

Open: with this command, we will be able to open an existing program

Save / Save as: save the current document/request where to save the current document

Examples: this is important; a whole list is scrolled to display the names of examples of existing programs; here, you can help you create your own programs.

The rest of the menus are not interesting for the moment, we will come back to this later, before starting to program.

Buttons

Now let's see what the buttons are for, framed in red and numbered by the number 2.

Button 1: This button allows you to check the program, it activates a module that searches for errors in your program

Button 2: Create a new file

Button 3: Save the current program Button 4: We do not touch it for the moment Button 5: Stop the verification

Button 6: Load an existing program

Button 7: Compile and send the program to the card

Finally, we will be able to take care of the equipment that you should all have right now: the Arduino board.

Equipment

I hope that you now have the equipment required to continue the book because, in this chapter, I will show you how your card looks, then how to test it to verify that it works.

Presentation of the Card

To start our discovery of the Arduino board, I will present the board itself. We will see how to use it and with what. I have represented in red on this photo the important points of the map.

Constitution of the Card

Let's see what these important points are and what they are used for.

The micro-controller - This is the brain of our map (in 1). It will receive the program you have created and store it in the memory and then execute it. Thanks to this program, it will know how to do things suchs as: flashing an LED, displaying characters on a screen, sending data to a computer, etc.

Materials

To operate, the card needs a power supply. The microcontroller operating under 5V, the card can be supplied with 5V by the USB port (in 2) or by an external power supply (in 3), which is between 7V and 12V. This voltage must be continuous and can, for example, be supplied by a 9V battery. A regulator then takes care of reducing the voltage to 5V for the proper functioning of the card. No danger of toasting everything. Please only respect the interval of 7V to 15V (even if the regulator can support more, do not bother to subtract it within its limits)

Visualization

The three "white dots" circled in red (4) are, in fact, LEDs whose size is of the order of a millimeter. These LEDs are used for two things:

The one at the top of the frame: it is connected to a pin of the microcontroller and will be used to test the hardware.

Note: When you connect the card to the PC, it flashes for a few seconds.

The two LEDs at the bottom of the frame: are used to display activity on the serial channel (one for transmission and the other for reception). The downloading of the program in the microcontroller being done in this way, one can see them flashing during the loading.

Connectivity

The Arduino board does not have any components that can be used for a program, put by the LED connected to pin 13 of the microcontroller, and it is necessary to add them. But to do this, you must connect them to the card.

For example, we want to connect an LED to an output of the microcontroller. connect it, with a resistor in series, to the card, on the card's connection plugs.

This connection is important and has a pinout that must be respected. We will see it when we learn to do our first program. It is with this connection that the card is "expandable," because you can connect all types of assemblies and modules. For example, the Arduino Uno board can be extended with shields, such as the "Ethernet Shield," which allows the latter to be connected to the internet.

Installation

To use the card, it must be installed. Normally, the drivers are already installed under GNU / Linux. Under Mac, double click on the.mkpg file included in the download of the Arduino application, and the installation of the drivers is executed automatically.

Windows

When you connect the card to your computer on the USB port, a small message appears at the bottom of the screen. Theoretically, the card you use should install itself. However, if you are on Win 7 like me, it may not work the first time. In this case, leave the card connected and then go to the control panel. Once there, click on "system," then in the left panel, select "device manager." Once this menu is open, you should see a component with a yellow "caution" sign. Right-click on the component and click "Update Drivers." In the new menu, select the option "Search for the driver myself." Finally, you have to select the right folder containing the driver. It is in the Arduino folder that you had to unzip a little earlier and is called "drivers" (be careful, don't go down to the "FTDI" folder). For example, for me, the path will be:

[The way-to-folder] \ Arduino-0022 \ Arduino-0022 \ drivers

It seems that there are problems using the French version of Arduino (the drivers are missing from the file). If this is the case, you will need to download the original version (English) to install the drivers. After installation and a sequence of flashing on the micro-LEDs of the card, it should be functional; a small green LED indicates that the card is properly powered.

Test your Equipment

Before starting to program the head down, it is first of all necessary to test the correct functioning of the card. Because it would be silly to program the card and look for errors in the program when the problem comes from the card. > <We will test our hardware by loading a program that works on the card. But we haven't made any programs yet?

Barely. But the Arduino software contains sample programs. Well, these are the examples that we will use to test the card.

1st step: Open a Program

We are going to choose a very simple example that consists of making a LED flash. His name is Blink, and you will find him in the Basics category.

Last Step

Now, we will have to send the program to the card. To do this, click on the Upload button.

At the bottom of the image, you see the text: "Uploading to I / O Board...," this means that the software is sending the program to the card.

The message display: "Done uploading" indicates that the program has been loaded onto the card. If your hardware is working, you should have a flashing LED on the board:

If you do not get this message but rather something in red, do not worry, the equipment is not necessarily defective.

Indeed, several errors are possible:

- IDE recompiles before sending code, check for error

- The serial channel may be badly chosen, check the connections and the choice of the serial channel

- the IDE is coded in JAVA, it can be capricious and bug from time to time (especially with the serial channel...): try the sending again.

The Arduino Language (1/2)

To be able to program our card, we need three things:

- A computer
- Arduino board
- And know the Arduino language

It is this last point that we must acquire. The purpose of this chapter is to teach you how to program with the Arduino language. However, this is only a book material that you can browse when you have to program your card by yourself. Indeed, it is by manipulating that you learn, which implies that your programming learning will be more consistent in the next chapters than in this book itself.

I specify a small hazard: the Arduino language not having the coloring of its syntax in the zCode. I will put it as C code because their syntax is very close:

Code: C

```
// here is colored Arduino code thanks to the "code: C" tag in zCode
void setup ()
{
//...
}
```

The Arduino language is very close to C and C ++. For those whose knowledge of these languages is based, do not feel obliged to read the two chapters on the Arduino language. Although there are some somewhat important points.

Language syntax

The syntax of a programming language is the set of writing rules linked to this language. We will, therefore, see in this section the rules which govern the writing of the Arduino language.

The Minimum Code

With Arduino, we have to use minimal code when creating a program. This code allows dividing the program that we are going to create into two big parts.

Code: C

```
void setup ()
{
// card initialization function
// content of initialization
```

```
}
void loop () // main function, it repeats (runs) endlessly
{
// content of your program
}
```

So you have before you the minimum code to insert into your program. But what can it mean for someone who has never programmed?

Function

In this code, there are two functions. Functions are portions of code.

Code: C

```
void setup ()
// card initialization function
{
// content of initialization
// we write the code inside
}
```

This setup () function is called only once when the program starts. This is why it is in this function that we will write the code which only needs to be executed once. This function is called: "initialization function." We will find the implementation of different outputs and some other settings. It's a bit like the start-up check-up. Imagine an airplane pilot in his cabin doing the inventory:

- leg 2 at the exit, high state?

- OKAY

- timer 3 to 15 milliseconds?

- OKAY

Once you have initialized the program, you must then create your "heart," in other words, the program itself.

Code: C

```
void loop () // main function, it repeats (runs) endlessly
```

```
{
// content of your program
}
```

It is, therefore, in this loop () function where we will write the content of the program. It should be known that this function is called permanently, that is to say, that it is executed once, then when its execution is finished, it is re-executed and again and again. We are talking about an infinite loop.

For your information, you don't have to write anything on these two functions. However, it is mandatory to write them, even if they do not contain any code.

The Instructions

In these functions, what do we write? This is precisely the purpose of this paragraph.

In your list for dinner tonight, you write down the important tasks that await you. These are instructions. The instructions are lines of code that say to the program: "do this, do that." It is very simple but very powerful because it is what will orchestrate our program.

The semicolons end the instructions. If, for example, I say in my program: "call the function cutFromSausage" I must put a semicolon after calling this function.

Semicolons (;) are synonymous with errors because they are very often forgotten at the end of the instructions. Therefore, the code does not work, and the search for the error can take us considerable time. So be careful.

The Braces

The braces are the "containers" of the program code. They are specific to functions, conditions, and loops. The program instructions are written inside these braces. Sometimes they are not mandatory in the conditions (we will see below what it is), but I recommend to put them all the time. This will make your program more readable.

Comments

Finally, we will see what a comment is. I already put some in the example codes. These are lines of code that will be ignored by the program. They are useless during the execution of the program. But then is it useless?

No, because this will allow the programmers who will read your code and us (if there are any) to know what the line of code you wrote means. It is very important to put comments, and it also makes it possible to resume a forgotten program more easily.

If, for example, you are unfamiliar with an instruction that you have written in your program, you put a comment line to remind you the next time you read your program what the line means.

Single comment line:

Code: C

// this line is a comment on ONLY ONE line

Line or paragraph on several lines:

Code: C

/ * this line is a comment, on SEVERAL lines

which will be ignored by the program, but not by anyone who reads the code

;) * /

Accents

It is strictly forbidden to put accents in programming, except in the comments.

The Variables

As we have seen, in a microcontroller, there are several types of memory. We will only deal with "live" memory (RAM) and "read-only" memory (EEPROM).

I'm going to ask you about a problem. Let’s say you’ve connected a push button to a pin on your Arduino board. How will you store the state of the button (pressed or off)?

What is a Variable?

A variable is a number. This number is stored in a memory space (RAM) of the microcontroller. The way to store them is similar to that used to store shoes: in a numbered locker.

Shoes stored in numbered boxes

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60

A Variable is a Number

This number has the distinction of changing its value. Strange, isn't it? Well, not that much, because a variable is the container of the number in question. And this container will be stored in a memory box. If we materialize this explanation by a diagram, it will give:

number => variable => memory

the symbol "=>" meaning: "is contained in..."

The Name of a Variable

The variable name accepts almost all characters except:

. (point)

, (the comma)

é, à, ç, è (the accents)

Well I will not give them all, it only accepts the alphanumeric alphabet ([a-z], [A-Z], [0-9]) and _ (underscore)

Define a Variable

If we give a number to our program, it doesn't know if it's a variable or not. You have to tell it. For that, we give a type to variables. Yes, because there are several types of variables. For example, the variable "x" has a value of 4, so it is an int type.

Code: C

Well, this code would not work because it is not enough. Indeed, there is a multitude of numbers: whole numbers, decimal numbers, etc. This is why we must assign a variable to a type.

These are the most common types of variables:

int integer -32,768 to +32,767 16 bit 2 byte

long integer -2 147 483 648 to +2 147 483 647 32 bits 4 bytes

whole char -128 to +127 8 bits 1 byte

decimal float -3.4 x to +3.4 x 32 bit 4 byte

double decimal -3.4 x to +3.4 x 32 bit 4 byte

For example, if our variable "x" only takes decimal values, we will use the types int, long, or char. If now the variable "x" does not exceed the value 64 or 87, then we will use the char type.

If, on the other hand, x = 260, then we will use the higher type (which accepts a greater quantity of number) than char, in other words, int or long. But you're not smart; to avoid overspending, we put everything in double or long.

Yes, but no. A microcontroller is not a 2GHz multicore computer, 4GB of RAM. Here we are talking about a system that works with a CPU at 16MHz (0.016 GHz) and 2 KB of SRAM for the RAM. So there are two reasons why you should choose your variables wisely:

- RAM is not extensible when there is more, and there is more.

- The processor is of the 8-bit type (on Arduino UNO), so it is optimized for processing on variables of 8-bit size, processing on a 32-bit variable will, therefore (much) take longer.

If our variable "x" never takes a negative value (-20, -78,...), then we will use an unsigned type. That is, in our case, a char whose value is no longer from -128 to +127, but from 0 to 255.

Here is the table of unsigned types, we identify these types by the word unsigned which precedes them:

Extended Precision Operations

To represent larger numbers, more bits are needed.

N bits can represent the unsigned range 0 to 2^N-1.

Bytes 1 Byte = 8 bits	Unsigned Range	*C* Data Type (PIC16)
1 (8 bits)	0 to 255	char
2 (16 bits)	0 to 65,535	int
4 (32 bits)	0 to 4,294,967,295	long

The size of *int*, *long* depends on the *C* implementation; on some machines both *int* and *long* are 4 bytes, with a *short int* being 2 bytes. On some machines a *long* is 8 bytes (64 bits).

V 0.1 1

Type what number does it store? Maximum values of the stored number, Number of bits and Number of bytes like below:

- unsigned non-negative integer char 0 to 255 - 8 bits, 1 byte
- unsigned int non-negative integer 0 to 65,535 - 16 bits, 2 bytes
- unsigned long non-negative integer 0 to 4,294,967,295 - 32 bits, 4 bytes

One of the peculiarities of the Arduino language is that it accepts a larger number of types of variables.

Type what number does it store? Maximum values of the stored number, Number on X bits Number of bytes like below:

- byte non-negative integer 0 to 255 8 bits 1 byte
- non-negative integer word 0 to 65535 16 bit 2 byte
- non-negative integer boolean 0 to 1 1 bit 1 byte

The Arduino Language (2/2)

I have a question. If I want to make the code I wrote repeat itself, do I have to copy it as many times as I want? Or is there a solution?

This is an excellent question because it is precisely the object of this chapter. We will see how to make a piece of code repeat itself. Then we will see, next, how to organize our code so that it becomes more readable and easy to debug. Finally, we will learn to use the tables that will be very useful.

Here is the program waiting for you.

```
import processing.video.*;
Movie movie;
void setup()
{
size(640,360);
background(0);
//load and play the video in a loop
movie = new Movie(this,"transit.mov");
movie.loop();
}
void movieEvent(Movie m)
{
m.read();
}
void draw()
{
//if (movie.available() == true)
//{
```

```
// movie.read();
//}
image(movie,0,0,width,height);
}
```

What is a Loop?

In programming, a loop is an instruction that makes it possible to repeat a piece of code. This will allow us to repeat an end of program or an entire program.

There are two main types of loops:

The conditional loop tests a condition and executes the instructions it contains as long as the condition tested is true.

The repeat loop executes the instructions it contains a predetermined number of times.

The while loop

Problem: I want the electric shutter of my window to close automatically when night falls. We will not take care of making the system that closes the shutter at the arrival of the night. The Arduino board has a sensor which indicates the position of the flap (open or closed). What we are trying to do is create a piece of code that brings the shutter down until it is closed.

To solve the problem, we will have to use a loop.

/ * HERE, a bit of program allows you to do the following things:

_ a sensor detects nightfall and daybreak o If it is night, then we must close the shutter

o Otherwise, if it is the day, we must open the shutter

_ the program reads the state of the sensor which indicates whether the shutter is open or closed

_ recording of this state in the String variable: position_volet

o If the shutter is open, then: position_volet = "open";

o Otherwise, if the shutter is closed: position_volet = "closed";

* /

while (position_volet == "open")

{

// instructions that lower the shutter

}

The do... while loop

This loop is similar to the previous one. But there is a difference that matters. In fact, if we pay attention to the condition in the while loop, we will notice that it is tested before entering the loop. While loop, the condition is tested only when the program has entered the loop:

Code: C

do

{

the instructions between these braces are repeated as long as the condition is false } while (/ * condition to test * /);

The word does come from English and translates as do. So the do while loop means "do the instructions, as long as the condition tested is false." While in a while loop, you could say, "as long as the condition is false, do the following."

What Does it Change?

Well, in a while loop, if the condition is true from the start, we will never enter this loop. Conversely, with a loop do while we enter the loop, then we test the condition. Let's take our counter:

```
let i = 5;
do {
  alert( i );
  i++;
} while (i > 5);
```

In this code, the counter value is defined from the start at 5. However, the program will enter the loop when the condition is false. So the loop is executed at least once. And whatever the veracity of the condition. In the test it gives:

Concatenation

A loop is an instruction that has been spread over several lines. But we can write it on one line:

while(…) {…}

This is why you should not forget the semicolon at the end (after the while). While in a simple while loop, the semicolon should not be put.

The for loop

This is a very special loop. What it will allow us to do is quite simple. This loop is executed X times. Unlike the two previous loops, we must give it three parameters.

First, we create the loop with the term for (means "for that"). Then, between the parentheses, we must give three parameters which are creation and assignment of the variable to a starting value followed by the definition of the condition to be tested followed by the instruction to execute

The Arduino language does not accept the absence of the following line:

We are forced to declare the variable we are going to use (with its type) in the for a loop.

So, if we link this line: "FOR counter = 0 and counter less than 5, we increment counter". More concisely, the loop is executed as many times as it takes counter to get to 5. So here, the code inside the loop will be executed five times.

The infinite loop

The infinite loop is very simple to achieve, especially since it is sometimes very useful. You need to use a while and assign it a value that never changes. In this case, we often put the number

We can read: "As long as the condition is equal to 1, we execute the loop". And this condition will always be met since "1" is not a variable but a number. Also, it is possible to put any other whole number, or the boolean "TRUE":

```
do {
   Block of statements;
}
while(1);
```

This will not work with the value 0. Indeed, 0 means "false condition," so the loop will stop immediately... The loop () function behaves like an infinite loop since it repeats itself after it has finished executing its tasks.

The Functions

In a program, the lines are often numerous. It then becomes imperative to separate the program into small pieces to improve the readability of it, in addition to improving the operation and facilitating debugging. We will see together what a function is, and then we will learn how to create and call them.

What is a Function?

A function is a "container" but different from the variables. Indeed, a variable can only contain a number, while a function can contain an entire program.

In fact, when we program our Arduino board, we will write our program in functions. For the moment, we only know 2: setup () and loop ().

In the previous example, instead of the comment, we can put instructions (conditions, loops, variables, etc.). This is these instructions that will make up the program in itself.

To be more concrete, a function is a piece of program that allows you to perform a specific task. For example, to format a text, you can color a word in blue, put the word in bold or even enlarge this word.

```
public class Foo { // Program by Joe
   public static void main(String[] args){
      String helloText = "Hello world";
      String goodbyeText = "Goodbye world";
      System.out.println(helloText);
      System.out.println(goodbyeText);
   }
}
```

Bold - to put the word in bold color, to put the word in blue enlarge, to increase the size of the word in programming, we will use functions. So these functions are "divided into two large families." What I mean by that is that there are ready-made functions in the Arduino language and others that we will have to create ourselves.

You cannot write a program without putting functions inside. We are forced to use the setup () and loop () function (even if we don't put anything in it). If you write instructions outside of a function, the Arduino software will systematically refuse to compile your program. There are only global variables that you can declare outside of the functions.

I did not really understand what it is for

The usefulness of a function lies in its ability to simplify the code and separate it into "little bits" that we will assemble to create the final program. If you want, it's a bit like plastic building sets: each piece has its own mechanism and performs a function.

For example, a wheel makes it possible to roll; a block makes it possible to join several other blocks together; a motor will move the created object forward. Well, all these elements will be assembled to form an object (car, house, etc.). like, the functions will be joined together to form a program. For example, we will have the function: "squaring a number"; the function: "add a + b"; etc. Which, in the end, will give the desired result.

Make a Function

To make a function, we need to know three things:

- What type of function do I want to create?
- What will his name be?
- What parameter (s) will it take?

Function Name

To start, we will, first, choose the name of the function. For example, if your function has to retrieve the temperature of a room supplied by a temperature sensor: you will call the function RoomTemperaturePiece, or room_temperature_piece, or lecture_temp_piece.

Well, names can be given full, but be logical about the choice of the latter. It will be easier to understand the code than if you call it tmp (for temperature). An explicit function name guarantees quick reading and easy understanding of the code. A reader should know what the function is doing by name, without reading the content.

Types and Parameters

The purpose of the functions is to split your program into different logical units. Ideally, the main program should only use function calls, with minimal processing. To be able to function, they mostly use "things" as inputs and return "something" as outputs. The inputs will be called function parameters, and the output will be called the return value.

The Settings

The parameters are used to feed your function; they are used to give information to the process it must carry out. Let’s take a concrete example.

To change the state of an output of the microcontroller, Arduino offers the following function: digitalWrite (pin, value). Thus, the reference explains to us that the function has the following characteristics:

- pin parameter: the number of the pin to be changed
- value parameter: the state in which to put the spindle (HIGH, (high, + 5V) or LOW (low, ground))
- return: no return of result

As you can see, the example is self-explanatory without reading the function code. Its name, digitalWrite ("digital writing" for anglophobes), means that we will change the state of a digital pin (therefore not analog). Its parameters also have explicit names, pin for the pin to change, and value for the state to give it.

When you go to create functions, it is up to you to see if they need parameters or not. For example, you want to do a function that pauses your program, and you can do a Pause () function that takes a parameter of type char or int, etc. (this will depend on the size of the variable). This variable will, therefore, be the parameter of our Pause () function and will determine the duration during which the program will be paused.

We will, therefore, obtain, for example, the following syntax: void Pause (char duration).

To sum up a bit, we have the choice of creating empty functions, therefore without parameters, or else "typed" functions that accept one or more parameters.

But what is "Void"?

I arrive there. Remember, a little above, I explained to you that a function could return a value, the famous output value, I will now explain how it works.

Empty Functions

We have seen that a function could accept parameters. But it is not compulsory. A function that does not accept parameters is empty.

We, therefore, use the void type to say that the function will have no parameters. A void type function cannot return a value. For example:

```
void Arrange(int left,int right){
    int i,j,x,w;
    i=left,j=right;
    x=(left+right)/2;
    do{
        while(struct[i].number < struct[x].number)i++;
```

```
        while(struct[j].number > struct[x].number)j--;
        if(i<=j){
            w=struct[i].number;
            struct[i].number=struct[j].number;
            struct[j].number=w;
            i++;j--;
    }}while(i<=j);
      if(left<j)
    return Arrange(left,j); //1st recursive call. It doesn't work.
      if(right>i)
    return Arrange(i,right); //2nd recursive call.It doesn't work either.
      };
```

This code will not work because the function () is of type void. However, it must return a variable that is of type int. Which is impossible.

The "Typed" Functions

Indeed, if we want to create a function that calculates the result of an addition of two numbers (or a more complex calculation), it would be good to be able to return the result directly rather than to store it in a variable which has a global scope and access that variable in another function.

Clearly, calling the function gives us the result directly. We can then do "what we want" with this result (store it in a variable, use it in a function, make it undergo an operation,...)

How to Create a Typed Function

In itself, this is not complicated, and you have to replace the void with the chosen type (int, long, etc.) Here is an example:

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

void loop() {
  int i = 10;
  int j = 44;
  int k;
```

```
  k = myMultiplyFunction(i, j); // k now contains 440
  Serial.println(k);
  delay(500);
}

int myMultiplyFunction(int x, int y){
  int result;
  result = x * y;
  return result;
}
```

Note that I have not put the two main functions, namely setup () and loop (), but they are mandatory.

In the loop () function, we do a calculation with the value returned by the function myFunction (). In other words, the calculation is: calculation = 10 * 44; Which gives us: calculation = 440.

Well, this is a very simple example to show you how it works. Later, when you are ready, you will certainly use this combination in a more complex way.

As this example is very simple, I did not write the value returned by the function myFunction () in a variable, but it is better to do so. At least, when it is useful, which is not the case here?

Functions with Parameters

It's all very nice, but now you're going to see something much more interesting. Here is a code; we will see what it does after:

```
void loop(){
int i = 2;
int j = 3;
int k;

k = myMultiplyFunction(i, j); // k now contains 6
}
```

What is going on?

I defined three variables: sum, x, and y. The function of my Function () is "typed" and accepts parameters. Let's read the code at the beginning:

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

void loop() {
  int i = 2;
  int j = 3;
  int k ;

  k = myMultiplyFunction(i, j); // k now contains 6
  Serial.println(k);
  delay(500);
}

int myMultiplyFunction(int x, int y){
  int result;
  result = x * y;
  return result;
}
```

We Declare our Variables

The loop () function calls the myFunction () function that we created

It is on this last point that we will look. We gave the parameters function. We use these parameters to "feed" the function. To put it simply, we say to the function: "These are two parameters, I want you to use them to do the calculation I want."

Then comes the signing of the function

The signature... what are you talking about?

The signature is the "full title" of the function. Thanks to it, we know the name of the function, the type of the returned value, and the type of the different parameters.

The function retrieves the variables sent to it from variables. In other words, in the variable param1, we find the variable x. In the variable param2, we

find the variable y.

Either: param1 = x = 64 and param2 = y = 192.

Finally, we use these two variables created "on the fly" in the function's signature to perform the desired calculation (a sum in our case).

What's the point of doing all this? Why don't we use the variables x and y in the function?

This will help us to simplify our code. For example, you want to do several operations (addition, subtraction, etc.), and instead of creating several functions, we will only create one that does them all. But, to tell it which operation to do, you will give it a parameter telling it: "Multiply these two numbers" or else "add these two numbers."

What it would do:

So if the operation variable is 0, we add the variables x and y; otherwise, if the operation is 1, we subtract y from x. Simple to understand, isn't it?

The Tables

As the name suggests, this part will talk about tables.

What is the point of talking about this? Well, think again, on the computer a table has nothing to do. If we had a lot to summarize, an array is a big variable. Its purpose is to store elements of the same types by putting them in boxes. For example, a teacher who stores the grades of his students will use a floating table (a floating-point number) with one box per student.

We will use this example throughout this section. Here are some details to understand everything: each pupil will be identified by a number going from 0 (the first pupil) to 19 (the twentieth pupil of the class) one starts from 0 because in data processing the first value in a table is 0.

A Table in Programming

A table, like in Excel, is a set made up of boxes, which will contain information. In programming, this information will be numbered. Each box in an array will contain a value.

What's the Point?

We will mainly use arrays when we need to store information without creating a variable for each information.

The Maximum Score

As the title indicates, we will look for the maximum grade (the best student in the class). The function will receive in parameter the array of float, the number of elements in this array, and will return the best note.

Code: C

```
float bestNote (float array [], int highNumber)

    {
    int i = 0;
    int max = 0; // variables containing the highest grade
    for (i = 0; i <highNumber, i ++)
    {
    if (table [i]> max) // if the grade read is better than the best current
    {
    max = array [i]; // then we save it
    }
    }
    return max; // we return the best grade
    }
```

What you do to read an array is exactly the same as when you initialize it with a loop.

It is quite possible to put the value of the searched box in a variable:

Code: C

```
    int value = array [5]; // we save the value of the box
    6 of the array in a variable
```

Well, it was not that hard; you can do the same to find the minimum value To practice.

Average Calculation

Here, we will look for the average of the grades. The signature of the function will be exactly the same as that of the previous function, unlike the name. Let you think, here is the signature of the function, the code is lower but try to find it yourself before:

Code: C

```
float mediumNote (float array [], int numberHigh)
```

A solution:

Secret (click to display)

Code: C

```
float mediumNote (float array [], int numberHigh)
{
int i = 0;
double total = 0; // add all the grades
average float = 0; // average of grades
for (i = 0; i <highNumber; i ++)
{
total = total + table [i];
}
mean = total / number High;
average return;
}
```

We finish with the tables, we may see more things in practice.

Now you are joyful because you have finished the first part.

You are ready to start using your card. So go to the next part of the book.

Chapter 2

Input / Output Management

Now that you have gained enough programming knowledge and some notions of electronics from the previous chapter and in this chapter, we will look at the use of the Arduino board. I will tell you about the inputs and outputs of the card. We are going to start simply, so do not be surprised if you go quickly to reading the chapters.

Do not neglect the basics; otherwise, you may not be able to follow the more complex chapters. A piece of advice – make sure you understand everything before you move on – there is no rush; everyone should work at their own pace.

Our First Program

You have finally arrived at the fateful moment when you will have to program. But before that, I will show you what will serve us for this chapter. In this case, learn to use an LED and the reference, present on the Arduino.cc site, which will be very useful when you need to make a program using a concept that is not covered in this book.

The LED / LED Light-Emitting Diode

The question is not to know which abbreviation to choose, but to know what it is.

A LED / LED: Light-Emitting Diode, or "Light Emitting Diode”. It is an electronic component that creates light when an electric current flows through it. I made you buy different colors. You can, for this chapter, use whatever you want. You see, a photo of a red LED. The size is not real, its "head" (in red) is only 5mm in diameter. It is this component that we will try to light with our Arduino board. But before, let's see how it works.

I will call the light-emitting diode, throughout the book, an LED. An LED is a diode that emits light. So I will talk to you about how LEDs work at the

same time as LEDs. There is very little difference between the two. The LED is simply a diode that emits light, hence the arrows on its symbol.

Direct Polarization

We speak of polarization when an electronic component is used in an electronic circuit in the "right way." In fact, when it is polarized, it is because we use it in the desired way.

To polarize the diode, we ensure that the current must flow from the anode to the cathode. In other words, the voltage must be higher at the anode than at the cathode.

Reverse Polarization

The reverse bias of a diode is the opposite of forwarding bias. To create this type of assembly, it is simply enough, in our case, to "turn over" the diode finally to connect it "upside down." In this case, the current does not pass.

Note: a reverse-biased diode will not burn out if used in the right conditions. In fact, it works "the same way" for the positive and negative current.

If you do not want to make your first diode go up in smoke, I advise you to read the next lines carefully. In electronics, we must take two parameters into account: current and voltage. For a diode, two voltages are important. These are the maximum voltage in forwarding polarization, and the maximum voltage in reverse polarization. Then, for the LEDs to work properly, the current is also important.

Maximum Direct Voltage

When using a component, we must get into the habit of using the "datasheet" ("technical documentation"), which gives us all the characteristics of the component. In this datasheet, we will find something called "Forward Voltage" for the diode. This indicator represents the voltage drop across the diode when current flows directly through it. For a conventional diode (type 1N4148), this voltage will be around 1V. For a led, we will rather consider a voltage of 1.2 to 1.6V.

Well, to make our small assemblies, we will not quibble, but it is the approach to do when we design an electrical diagram and that we

dimension its components.

Maximum Reverse Voltage

This voltage represents the maximum admissible difference between the anode and the cathode when the latter is connected "upside-down." In fact, if you put too much voltage on these terminals, the junction will not be able to support it and will go up in smoke. We find this voltage under the name of "Reverse Voltage" (or even "Breakdown Voltage"). If we take the diode 1N4148, it will be between 75 and 100V. Beyond this voltage, the junction breaks, and the diode becomes unusable. In this case, the diode becomes either a short circuit or an open circuit. Sometimes this can cause significant damage to our electronic appliances. Anyway, we will never handle 75V.

The Flowing Current

For an LED, the current flowing through it is important. If you connect the led directly to a battery, it will light up, then sooner or later, will eventually go out.

If we do not limit the current flowing through the LED, it will take the maximum current, and that is not good because it is not the maximum current that it can support. To limit the current, a resistor is placed before (or after) the LED. This resistance, expertly calculated, will allow it to ensure optimal operation.

But how do you calculate this resistance?

Simply with the basic formula, ohm's law. Little reminder:

Then, we will take for the example a supply voltage of 5V (at the output of the Arduino, for example) and a voltage at the terminals of the LED of 1.2V in normal operation. We can, therefore, calculate the voltage which will be at the terminals of the resistor:

Finally, we can calculate the resistance value to use:

And voila, you know the value of the resistance to use to be sure not to burn LEDs out of the arm. Is it better to use a higher value or lower value resistor?

Where do we start?

The Goal

The aim of this first program is to light an LED, however, it may feel somewhat tricky here but everything will be in control over time. The plan is to show you two or three things that can help you when you want to leave the nest and take the flight to new skies.

Equipment

To be able to program, you obviously need an Arduino board and an USB cable to connect the board to the PC. But to see the outcome of your program, you will need additional items. In particular, an LED and a resistor.

Production

With the pinout of the Arduino board, you will need to connect the larger tab to the + 5V (5V pin). The smallest tab being connected to the resistor, itself connected to pin number 2 on the card. We could do the opposite, connect the LED to the ground and turn it on by supplying 5V from the signal pin. However, components like microcontrollers do not like to deliver current too much, and they prefer to absorb it. For this, we will, therefore, prefer to power the LED by placing it at + 5V and putting the Arduino pin to ground to pass the current. If we put the pin at 5V, in this case, the potential is the same on each side of the LED, and it does not light up.

Create A New Project

To be able to program our card, we must create a new program. Open your Arduino software. Go to the File menu and choose the Save as... option:

What is it?

Arduino is a project in which the community is very active, offers us on its website a reference. But what is it? Well, it is simply the "user manual" of the Arduino language.

More precisely, an internet page on their site is dedicated to referencing each code that can be used to make a program.

How to use it?

To use it go to the page of their site. What we see when we get to the page is three columns, each with an element that makes up the Arduino languages.

Structure: this column references the elements of the Arduino language structure. It contains conditions, operations, etc.

Variables: As its name suggests, it brings together the different variables that can be used, and certain specified operations

Functions: Here it is everything else, but especially the read / write functions of the pins of the microcontroller (and other very useful functions)

It is very important to know how to use the documentation that Arduino offers us. Because knowing this, you can make programs without having previously learned to use one function or another.

Turn on our LED

1st Step

First of all, we must define the pins of the microcontroller. This step itself makes up two sub-steps. The first being to create a variable defining the pin used, then define if the pin used should be an input of the microcontroller or an output.

The term const means that we define the variable as being constant. We change the nature of the variable which then becomes constant.

The term int corresponds to a type of variable. By defining a variable of this type, it can store a number ranging from - 2149531648 to +2197438647. That is enough for us.

We are, therefore, in the presence of a variable named led_red, which is a constant which can take a value ranging from -2147483648 to +2147483647. In our case, we assign this variable to 2.

When your code is compiled, the microcontroller will know that on its pin number 2, there is a connecting element.

Well, that is not enough to define the pin used. We must now say whether this pin is an input or an output. Yes, because the microcontroller can use some of its pins for input or output. It is fabulous. It suffices simply to interchange ONE line of code to say that it is necessary to use a pin as input (retrieval of data) or as output (sending of data).

This line of code precisely, let’s talk about it. It must be in the setup () function. In the reference, what we need is in the Functions category, then in Digital I/o. I/o for Input / Output, which means: Input / Output.

The function is pinMode (). To use this function, you must send it two parameters: The name of the variable that you defined on the pin.

2nd Step

This second step is to create the content for our program. Whoever is going to replace the comment in the loop () function, achieving our aim: turn on the LED.

Again, we do not snap our fingers to have the program ready. You have to go back to the Arduino reference to find what you need.

Yes, but now we don't know what we want?

We are looking for a function that will allow us to light this LED, so we have to find it. Let's be a bit logical if you don't mind. We know it's a function we need (I said it a moment ago), so we look in the Functions category of the reference.

If we keep our logical mind, we will take care of lighting an LED, so to say what is the output state of pin number 2 where which is connected to our LED. So it's a safe bet that it's in Digital I/o. Well, there is a suspicious function called digitalWrite (). In French, this means "digital writing." It is, therefore, the writing of a logical state (0 or 1).

What is the first sentence in the description of this function? This one: "Write a HIGH or a LOW value to a digital pin." According to our bilingual level, we can translate by Writing a HIGH value or a LOW value on a digital output. Bingo. This is what we were looking for.

What does "HIGH Value or LOW Value" Mean?

In digital electronics, a high level will correspond to a voltage of + 5V, and a so-called low level will be a voltage of 0V (generally ground). Except that we have connected the LED to the positive pole of the power supply so that it lights up, we must connect it to 0V. Therefore, a low state must be set on the pin of the microcontroller. Thus, the potential difference across the LEDs will allow it to light up.

Let's take a look at how digitalWrite () works by looking at its syntax. It requires two parameters. The name of the spindle that we want to put in a logical state and the value of this logical state.

We will therefore write the following code, using this syntax:

Code: C

```
digitalWrite (led_red, LOW); // write output (pin 2) of a LOW state
```

If we test the whole code:

Code: C

```
const int led_red = 2; // definition of pin 2 of the card as a variable
void setup ()
// card initialization function
{
pinMode (led_red, OUTPUT); // initialization of pin 2 as an output
}
void loop () // main function, it repeats (runs) endlessly
{
digitalWrite (led_red, LOW); // write output (pin 2) of a LOW state
}
```

We see the LED light up... It's fantastic.

Now you know how to use the outputs of the microcontroller, so we can get down to business and make our LED flash.

Enter time

It is fine to light an LED, but if it does nothing else, it is not very useful. You might as well plug it directly into a battery (with resistance all the same.). So let's see how to make this LED interesting by making it flash.

For this, we will have to introduce the concept of time. Well, guess what? There is a ready-made function there again. Let's go to practice.

How to do this?

Have a look for yourself; it will help you learn

I let you search a little by yourself, and it will train you.

For those who have attempted to search and have not found the function, here it is: delay ()

A short description of the function will be used to pause the program for a predetermined time.

Use the Command

The function accepts a parameter, which is the time during which we want to pause the program. We must give this time in milliseconds. If you want to stop the program for 1 second, you will have to give the function at the same time, written in milliseconds, or 1000ms.

Practice: Flash an LED

So, if we want to make our LED flash, we will have to use this function.

The LED lights up. Then, we use the delay () function, which will pause the program for a certain time. Then we turn off the LED. We pause the program. Then we return to the start of the program. We start again, and so on. It is this command sum, which forms the process that causes the LED to flash.

Henceforth, get into the habit of making this kind of diagram when you make a program.

Now we need to translate this diagram, bearing the name of the organization chart, into code. You have to replace the sentences in each frame with a line of code.

The Program

Normally, its design should not cause you any problems. It suffices to retrieve the code of the previous program ("light a group of LEDs") and

modify it according to our needs.

This code, I give it to you, with the comments that go well:

Code: C

```
// we keep the same start as the previous program
const int L1 = 2; // pin 2 of the microcontroller is now called: L1
const int L2 = 3; // pin 3 of the microcontroller is now called: L2
const int L3 = 4; //...
const int L4 = 5; const int L5 = 6; const int L6 = 7;
void setup ()
{
pinMode (L1, OUTPUT); // L1 is a pinMode output pin (L2, OUTPUT); // L2 is a pinMode output pin (L3, OUTPUT); //...
pinMode (L4, OUTPUT); pinMode (L5, OUTPUT); pinMode (L6, OUTPUT);
}
// we  change the inside of the loop to reach our goal
void loop () // the loop () function executes the following code by repeating it in a loop
{
digitalWrite (L1, LOW); // switch on L1 delay (1000); // wait 1 second
digitalWrite (L1, HIGH); // we extinguish L1
digitalWrite (L2, LOW); // we switch on L2 at the same time as we switch off L1
delay (1000); // we wait 1 second digitalWrite (L2, HIGH); // we turn off L2 and digitalWrite (L3, LOW); // we immediately turn on L3 delay (1000); //...
digitalWrite (L3, HIGH); digitalWrite (L4, LOW); delay (1000); digitalWrite (L4, HIGH); digitalWrite (L5, LOW); delay (1000); digitalWrite (L5, HIGH); digitalWrite (L6, LOW); delay (1000); digitalWrite (L6, HIGH);
}
```

As you can see, this code is very heavy and not practical. We will see later how to make it lighter. But before that, a TP arrives...

Millis () Function

We will end this chapter with a point that can be useful, especially in certain situations where we do not want to stop the program. If you want to flash an LED without stopping the execution of the program, you cannot use the delay () function, which pauses the program for the defined time.

Limits of the Delay () Function

You have probably noticed that when you use the "delay ()" function, our whole program stops waiting for you to wait. Sometimes, this is not a problem, but in some cases, it can be more troublesome.

Imagine, you are moving a robot forward. You put your motors at medium speed, quiet, until a small button on the front is pressed (it clicks when you touch a wall, for example). During this time, you decide to make signals by flashing your LEDs. To make a nice flash, you turn on a red LED for one second and then turned it off for another second. This is, for example, what we could do as code:

Code: C

```
void setup ()
{
pinMode (motor, OUTPUT); pinMode (led, OUTPUT); pinMode (button, INPUT);
digitalWrite (motor, HIGH); // we start the engine
digitalWrite (led, LOW); // we turn off the LED
}
void loop ()
{
if (digitalRead (button) == HIGH) // if the button is clicked (we enter a wall)
{
digitalWrite (engine, LOW); // we stop the engine
}
else // otherwise we blink
{
digitalWrite (led, HIGH); delay (1000); digitalWrite (led, LOW); delay (1000);
}
```

Please note this code is not at all rigorous or even false in its writing; it serves to understand the principle.

Now imagine you drive, test the button that is not pressed, so blink the LEDs (case of else). The time you do, the entire display runs out two long seconds. The robot has been able to take the wall in the middle of this pearl for eternity, and the motors continue to advance head down until smoking. It is not good at all.

Chapter 3

Programming With Arduino

This chapter starts much theory, we begin directly with programming. "Learning by doing" is the magic word here. While the left side of the so-called "sketches" are printed and are located in the right pane Reviewing the code.

Anyone who works through the programs under this system will be able to see through the program code in a short time and apply yourself. Later even become familiar with other functions you then. This guide is only an introduction to the Arduino platform. All possible program functions or

Program code is on the website "www.arduino.cc " under the point called "reference."

Before we start a piece of brief information on possible errors that might arise while working with the Arduino software. The most common are the following two:

1. The board is not installed properly or selected the wrong board. When Upload, it displays the sketch an error message at the bottom of the software that roughly looks like as shown at right. In error, the text is then a Noted "not in sync."

2. There is an error in the sketch. For example, a word is misspelled, or it lacks only a clamp or a semicolon. In the example, the left brace that introduces the loop part is missing. The error message then begins often with "expected...” This means that the program expects something that does not yet exist.

The Basic Structure of a Sketch

We may divide a sketch into three areas.

1. Variables to Name

In the first area, elements have named the program (What that means, we learn in the program No. 3). This part is not mandatory.

2. Setup (essential in the program)

The setup is performed by the Board only once. Here, dividing the program with, for example, which pin is (slot for cable) on the microcontroller board, an output or an input.

Here, a voltage at the output Board shall: defined as the output. Example: This pin an LED to be illuminated.

Defined as input: Here, a voltage to be read by the Board. Example: a switch is pressed, and the Board noted the fact that he recognizes this input pin voltage.

3. Loop (essential in the program)

The loop portion is repeated continuously by Board. It processes the sketch once completely to the end and then starts again at the beginning of the loop part.

Instructions

A Flashing LED

Task: A light-emitting diode to blink.

Material: Only the microcontroller board with the USB cable.

On the Arduino, an LED is connected to pin 13 is already installed (for test purposes). Often this light blinks on when you connect a new Arduino board, as the flashing program is already pre-installed for testing the board depending on the manufacturer. We will program this flashing now itself.

Circuit

The existing on the board

LED is circled in red in the diagram.

You have to connect to the computer, only the board via an USB cable.

1.1 Program Section 1: variable name

- Here we go, first nothing.

1.2 Program Section 2: Setup

We have only one exit - to pin 13 to a voltage output (the LED will light up eventually.).

We write in the middle of the white box the Arduino software:

```
void set up()   // Here the setup starts

{  // brace on - Here begins a program section.

}  // curly bracket - Here ends a program section.
```

In part between the braces, we now introduce the setup information. In this case, "an output pin 13 is supposed to be."

```
void set up() // Here the setup starts
{  // Here a program section starts.
pinMode(13, OUTPUT); //          Pin 13 is what the output should be.
}  // Here, a program segment ends.
```

1.3 Program section 3: Loop (main body)

```
void set up()   // Here the setup starts
{  // Here a program section starts. pinMode(13, OUTPUT);              //
pin 13 is to be a starting
} // Here a program segment ends. void loop()   // Here the main
program begins
{  // Here a program section starts.
}  // Here a program segment ends.
```

Now the contents of the loop part, ie, the main program is introduced: THIS IS THE COMPLETE SKETCH:

```
void set up()   // Here the setup starts
```

```
{ // Here a program section starts.
pinMode(13, OUTPUT);    // pin 13 to be an output.
} // Here a program segment ends. void loop()   // Here the main program begins
{ // Program section begins.
digitalWrite(13, HIGH);   // Turn the the voltage at pin 13 a (LED on).
delay(1000);              // wait 1000 milliseconds (one second)
digitalWrite(13, LOW);    // Turn the the voltage at pin 13 from (LED off). delay(1000);              // wait 1000 milliseconds (one second).
} // Program section         completed.
```

Finished. The sketch should now look exactly as it appears on the screen is shown on the right. It has to be now uploaded to the board. This works with the red circled button (top left of the software).

1.4 Now we can create variation to the program. For example, the LED will blink rapidly. To do this, we reduce waiting times (from 1000ms to 200ms)

```
void set up()  // Here the setup starts
{ // Here a program section starts.
pinMode(13, OUTPUT);    // pin 13 to be an output.
} // Here, a program segment ends. void loop()  // Here the main program begins
{ // Program section begins.
digitalWrite(13, HIGH);   // Turn the the voltage at pin 13 a (LED on).
delay(200);               // Wait 200 milliseconds
digitalWrite(13, LOW);    // Turn the the voltage at pin 13 from (LED off). delay(200);              // Wait 200 milliseconds
} // Program section         completed.
```

The new Sketch must now again be uploaded to the board. If all goes well, the LED is now flashing faster.

The Change Indicators

Material: Arduino / two light-emitting diodes (blue) / two resistors 100 ohms / Breadboard / cable

Sketch:

void set up()

{ // // We start with the Setup

pinMode(7, OUTPUT); // pin 7 is an output. pinMode(8th, OUTPUT); // pin 8 is an output.
}

void loop()

{ // The main program starts.

digitalWrite(7, HIGH); to Pin7 // Turn on the LED. delay(1000); // wait 1000 milliseconds. digitalWrite(7, LOW); // Turn the LED off at Pin7. digitalWrite(8th, HIGH); // Turn the LED on a Pin8. delay(1000); // wait 1000 milliseconds. digitalWrite(8th, LOW); // Turn the LED off at Pin8.
} // Here at the end of the program jumps to the start of the loop part. So…
//... turn the LED on pin 7.
//... etc... etc... etc...

An LED Pulsing Leave

Task: An LED to be pulsed light and dark. (Also referred to as English. "Thread")

Material: Arduino / an LED (blue) / A resistor 100 Ohm / Breadboard / cable assembly:

The Arduino is a digital microcontroller. We know it outputs only "5 volts" or "5V off". To vary the brightness of an LED, you would need, but the voltage may vary. For example, 5V when the LED light illuminates. 4 volts when darker lit something so GOING TO DIGITAL PINS BUT NOT. There is an alternative, however. It is called pulse width modulation (PWM called).

The PWM pulsates the 5V voltage. The voltage is thus a millisecond on and off. At a high PWM, the 5V signal is almost continuously on the respective

pin. At a low PWM, the 5V signal is virtually non-existent (since this is a very compact summary, you should look on the Internet for further explanation).

This PWM, you can achieve a similar effect with LEDs as if you were the voltage varies. Not all digital pins on the board have the PWM function. The pins on which the PWM works are specially marked, e.g. through a small wave of the number with the pin number. Here we go.

Sketch:

```
int LED = 9; // The word "LED" is now available for the value. 9

int Brightness = 0; // The word "brightness" is now available for the value in the
// PWM output. The number 0 is only an arbitrary starting value.
int fade steps = 5;  // fade steps: determines the speed of the "thread"
void set up()
{ // Here the setup starts.

pinMode(LED, OUTPUT); //// The pin with the LED (Pin9) is an output
}
void loop()
{
analog write(LED, brightness); // The function analog write is here at the
// Pin activated with the LED (Pin9) the PWM output. The PWM value is the value
// stored under the name of "brightness". In this case, "0" (See
// first program section)

brightness + = brightness fade steps; // Now, the value of the PWM output
// changed. Under the value "brightness" is now the previous brightness
// the value for the fade steps added. In this case: brightness = 0 + 5. The
```

```
// new value for "brightness" is therefore no longer 0 but 5. Once the loop
// part is once gone through, he repeats. Then the value is for
// the brightness of 10. In the next cycle 15, etc., etc...

delay(25); // The LED should for 25ms (milliseconds), so only very briefly
// maintained brightness. Decreasing this value, the pulsation
// also faster.

if(Brightness == 0 || brightness == 255) { // Command explanation: When the
// brightness has reached the value 0 ORDER 255, the value for the change
// "fade steps" from positive to negative or vice versa. Reason: The LED is
always a bit brighter // first with each pass of the loop part.
// However, at some point the maximum value for the PWM output with the value 255
//reached. The LED will then gradually become darker. So
// the value for the "fade steps" is at this point Negating (A
// minus sign is placed in front of it)

fade steps = -fadeschritte; // That means for the next pass that in
// the line "brightness = brightness + fade steps," the brightness decreases.
// Example: "brightness = 255 + (- 5)". The value of brightness is from then 250. In
// next pass 245, etc., etc... Once the value for brightness at 0
// has arrived, returns the sign. (Consider the old
// mathematical rule: "minus and minus gives plus").

}

} // This last clip of the loop portion is closed.
```

Simultaneous Lighting and Sound

Task: An LED and a piezo speaker will continuously flash or beep. Material: Arduino / a LED / A resistor 200 Ohm / A piezo speaker / Breadboard / cable

Sketch:

```
// This time, we also use the first program section. We enter
// variables. This means that behind a letter or
// one word hides a number. For us, the LED on pin 4 is connected and
// the piezo speaker to pin 5. This is the two pins later not
// confused, we call Pin4 and Pin5 simply.

int LED = 4; // The word "LED" is now available for the number "4".

int pieps = 5; // The word "beep" is now available for the number "5".
void set up()
{  // We start with the setup.

pinMode(LED, OUTPUT); // pin 4 (pin "LED") is an output.
pinMode(Beeping, OUTPUT);                    // Pin 5 (Pin "beep") is
an output.
}

void loop()

{  // The main program starts. digitalWrite(LED, HIGH);            //
Turn the light on
digitalWrite(Beeping,  HIGH);              // Switch  to  the  piezo
speaker. delay(1000);            // wait 1000 milliseconds. (It beeps and
lights up.)digitalWrite(LED, LOW);                // Switch from LED.
digitalWrite(Beeping, LOW);                  // Turn on the piezo out.
delay(1000);  // wait 1000 milliseconds. (No noise, no light)

} // Here at the end of the program jumps to the start of the loop part.
so becomes
// it beeps again and lights. If one reduces the pause (delay)
// or enlarged beeps and lights it faster or slower.
```

Activate a LED Touch of a Button

Task: An LED to be lit for 5 seconds when a button has been operated.

Material: Arduino / an LED (blue) / A resistor 100 Ohm / A resistor 1K ohms (1000 ohms) / Breadboard / cable / probe

The microcontroller can output to its digital pins, not only tensions but also read out. We want to try this program. In the structure, however, there is a special feature. If you simply connect the switch only to the microcontroller, then up to the pin of the microcontroller to a voltage as soon as the button is pressed.

You can think of it that way as if many electrons are floating around the said pin. If we then release the button, no new electrons are more being added to the pin on the microcontroller. Now the rub comes. The electrons that have made it before relaxing on the pin, then still there and escape only very slowly over small leakage currents.

The microcontroller then so thinks that the button is not pressed only briefly, but that it is pressed very long. Namely, until no electrons spend more on the pin, this problem can be correct in that one grounding pin via a resistor to about 1000 ohms (1K ohms). The electrons can thus very quickly drain away from the pin, and the microcontroller detects that the button has been "touched" only briefly.

Since the resistance, the voltage at the input pin to 0V always "pulling down," it is also referred to as a "pull-down" resistor. NOTE: If you use it too small, resistance can cause a short circuit to the microcontroller when the button is pressed that one grounding pin via a resistor to about 1000 ohms (1K ohms).

The electrons can thus very quickly drain away from the pin, and the microcontroller detects that the button has been "touched" only briefly. Since the resistance, the voltage at the input pin to 0V always "pulling down," it is also referred to as a "pull-down" resistor. NOTE: If you use it too small, resistance can cause a short circuit to the microcontroller when the button is pressed that one grounding pin via a resistor to about 1000 ohms (1K ohms).

The electrons can thus very quickly drain away from the pin, and the microcontroller detects that the button has been "touched" only briefly.

Since the resistance, the voltage at the input pin to 0V always "pulling down," it is also referred to as a "pull-down" resistor. NOTE: If you use it too small, resistance can cause a short circuit to the microcontroller when the button is pressed.

Sketch:

```
int LEDblue = 6;     // The word "LEDblue" is now available for the value. 6

int taster = 7;// The word "taster" is now available for the value. 7

int button status = 0;       // The word "taster status" is now available for the first
// value 0 is later stored under this variable, if the button is pressed
// or not. void set up()
{  // Here the setup starts.

pinMode(LEDblue, OUTPUT);  // The pin with the LED (pin 6) is now an output. pinMode(Push buttons, INPUT); // The pin with the button (pin 7) is now an entrance.
}

void loop()
{ // This clip the loop portion is opened taster status =digital read(Button);              // Here the Pin7 is read
// (Command: digital read). The result is below the variable "taster status" with
// the value "HIGH" for 5V or "LOW" saved for 0Volt.

if (Taster status == HIGH)       // Processing: When the pushbutton is pressed (The
// voltage signal is high)..

{ Open // Program section of the IF command. digitalWrite(LEDblue, HIGH);            //..dann should light up the LED..
delay(5000);  Although for //..und for 5 seconds (5000 milliseconds). digitalWrite(LEDblue, LOW);              // then the LED off should be.
} // close the program section of the IF command. else                //... otherwise...
```

{ Open // Program section of the else command digitalWrite(LEDblue, LOW); //... to be the LED off.
} // close the program section of the else command

} // This last clip of the loop part closed.

An RGB LED Drive

Task: An RGB LED should light up in different colors

Material: Arduino board / RGB-LED / three resistors with 200 Ohm / Breadboard / Cable

What is an RGB LED?

An RGB LED is an LED that can light up in different colors. After the name of the RGB Color Bet "Red," "Green" and hide

"Blue." The LED is in the interior of three individually addressable LEDs, which shine in the three colors. Therefore, an RGB LED also has so many legs, for example, 4/6/8 legs . The longest of the four legs depending on the version of the common anode (+) or

Cathode (-). With the three shorter legs of the RGB LED, we can control each color.

Version a "common cathode" - the longest leg of the LED is "-,," and the three shorter legs are controlled by "+" (voltage).

Version b) "common anode" - the longest leg of the LED is "+," and the three shorter legs are - controlled (GND). ""

Through a blend of colors, many other colors can be produced very. For example, results from the control of the color "blue" and "green" color

"Turquoise."

Which version you own, you can find by simply repositioning of "+" and "-" on the LED out (Info: An LED lights up only when properly mated)

The Arduino is a digital microcontroller. He knows his digital outputs only "5 volts" or "5V off". So you can generate many different colors with an

RGB LED each color of the LED but must be controlled more accurately. This can be done with pulse width modulation and the PWM (pulse width modulation) can be used to the digital pins on the board at which it prints a small shaft.

The PWM allows the voltage between + 5V and 0V pulsate. The voltage is thus a millisecond on and off. At a high PWM, the 5V signal is almost continuously on the respective pin. At a low PWM signal is the 5V virtually non-existent (Since this is a very compact summary, you should look on the Internet for further explanation).

This PWM can achieve a similar effect in LEDs, as would be the voltage varies. The following codes work for both RGB versions alike. It needs only one thing to be observed: In the LED version b (Common anode) should be the value of the "Dark" is set to the 255th. This has the consequence that then, not only a positive voltage at the common positive terminal of the LED but also to the corresponding color. Then, the LED can be no current flowing, and the respective color of the LED remains off between the two contacts.

For this reason, that in this version of the LED, the color is brighter when the value is smaller is also noted for the color mixture. Thus the color lights blue to pin 3 in the sketch bright, if we choose the code for the blue color like this:

int brightness1a = 0;

Sketch 1:

In this code, the three individual colors are sequentially switched on and off.

```
int LEDblue = 3; // color blue to pin 3 int LEDrot = 5; // color red on pin 5 int LEDgreen = 6; // color green to pin 6
int p = 1000; // p is a break with 1000ms So one second

int brightness1a = 150; // numerical value between 0 and 255 - are
// the luminance of each color at

int brightness1b = 150; // numerical value between 0 and 255 - are
```

```
// the luminance of each color at

int brightness1c = 150; // numerical value between 0 and 255 - are
// the luminance of each color at
int dark = 0; // Neumericalvalue 0 means 0V - so LED off void set up()
{
pinMode(LEDblue, OUTPUT); pinMode(LEDgreen, OUTPUT);
pinMode(LEDrot, OUTPUT);
}
void loop()
{
analog write(LEDblue, brightness1a); // turn blue delay(P); // Break
analog write(LEDblue, dark); off // blue analog write(LEDrot,
brightness1b); // turn on red delay(P); // Break
analog write(LEDrot, dark); rotausschalten //
analog write(LEDgreen, brightness1c); turn green // delay(P); // Break
analog write(LEDgreen, dark); // greenausschalten
}
```

Sketch 2:

In this code, the three individual colors are in pairs on and off in sequence. Thereby, the color mixing yellow, turquoise, and purple arise.

```
int LEDblue = 3; // color blue to pin 3 int LEDrot = 5; // color red on
pin 5 int LEDgreen = 6; // color green to pin 6
int p = 1000; // p is a break with 1000ms So one second
int brightness1a = 150; // numerical value between 0 and 255 - are
// the luminance of each color at
int brightness1b = 150; // numerical value between 0 and 255 - are
// the luminance of each color at
int brightness1c = 150; // numerical value between 0 and 255 - are
// the luminance of each color at
int dark = 0; // value 0 means 0V - so LED
//out
void set up()
{
```

```
pinMode(LEDblue, OUTPUT); pinMode(LEDgreen, OUTPUT);
pinMode(LEDrot, OUTPUT);
}
void loop()
{
analog write(LEDgreen, brightness1c); // green and red on yellow =
analog write(LEDrot, brightness1b);
delay(P);
analog write(LEDgreen, dark); // green and red from yellow = from
analog write(LEDrot, dark);
analog write(LEDgreen, brightness1c); // green, and blue =
//turquoise
analog write(LEDblue, brightness1b); delay(P);
analog write(LEDgreen, dark); // green and blue from turquoise from =
analog write(LEDblue, dark);
analog write(LEDrot, brightness1b); // red and blue = purple analog
write(LEDblue, brightness1b);
delay(P);
analog write(LEDrot, dark); // red and blue from purple from = analog
write(LEDblue, dark)
}
```

The Motion

Task: A piezo speaker to beep when motion is detected. Material: Arduino / Motion / Breadboard / cable / piezo speaker

Course content: voltage of a motion to read and use for output.

The motion, also called the PIR sensor, is very simple in construction.

Once it detects motion, it is on a pin a voltage

of 5 volts. This only needs to be read and processed by the microcontroller.

Two knobs can ad the duration of the output signal and sensitivity (range).

The plastic lens is lightly connected. If one sets them apart can be seen the infrared detector and can be seen from the label under the lens, as the sensor has to be wired: GND (-), OUT (output of the signal), VCC (+).

There is also a jumper which allows you to switch between two modes at the bottom of the movement.

1. A jumper is quite outside: it detects The output signal after it maintained a motion for a certain time and then deactivated definitely be back, even if there is a movement that could be detected in the action area of the motion. After a certain time, the output signal is generated again.
2. The jumper is like a right slightly offset on the image inside. The output signal remains constantly active as long as the motion is detected a movement and this mode is recommended for Arduino projects.

Sketch:

```
int piezo = 5; // The word "piezo" is now available for the value. 5

int motion = 7; // The word "movement" is now available for the value.
7

int motion status = 0; // The word "movement status" first, is now
available for the
// value 0 is later stored under this variable, whether a movement
// is recognized or not.

void set up() // Here the setup starts.

{
pinMode(Piezo, OUTPUT); // The pin with the piezo (pin 5) is now an
output. pinMode(Move, INPUT); // The pin with the motion detector
(pin 7) is now
//an entrance.

}

void loop() // The loop part begins
{ // This brace is the Loop section opened. motion status =digital
read(Move); // ier of Pin7 is read. The
```

// result is under the variable "moving status" with the value "HIGH" for
// 5V or "LOW" saved for 0Volt.

if (Movement status == HIGH) // processing: When motion is detected
// (The voltage signal is high)

{ Open // Program section of the IF command. digitalWrite(Piezo, HIGH); // then the piezo will beep. delay(5000); Although //...and for 5 seconds. digitalWrite(Piezo, LOW); //...then it should be quiet.
} // close the program section of the IF command. else //otherwise...
{ Open // Program section of the else command.

digitalWrite(Piezo, LOW); //...should the piezo speaker to be.
} // close the program section of the else command.

} // This last clip of the loop portion is closed.

Read Brightness Sensor

Now it's getting more complicated.

Task: An LED should light when it gets dark or when a photoresist is covered.

Material: Arduino / a LED / resistor with a 200 Ohm / a resistor of 10K ohms / Breadboard / cable / photoresistor

Read Voltages

The microcontroller is to read about a photoresistor how bright it is. For this, one uses a simple physical principle. If in a circuit two consumers are connected one behind the other (in series), then "shared" it is also commonly applied voltage. An example: two identical lamps connected in series, and there is applied a voltage of 6 volts.

Then you can determine with a voltmeter that each abut the lamps only 3 volts. If we connect two unequal lamps (One has a lower resistance), then one can use two different voltages to the two lamps measure, for example. 1.5 volts and 4.5 volts. A photoresistor changes its resistance as a function of light intensity. This effect is exploited so one can generate a voltage

division here at all, and it closes the photoresistor and a resistor (1 - 10 K ohms, depending on the photoresistor. The resistor should have a similar resistance value as that of the photoresistor have.) in series and connects it to 5 volts and the "grounding" (ground / GND) - see construction. The microcontroller board is to measure capability of analog signals (voltage) and to process these.

We can do this with the analog inputs on the board and this converts the measured voltage value into a number that can then be further processed. 0 volt corresponds to the number 0, and the highest measured value 5 volts corresponds to the number 1023 (0 to 1023 equals 1024 digits = 10 bits). Example: A voltage of 2.5 volts is measured, then the microcontroller returns the value 512 (1024: 2).

The "Serial Monitor"

The "serial monitor" is an important part of the Arduino software. This "serial monitor" you can display data on the PC, which the microcontroller board to the PC sends (numbers or text). This is very useful because you do not an LCD display that is always connected to the microcontroller, on which one could read certain values.

In the sketch, the "serial monitor" is used to display the values that read the board of the photoresistor. Why is this useful? Suppose the LED to begin only at the onset of darkness to light. Then it must be in an area Sketch gives, which has the function: "If the value of the photo resistance below the value x, then the LED lights will." It would have to know how big the value of x at the onset of twilight.

Solution: (. Eg dusk) I send the read value "x" of the voltage across the photoresistor with appropriate brightness at the "serial monitor" and let him see me there. With this knowledge, I can later modify the program in the following form. "If the voltage output of the photoresistor has a value of" x "below, then turn on the LED."

Sketch:

```
int input = A0;       // The word "input" is now available for the value
"A0" (name
// the analog port 0)
```

```
int LED = 10; // The word "LED" is now available for the value 10
int Sensor value = 0;        // variable for the sensor value with 0 as the start value

void set up()   // Here the setup starts.
{

Serial.begin(9600); // Communication with the serial port
// started. This one needs to be the actual value read later
let // Display serial monitor in.

pinMode (LED,OUTPUT); // The pin with the LED (pin 10) is now an output

}

void loop()
{ // This brace is the Loop section opened. Sensor value =analog read(entrance);              // Read the voltage on the photoresistor
// store and under the variable "sensor value". Serial.print("Sensor value =");// Output on Serial Monitor: The word "sensor value"

Serial.print(Sensor value);        // Output on Serial Monitor. The command
the sensor value of the photoresistor is in the form of a number //Serial.print
// sent 0-1023 to the serial monitor.

if(Sensor value> 512)       // If the sensor value is over 512....
{

digitalWrite(LED,HIGH);  //... is to light the LED...

}

else

{

digitalWrite(LED,LOW);   //... otherwise, they will not shine
```

```
}

delay(50);      // A short break, during which the LED is on or off

}  // This last clip of the loop portion is closed.

// Now, if the sensor value, for example, at normal brightness. Only the value of 100 has
// (The value depends on the used resistors of the brightness
// and of the current direction), then take the place of the value 512
// much smaller value at which the LED will begin to glow. For example.
// then you take the value of 90. The current sensor value you can now use
// using the "Serial monitor" display. Click at the top "Tools."
// and then click "serial monitor."
```

Knobs

Task: An LED should blink. The blink rate is to be added with a knob.

Material: Arduino / a knob (potentiometer) / Breadboard/cable read voltage of a rotary control, processing sensor values, and mathematically to use for output (in this case, for the duration of a pause): learning content.

A knob has three terminals. Outside is + and - connected. From the middle, pin cable goes to an analog input pin on the microcontroller board. By turning the knob, the middle pin outputs a voltage between 0 and 5 volts. Knobs far left: 0 V and knobs on the far right: 5V, or mirror-inverted depending on the wiring.

As LED should blink, we use as the first sketch the LED, which is secured by pin 13 on the microcontroller. In addition, it still connects further LEDs.

Sketch:

```
int input = A0;       // The word "input" is now available for the value "A0" (name
// the analog port 0)

int LED = 13; // The word "LED" is now available for the value 13
```

```
int Sensor value = 0;        // variable for the sensor value with 0 as the start value void set up()
{  // Here the setup starts.

pinMode (LED,OUTPUT); // The pin with the LED (pin 13) is now an output.

}

void loop()

{  // This brace is the Loop section opened.

Sensor value = analog read(entrance);        // The voltage on the control dial will read
// and in the previous sketch as a number from 0 to 1023 under the variable
// "sensor value" stored.

digitalWrite (LED,HIGH); // The LED is turned on

delay(Sensor value);        // The LED remains on for so many milliseconds,
// as the value of "sensor value" it has stored. digitalWrite(LED, LOW);              // The LED is turned off.
delay(Sensor value);        // The LED remains off for so many milliseconds,
// as the value of "sensor value" it has stored.

}  // This clip the loop portion is closed

// The loop part is now restarted. If the value of the read
// knob changes, then the time between the input and output changes
// phases of the LED. The flashing is faster and slower. The longest
In this skit // delay is 1023ms (milliseconds). If you long delays
// required then builds to a small mathematical line in the code.
// For example, to change the line "sensor value = analog read (input)," in
// "sensor value = analog read (input) * 2;" This is the stored sensor value
```

// magnified by a factor 2nd Since longest delay would be 2046ms etc.

Measure Temperatures

Task: With the temperature sensor TMP36, the temperature should read and display monitor serial-by.

Material: Arduino / Breadboard / cable / Temperature Sensor TMP36 / External Power Supply

The sensor has three connections. When looking at the flat side of the sensor: left 5V, GND, and right in the middle of the pin for the temperature signal. On this pin, the sensor outputs a voltage between 0 and 2.0 volts. Wherein 0V -50 ° C and the corresponding value of 2.0V corresponds to 150 ° C.

According to the manufacturer, the sensor is between -40 ° C and + 125 ° C with reasonable accuracy (± 2 ° C). The voltage of this pin has to read from the microcontroller board and converted into a temperature value.

NOTE:

- If the sensor is connected incorrectly, it will burn out.
- In the structure of an external power supply should be used if possible, as it improves considerably the sensor accuracy (9V AC adapter or 9V battery).

Sketch:

```
int TMP36 = A0;     // The sensor A0 is to be connected to the analog pin. We
// call the pin from now "TMP36"

int temperature = 0; // Under the variable "temperature" is later
// temperature value stored

int temp [10];// This yields good values, you first read multiple values
// and then averages the results. The square bracket "[10]" generated here
// equal to ten variables named "temp [0],” "temp [2],” "temp [3]"... to...
```

```
//"temp[9]".Mit the notation [10] So you can save only a little Place.

int time= 20; // The value for "time" is in the code, the time intervals
// ago between the individual measurements. void set up() {
Serial.begin(9600); // In the setup we start serial communication so
// we can show us the temperature later. About the serial
// communication the board, sending measurements to the computer. In the Arduino
// software can be under "Tools" the "Serial Monitor" start the readings
// to see.

}
void loop() {
temp [0] = map(analog read(TMP36),0 410 -50, 150);
delay(time);
temp [1] = map(analog read(TMP36),0 410 -50, 150);
delay(time);
temp [2] = map(analog read(TMP36),0 410 -50, 150);
delay(time);
temp [3] = map(analog read(TMP36),0 410 -50, 150);
delay(time);
temp [4] = map(analog read(TMP36),0 410 -50, 150);
delay(time);
temp [5] = map(analog read(TMP36),0 410 - 50, 150);
delay(time);
temp [6] = map(analog read(TMP36),0 410 -50, 150);
delay(time);
temp [7] = map(analog read(TMP36),0 410 -50, 150);
delay(time);
temp [8] = map(analog read(TMP36), 0, 410, -50, 150); delay(time);
temp [9] = map(analog read(TMP36), 0, 410, -50, 150); // From here is ten times the
read // temperature. In between is ever a break with the duration
// "time" in milliseconds. But what happens here, exactly? Let's look at the
// command once more closely.
// temp [1] = map (analog read (TMP36), 0, 410, -50, 150);
```

```
//Sequentially:
// temp [1] - is the name of the first variable.
// "map (a, b, c, d, e)" - This is the so-called "Map command". Allows
// to a read-in value (a) in which the values between (b) and (c) lie,
// convert it into a different speed range, and that (d) in the area between
// and (e).
// following our command happens:
// the value of the sensor is read out directly in the "Map" command with
// "analog read (TMP36)". The measured values should be between 0 and 410th The
// corresponds to the analog port of the values between 0V and 2V. is the voltage
// the sensor out at temperatures between -50 ° C and + 150 ° C. These values on
// analog port will now be through the "Map command" directly to the Celsius values
// -50 to 150 converted.

temperature = (temp [0] + temp [1] + temp [2] + temp [3] + temp [4] + temp [5] + temp [6] + temp [7] + temp temp [8] + [ 9]) / 10; // All in a row. Here every ten are found
// temperature values added together and divided by ten. The
// average value is stored under the variable "temperature"

Serial.print(temperature);        // Now, the value of "temperature" over the serial
// sent communications to the PC. Now you have the serial monitor in the
// Open Arduino software to the temperature at the sensor to read.
Serial.println("Degree");
}
```

Expansion of the Program:

Once the temperature has reached 30 ° C, to a warning sound (beep of a piezo speaker).

```
int TMP36 = A0;

int temperature = 0; int temp [10];

int time= 20;

int piezo = 5;       // The word "piezo" is now the number 5, that is to port 5 of the

// Piezo connected. void set up() { Serial.begin(9600);

pinMode (Piezo, OUTPUT);  // The piezo speaker to pin 5 should be a starting


// (logical because of the yes from the microcontroller board yes a voltage required to

// beeping.


}


void loop() {

temp [0] = map(analog          0       410    -50,    150);

read(TMP36),

delay(time);                              0       410    -50,    150);

temp [1] = map(analog read(TMP36),


temp [9] = map(analog          0       410    -50,    150);

read(TMP36),
```

Measure Distance

Task: the ultrasonic sensor HC-SR04 and a microcontroller Arduino a distance to be measured and displayed with the "serial-monitor".

How does the ultrasonic sensor? The sensor has four terminals.

a) 5V (+) b) GND (-) c) echo d) trigger

The connections 5V and GND are self-evident; they supply the sensor with power.

The "trigger" of the contacts inserts a short signal (5 V) into the microcontroller board, causing an acoustic wave through the ultrasonic sensor. Once the sound wave hits a wall or other object, it is reflected and at some point reaches the ultrasonic transducer again.

Once the transducer detects these reflected sound waves, it sends a 5V signal to the "echo" contact on the microcontroller board. The microcontroller only measures the time between the radiation and the return of the sound wave and converts this time into distance. Here we go.

Material: microcontroller board / cable / Breadboard / Hc-SR04 ultrasonic sensor

Code:

```
int trigger = 7;        trigger // The word is now the number. 7 int echo =
6;              // The word echo is now the number sixth
long time = 0;          life // The word is now a variable under which the
time
// is stored, to the reflection and requires a sound wave back.
// Start value here 0th

long distance = 0;   // The word "distance" is now variable, under the
is stored // the calculated distance.

// Info: Instead of "int" is faced with the two variables "long." That has
// the advantage that a larger number can be saved. Disadvantage:
// variable takes up more space in memory. void set up()
{

Serial.begin (9600);// start communication Serial, so you later
```

```
can view the serial monitor // values.

pinMode(Trigger, OUTPUT);    // "trigger" (pin 7) is an output.
pinMode(echo, INPUT);              // "echo" (pin 6) is an input.
}

void loop()

{

digitalWrite(Trigger, LOW);      // Here, take the voltage for a short period
// trigger pin, so that one later sending the trigger signal a noise-free
// Signal.

delay(5);        // Duration: 5 milliseconds

digitalWrite(Trigger, HIGH);                  // Now you send off an ultrasonic wave delay(10);                  // This "tone" is heard for 10 milliseconds. digitalWrite(Trigger, LOW);              // Then the "tone" is turned off.
duration = pulseIn(echo, HIGH);        // The command "pulseIn" is one of the
// microcontroller the time in milliseconds until the sound to
// ultrasonic sensor returns.
Distance = (time / 2) / 29.1;       // Now we calculate the distance in
// centimeters. First, you divide the time by two (because only one
// Section wants to calculate and not back the track and back). The value
// can still be divided by 1.29 and then receives the value in centimeters because of the
// sound in the air at a speed of one centimeter per 29.1
// microseconds.
// Example: The sensor measures time from 2000 microseconds. These are then
// 1000 microseconds each way. Now one in 1000 because of the sound divided by 1.29 (
// yes 1cm travels per 29.1 microseconds) and receives the value 34,3cm
```

```
if (Distance> = 500 || distance <= 0) // If the measured distance over
// is 500cm or 0cm...

{

Serial.println("No reading");    // then is to spend the serial monitor "No
// reading "because readings are wrong or inaccurate in these areas.
}

else      //otherwise..
{

Serial.print(distance);      //... should the value of the distance to the serial monitor
// output here.

Serial.println("cm");
}

delay(1000);  // The delay of one second provides in about every new second for
// a new measurement.
}
```

Expansion of the Program:

When a distance is measured 80cm, a piezo speaker to beep.

```
int trigger = 12; int echo = 13; long time = 0;
long distance = 0;

int piezo = 5; // The word piezo is now the number 5 void set up()
{

Serial.begin (9600); pinMode(Trigger, OUTPUT); pinMode(echo, INPUT);
pinMode(Piezo, OUTPUT);        // The piezo speaker to pin 5 should be a starting
// (logical because the yes from the microcontroller board yes a voltage required to
```

```
beeping //

}

void loop()


{

digitalWrite(Trigger, LOW);

delay(5); digitalWrite(Trigger, HIGH); delay(10); digitalWrite(Trigger,
LOW); duration =pulseIn(echo, HIGH); Distance = (time / 2) / 29.1;
if (Distance> = 500 || distance <= 0)

{

Serial.println("No reading");
}

else
{

Serial.print(distance); Serial.println("cm");
}

if (Distance <= 80) // If the value for the distance below or equal to 80
//, then...

{

digitalWrite(Piezo,HIGH);          //... start beeping.
}

else       // And if that is not so...

{
digitalWrite(Piezo,LOW); //... then be quiet.

}
delay(1000);
```

```
}
```

Enhancements: Reversing Warning

With this code, we can construct a reverse warning system, on pin 12 besides "measure distance" to the already connected ultrasonic sensor from the Sketch 10 (connects an LED.

```
... you can wrap an image already follows construct the reverse warning anyway?
int trigger = 7; int echo = 6; long time = 0; int LED = 12;
long distance = 0;

void set up()
{
Serial.begin (9600); pinMode(Trigger, OUTPUT); pinMode(echo, INPUT); pinMode(12, OUTPUT);
}

void loop()
{
digitalWrite(Trigger, LOW); delay(5); digitalWrite(Trigger, HIGH); delay(10); digitalWrite(Trigger, LOW); duration =pulseIn(echo, HIGH); Distance = (time / 2) / 29.1;

if (Distance> = 500 || distance <= 0)
{
Serial.println("No reading");
}

else
{
Serial.print(distance); Serial.println("Cm");
}

if (Distance <= 40)
{
digitalWrite(LED, HIGH); delay(Distance * 3); digitalWrite(LED, LOW); delay(Distance * 3);
}
```

}

Infrared Remote Control for Controlling Arduino Microcontrollers

Material: Arduino / Breadboard / cable / infrared sensor / infrared remote control

With an infrared receiver that an Arduino board evaluates the commands of an infrared remote control. The data is sent in the form of infrared light from the remote control to the receiver. Since our eyes can not perceive this light, we can't see it.

With a little trick, however, you can test whether a remote control, for example, sends an infrared signal. This involves taking a digital camera (for example, by phone) and viewed via the display, the infrared diode. Now, if we press the remote control, you can see the lights infrared diode because the sensors of digital cameras perceive infrared light and can represent. The light flickers easily, since the infrared diode toggle very quickly and outgoing. Underlying this is a very specific rhythm that can test the basis of which the infrared receiver later, which key was pressed on the remote control.

This sketch is a slight modification of Sketches "IRrecvDemo," default in the Arduino main program can not find which under the examples.

This sketch is very short and therefore lends itself very well for the first experiments.

Code:

```
/ *
* IrRemote: IRrecvDemo - Demonstrates receiving IR codes with IRrecv
* An IR detector/demodulator must be connected to the input RECV_PIN.
*  Version 0.1 July 2009
*  Copyright 2009 Ken Shirriff
*  http://arcfn.com
```

```
* /// Information over the original program "IrrecvDemo". #include <Irremote.h> // The program encroaches on a
// "Library" back. This does a lot of work. Because of the infrared light
// will be encrypted with a code. To read this code itself
// convert and appropriate values, a lot of lines of code would be required.

int RECV_PIN = 11; // outputs The contact of the infrared sensor data,
// is connected to pin 11 of the Arduinoboards.

IRrecv irrecv (RECV_PIN); // At this point, an object is defined that
// reads the infrared sensor to pin. 11

decode_results results; // This command ensures that the data by
// infrared is read stored under "results."

void set up()
{

Serial.begin(9600); // In the setup the serial connection is started, so
// you look at the data to receive the remote control via the serial monitor
// can.

pinMode (13, OUTPUT);

irrecv.enableIRIn();// This command initializes the infrared receiver.

}

void loop()
{ // The loop portion turns out very shortly by resorting to the "library". if (irrecv.decode(Results)) { // If data has been received, Serial.println (results.value, DEC); // they are as a decimal (DEC) to the
// Serial Monitor output.
```

```
irrecv.resume();      // The next value to be read from the IR receiver
}
```

One press on the button "1" of the infrared remote control unit causes the number "16724175" is output at the Serial Monitor. This is the decoded numerical code behind this button.

If you hold the button down permanently, the number "4294967295" is displayed permanently. This indicates the code that a key is pressed permanently. This number is regardless of which button is pressed, but it can also appear other numbers if a button is pushed only very short or pulsed. In this case, the sensor can read no unique value.

Expansion of the Program:

When pressing the "1" LED will go on and on pressing the button "2," the LED will go out.

```
#include <Irremote.h> int RECV_PIN = 11;
IRrecv irrecv (RECV_PIN);
decode_results results; void set up()
{

Serial.begin(9600);

pinMode (13, OUTPUT); // To pin 13 an LED is connected.
digitalWrite(13, LOW);                    // This should first of all be
irrecv.enableIRIn();
}

void loop() {

if (irrecv.decode(Results)) { Serial.println(Results.value, DEC);
if (Results.value == 16724175) // If the infrared receiver, the number
// read "16724175" (the key According to "1" on the remote control)
has

{digitalWrite (13, HIGH)}        // the LED should go on.

if (Results.value == 16718055) // If the infrared receiver, the number
```

```
// read has "16718055" (corresponding to "2" of the remote control button)

{digitalWrite (13, LOW)} // the LED should go out. irrecv.resume();
}

}
```

Servo Drive

Task: A servo is to be controlled by an Arduino microcontroller. The servo should drive to three different positions in this example and wait a short time between positions.

Material: A microcontroller board, a servo, three plug-in cable

Code:

```
#include <Power.h>        // The servo library is called.

Power servo-blue;   // Create the program a servo with the name
// "servo blue"
void set up()
{

servo blue.attach(8th);     // servo is connected to Pin8
}

void loop()
{
servo blue.write(0);              // Position 1 controlled with the angle 0 °
delay(3000);                      // The program stops for 3 seconds servo
blue.write(90);                       // position 2 controlled with the angle
90 ° delay(3000);                 // The program stops for 3 seconds servo
blue.write(180);                                  // position 3 controlled with
the angle 180 ° delay(3000);                      // The program stops for 3
seconds servo blue.write(20);                // position 4 controlled with the
angle 20 ° delay(3000);             // The program stops for 3 seconds
}
```

LCD Display

Task: An LCD display is to be controlled with an Arduino microcontroller. After that, a predetermined text should appear as in the following example photo on display.

Material: Microcontroller Board (In this example, UN R3), a wheel (or potentiometer), jumper cables 14, Breadboard

Based on the material and the sketch, you quickly realize that the wiring is not so easy. This is because the LCD display must be connected to a large number of cables. Another difficulty is with which one could connect the cables missing sockets.

Therefore, it is advisable to solder a terminal strip or to solder the cable directly to the LCD. In the second case, it is advisable to use a ribbon cable (for example, from old IDE drives like hard disks, CD or DVD drives). Without a soldered connection, it is almost impossible to achieve good results.

The knob is there to ad the contrast of the LCD. The LED backlight, as shown in the sketch supplied with 5V. Since it would not recognize the label on the LCD in the sketch, it is not shown. You have, therefore, the position of the cable to the LCD counting (Example: The LCD is the first cable from the right GND.

The second cable from the right is connected to 5V, etc.). Info: For quick tinkering, many hobbyists prefer to use an LCD Keypad Shield or I2C LCD because you do not have to worry about the two alternatives to the wiring of the LCD must. However, the two alternatives mentioned above are also more expensive.

If you have successfully made the wiring, you can take the software with many other components, and use is made here to a "Library." The Library for the LCD display is part of the Arduino software and must not be installed separately.

Code:

```
#include <Liquid Crystal.h>     // Load LCD library
```

```
Liquid Crystal LCD (12, 11, 5, 4, 3, 2);        // This line is set,
// which pins is the microcontroller boards used for the LCD (Best
// first do not change). void set up() {
lcd.begin(16, 2);     // In the Setup specifies how many characters and
lines are applied. Here: 16 characters in 2 lines

}

void loop() {

lcd.setCursor(0, 0); Set the LCD // start position of the display.
// Means: The first character in the primal line.

lcd.print("Www.xyz.com");        // There should text "Www.xyz.com"
//appear.

lcd.setCursor(0, 1); // lcd.setCursor (0.1) means: First character in the
// second line.

lcd.print("I wish you success...");       // There will be the text "Good
luck ..."
//Pop up.

}
```

A variation: to alternately first up and then down a text message display. In this example, the text "up" and "down".

```
#include <Liquid Crystal.h> Liquid Crystal LCD (12, 11, 5, 4, 3, 2);
void set up() {
lcd.begin(16, 2);

}

void loop() {

lcd.setCursor(0, 0); // start at the first character in the first line with the
// text "Up." lcd.print("Above");
delay (2000);                Wait // Two seconds. lcd.clear();                //
Clear display.
```

```
lcd.setCursor(5, 1); // Renewed beginning at the fifth character in the second
// line with the text "Down". lcd.print("Below");
delay (2000);              Wait // Two seconds. lcd.clear();              //
Clear display.
}
```

An LCD is particularly well suited to sensor values or other issues of microcontroller boards display. For more help, you get, for example, in the Arduino software. Among the sample Sketches, one finds a number of different examples under the menu item "Liquid Crystal."

Relay Card

When tinkering with Arduino relay is very important. Relays are switches that can flow a larger current than would be possible by the microcontroller. With the Arduino, so you now enabled only a very small current relay, which can then also run large electrical items.

The relay card is connected to the Arduino to the contacts (in the photo on the right edge, between the red and green LED). The card is in operation permanently + 5V and GND - connected (). The pin labeled "IN" is connected to a pin of the digital Arduino boards. As long as at pin "IN" no signal (output from the digital pin of Arduino), the screw terminals A and B are connected together. As soon as a signal is present, contacts B and C are connected to each other.

Warning: relay cards that switch when applied to the pin "IN" GND, and there are relay cards that switch when applied to the pin "IN," a voltage of 5V +. Which version you can easily tell by combining the "signal" Pin once with GND and once with 5V + Arduino boards. A loud crack can clearly see the switching of the relay card.We can connect an electric device, at which a higher electric current flows as if it could provide the Arduino to the bottom contacts (A, B, C). For example, a large electric motor, a large lamp, etc.

As an example code, in this case, the simple "blink" code can be used. In place of the LED, you to close the output pin of the Arduino boards to the "signal" Pin the relay card. The relay will then switch every second.

Code:

```
void set up()
{
pinMode(6, OUTPUT);
}

void loop()
{
digitalWrite(6, HIGH);     // At this point, would one turn on the relay
delay(1000);               //... wait a second
digitalWrite(6, LOW);      And off again //

delay(1000);  //... and wait a second.
}
```

Stepper Motor

In this step, the motor is a stepper motor suitable for small applications with the Arduino board. The special feature is that we can operate it without an external power supply. The motor develops a relatively high torque. This is realized by a transmission, which has been installed within the metal casing before the actual step motor.

This makes it possible at all in this compact design, is that a complete revolution of the driveshaft to 2048 individual steps can be divided. A small resultant disadvantage is the slow maximum rotational speed.

We connect the stepper motor to a motor control board. This supplies the engine with sufficient electrical power so that the power need not apply to the digital pins on the Arduino board. The control board is available in two versions, in which the side-mounted pins are protruding either upward or downward out of the board. However, the pin assignment is the same.

Cabling

PIN2 the Arduino board to IN4 of the motor control board - 2 to 4

PIN 3 on the Arduino board to IN2, the motor control board - 3 to 2

PIN4 the Arduino board to IN3 of the motor control board - 4 to 3

PIN 5 on the Arduino board to IN1 of the motor control board - 5 to 1

PIN "GND" on the Arduino board to GND, the motor control board

PIN "5V" on the Arduino board to VCC, the motor control board

This is an example code that the motor turns by 2048 steps (corresponding to a full revolution) above and can turn back.

Code

```
#include <stepper.h> int SPMU = 32;
stepper myStepper (SPMU, 2,3,4,5); void set up()
{
myStepper.setSpeed(500);
}
void loop() {MyStepper.step(2048); delay(500);
myStepper.step(-2048); delay(500);
}
```

Instructions for a Humidity Sensor

With a moisture sensor (Moisture sensor), you can, as the name suggests measuring the moisture. However, this refers to the directly adjacent moisture, such as skin moisture or soil moisture, but not from humidity. You can use it, for example, to measure the moisture in the soil of plants. In the case of dryness, then an alarm signal may be heard, or an electric water pump could automatically the plant with water supply.

The sensor is also suitable to measure the water level in the sensor's range. The operation is simple. a voltage is applied to the two contacts of the sensor. The higher the moisture between the two contacts, the better the current to flow to the other from a contact.

This value is processed electronically in the sensor and transmitted to an analog signal on the molded card. Since the card, as described in previous manuals, cannot measure voltage per se, it converts the voltage present on the analog contact to a digital value. From 0 to 12 V corresponds to a digital value of 0 to 1023 (the digits 1024, since zero, is considered first).

In the humidity sensor, however, the upper limit is about at the numerical value of 800 when the sensor is completely immersed in the water. The exact calibration is, however, dependent on the sensor and of the kind that is measured of the liquid/moisture (e.g., saltwater has better conductivity, and the value would be correspondingly higher).

The programming is very simple and very similar to the very reading of potentiometers, as an analog value is read.

```
int reading = 0;     // Under the variable "reading" is later, the measured value of the
// sensor stored.

void set up()
{  // Here the setup starts.
Serial.begin(9600); // Communication with the serial port is started.
// This is required to display the value read in the serial monitor to
//to let.

}

void loop()
{  // This brace is the Loop section opened.

reading =analog read(A0);        // The voltage on the sensor is read out and
// under the variable "reading" saved.

Serial.print("Humidity Reading");      // Output on Serial Monitor: The word
// "Humidity Reading"
Serial.println(Measured value);// and following the actual measured value delay(500);              // Finally, a little break, so not too many
// rushing numbers on the Serial Monitor.
}
```

Expansion of the Program:

Now, an alarm signal with the aid of piezo speakers to sound as soon as the measured humidity falls below a certain limit.

As the limit of the value "200" is set in this case.

Code:

```
int reading = 0;

int Beep = 6; // The name "PIEPS" the Pin6 is now referred to which a piezo speaker is connected.

void set up()
{
Serial.begin(9600);
pinMode (6,OUTPUT);      // In the setup of the pin 6 is set as output
}

void loop()

{

reading =analog read(A0); Serial.print("Humidity Reading"); Serial.println(Measured value);
delay(500);

if (Reading <200)   // Here the query begins: When the sensor value is less
// is called "200," then...

{

digitalWrite(PIEPS, HIGH);       //... to beep the piezo speaker

}

else      //…otherwise…

{

digitalWrite(PIEPS, LOW);        //... he should be quiet.

}

}
```

Drop Sensor

A drop sensor or liquid sensor, as the name suggests, can detect a liquid. For this purpose, the liquid must be located directly on the sensor. It is enough, a small drop, to get a clear reading.

You can use the sensor, for example, as a rain sensor. Once a drop strikes the sensor, the Arduino board can perform an action such as roll-up an awning, blinds close, trigger an alarm, or actuate a wiper.

The operation is simple. Traverse the sensor at the long contact point is a voltage (either + or -). As soon as a liquid, for example. Connects through a tropical two contacts, a small current flows from one contact to another. This value is processed electronically in the sensor and transmitted as an analog signal to the analog input of the card in a form.

Since the card cannot measure the voltage per se as described in previous instructions, it converts the voltage applied to the analog pin into a digital value. From 0 to 12 volts corresponds to a digital value of 0 to 1023 (the number 1024 because zero is considered the first value).

In the liquid sensor, the value in the dry is zero "0". When a drop of water hits the contacts of the sensor, the Who at about "480" is located. The more drops are on the sensor, the higher the value.

The first code it's all about, read the Sensorvalue with the Arduino board and display with the "serial monitor."

The programming is very simple and very similar to the very reading of potentiometers or the reading of the humidity sensor, as an analog value is read.

```
int reading = 0;     // Under the variable "reading" is later, the measured value of the
// sensor stored.

void set up()
{  // Here the setup starts.

Serial.begin(9600); // Communication with the serial port is started.
// This is required to display the value read in the serial monitor to
```

```
//to let.

}

void loop()
{  // This brace is the Loop section opened.

reading =analog read(A0);        // The voltage on the sensor is read out and
// under the variable "reading" saved.
Serial.print("Humidity Reading");       // Output on Serial Monitor: The word
// "Humidity Reading"
Serial.println(Measured value);// and following the actual measured value delay(500);              // Finally, a little break, so not too many
// rushing numbers on the Serial Monitor.
}
```

Expansion of the Program:

Now an alarm signal using piezo speakers to alert you when a raindrop hits the sensor. As the limit, in this case, the measured value is determined 400, as expected with a drop on the sensor, a measured value of about 480 for sale.

Code:

```
int reading = 0;

int Beep = 6;  // The name "PIEPS" the Pin6 is now referred to which a piezo speaker is connected.

void set up()
{
Serial.begin(9600);

pinMode (6,OUTPUT);      // In the setup of the pin 6 is set as output
}

void loop()
```

```
{

reading =analog read(A0); Serial.print("Humidity Reading");
Serial.println(Measured value);
delay(500);

if (Reading > 400)  // Here the query begins: When the sensor value is
less
// is called "400," then...

{

digitalWrite(PIEPS, HIGH);       //... to beep the piezo speaker

}

else      //…otherwise…

{

digitalWrite(PIEPS, LOW);        //... he should be quiet.

}

}
```

RFID Kit

The RFID ("radio frequency identification") reader is used (also called "RFID tags") of RFID transmitters read a particular code by radio. Each transmitter has a unique case, only own individual code. Thus can be realized with the Arduino locking systems or similar projects where a person to identify with a transmitter.

RFID tags can have different forms, such as key fobs or cards in credit card format.

How does it work? An RFID receiver contains a small copper coil that generates a magnetic field. An RFID transmitter also includes a copper coil that picks up the magnetic field and generates an electric voltage in the transmitter. This voltage is then used to bring a small electronic chip to emit an electrical code by radio. This code is then received directly from the

transmitter and processed so that the Arduino microcontroller may further process the received code.

You can also describe RFID tags with a code. This is not taken into account because of the complexity of this manual. Other guides are available online.

Read an RFID tag with Arduino and Process the Data

Material: UN or MEGA Arduino, RFID reader, at least one RFID tag, display plate, display plate cable, LED, 200-ohm resistor.

Task: Using an Arduino microcontroller, an RFID tag to be read. Unless it is the correct TAG, it is to light an LED for 5 seconds.

Wiring the Arduino board with the RFD-receiver:

Board:	Arduino Uno	Arduino Mega	MFRC522 READER
Pin code:	10	53	SDA
Pin code:	13	52	SCK
Pin code:	11	51	MOSI
Pin code:	12	50	MISO
Pin code:	unassigned	unassigned	IRQ
Pin code:	GND	GND	GND
Pin code:	9	5	RST
Pin code:	3.3V	3.3V	3.3V

In this example, the 90° curved contact pins are soldered to the RFID receiver so that the receiver can be placed vertically in a breadboard.

Preparation - the first sketch with the RFID READER

First, we will read the UID ("Unique Identification Number"), i.e., the individual name of an RFID tag. We use the following program (Attention, the program only works if we added the library to the Arduino software). The program is provided an R3 microcontroller for the UN. In MEGA2560 and other controllers, the pins have to be added accordingly.

```
#include <SPI.h> // SPI Bibliotheque Add #include <MFRC522.h> // add RFID Bibliotheque #define SS_PIN 10 // SDA to pin 10 (MEGA otherwise) #define RST_PIN 9 // RST to pin 9 (MEGA otherwise) MFRC522 mfrc522 (SS_PIN, RST_PIN); // RFID receiver name

void set up() // start the setup:
{
Serial,begin(9600); // start serial connection (Monitor)
SPI,begin(); // build SPI connection mfrc522.PCD_Init (); // initialization of the RFID receiver
}

void loop() // Here, the loop part begins

{

if (. Mfrc522.PICC_IsNewCardPresent ()) // If a card within reach
// is...
{

return; // go on...

}

if (. Mfrc522.PICC_ReadCardSerial ()) // When an RFID transmitter was selected

{

return; // go on...

}

Serial,print("The ID of the RFID TAGS is:"); // "The ID of the RFID TAGS
// reads: "is written to the Serial Monitor for (byte i = 0; i <mfrc522.uid.size; i ++)

{
Serial,print(Mfrc522.uid.uidByte [i], HEX); // Then the UID is read,
// each of four blocks is and sequentially to the serial
```

// Monitor sent. The ending Hex means that the four blocks of the UID

// is (including letters) output as a HEX number

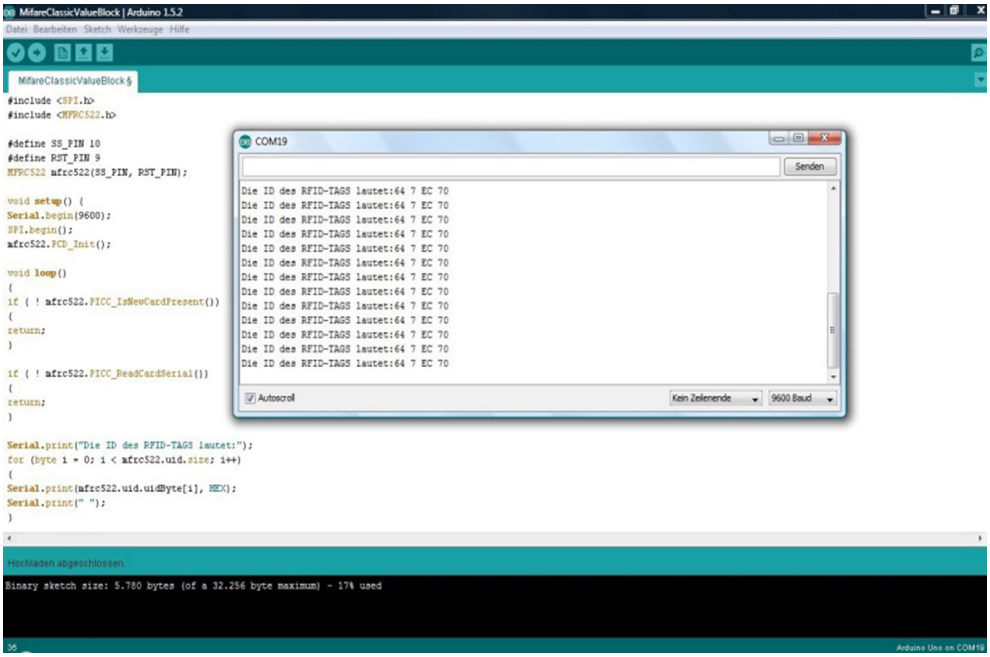
Serial,print(""); // The "Serial.print (" ") command;" ensures that
// stands between the individual blocks read a space.

}

Serial,println(); // This line is on the Serial Monitor only
// line break made.

}

If all is well, the result of the Serial Monitor looks (Apart from the own UID) like this:



HEX number with this written succession does not work well. So we change the line "Serial.print (mfrc522.uid.uidByte [i], HEX)," To

Then one gets as the result of the individual blocks of the UID code is output as a decimal number; "Serial.print (mfrc522.uid.uidByte [i], DEC".

RFID Sketch 2

Now the UID code is being issued as a decimal number, but it is still divided into four blocks. We change the code now with a little math to the effect that we obtain a single contiguous normal number for the UID (decimal).

Why do we do that? If we want to use the sketch later to trigger a correctly read RFID tags (e.g., an LED should light, or a servomotor is to be rotated 90 degrees...), we can better use an IF command with an associated number. For example:

"If the RFID code = is 1031720, then an LED for 5 seconds to light up."

Harder would contrast with the command "If the first block is 195 and the second block 765 reads and the third block 770 blocks 233 and the fourth is... Then turn the LEDs for 5 seconds.

However, a disadvantage is that the sketch is thus somewhat uncertain because all the numbers of the four blocks (max. 12 digits) can not be represented in a continuous number. If there is to be more secure, you would have every single block a query.

```
#include <SPI.h> #include <MFRC522.h> #define SS_PIN 10
#define RST_PIN 9
MFRC522 mfrc522 (SS_PIN, RST_PIN);

void set up()
{
Serial.begin(9600); SPI.begin(); mfrc522.PCD_Init ();
}

void loop()
{

if (. Mfrc522.PICC_IsNewCardPresent ())
{
return;
}

if (. Mfrc522.PICC_ReadCardSerial ())
{
```

return;
}

long code = 0; // As a new variable we add "code" under which the UID later is output as a continuous number. Instead of int, we now use the number range "long" because they can store a larger number.

for (byte i = 0; i <mfrc522.uid.size; i ++)
{
code = ((code + mfrc522.uid.uidByte [i]) * 10); // Now, as also previously read out the four blocks, and in each pass of the code is "stretched" by a factor of 10 degrees. (, here you would use the value in 1000, but the number would become too large.
}

Serial.print("The card number is ::"); // Finally, the number code (you can not call him as a UID more) output.

Serial.println(code);
}

Great, now we can read a unique identification number of an RFID tag (it displays The number on the Serial Monitor). In this case, the identification number of that individual RFID tags 1031720 is.

And how does it continue? Now we want to turn on an LED for 5 seconds if it holds the desired RFID tag in front of the RFID READER.

Sketch 3

#include <SPI.h> #include <MFRC522.h> #define SS_PIN 10
#define RST_PIN 9

MFRC522 mfrc522 (SS_PIN, RST_PIN);

void set up()
{

Serial.begin(9600); SPI.begin(); mfrc522.PCD_Init ();

```
pinMode (2, OUTPUT); // The pin 2 is now an output (Connects a LED)
}

void loop()
{
if (. Mfrc522.PICC_IsNewCardPresent ())
{
return;
}

if (. Mfrc522.PICC_ReadCardSerial ())
{
return;
}

long code = 0;

for (byte i = 0; i <mfrc522.uid.size; i ++)
{
code = ((code + mfrc522.uid.uidByte [i]) * 10);
}

Serial.print("The card number is"); Serial.println(code);
// From here takes place the expansion of the program.

if (Code == 1031720) // If the number code is 1031720...

{ // Open Programmabschniss

digitalWrite (2, HIGH); should //...dann the LED on pin 2 light... delay (5000); // for 5 seconds
digitalWrite (2, LOW); //... and then go out again

} // close the program section


} Completing // Sketch
```

Chapter 4

Accessories for Arduino

Keypad Shield

A Keypadshield has that one, the LCD display need not wire up in a complicated way, and that six additional buttons has that you can use to advantage. The special lies with the keys are that we can read them using an analog pin in the program code. Because it connects the keys via different resistances, all with an analog pin (A0). Pin A0 can, therefore, be restricted only used for other things. The reason is on the Shield no slot for the A0 pin.

The Keypadshield is placed such a way so that the pins for the power right into the pins of the power supply on the Arduino board fits (see in the image below in the middle of the pins for the power supply. A clue can be the pin that says "VIN"). The upper slots of the Arduino boards also are covered by the shield (Pin 0-13). Some may anyway not be used because the LCD display uses. The unneeded pins were summarized in a series of slots (see photo above).

If you want to use these slots, you should solder the cable sockets there. As an example, we can use the following code:

Code:

```
// Sample using Liquid Crystal library #include <Liquid Crystal.h>
/ ********************************************** ******
This program will test the LCD panel and the buttons Mark Bramwell, July 2010
********************************************** ******
/

// select the pins used on the LCD panel Liquid Crystal lcd (8, 9, 4, 5, 6, 7);
```

```
// define some values used by the panel and buttons int lcd_key = 0;
int adc_key_in = 0; #define btnRIGHT 0
#define btnUP 1
#define btnDOWN 2
#define btnLEFT 3
#define btnSELECT 4
#define btnNONE 5

// read the buttons int read_LCD_buttons ()
{
adc_key_in = analog read(0); // read the value from the sensor
// my buttons When read are centered at Valie thesis: 0, 144, 329, 504, 741
// we add approx 50 tons Those values and check to see if we are close
if (Adc_key_in> 1000) return btnNONE; // We make this the 1st option for speed reasons since it will be the most likely result

if (Adc_key_in <50) return btnRIGHT; if (Adc_key_in <195) return btnUP; if (Adc_key_in <380) return btnDOWN; if (Adc_key_in <555) return btnLEFT;
if (Adc_key_in <790) return btnSELECT;

return btnNONE; // When all others fail, return this...
}

void set up()
{
lcd.begin(16, 2); // start the library lcd.setCursor(0,0);
lcd.print("Message"); // print a simple messagepinMode (2, OUTPUT);
}

void loop()
{
digitalWrite (2, HIGH);
lcd.setCursor(9,1); // move cursor to second line "1" and nine spaces over LCD.print(millis() / 1000); // display seconds elapsed since power-up
```

```
lcd.setCursor(0.1); // move to the begining of the second line lcd_key =
read_LCD_buttons (); // read the buttons


switch (Lcd_key) // DEPENDING ON whichButton was pushed, we
perform in action
{
case btnRIGHT:
{
lcd.print("RIGHT"); digitalWrite (2, LOW); break;
}
case btnLEFT:
{
lcd.print("LEFT"); break;
}
case btnUP:
{
lcd.print("UP"); break;
}
case btnDOWN:
{
lcd.print("DOWN"); break;
}
case btnSELECT:
{
lcd.print("SELECT"); break;
}
case btnNONE:
{
lcd.print("NONE"); break;

}
}
}
```

Instructions for LCD Display with I2C Port

The LCD module with a soldered I2C bus allows the use of an LCD module with simple wiring. This is particularly advantageous in more complex projects. Another difference from the normal LCD display is that a knob is located on the back of the display, with the brightness of the LCD can be added.

Note:
These instructions will only work with an I²C module on the back of the display without three solder points labeled A0, A1, and A2.

Material: microcontroller (in this example, UN R3), LCD with I2C module, cable

Wiring: The wiring is very simple. On the I2C LCD module, only four contacts are available. GND is connected to the GND contact on the microcontroller. VCC 5V with the contact on the microcontroller, SDA to the analog input A4 and SCL with the analog input A5.

Attention: In the MEGA2560 R3, microcontrollers are available for the SDA - and SCL contacts separate entrances on the board under 20 and 21.

Program

To work with the I2C LCD module needs a library that is not pre- installed in the Arduino program. This can be downloaded, for example, https://github.com/fdebrabander/Arduino-LiquidCrystal-I2C-library as a.ZIP file. After that, the library must be added to the program in Arduino.

This can be done under the "Sketch" then "Include Library" and "add.ZIP Library..". Now it can be accessed in the code to the library.

Sketch:

```
#include <Wire.h> // Wire Library Upload

#include <LiquidCrystal_I2C.h>// Previously added
LiquidCrystal_I2C
// Library Upload

LiquidCrystal_I2C lcd (0x27, 16, 2);    // This determines what kind
// a display it is. In this case, a 16-character 2
```

```
// lines.

void set up()
{
lcd.begin(); // In the setup, the LCD is started (unlike the simple LCD module without 16.2 in brackets for the previously specified
}

void loop()
{
lcd.setCursor (0,0); // From here the I2C LCD module as the simple LCD module can be programmed exactly.

lcd.print("XYZ GmbH");
lcd.setCursor (0.1); // lcd.setCursor to characters and line indicate

lcd.print("I wish you success.");//lcd.print something on the screen to show
//to let.
}
```

Example of use:

With the I2C LCD module can as with the simple LCD module, it displays also measured values.

Here is an example code in which a moisture sensor to pin A0 is connected:

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h> LiquidCrystal_I2C lcd (0x27, 16, 2);
int reading = 0;
void set up()

{

lcd.begin();

}

void loop()
```

```
{

reading =analog read(A0); // The value from analog input A0 to
// read, and the variable "reading" are stored.

lcd.setCursor(0,0); // The first line of the text is "reading"
//are displayed.

lcd.print("Reading");

lcd.setCursor(0.1);  // In the second line to the measured value from
// moisture sensor has been determined to be displayed.
lcd.print(Measured value);
delay(500);

}
```

Two I²C Displays

Use two I²C displays at the same time, Arduino.

Caution:

This structure and the associated change in the I²C address is only possible with displays that have a jumper function. We can see this in the following image to the red area. On the points A0, A1, and A2, we can solder a contact bridge.

In addition, the NewliquidCrystal_1.3.4 Library is required for these instructions which can be downloaded here: https://bitbucket.org/fmalpartida/new- liquid crystal/downloads

The library needs to be added to the Arduino software (see previous instructions I²C display).

Material: Arduino / 2 displays with I²C module / Breadboard / cable

Task: Two LCDs with I²C module to be driven simultaneously by an Arduino microcontroller and display two different texts. For this, the I²C addresses are the displays that are found earlier, and we can change the address of display.

First, a Brief Explanation of the I²C Address

Each I²C module has a so-called "HEX address." Using this address, the I²C module reacts to the data being sent from the Arduino on the data bus at exactly that address. Many I²C LCDs also have the same HEX address. That means it would react with the use of two displays, both displays to the transmitted data from the Arduino board. One might, therefore, represent two displays with no different data.

The HEX address can at the screen using the A0, A1, and A2 solder joints. However, it changed. As established, all three solder joints are not connected depending on the combination, which one of the bridged areas with a solder joint, so eight different addresses are possible. Depending on display type, this address may initially be 0x27 or 0x3F (we can find it with the address "scanner," more on that later).

Tables HEX addresses depending on the soldered points (I = connected: = not connected):

A0	A1	A2	HEX address	HEX address
:	:	:	0x27	0x3F
I	:	:	0x26	0x3E
:	I	:	0x25	0x3D
I	I	:	0x24	0x3C
:	:	I	0x23	0x3B
I	:	I	0x22	0x3A
:	I	I	0x21	0x39
I	I	I	0x20	0x38

We come to the I²C addresses "Scanner":

The "Scanner" is basically where the LCD module is connected; only one code is uploaded to the Arduino and then displays the HEX address to the serial monitor.

Step Guide to I²C addresses "Scanner":

1. The same LCD module with the Arduino connected:

- I²C LCD module >> Arduino 5V VCC >>
- GND GND SDA >> >> >> A4 SCL A5

2. Load following sketch to the microcontroller:

```
// I2C scanner
// Written by Nick Gammon
// Date: 20th April 2011 #include <Wire.h>
void setup () {

Serial.begin (115200); // Leonardo: wait for serial port to connect while (.
// Serial)
{
}

Serial.println ();

Serial.println ( "I2C scanner scanning...."); byte count = 0;
Wire.begin ();

for (i byte = 8; i <120; i ++)

{

Wire.beginTransmission (i);

if (Wire.endTransmission () == 0)

{

Serial.print ( "Found address"); Serial.print (i, DEC); Serial.print ( " (0x"); Serial.print (i, HEX); Serial.println ( ")");
count ++;

delay (1); // maybe unneeded?

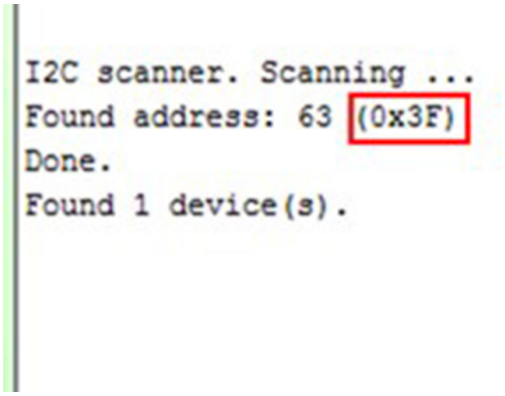
} // end of good response

} // end of for loop Serial.println ( "Done."); Serial.print ( "Found"); Serial.print (count, DEC); Serial.println ( "device (s).");
```

}
void loop () {}

3. Open Serial Monitor in the Arduino software
4. Change baud rate to 115200
5. Possibly. Press the Reset button on the Arduino

In the serial monitor, the following should be shown (HEX address is highlighted in red):



Again, the actual instructions:

To be able to display on two I²C displays simultaneously two different texts, the displays must have different HEX addresses. So we solder in our example, at one of the displays the A1 contact, so that it now has the address 0x3D (can be checked with the addresses scanner again). We call this display from now on display 2 and the other display. 1

Code:

#include <Wire.h> integrate //Wire.h Library
#include <LiquidCrystal_I2C.h> integrate //LiquidCrystal_I2C.h Bibliothel LiquidCrystal_I2C LCD1 (0x3F, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE); // Here is
// display one named as LCD1, given the address "0x3F" and the
// Pinassignment through the I²C module specified
LiquidCrystal_I2C LCD2 (0x3D, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);// Here is
// Display 2 named as LCD2, the address specified "0x3D" and the
// Pinassignment through the I²C module specified void set up()

```
{
LCD1.begin(16.2); // Display 1 is started and the size of the display
// is specified (16 characters, two lines) LCD1.backlight(); Switch //
Lighting Display 1
LCD2.begin(16.2); // Display 2 is started and the size of the display
// is specified (16 characters, two lines) LCD2.backlight(); Switch //
Lighting Display 2
}
void loop()
{
LCD1.setCursor(0,0); // text is the first character in the front row
// start at Display 1.. LCD2.setCursor(0,0); // display at 2 also..
LCD1.print("Display 1"); // Display 1 to the top "Display 1" Show
LCD2.print("Display 2"); // Display 2 should show up "display 2"
delay(1000); // wait a second
LCD1.setCursor(0.1); // text is the first character in the second row
// start at Display 1.. LCD2.setCursor(0.1); // display at 2 also..
LCD1.print("XYZ GmbH"); // Display 1 to bottom "XYZ GmbH"

LCD2.print("Www.xyz.com");// Display 2 "Www.xyz.com"
}
```

Arduino RTC

Time Module for Arduino: Real-Time Clock - RTC compatible with Arduino Real Time Clock DS1307 I2C can display time and date in near real time.

Material: Microcontroller Board (In this example, UN R3). Navigation wheel (or potentiometer), dashboard, LCD display, DS1307 I2C Real-Time Clock, connection cable.

Cabling:

We wire the LCD display in principle as in the instruction no. 13. These then must be added to the RTC, which is again connected to 5V and GND, the SDA contact to the analog input A4, and the SCL contact to the analog input A5.

Caution.: In the MEGA2560 R3, microcontrollers are available for the SDA - separate entrances on the board at 20 and 21 and SCL contacts.

If everything is wired, you can also be able to start correctly with programming, almost. Before must need for programming, download libraries and added to the Arduino software.

If you have downloaded both libraries, you have to add them to the Arduino software to the existing libraries. For this, open the Arduino software and selects the Punk "sketch" the "Include Library" box. Then you can select "Add.ZIP Library" and add the previously downloaded Time and DS1307RTC Library.

We can now draw this in code.

Next, you have to the so-called "Set Time" Sketch Upload the time and date from the device (computer, laptop, etc.), on which the microcontroller is connected to add to the real-time clock.

This is done by then "DS1307RTC" select "File" to "examples" to select "Set Time." This opens the appropriate Sketch which we now upload on the UN R3.

If you have done all, you can get to the actual code.

Code:

```
#include <Time.h> Libraries invite #include <Wire.h>
#include <DS1307RTC.h> #include <Liquid Crystal.h>
Liquid Crystal LCD (12, 11, 5, 4, 3, 2); which pins is the microcontroller boards used for the LCD // This line is set,
void set up()
{
lcd.begin(16, 2); // In the Setup specifies how many characters and lines are applied.
Serial,begin(9600); // Opens the serial port and sets the baud rate (9600) determined for the serial transmission. setSyncProvider (RTC.get); // This is to get the function to the time and date of the RTC
}
void loop()
{
```

```
Serial,print(Hour ()); //Serial.print is the command show something in the serial monitor (hour, minute, second, space, day, spaces, etc.)
print digits (minute ()); // in minutes and seconds, the command is
print digits (second ()); // print digits specified which will be determined at the end of the code.
Serial,print(""); Serial,print(Day ()); Serial,print(""); Serial,print(Month ()); Serial,print( " "); Serial,print(Year ()); Serial,println();
delay(1000); //wait a second
lcd.setCursor(2, 0); // setCursor indicates where the text should begin. In this case, the third character in the front row.lcd.print(Hour ()); // Here is the time to be displayed now, so "hour" "" "minute," etc..
lcd.print ( ":"); lcd.print (minute()); lcd.print(":"); lcd.print(Second ());
lcd.print("");
lcd.print("Clock"); // Here the word "clock" is after the time are displayed to have to follow NOCH 3 spaces, otherwise in numbers <10 would always another "r" are displayed behind PM. This is because the LCD Library command a view 0 before numbers <10 does not exist.
lcd.print("");
lcd.print("");
lcd.print("");
lcd.setCursor(1, 1); // The next "text" will now start at the second character in the second row.
lcd.print(Day ()); // The date is now as
lcd.print("."); // "day," "." "Month," etc. are given. lcd.print(Month ());
lcd.print(".");
lcd.print(Year ());
}

void print digits (int digits) // This section fixed that automatically becomes 0 before the digits for numbers <10 in the serial monitor. This only applies to the serial monitor instead of the LCD display.
{
Serial,print(":"); if(Digits <10) Serial,print('0'); Serial,print(Digits);
}
```

Note: The DS1307 Real Time Clock is not accurate to the second. The reason is that between uploading the Set Time Sketch and uploading the

final sketch, a certain period of about 15 seconds can not compensate for the RTC.

In addition, you can still display the week before the date.

For this purpose just before the date in lcd.print section of printDay () command; are added and are inserted after the loop following code:

```
void printDay () is // This sets a meaning for those already used in the code before command printDay
{
int day;
day = weekday (); // The weekdays are to be displayed depending on the date.
if(Day == 1) {lcd.print("So,")} // If this is Day 1 to display "Sun" and so on.
if(Day == 2) {lcd.print("Mo,")}
if(Day == 3) {lcd.print("Di")}
if(Day == 4) {lcd.print("Mi")}
if(Day == 5) {lcd.print("Do,")}
if(Day == 6) {lcd.print("Fri")}
if(Day == 7) {lcd.print("Sa")}
}
```

Extension:

Two I²C Activate components simultaneously: the time and date with a Real Time Clock with I²C bus, LCD display on a I²C.

To accommodate the two I²C components actuate the same time, one of the components must be another I²C address than the other. In our case, we can only change the address of the LCDs. Provided that the three soldering points A0, A1, and A2 on the rear of the module has I²C.

The I²C address of the Real-Time Clock is set and can not be changed.

To change the address of LCDs, the solder joints need to be connected differently (separated all three solder joints) at the beginning. Thus, the HEX address changes of LCDs (s.

Table) (I = connected,: = not connected):

A0	A1	A2	HEX Adresse	HEX Adresse
:	:	:	0x27	0x3F
I	:	:	0x26	0x3E
:	I	:	0x25	0x3D
I	I	:	0x24	0x3C
:	:	I	0x23	0x3B
I	:	I	0x22	0x3A
:	I	I	0x21	0x39
I	I	I	0x20	0x38

"simultaneously control two displays with I²C module" are checked with the "Scanner" from the manual.

If the address of LCDs has been changed, we can now proceed to the wiring:

For programming as already mentioned above, the time and the DS1307RTC Library needed. In addition, the NewliquidCrystal_1.3.4 Library is required for the I²C LCD.

All libraries have the Arduino software is added:

Select Sketch> Include Library> Add ZIP library> previously downloaded file Next, you have to the so-called "Set Time" Sketch Upload the time and date from the device (computer, laptop, etc.) On which it connects the microcontroller, on the set of real-time clock.

This is done by "File" to "examples" then "DS1307RTC" the point

"Set Time" selects. This opens the appropriate Sketch which we now upload on the UN R3.

If you have done all you can get to the actual code.

```
#include <Time.h>
#include <Wire.h> #include <DS1307RTC.h>
#include <LiquidCrystal_I2C.h> // Load Libraries
LiquidCrystal_I2C lcd (0x3D, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE); // The I²C Display rename and the HEX address // Insert (for us 0x3D)
void set up() {
lcd.begin(16, 2); // Start the display, specify that it is a display with 16 characters in 2 lines are //
lcd.backlight(); Switch // lighting of the display Serial,begin(9600); // start serial connection with baud rate 9600 setSyncProvider(RTC.get); // Retrieve data from the RTC
}
void loop() {
Serial.print(hour()); //Serial.print is the little display in the serial monitor (hour, minute, second //, space, day, spaces, etc.) Command print digits (minute()); // in minutes and seconds, the command is print digits (second()); // print digits used, which is still set at the end of the code
Serial.print(""); Serial.print(day()); Serial.print (""); Serial.print(month()); Serial.print(""); Serial.print (year ()); Serial,println();
delay(1000); // wait one second
lcd.setCursor(2, 0); // setCursor indicates where the text should begin. In this case, the third character in // the front row.
lcd.print(hour()); // The time is to be displayed in the format: lcd.print(":"); //Hours minutes seconds
lcd.print (minute()); lcd.print(":"); lcd.print(second()); lcd.print("");
lcd.print("Clock"); // Behind, the word "clock" display lcd.print("");
lcd.print("");
lcd.print("");
lcd.setCursor(1, 1); // The second line will display the date // be
lcd.print(day());
lcd.print(".");
lcd.print(month());
lcd.print(".");
lcd.print(year());
}
```

```
void print digits (int digits) { // The print command digits for the serial
monitor Serial,print(":");
if(Digits <10) Serial,print('0'); Serial.print(Digits);
}
```

Keypad

A keypad on the Arduino.

Task: With the Arduino, a keypad (keyboard) to read out, and the pressed key are displayed on the Serial Monitor.

These instructions a 3 x 4 keypad used. Alternatively, a4 x 4 keypad is used here pins, and defined keys (COLS / HEXAKeys) need to be added (see below).

Material: Arduino / cable (7x) / keyboard / power supply

To use the keypad, you need the keypad Library. The Library of the Arduino IDE to add:

- Open the Arduino IDE
- Menu item: - embed> Library - Sketch> Library Management

Search -with the search line to "Keypad," find and install (the version of Mark Stanly)

The keypad comprises a 3 x 4 array of 12 keys.

To understand the functioning of the keypad you have to imagine how a coordinate system. Each key is associated with two pins; a pin for the columns (COLS) and one for the rows (ROWS). If a key is pressed, the corresponding pins are connected. These pins can read the Arduino with the digital read () function. (The keyboard requires no external power supply.)

The Sketch:

```
#include <keypad.h>
// Here, we define the size of the keypad const byte COLS = 3; // 3
columns
const byte ROWS = 4; // 4 lines
```

```
// The digits / characters:
char HEXAKeys [ROWS] [COLS] = {
{.# '..0 '..* '},
{.9 '..8th'..7 '},
{.6 '..5 '..4 '},
{.3 '..2 '..1'}
};
byte colPins [COLS] = {8, 7, 6}; // definition of the pins for the 3 columns byte rowPins [ROWS] = {5, 4, 3, 2};// definition of the pins for the 4 lines char pressedKey; // pressedKey corresponds in the future, the keys pressed keypad myKeypad = keypad(makeKeymap(HEXAKeys) rowPins, colPins, ROWS,
COLS); // The keypad can be addressed from now on // with myKeypad void set up() {
Serial.begin(9600);
}
void loop() {
pressedKey = myKeypad.getKey(); // pressedKey corresponds to the pressed key if (PressedKey) { // If a key is pressed
Serial,print("The key "); Serial,print(PressedKey); Serial,print(" was pressed");
Serial,println(); // Tell us the Serial Monitor the pressed key with
}}
```

Using a 4 x 4 keypad (the sections are changed):

```
> const byte COLS = 4; // 4 columns
> byte colPins [COLS] = {9, 8, 7, 6}; // definition of the pins for the 4 columns

> char HEXAKeys [ROWS] [COLS] = {
{, D '. , #.. , 0.. *.},
{, C '..9 '..8th'. , 7 '},
{, B '. , 6.. , 5...4 '},
{, A '..3 '. 2...1'}
};
```

Keypad - Castle

With the number keypad code lock programming Task: With the entry of a code 3 digit on the keypad to an LED light up and a servo occupy a certain position. The servo example could be a bolt are attached that unlocks a door.

Material: Arduino / Breadboard / keypad (in this example, 4 × 4) / cable / a red LED / green LED / two 100 Ohm resistors / Servo

We want to leave a green LED light up, typing a 3-digit password on the keypad and can occupy a certain position a servo. If the "lock" is closed to light up and the servo taking another position, a red LED. This tutorial serves as inspiration and an example of how a "safe" can be built from these simple components, for example.

We now come to the construction. This is quite simple and goes from the following Fritzing sketch out.

For Better Orientation in Wiring the Keypad:

At the extreme contacts of the keypad to enter the numbers, you can see one and seventh it connects The contact 1 at the keypad to pin 2 on the Arduino. Ascending, it then proceeds to contact 7 which is connected to the pin 8 at the Arduino, the keypad.

For the code, we need the keypad library, what the Arduino software has to be added.

Open Arduino Software> "Sketch"> Select "Include Library"> choose "Manage Libraries.." select> In the search bar of the new window "Keypad"> enter the top library Select by Mark Stanley and install.

From now on, we can use the keypad Library in the code.

Code:

```
#include <keypad.h> // keypad and servo Library is integrated
#include <Power.h>
Power servo-blue; // servo is now addressed with "servo blue"
char* Password = "123"; // The password is set. In this example
// "123"
int position = 0;
```

```
const byte ROWS = 4; // This indicates how many rows and columns the const byte COLS = 3; // keypad has
char keys [ROWS] [COLS] = { be // The numbers and characters on the keypad
{# ". '0'. * '}, // specified
{9 '. ,8th'. , 7 '},
{6 '. , 5 '. , 4 '},
{, 3 '. , 2 '. ,1'}
};
byte rowPins [ROWS] = {5, 6, 7, 8}; // The connection with the Arduino is
byte colPins [COLS] = {2, 3, 4}; // set
keypad = keypad( makeKeymap(Keys), rowPins, colPins, ROWS, COLS); int RedLED = 12; // The red LED is connected to pin 12
int GreenLED = 13; // The green LED is connected to pin 13 void set up()
{
pinMode(RedLED, OUTPUT); // The LEDs are set as output pinMode(GreenLED, OUTPUT);
servo blue.attach(11); // The servo is connected to pin 11

setLocked (true);
}
void loop()
{
char key = keypad.getKey();
if (Key == * ' || key ==# ") // when the lock is unlocked it can with the
// key or "#" are blocked again "*"
{
position = 0;
setLocked (true); Lock // castle once was * or press #
}
if (Key == password [position])
{
position ++;
}
if (position == 3) // This line length indicates the password. In our
```

```
// Case 3 points.
{
setLocked (false);
}
delay(100);
}
void setLocked (int locked)
{
if (Locked) // If the lock is locked to..
{
digitalWrite(RedLED, HIGH); //..die red LED lights..
digitalWrite(GreenLED, LOW); //..die green LED does not light..
servo blue.write(90); // and the servo is to control 0 degrees.
}
else // when the lock is unlocked to..

{
digitalWrite(RedLED, LOW); //..die red LED does not light..
digitalWrite(GreenLED, HIGH); //..die green LED lights.. servo
blue.write(0); //..und the servo drive 90 degrees.
}
}
```

This guide serves as an example and foundation of how a simple "castle" using fewer components and can construct the Arduino.

Color Sensor

Task: Using a color sensor, the colors are red, green, and blue are detected, and a corresponding LED light up.

Material: Arduino / cable / color sensor / Red LED / Green LED / Blue LED / resistors for LEDs are to obtain more accurate values 4 white LEDs on the color sensor. In the center of the LEDs is a small square of photodiodes, which may filter the intensity of a color. There are photodiodes for the green, red and blue color, and photodiodes without color filters present. This "filter" out what color is held before the color sensor and convert them to values around that can handle the microcontroller.

We can use the color sensor, for example, in the construction of a sorting machine.

For our test setup, we made the following structure with appropriate wiring

Cabling:

Color sensor >>> Arduino VCC 5V >>>
GND GND >>>
S0 >>> 8th
S1 >>> 9
S2 >>> 12
S3 >>> 11
OUT >>> 10
OE >>> GND
LEDs >>> Arduino Red LED >>> 2
green LED >>> 3 Blueness LED >>> 4

This is a sample code with the three colors of red, green and blue are read by Arduino. At the same time indicate three correspondingly colored LEDs which color has been recognized by the sensor.

```
const int s0 = 8; // Set the connection of the color sensor contacts with the Arduino const int s1 = 9;
const int s2 = 12; const int s3 = 11; const int out = 10;
int RedLED = 2; // Set the connection of the LEDs with Arduino int GreenLED = 3;
int BlueLED = 4;
int red = 0; call // variables for LEDs int green = 0;
int blue = 0; void set up()
{
Serial.begin(9600); // Start Serial Communications
pinMode(S0, OUTPUT); // The contacts of the color sensor are used as output or pinMode(S1, OUTPUT); // Input festgelgt
pinMode(S2, OUTPUT); pinMode(S3, OUTPUT); pinMode(out, INPUT);
pinMode(RedLED, OUTPUT); // The LEDs are defined as output pinMode(GreenLED, OUTPUT);
pinMode(BlueLED, OUTPUT);
```

```
digitalWrite(S0, HIGH); // The four white LEDs at the color sensor will light up digitalWrite(S1, HIGH);
}
void loop()

color ();// This function is determined at the end of the code (s. "Void color ();") Serial.print("Value Red"); // On the serial monitor is to "value" each Serial.print(red, DEC); // with the appropriate color display Serial.print("Value Green");// be behind it and the value obtained in the Serial.print(green, DEC); // void color (); Function was read out.
Serial.print("Value Blue"); Serial.print(blue, DEC);
// Here are the commands for the LEDs
if (Red <blue && red <green && red <20) // If the filter value for red
// is smaller than all other values..
{
Serial.println(" - (Red color)"); are displayed on the serial monitor //
//..should "red color" and..
digitalWrite(RedLED, HIGH); //... ie red LED lights up, the others
digitalWrite(GreenLED, LOW); // stay out
digitalWrite(BlueLED, LOW);
}
else if (blue <red && blue <green) // The same in blue and green
{
Serial.println (" - (Blue colour)"); digitalWrite(RedLED, LOW);
digitalWrite(GreenLED, LOW); digitalWrite(BlueLED, HIGH);
}
else if (Green <red && green <blue)
{
Serial.println(" - (Green colour)"); digitalWrite(RedLED, LOW);
digitalWrite(GreenLED, HIGH); digitalWrite(BlueLED, LOW);
}
else{ // If there are no values..
Serial.println(); Show //..nichts on the serial monitor and..
}
delay(300);
```

```
digitalWrite(RedLED, LOW); //... turn off all LEDs
digitalWrite(GreenLED, LOW); digitalWrite(BlueLED, LOW);
}
void color () // This is where the values are read by the color sensor and under the
// corresponding variables stored
{
digitalWrite(S2, LOW); digitalWrite(S3, LOW);
red = pulseIn(out, digital read(Out) == HIGH ? LOW : HIGH);
digitalWrite(S3, HIGH);
blue = pulseIn(out, digital read(Out) == HIGH ? LOW : HIGH);
digitalWrite(S2, HIGH);
green = pulseIn(out, digital read(Out) == HIGH ? LOW : HIGH);
}
```

To check our building, we have printed us a table with a red, green and blue surface.

Depending on what color you hold the color sensor, the corresponding LED should now light up.

Generate Tones

Produce with the Arduino and a speaker sounds.

With the Arduino can produce sounds in different ways. The simplest way is the tone generation with an active loudspeaker (active buzzer), which is connected to only the voltage of 5V. The sounds produced by a built inside electronics. The disadvantage is that a

"Active buzzer" can only produce a single tone - tunes or siren sounds are not possible.

With a passive Loudspeaker (passive buzzer), you have the option using the Arduino microcontroller different sounds, tunes, or siren signals to generate, since buzzer in passive no electronics is present, sets a tone.

Task: Passive speakers are different tones and a melody to be generated.

Material: Arduino microcontroller / A passive speaker (passive buzzer) / Breadboard / cable

The generation of sound is largely based on the command "tone (x, y),” where the x value indicative of the pin on which it connects the speaker to the positive side and the Y value that indicates the pitch.

An example:

tone (8, 100); // The speakers on pin 8 is activated with the pitch "100."

delay (1000); // break with 1000 milliseconds, or 1 second - The speaker remains active for this time.

noTone (8); // The speakers on pin 8 is disabled

Code 1: Simple Tone

```
void set up() // The Setup information is not required.
// The voltage output for the piezo speaker no command "tone" is automatically set in the sketch by the Arduino,
{
}

void loop()
{
tone(8, 100); // The main part will be given the command "tone (x, y)" a sound.
delay(1000); // with a duration of 1 second noTone(8th); // The sound is turned off
delay(1000); // The speaker one second remains off
}
```

Code 2: Alternating Pitches

In the second example, only a few lines are complemented in the main part (loop). Characterized sounding two tones alternately with the pitch "100" or "200,” as in a siren. A break between the tones does not exist.

```
void set up()
{
}
void loop()
{
tone(8, 100); delay(1000);
```

noTone(8th); // At this point, the first sound goes out.
tone(8, 200); sounds // The second tone with the new pitch "200". delay(1000); //... and that for a second...
noTone(8th); // At this point, the second sound goes off, and the loop part of the sketch begins again.
}

Code 3: Tone Generate by Keystroke

In the third example, a button on pin 6 is added to the construction. In the main part of the button is read by an "IF query." If the button is pressed, you will hear a tone for one second with the pitch "300".

```
int Taster1 = 6;
int Button status = 0; void set up()
{
pinMode(Taster1, INPUT);
}
void loop()
{
Probe status = digital read(Button); if (Button == Status HIGH)
{
tone(8, 300); noTone(8th);
}}
```

Code 4: Sounds Depending on Different Keys

In the fourth example, different keys should be decided depending on the actuation of which sound is emitted from the speaker. For this purpose, it connects a probe connected to pins 6 and 7 in each case. The main part is decided by two "IF statements," in which sound is emitted.

Code:

```
int Taster1 = 6; // Taster1 connected to pin 6 int Button2 = 7; // button2 connected to Pin7
int Tasterstatus1 = 0; save // variable to indicate the status of the probe. 1 int Tasterstatus2 = 0; save // variable to indicate the status of the probe. 2

void set up()
```

```
{
pinMode(Taster1, INPUT); // Set Taster1 as input pinMode(Button2,
INPUT); // Set button2 as input
}

void loop()
{
Tasterstatus1 = digital read(Taster1);// Status of Taster1 read (HIGH or
LOW)
Tasterstatus2 = digital read(Button2); // Status of button2 read (HIGH
or LOW)

if (Tasterstatus1 == HIGH) // If the Taster1 is pressed, then...
{
tone(8, 100); // outputting a sound with the pitch 100th delay (1000); //
with the duration of one second noTone(8th); // off the sound.
}

if (Tasterstatus2 == HIGH)
{
tone(8, 200);
delay (1000); // with the duration of one second noTone(8th); // off the
sound.
}
}
```

Create Melodies

In the Arduino software is a file containing specially designed to generate sounds using a speaker (passive buzzer). Instead of the pitch in the form of numbers such as. "Tone (8, 200),” and sounds from the Sounds can now be selected.

As an example, there is for it in the Arduino software to sketch "toneMelody."

To see "examples" -> "02.Digital" -> "toneMelody"

In the example, the file is already stored, the association is in the pitch and numerical value. She has the name "pitches.h." The content you can see if

you click in the opened sample program on the second tab in the sketch.

To have access to the file must be in the sketch only the command #include

"Pitches.h" be installed and must be opened as a tab file. This is best done by opening the sketch "toneMelody" from the examples in the Arduino software and then edited. Thus, the "pitches.h" file is automatically invoked as a tab. This is clear in the following example.

This is a very small sketch of the alternating plays, two tones from the file "pitches.h".

```
#include "Pitches.h" void set up()
{
pinMode (8th,OUTPUT); // speaker Pin8
}

void loop()
{
tone(8, NOTE_C4, 1000); // At Pin8 played the note C4 for 1000ms
delay(3000); // After the note sounds, the Sketch for 3 seconds pause. This means that after it plays the sound to the end, the remaining two-second pause with no sound.
tone(8, NOTE_G3, 1000); // At Pin8 played the note G3 for 1000ms
delay(3000); // After the note sounds, the Sketch for 3 seconds pause. This means that after it plays the sound to the end, the remaining two-second pause with no sound.
}
```

In this sketch, the "Tone" command is no longer needed to end the sound through the extension.

Command:

"tone (x, y, z)" x = Pin of the speaker, y = pitch = z duration of tone in milliseconds.

Arduino Tilt Sensor SW-520D

Sketch: Reading With the Arduino a tilt sensor SW-520D

The Arduino can be in different inclinations measures. Very accurate measurements you get, for example, with an acceleration sensor in which one can almost measure any angle exactly. It is, however, also easy with the tilt sensor (SensorTilt or Tiltswitch).

However, this sensor detects only two stages, namely "inclined" or "not inclined." For this purpose, the sensor roll inside around two free-moving metal balls. Once the sensor is tilted so that one of the balls inside against the two contacts about these two contacts are interconnected. This is then as if a button is pressed, and the electric current can flow from one contact to another contact.

Therefore, the sketch and the structure is very similar to how the Stretch with the button.

When the sensor is used as in the picture vertically, for example, if it is simply plugged into the breadboard, the circuit between the two contacts is closed, and the current can flow from one contact to another.

Once the sensor is inclined by more than 45 °, the balls of the two contacts away and the connection is broken.

Task:

With a Tilt sensor to be turned on, an LED, when an inclination of more than 45 ° is present

Material: Arduino / a LED / A resistor 100 Ohm / A resistor 1K ohms (1000 ohms) / Breadboard / cable / inclination sensor

The Sketch:

```
int LED = 6; // The word "LED" is now available for the value. 6
int TILT = 7; // The word "TILT" is now available for the value 7 - The inclinometer is thus connected to Pin7
int TILT = 0; // The word "TILT" is now first for the value 0. Later will be saved under this variable, whether a slope of more than 45 degrees or not.
void set up() // Here the setup starts.
{
```

pinMode(LED, OUTPUT); // The pin with the LED (pin 6) is now an output. pinMode(TILT, INPUT); // The pin with the tilt sensor (pin 7) is now an entrance.
}

void loop()
{ // This brace is the Loop section opened.
TILT =digital read(TILT); // Here the Pin7, so the inclination sensor is read (: digital read command) is. It stores the result under the variable "TILT" with the value "HIGH" for 5V or "LOW" for 0Volt. When the sensor is even, the current can flow between the two contacts of the sensor. It is then up to Pin7 to a voltage. The status would be so "HIGH." When the sensor 45 or more degrees is inclined, no current can flow, and the status would be "LOW."

if (TILT == HIGH) // processing: If the sensor is inclined less than 45 degrees
{ Open // Program section of the IF command. digitalWrite(LED, LOW); // then the LED will not light
} // close the program section of the IF command. else //...otherwise...
{ Open // Program section of the else command. digitalWrite(LED, HIGH); //...should the LED lights.
} // close the program section of the else command.
} // This last clip of the loop portion is closed.

Temperature Measured with the LM35

Task:

With the temperature sensor, LM35 temperature is to be read and displayed monitor serial-by.

Material:

Arduino / cable / LM35 temperature sensor/screw Shield or cable for soldering to the sensor

The sensor has three connections. The red Contact 5V, the black GND, and the yellow is the contact for the temperature signal. At this contact, the

sensor outputs a voltage between 0 and 1.5 volts. Wherein 0V 0 ° C and the corresponding value of 1.5V corresponds to 150 ° C. The voltage of this pin has to read from the microcontroller board and converted into a temperature value.

- The special at this temperature sensor is its waterproof feature. Thus, the temperature can be measured by liquids with the LM35.
- ***NOTE*** : If the sensor is connected incorrectly, it will burn out.
- In the structure of an external power supply should be used if possible, as this is it improves considerably the sensor accuracy (9V AC adapter or 9V battery).

Code:

```
int LM35 = A0; // The sensor A0 is to be connected to the analog pin. We call the pin from now "LM35."

int temperature = 0;// Under the variable "temperature" the temperature value is saved later.

int temp [10]; // This yields good values, one first read several values and then averages the results. The square bracket "[10]" produced here equal to ten variables named
"Temp [0],” "temp [2],” "temp [3],” … to … "Temp [9]." With the notation [10], So it saves only a little Place.

int time = 20; // The value for "time" specifies the time intervals between the individual measurements in the code.

void set up()
{
Serial.begin(9600); // In the setup we start serial communication
}

void loop()
{

// more detailed explanation under the code * 1
```

```
temp [0] = map(analog read(LM35),

delay(Time);  0      410   -50,   150);
temp [1] = map(analog read(LM35),  0      410   -50,          150);
delay(Time);
temp [2] = map(analog read(LM35),  0      410   -50,          150);
delay(Time);
temp [3] = map(analog read(LM35),  0      410   -50,          150);
delay(Time);
temp [4] = map(analog read(LM35),  0      410   -50,          150);
delay(Time);
temp [5] = map(analog read(LM35),  0      410   -50,          150);
delay(Time);
temp [6] = map(analog read(LM35),  0      410   -50,          150);
delay(Time);

temp [7] = map(analog read(LM35),  0      410   -50,          150);
delay(Time);

temp [8] = map(analog read(LM35), 0, 410, -50, 150); delay(Time);
temp [9] = map(analog read(LM35), 0, 410, -50, 150);

temperature = (temp [0] + temp [1] + temp [2] + temp [3] + temp [4] +
temp [5] + temp [6] + temp [7] + temp temp [8] + [ 9]) / 10;

Serial.print(temperature); Serial.println(" Centigrade");
}
```

Detailed Explanation for Loop * 1:

Here the temperature is ten times read in between is ever a break with the duration

"Time" in milliseconds. But what happens here, exactly? Let's look at the command once more closely.
temp [1] = map (analog read (LM35), 0, 307, 0, 150); Sequentially:
temp [1] - is the name of the first variable.
"Map (a, b, c, d, e)" - This is the so-called "Map" command. This may be a read-in value (a) in which the values between (b) and (c) are,

convert them into a different speed range, in the range between (d) and (e).

In our command, the following happens:

The value of the sensor is directly read out in the "Map" command with "analog read (LM35)". The measured values should be between 0 and 307th. This corresponds to the analog port of the values between 0 V and 1.5V. The voltage is the sensor at temperatures between 0 ° C and

+ 150 ° C from. These values on the analog port will now be through the "Map command" directly into the

° C values between 0 and 150 are converted.

Then every ten ascertained temperature values are added together and divided by ten. The average value is stored under the variable "temperature."

Finally, the value of "temperature" on the serial communication to the PC is sent. Now the serial monitor you have to open the Arduino software to read the temperature on the sensor.

Expansion of the Program:

Once the temperature has reached 30 ° C, to a warning sound (beep of a piezo speaker).

Code:

```
int LM35 = A0;
int temperature = 0; int temp [10];
int time = 20;

int piezo = 5; // The word "piezo" is now the number five, so five of the piezo is connected to the port.

void set up()
{
Serial.begin(9600);
```

```
pinMode (Piezo, OUTPUT); // The piezo speaker to pin 5 should be (logic one output, because the yes from the microcontroller board yes a voltage needed to beep.
}

void loop() {
temp [0] = map(analog read(LM35),  0      307   0          150);
delay(Time);
temp [1] = map(analog read(LM35),  0      307   0          150);
delay(Time);
temp [2] = map(analog read(LM35),  0      307   0          150);
delay(Time);
temp [3] = map(analog read(LM35),  0      307   0          150);
delay(Time);
temp [4] = map(analog read(LM35),  0      307   0          150);
delay(Time);
temp [5] = map(analog read(LM35),  0      307   0          150);
delay(Time);
temp [6] = map(analog read(LM35),  0      307   0          150);
delay(Time);
temp [7] = map(analog read(LM35),  0      307   0          150);
delay(Time);
temp [8] = map(analog read(LM35),  0      307   0          150);
delay(Time);
temp [9] = map(analog read(LM35),  0      307   0          150);

temperature = (temp [0] + temp [1] + temp [2] + temp [3] + temp [4] + temp [5]

+ Temp [6] + temp [7] + temp [8] + temp [9]) / 10; // All in a row.Serial.print(temperature);
Serial.println(" Centigrade");

if (Temperature> = 30) // Creates a IF condition: If the value for the temperature above or equal to 30, then...
{
digitalWrite(Piezo,HIGH); //... start beeping.
}
```

```
else // And if that is not so...

{

digitalWrite(Piezo,LOW); //... then be quiet.
}
}
```

Measure the Voltage

With the Arduino microcontroller board and a voltage sensor, one can measure the electrical voltage. We use a sensor that can lie to Measuring voltage in a range between 0 and 25V DC (direct current).

Task: A voltage with a voltage sensor (voltage sensor) measure and display on the serial monitor.

Material: Arduino / cable / voltage sensor

We want to measure a voltage of a voltage sensor and display them on the serial monitor. For example, a voltage sensor makes sense if you want to measure a voltage exceeding the maximum measurable voltage (5V) from the Arduino.

For example, a voltage sensor can measure in the range of 0-25V voltages. The module measures the voltage and converts them into measurable for the Arduino values.

By mathematical formulas in the code, it is possible to specify a voltage indicated on the serial monitor.

Cabling:

This is simply because the sensor has only three contacts, which must be connected to the Arduino.

Voltage sensor >>> >>> Arduino S A1
+ 5V >>>
– >>> GND

On the other side, there are labeled GND and VCC contacts to which is connected to the voltage to be measured.

We now come to the code:

```
int value1;
float value2;
void set up()
{
Serial,begin(9600); // start serial connection
}

void loop()
{
float temp;
value1 =analog read(1); // read voltage value at analog input 1 temp =
value1 / 4,092; // value mathematically convert to the voltage value to
obtain in volts
value1 = (int) Temp; value2 = ((% value1 100) /10.0);
Serial,println(Value2); Show // Final voltage value in the serial monitor
delay(1000); // wait a second
}
```

If you now open after successful upload of the code the serial monitor, you can now see a voltage value every second. Since it still connects no power source, this should be at 0.00.

To see if everything is working, you can test as to connect GND contact from the sensor to the GND contact from the Arduino and the VCC contact from the sensor to the 3.3V contact from the Arduino. Now a value should show up to 3.3V on the serial monitor. The sensor is not quite exact. So it was with us 3.2V.

Four-Digit 7-Segment Display

Task: We want to display any number on the four-digit seven-segment display.

Material: Arduino / cable / 4 digit 7-segment display / 1K ohm resistors

The 7-segment display has 12 pins at the back, six at the top, and six at the bottom. Four of these pins correspond to each number. They can be "common anode," "common cathode," or. On which screen you can try the

code, but we'll talk about that later. The other eight pins belong to each segment and the dot next to the number.

On a 7-segment display with only one or two digits, the segments are controlled by each digit separately. However, since this even greater confusion on the cable will be four digits than it already is, these displays work with "multiplexing." This means that if, for example, the four digits have to be checked at the same time, they will be checked very quickly. This happens so quickly that it seems to the human eye as if all four digits are displayed at the same time.

Explanation of Contacts:

Programming

A 7-segment display to program without endless code, you need a library that is not installed in the Arduino software. This "SevenSeg" Library of Dean Reading can be downloaded here: https://github.com/DeanIsMe/SevSeg, and The library must then be known as already from the previous instructions, added to Arduino software.

This goes on easiest in the Arduino software at Sketch> Include Library> Add.ZIP Library.

Code:

```
#include "SevSeg.h" // Load The previously added Library SevSeg
sevseg; // initialize a seven segment object

void set up() {

byte numDigits = 4; // Here the number of digits is specified
byte digitPins [] = {2, 3, 4, 5}; // The pins on points are set
byte segmentPins [] = {6, 7, 8, 9, 10, 11, 12, 13}; // The pins on the
// segments are defined

sevseg.begin(COMMON_CATHODE, NumDigits, digitPins,
segmentPins); //In this
// section you can either test what kind of display you own or
```

```
// you can specify it. Try both as the display works only when the correct //type is entered - COMMON_CATHODE or
// COMMON_ANODE display. The wrong type will make all digits, and //segments light up simultaneously.

}

void loop() {

sevseg.SetNumber(1234.3); // Here we can now enter the desired number.
// 1234 is the example we used. The number after the comma represents
// the lit-up dot, i.e., 3 indicates the dot beside the first digit and 0 //indicates the dot beside digit 4.  If you do not want a dot showing,
// input 4.

sevseg.refresh display(); // This is where the numbers are started on the displays

sevseg.SETBRIGHTNESS(90); // Here the brightness of the display can be
//added in a range of 0-100 where 100 is the brightest.
//0 doesn't indicate a completely dark display. Only using
// ,,sevseg.refreshDisplay ();controls what is on the display

}
```

In the sevseg library, there are some interesting sample codes available. These can be in File> Samples> SevSeg-master.

Conclusion

Well, here in the closing of this book, we want to detail the reasons why we think Arduino should be in your life, both professional and private.

Arduino boards are interesting, especially for students and teachers because of their low cost. For less than 100 dollars you can buy the official Arduino Starter Kit that comes with all the necessary components to get started in this world, and if it seems expensive, there are always Arduino starter kits composed of fully functional Arduino board clones (advantages of free hardware on which Arduino is based), and for less than 50 dollars you can get more than enough to start. Even if you're not a student and you're only interested in finding a hobby to spend time with, Arduino is going to be cheaper than photography, painting, or gardening.

With Arduino, you can work on almost all computer platforms, from Mac OS X to Linux through Windows. Thanks to the fact that it is open-source, multiple tools have appeared that make it easy to use, for example, Scratch or its equivalent for Arduino Scratch for Arduino. Apart from the software, it is also flexible when using different accessories available in different kits, so you have the freedom to choose the components to use in your project. There is also flexibility regarding the Arduino itself since depending on the project you have in mind there is an Arduino for each project, from the smallest to make wearables such as the Arduino Gemma to other more powerful ones such as the Arduino Mega or the Arduino Yun in which you can install a network server.

The limit is your imagination. If you have been reading about Arduino for a while, you will have realized that many projects can be done with Arduino, from the simple ones that come in the starter kits to 3D printers or robots such as those from Star Wars, R2D2 or new BB8.

Or how about making a surveillance system for your home or building your own drone. All these things you can do with Arduino. Now is the time to get started!

References

https://stackoverflow.com/questions/15113128/ir-hex-comparison

https://www.copyscape.com/prosearch.php

https://arduino.stackexchange.com/questions/18503/ script-stops-functioning-after-calling-irsend-function-in-irlibrary

http://archive.fabacademy.org/fabacademy2017/fablabbottrophrw/students/64/ week13.html

https://forum.sparkfun.com/viewtopic.php?t=46505

https://github.com/miguelbalboa/rfid/issues/496

https://community.particle.io/t/rfid-code-explanation-needed/53961

https://forum.arduino.cc/index.php?topic=424620.0

https://community.platformio.org/t/wire-h-is-missing-and-cause-a-simple-program-to-not-compile/550

https://forum.arduino.cc/index.php?topic=424620.0

https://media.digikey.com/pdf/Data%20Sheets/DFRobot%20PDFs/DFR0009_Web.pdf

https://community.platformio.org/t/wire-h-is-missing-and-cause-a-simple-program-to-not-compile/550

https://www.geeetech.com/wiki/index.php/Arduino_1602_LCD_KeyPad_Shield

https://protosupplies.com/determining-unknown-i2c-addresses/

https://forum.arduino.cc/index.php?topic=315058.15

https://www.instesre.org/ArduinoWeatherStationCode.txt

http://arduino-tutorials.eu/real-time-clock-tutorial

http://arduino-tutorials.eu/arduino-keypad-lock-tutorial

https://www.instructables.com/id/EAL-MM-Sorting-Machine/

http://tinkbox.ph/sites/mytinkbox.com/files/downloads/TCS230_COLOR_SENSOR.pdf

https://github.com/Nikaoto/Shamen/blob/master/lib/deep.lua

http://arduino-tutorials.eu/arduino-four-digit-seven-segment-display

ARDUINO PROGRAMMING

Advanced Methods and Strategies to Learn Arduino Programming

STUART NICHOLAS

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.

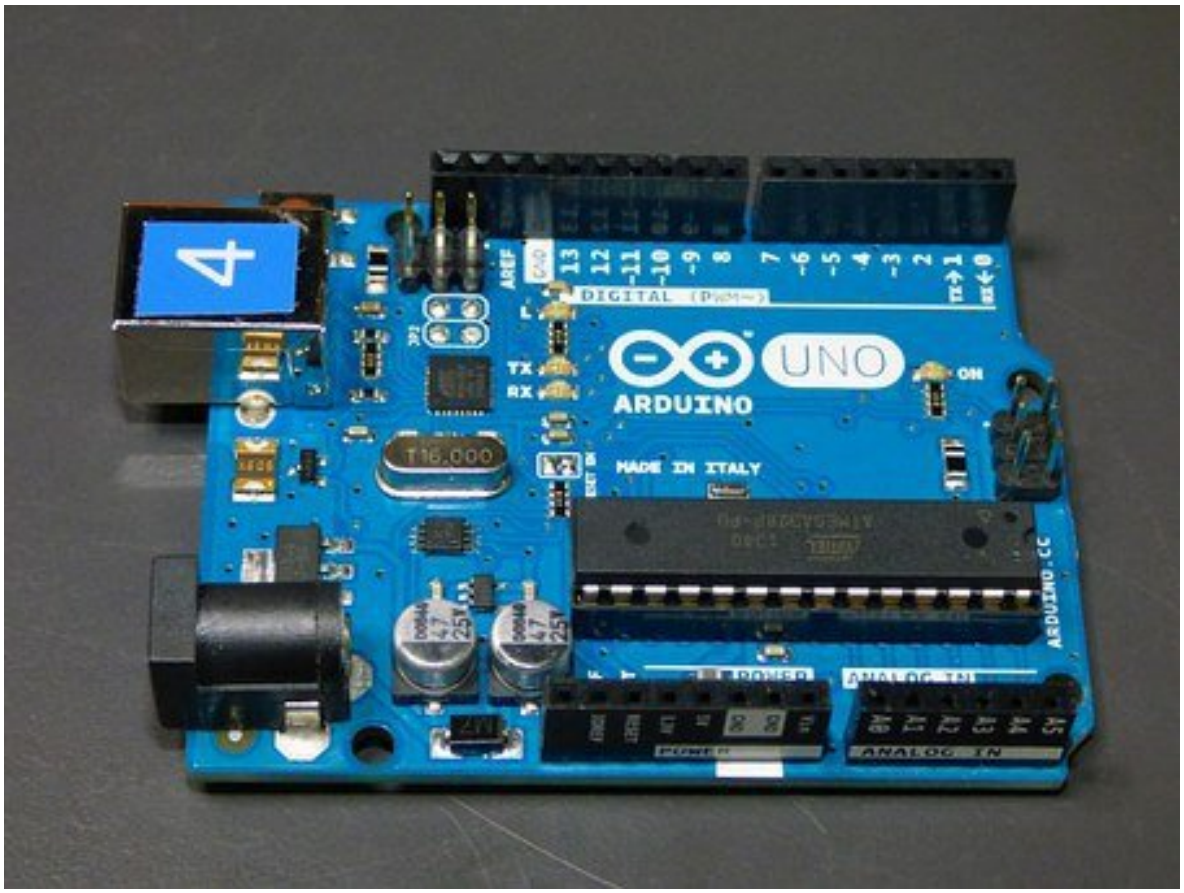
In this book, we will try to explore Arduino's capabilities to the furthest end and try to implement the knowledge we have learned into practicing building Arduino projects. Only talking about the theoretical possibilities of the techniques and methods is just not enough. Putting them into practice and 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.

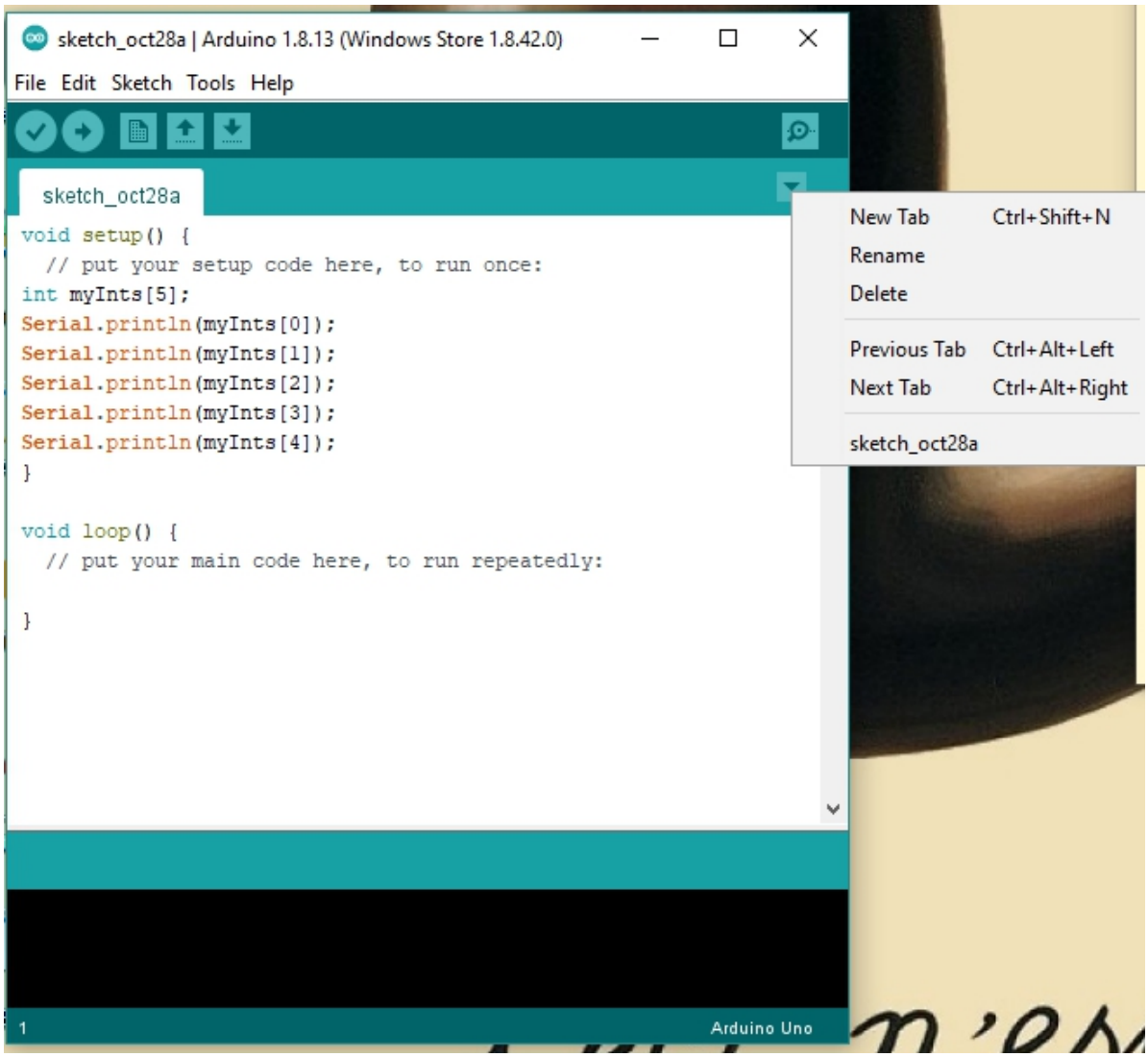
‘some_data’ will only store one value at a time, shifting from the 20 of int ‘i,’ to the 3.14 of the double ‘d,’ to finally the character array storing the value ‘Arduino’ (and its null terminator).

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

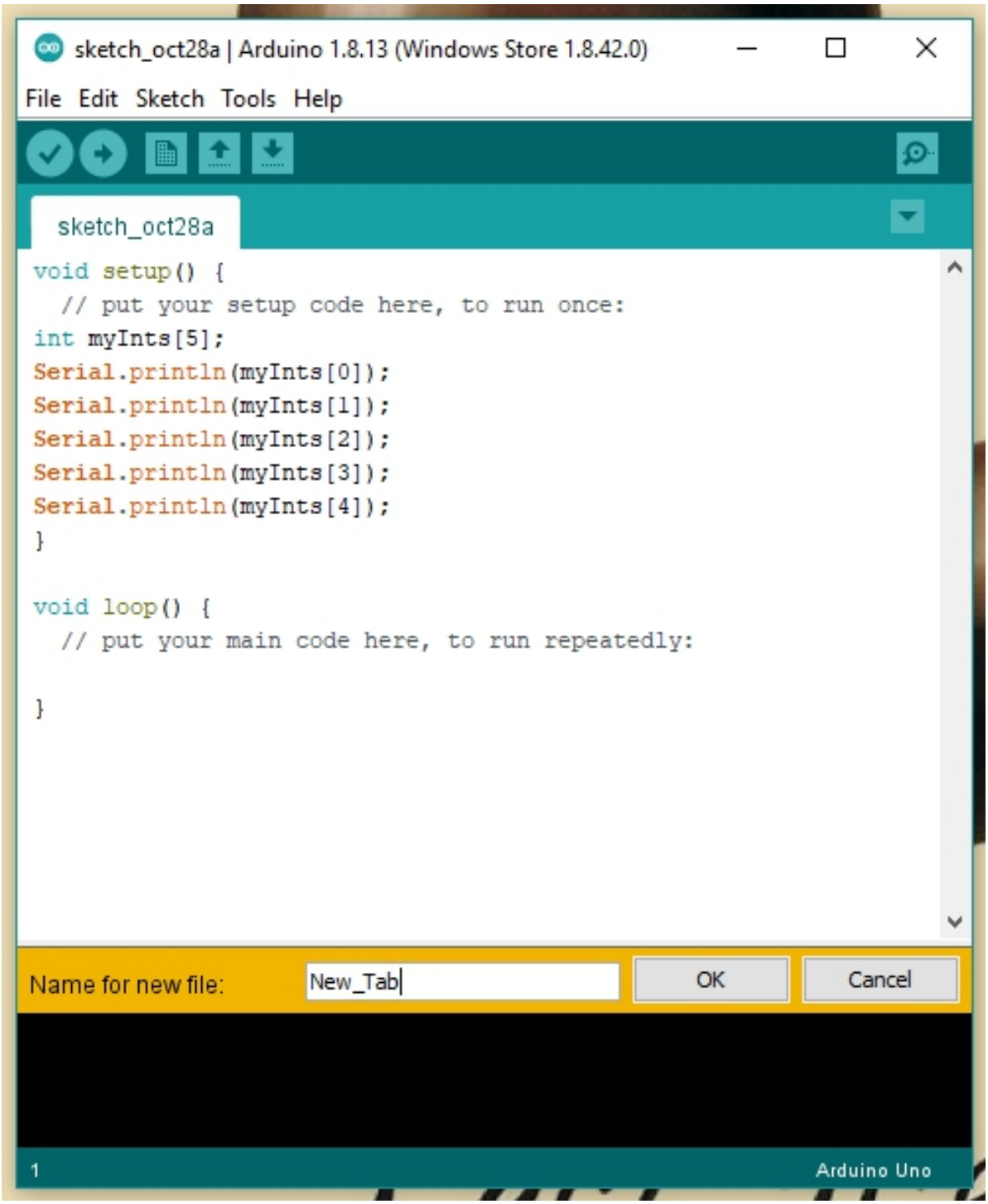
Adding Tabs

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:

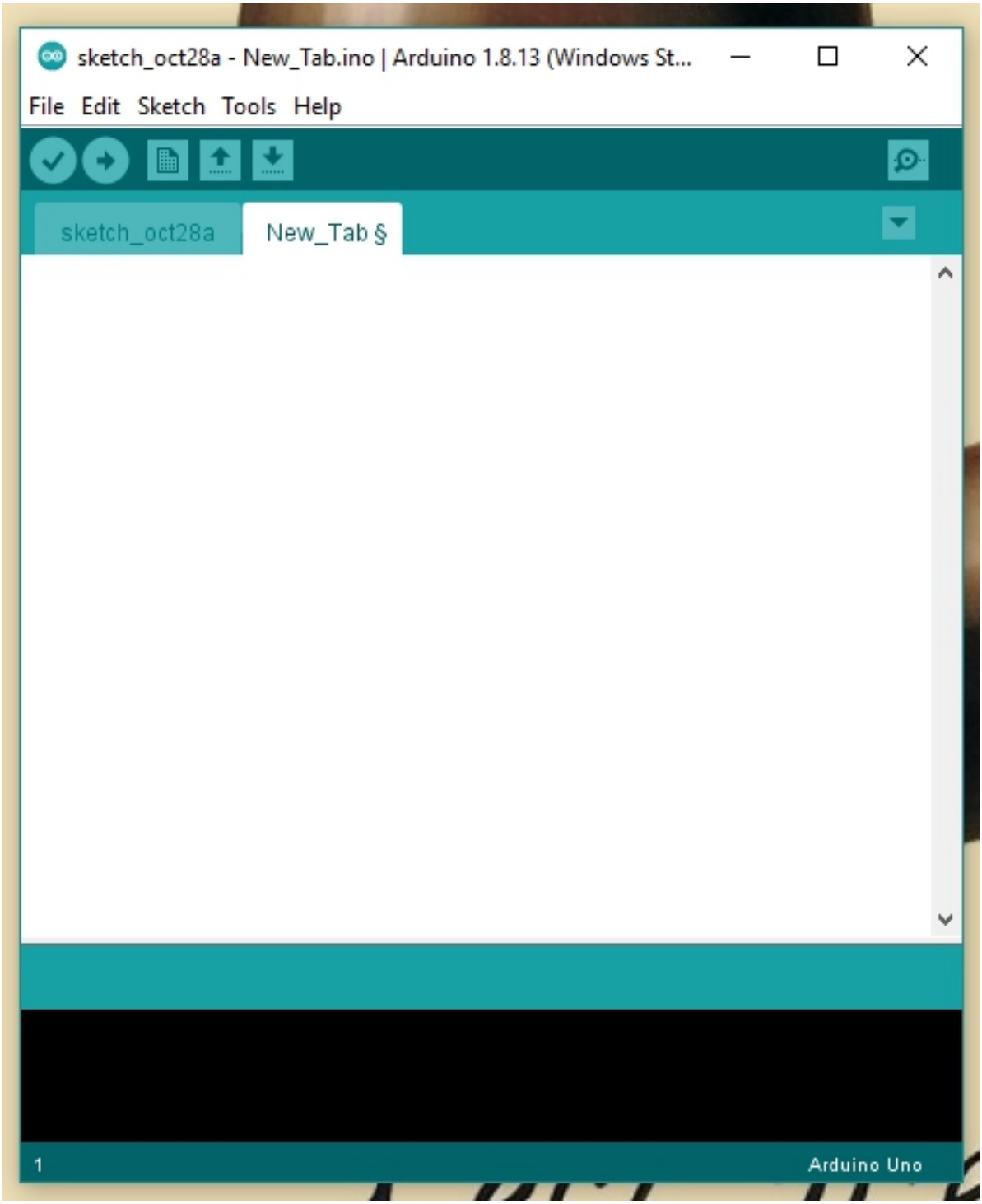
Click on the button at the IDE's top-right, with an upside-down triangle icon, like so:



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); \\ returns 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

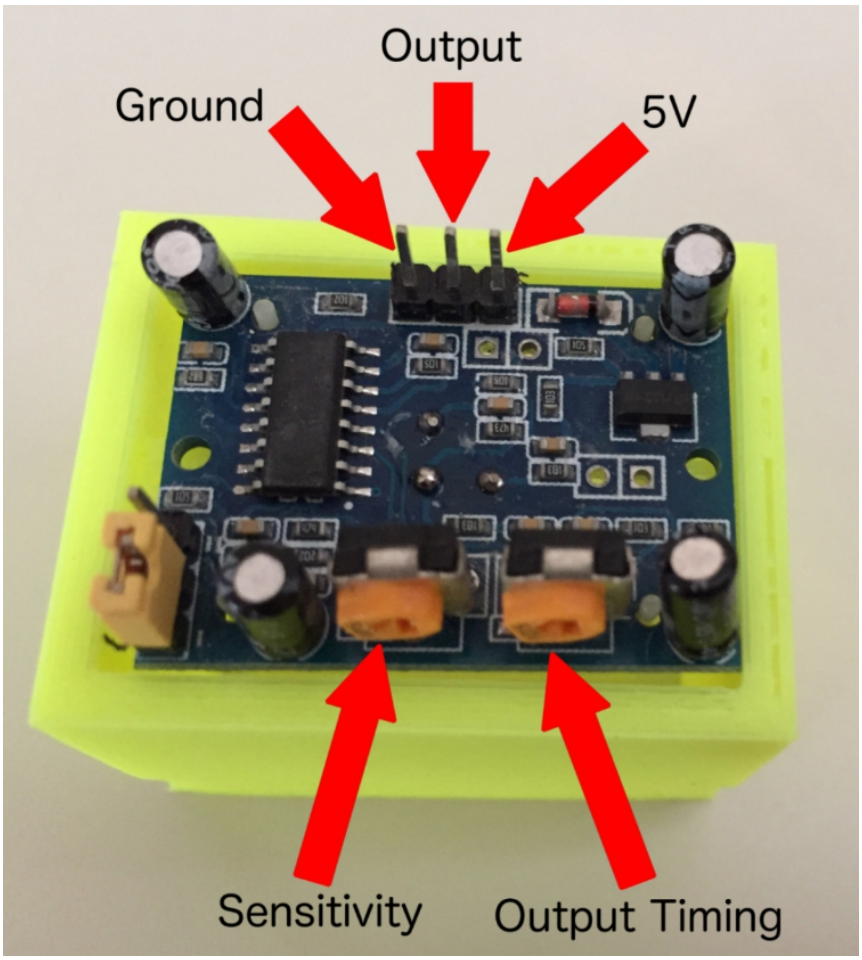
Most common motion sensors will use infrared light to track whether something’s passed in their vicinity. These sensors, termed **Passive Infrared Sensors** (PIR Sensors), detect motion in a small range around them (usually up to two meters for simpler sensors, but industrial and military models with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

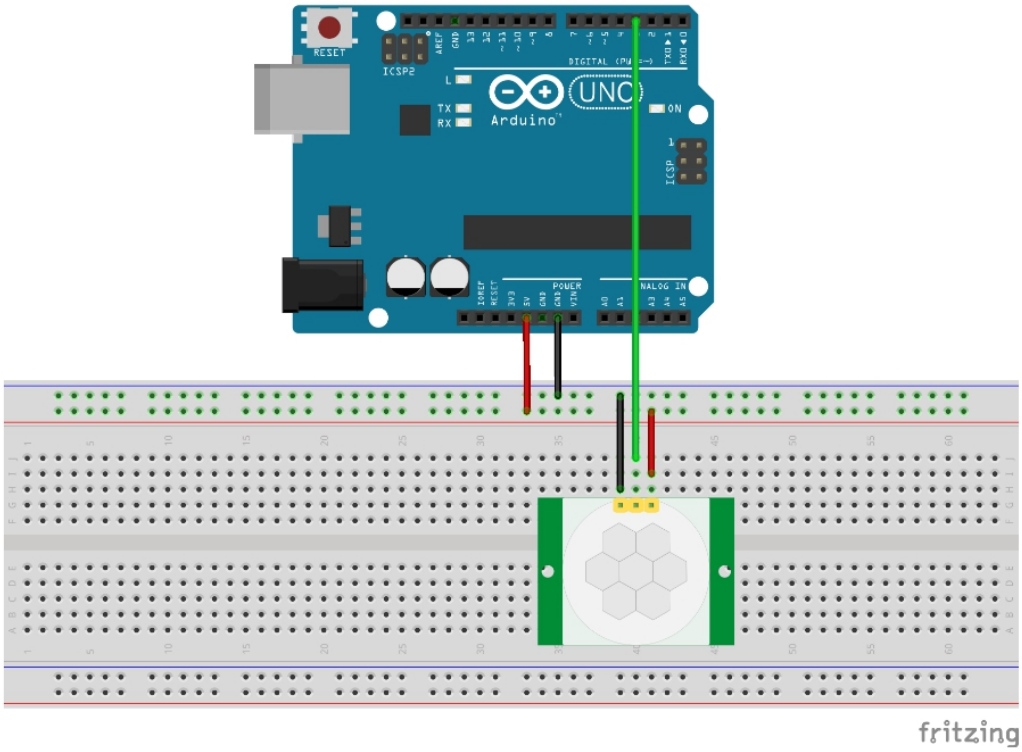
Components

For our circuit we shall require the components listed:

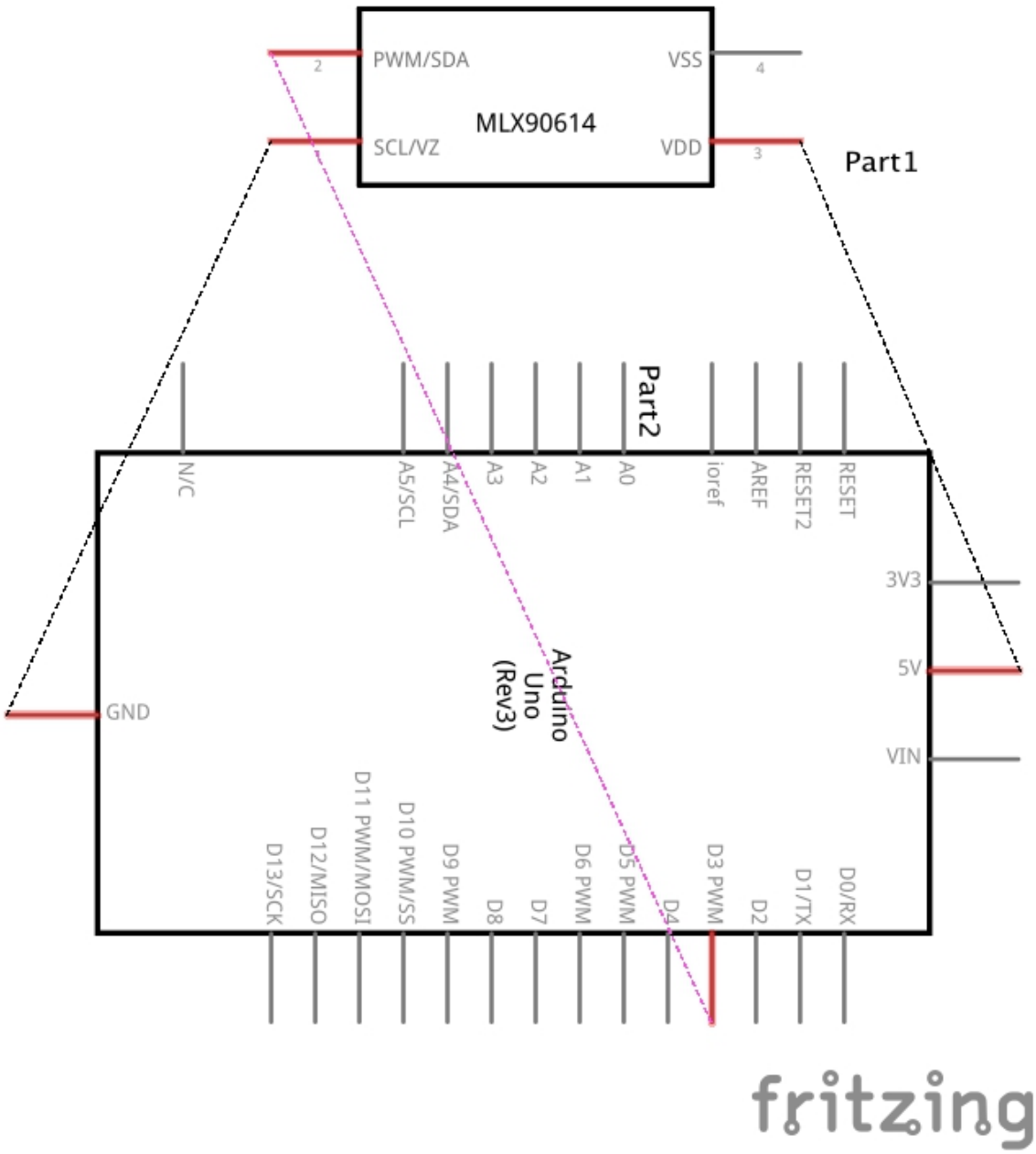
- Arduino Uno (or similar)
- HC-SR501 Motion Sensor
- Jumper Wires
- **Optional:** LED
- **Optional:** Breadboard

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.

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:

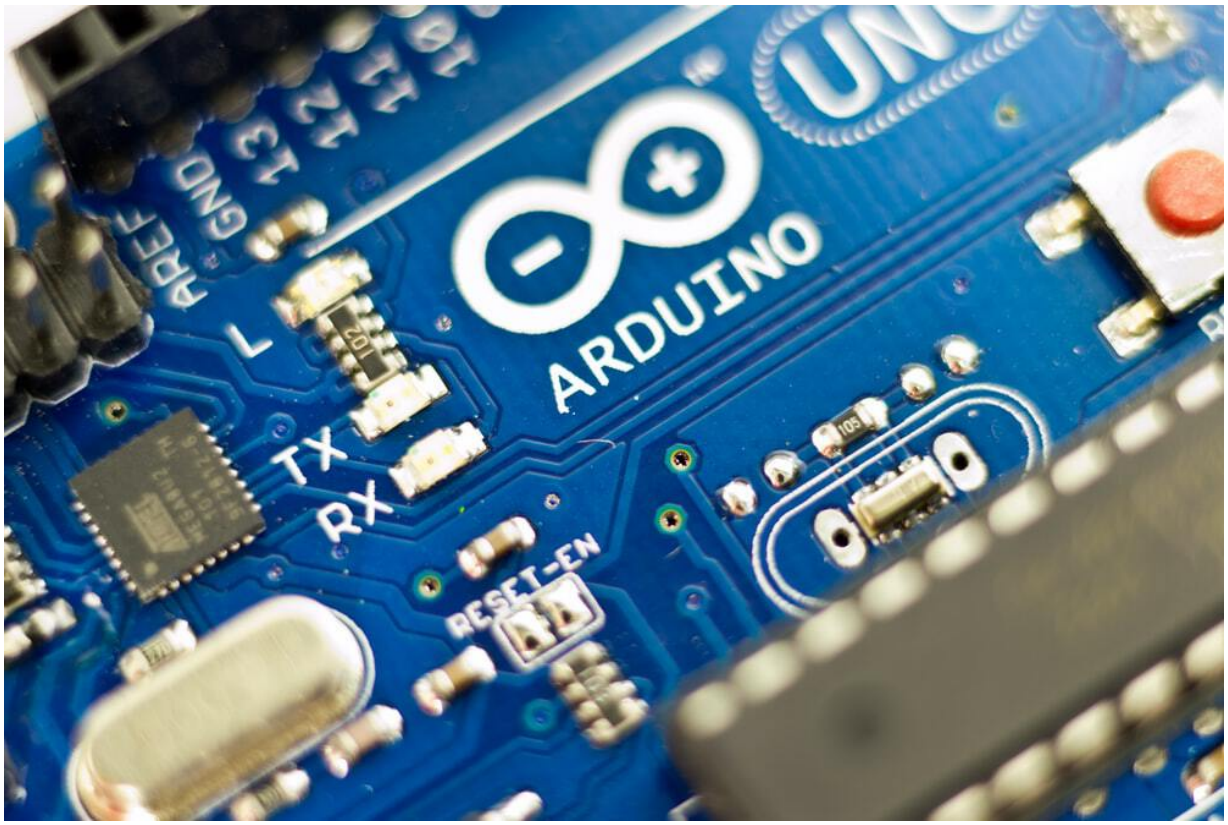
```
while(analogRead(sensorPin) > 100)
{
if(digitalRead(switchPin) == HIGH)
{
break; //exit the loop if the switch is pressed
}
flashLED(); // call a function to turn an LED on and off
}
```

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