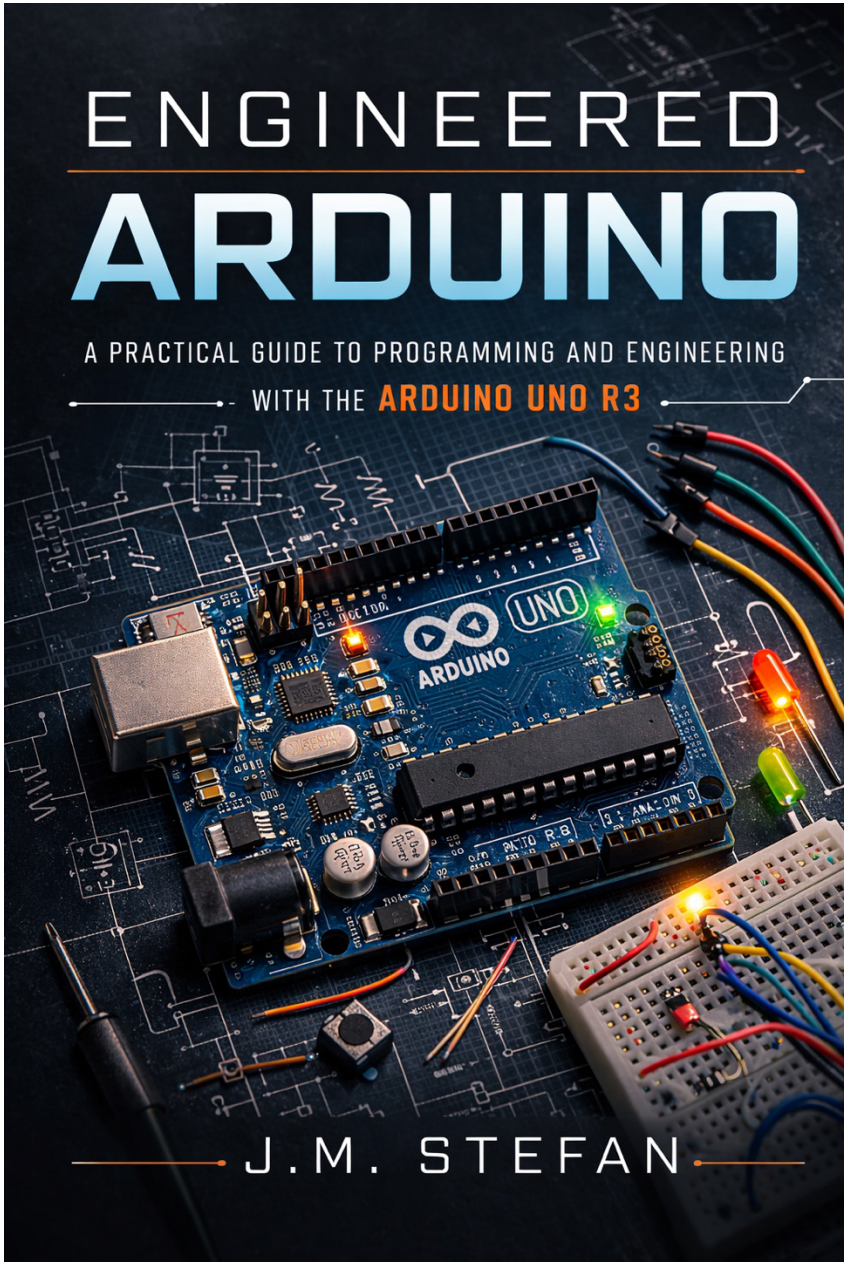

ENGINEERED
ARDUINO
A PRACTICAL GUIDE TO PROGRAMMING AND ENGINEERING
WITH THE ARDUINO UNO R3
ARDUINO
UNO
J.M. STEFAN

# ENGINEERED ARDUINO

This book may be downloaded from GitHub at Engineered Arduino



## Disclaimer

This book is for educational purposes only and the author shall not be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or text or the use of other dealings in the software or text.

## MIT License

**CHAPTER ONE ..... 11**

Engineered Arduino ..... 11
Arduino Microcontroller Boards ..... 11
Microcontrollers vs. Microprocessors ..... 12
How This Book is Structured ..... 12
Chapter One ..... 13
Chapter Two ..... 13
Chapter Three ..... 13
Chapter Four ..... 13
Chapter Five ..... 14
Chapter Six ..... 14
Chapter Seven ..... 14
Chapter Eight ..... 14
Chapter Nine ..... 14
Chapter Ten ..... 14
Chapter Eleven ..... 15
Chapter Twelve ..... 15
Example Project: Control an LED ..... 16
*Requirements and Comments ..... 16*
*The Circuit ..... 17*
*ATmega 328P to UNO R# Pinout Map ..... 19*
*Breadboard Layouts ..... 19*
*Flowchart/State Machine Diagram ..... 20*
*The Code ..... 23*
*Code Walk Through ..... 24*
*Going Further ..... 25*

**CHAPTER TWO ..... 27**

The ATmega 328p Microcontroller ..... 27
The Arduino UNO ..... 29
Architecture ..... 29
Damaging Your Arduino ..... 30

**CHAPTER THREE ..... 31**

Development Environments ..... 31
Arduino ID ..... 31
VSCode/PlatformIO ..... 32

**CHAPTER FOUR ..... 34**

Sensor Technologies ..... 34
Transducers and Sensors ..... 34
Passive and Active Sensors ..... 35

Sensor Properties ..... 35
Damping ..... 37
The Big View ..... 38
Sensor Technologies ..... 39
Resistive Sensors and Voltage Dividers ..... 39
Resistive Sensors ..... 42
Capacitive Sensors ..... 43
Pull Up and Pull Down Circuits ..... 43
Operational Amplifiers ..... 46
Open Loop Differential Amplifier ..... 47
Buffers ..... 47
Gain and Non-Inverting Amplifiers ..... 51
Comparators ..... 51
Non-inverting Comparator ..... 52
Inverting Comparator ..... 53
LM393 Comparator ..... 55
General 4 Pin Sensor Module Schematic ..... 57
The Importance of Data Sheets ..... 57
Sensor Configuration ..... 58
3 Pin Configuration ..... 58

**CHAPTER FIVE ..... 59**

2-3 Pin Input Sensors ..... 59
Tilt Ball Switch ..... 60
*State Diagram ..... 60*
*The Circuit ..... 61*
*Specifications ..... 62*
*The Code ..... 63*
*Code Walk Through ..... 64*
*Going Further ..... 65*
DHT11 Temperature and Humidity Module ..... 66
*Requirements and Comments ..... 66*
*The Circuit ..... 66*
*Library Functions ..... 67*
*Flowchart ..... 68*
*The Code ..... 68*
*Code Walkthrough ..... 70*
*Going Further ..... 71*
*The Circuit ..... 71*
*The Code ..... 72*
*Code Walkthrough ..... 73*
5V Relay ..... 76

*Requirements and Comments* ..... *76*
*Relay With Terminal Board* ..... *76*
*The Circuit* ..... *77*
*Flowchart* ..... *77*
*The Code* ..... *78*
*Code Walk Through* ..... *79*
*Going Further* ..... *79*
*The Circuit* ..... *81*
*The Code* ..... *82*
*Code Walkthrough* ..... *82*
TRANSISTORS ..... 83
KY-001 TEMPERATURE MODULE ..... 85
*Requirements and Comments* ..... *85*
*The Circuit* ..... *85*
*Flowchart* ..... *86*
*The Code* ..... *87*
*Code Walk Through* ..... *87*
*Going Further* ..... *88*
KY-031 PERCUSSION KNOCK SENSOR ..... 90
*Requirements and Comments* ..... *91*
*State Chart* ..... *91*
*The Circuit* ..... *91*
*Specifications* ..... *92*
*The Code* ..... *93*
*Code Walk Through* ..... *93*
*Going Further* ..... *94*
KY-002 SHOCK MODULE ..... 96
*Requirements and Comments* ..... *96*
*State Chart* ..... *97*
*The Circuit* ..... *97*
*Specifications* ..... *98*
*The Code* ..... *98*
*Code Walk Through* ..... *98*
*Going Further* ..... *99*
HC-SR501 MOTION SENSOR ..... 101
*Specifications* ..... *102*
*Requirements and Comments* ..... *102*
*State Chart* ..... *102*
*The Circuit* ..... *102*
*The Code* ..... *104*
*Code Walk Through* ..... *104*
*Going Further* ..... *105*

K-0135 Water Level Sensor ..... 107
*Flowchart* ..... *108*
*The Circuit* ..... *110*
*The Code* ..... *111*
*Code Walk Through* ..... *113*
*Going Further* ..... *116*
TTP 223 Capacitive Touch Sensor ..... 119
*The Circuit* ..... *119*
*State Chart* ..... *120*
*The Code* ..... *121*
*Code Walk Through* ..... *121*
*Going Further* ..... *122*

**CHAPTER SIX ..... 123**

4+ Pin Input Sensors ..... 123
TAC PushButton ..... 123
*Requirements and Comments* ..... *123*
*The Circuit* ..... *124*
*The Code* ..... *125*
*Code Walk Through* ..... *125*
*Going Further- Switch Debouncing* ..... *126*
*The Code* ..... *127*
*Code Walk Through* ..... *128*
HC-SR04 Ultrasonic Sensor ..... 130
*How it Works* ..... *130*
*The Circuit* ..... *131*
*The Code* ..... *132*
*Code Walkthrough* ..... *133*
*Going Further* ..... *134*
*The Code* ..... *135*
*Code Walk Through* ..... *136*
*Oscilloscope Traces* ..... *136*
KY-026 Flame Sensor ..... 139
*Specifications* ..... *139*
*The Circuit* ..... *139*
*The Code* ..... *141*
*Code Walk Through* ..... *142*
KY-038 Sound Modules ..... 144
*Requirements and Comments* ..... *145*
*The Circuit* ..... *149*
*Specifications* ..... *149*
*The Code* ..... *150*

*Code Walk Through* ..... *151*
*Going Further* ..... *151*
LINEAR HALL ..... 155
*Requirements and Comments* ..... *155*
*The Circuit* ..... *156*
*Specifications* ..... *156*
*The Code* ..... *157*
*Code Walk Through* ..... *157*
*Going Further* ..... *158*
MAGNETIC SPRING SWITCH ..... 163
*Requirements and Comments* ..... *163*
*The Circuit* ..... *164*
*Specifications* ..... *164*
*The Code* ..... *165*
*Code Walk Through* ..... *165*
*Going Further* ..... *166*
METAL TOUCH ..... 167
*Requirements and Comments* ..... *167*
*The Circuit* ..... *168*
*Specifications* ..... *168*
*The Code* ..... *169*
*Code Walk Through* ..... *169*
*Going Further* ..... *170*
KY-028 DIGITAL TEMPERATURE SENSOR ..... 171
*Requirements and Comments* ..... *171*
*Flowchart* ..... *171*
*The Circuit* ..... *172*
*The Code* ..... *173*
*Code Walk Through* ..... *173*
*Going Further* ..... *174*
*Flowchart* ..... *175*
*The Code* ..... *177*
*Code Walkthrough* ..... *178*
*Structure/Pointer Template* ..... *181*
*Code Walkthrough* ..... *181*

**CHAPTER SEVEN ..... 184**

SPECIALIZED SENSORS ..... 184
FSR PRESSURE SENSORS ..... 184
*Mass, Weight and Inertia* ..... *186*
*The Circuit* ..... *186*
*Code Walk Through* ..... *191*

YL-69 Resistive Moisture Sensor ..... 193
*Flowchart* ..... *194*
*The Circuit* ..... *195*
*The Code* ..... *196*
*Code Walk Through* ..... *198*
*Going Further* ..... *200*
HW-390 Capacitive Moisture Sensor ..... 201
*Flowchart* ..... *201*
*The Circuit* ..... *202*
*The Code* ..... *203*
*Code Walk Through* ..... *205*

**CHAPTER EIGHT ..... 209**

Output Devices ..... 209
LCM1602A LCD Display Module ..... 209
*Requirements and Comments* ..... *210*
*The Circuit* ..... *211*
*The Code* ..... *212*
*Code Walk Through* ..... *214*
The Circuit ..... 215
*The Code* ..... *216*
*Platform.ini File* ..... *218*
*Code Walkthrough* ..... *218*
1602 LCD Keypad Shield ..... 220
*The Circuit* ..... *220*
*Specifications* ..... *220*
*The Code* ..... *221*
*Code Walk Through* ..... *223*
Power Supply Module ..... 226

**CHAPTER NINE ..... 228**

*Serial Communications* ..... *228*
*Duplex Modes* ..... *228*
*Parallel vs. Serial Communication* ..... *229*
*UART* ..... *229*
*Parity* ..... *231*
*Errors* ..... *231*
*Serial Trace Analysis* ..... *231*
*Oscilloscope Traces* ..... *232*
I2C ..... 236
*I2C Address Finder Code* ..... *240*
*platform.ini* ..... *241*

*Code Walkthrough* ..... *242*

**CHAPTER TEN** ..... **243**

FUNCTIONS AND DATA STRUCTURES ..... 243
FUNCTIONS ..... 243
*Code Walkthrough* ..... *244*
*Arrays* ..... *246*
*For Loops* ..... *247*
*Example Code* ..... *249*
*Code Walk Through* ..... *250*

**CHAPTER ELEVEN** ..... **251**

INFORMATION REPRESENTATION ..... 251
*Digital Logic* ..... *251*
*Basic Binary Operations* ..... *251*
*Binary Conversions* ..... *253*
*Converting From Decimal to Binary and Back Again* ..... *253*
*Bit Operation Code* ..... *256*
*Code Walkthrough* ..... *257*
*Code Walkthrough* ..... *259*
*Code Walkthrough* ..... *261*
*Bit Operations and Ports* ..... *264*
*The Code* ..... *268*
*Code Walkthrough* ..... *269*
*Going Further* ..... *270*
*ASCII Chart* ..... *272*

**CHAPTER TWELVE** ..... **276**

PLATFORMIO ..... 276

**CHAPTER THIRTEEN** ..... **278**

THE EMBEDDED SYSTEMS WORLD ..... 278

**ABOUT THE AUTHOR** ..... **279**

# CHAPTER ONE

## Engineered Arduino

Welcome to Engineered Arduino. Here you will learn how to interface a variety of sensors to an Arduino Uno R3. You will also learn a good deal about sensor technologies- not only how they work but how to use some of the important support components that are frequently used with sensors, such as transistors and operational amplifiers. Along with the extensive sensor sections there is also a section on digital logic and a section on C programming. The C programming section delves into useful topics such as functions, arrays and pointers. All of these can be used in sophisticated and efficient Arduino applications.

How do machines make sense of the analog world? Via sensors and microcontrollers. Analog data represents events in the real world, and that data comes from sensors.

Sensors, in many cases, either have tiny microcontrollers on board or are connected to microcontrollers. Microcontrollers are in almost everything, from spacecraft to toys. Programming microcontrollers used to be, and still is at a deep level, a highly specialized pocket of engineering knowledge and experience. It was much more difficult to get started programming microcontrollers in the past. Development boards were expensive and so were development environments. There were a few odd reasonably priced kits here and there, but most were prohibitively expensive. The democratization of the microcontrollers took a quantum leap with the emergence of the Arduino.

### Arduino Microcontroller Boards

The Arduino was conceived in a town in Italy called Ivrea and is named after a local bar, the Bar di Re Arduino. The first board emerged in 2005 using the Atmega8 along with version 0001 of the Arduino IDE (Integrated Development Environment). An enhanced Arduino IDE was released later along with the more powerful Atmega168. In 2010 the Arduino Uno was released. We will be working with the Arduino UNO R3. This book's focus is on sensors and interfacing sensors to the Arduino Uno.

Engineered Arduino takes a systematic approach to working with Arduino sensor projects. Instead of just providing a breadboard layout

of a project and source code, we dive into the details of the embedded systems engineering process used to develop applications. For each project an explanation of the sensors used is provided, including specifications along with the requirements for each project, just like in real life engineering. A circuit diagram and/or schematic is provided. Breadboard layouts are not used, since the wiring can be easily connected using the circuit diagrams. The code (or sketch in Arduino-speak) for each sensor application is provided along with a detailed code walkthrough. Where relevant, oscilloscope traces are provided to further enhance understanding how the code and hardware work together.

We will only use the Arduino IDE as a code proving ground or "sandbox". The final code will be ported to PlatformIO running under VSCode. The Arduino code ports directly to PlatformIO with very few changes, so this is a painless process. If you like using the Arduino IDE go right ahead. It's a great development environment and is constantly improving. The Arduino IDE is great, but PlatformIO is much better suited to more advanced and sophisticated projects. If you aspire to do more in the embedded systems world, it will serve you well to understand and use PlatformIO. Like the Ardiuno IDE, VSCode and PlatformIO are open source and free to use. That being said, the projects will run under the Arduino IDE.

## Microcontrollers vs. Microprocessors

Arduino boards are microcontroller based versus microprocessor based. Microcontrollers are generally less complex that microprocessors and are used for practical applications, such as monitoring inputs, controlling outputs and interfacing to a wide variety of sensors where microprocessors possess a wider range of capabilities beyond microcontrollers. Microcontrollers, just by their very nature, consume much less overhead than a microprocessor. Microcontrollers are designed to connect to and interact with sensors and therefore, the real world.

## How This Book is Structured

This section discusses the approach Engineered Arduino takes. As the author I suggest you to take your time, build the circuits, use the code and above all experiment with the material presented in this book. You learn best by *doing*. This book is designed to help you do just that and emphasizes basic embedded system engineering practices.

The following is a summary of what to expect from Engineered Arduino.

## Chapter One

Chapter One familiarizes you with microcontrollers, discusses sensor technologies and provides an example project that gives an indication of how the projects in the book are presented. The built in LED is blinked using a simple sketch using the Arduino IDE. Next is a more detailed example where we blink the built in LED and an external LED in sequence. There are several sections of any given project, including full code listings, code walkthroughs, flow charts and diagrams. Many of the projects have captured oscilloscope traces. This reflects the nature of the rest of the projects in the book.

## Chapter Two

Chapter two provides an overview of the Atmega 328p microcontroller and the Arduino UNO R3.

## Chapter Three

In this chapter the Arduino IDE and VSCode/PlatformIO programming environments are presented, including the advantages and disadvantages of using each. The Arduino IDE is discussed in this chapter. This is the default programming environment for producing Arduino sketches and is rich in features and examples. The Arduino IDE is good for a lot of applications, but a more sophisticated application environment is VSCode running the PlatformIO extension. PlatformIO supports over 1000 boards, built-in debugging, remote development and many more features.

## Chapter Four

Sensor technology is covered here, primarily resistive, capacitive and inductive sensor types. This chapter discusses sensor design concepts and the characteristics of different sensor technologies. It also talks about what to take into consideration when designing your own sensors. In the following sections, explanations of the sensors, sketches, schematics and code walkthroughs are provided. For many of the sensors, oscilloscope traces are shown to gain insight into sensor response and behavior.

## Chapter Five

This chapter covers common 2-3 input sensor modules that come with most Arduino sensor kits. A sampling of these sensors include tilt ball switches, a 5V relay, water and motion sensors and many more. It also contains a section on transistors.

## Chapter Six

Sensors with four or more pins are covered in this chapter. These sensors also come with most Arduino sensor kits. Some of the sensors include the HC-SR04 ultrasonic sensor, KU-039 sound modules, GY-5213 3 axis gyro modules and many more interesting sensors. It also provides a section and template for C structures and pointers.

## Chapter Seven

This chapter covers interesting sensors that are highly specialized, such as resistive and capacitive pressure sensors and sensors.

## Chapter Eight

This chapter covers output devices that come with most Arduino sensor kits. Example output devices are the LCM1602A LCD display module, buzzers and drivers among other devices. It also describes the Arduino power supply module.

## Chapter Nine

Chapter Nine is all about serial communications. Serial communications is the main mechanism for communicating with an Arduino and PC. This chapter goes into operation of serial communications, particularly paying attention the UARTs. It also contains a section on I2C and how to find the address of an I2C device.

## Chapter Ten

This section dives into C functions and data structures. Being able to utilize these elements of C programming makes developing applications much more efficient. This is core knowledge a competent embedded systems programmer should possess.

## Chapter Eleven

This chapter covers the basics of digital and analog information representation, including binary conversions, analog to digital conversion, and much more. It also includes a sketch that demonstrates bit operators.

## Chapter Twelve

The embedded world is all around us and this very short chapter encourages you to go farther.

This is the journey this book takes. Come along and you should find yourself with a proficient skill set to imagine and tackle your own Arduino projects.

## Example Project: Control an LED

The fundamental Arduino program is the basic blink LED sketch. Blinking an LED on a circuit board is the "Hello, world" in the embedded systems domain. Here we will take a more robust approach using PlatformIO running under Visual Studio Code. This will give you an indication of how the book is structured and how it may differ from other Arduino books. That being said, if you want to stick with the Arduino IDE, go right ahead. It's powerful and easy to use.

Whenever PlatformIO is referenced, we will write the code using a #define guard to easily switch from the Arduino IDE to PlatformIO. We will also use an oscilloscope to look at the output trace in many of the applications. This will act as a foundation for more sophisticated embedded systems programing that goes beyond the Arduino. It will also introduce you to how the sensor programming sections of this book are laid out.

In this example the code blinks the Arduino onboard LED, then blinks an external LED. The intent of this program is to introduce a wider programming style and to get you used to interpreting an oscilloscope trace. The code for this example was created using PlatformIO running under VSCode. This will be our development platform of choice for many reasons, but as said before, all code runs on the Arduino IDE.

Here's how I work, and maybe you can consider it also. I use the Arduino IDE as a sandbox to prototype code and to prove out ideas and applications. The only source control I use is a revision history in the program headers. Once the proof-of-concept or prototype code settles, I then port the code to PlatformIO and keep it under source control using git and GitHub. For production code, I much prefer developing code in PlatformIO. That being said, I have to repeat that the Arduino IDE is a wonderful and easy to use environment. One advantage of using the Arduino IDE is that libraries are easy to pull in and use. It's not as easy using PlatformIO. The advantage of using PlatformIO is that is supports hundreds of boards, making it easy to port code from one board, or hardware platform, to another.

So let's dive into the first application and see the flow of information.

### Requirements and Comments

We start with some basic requirements regarding what we want our code to do. For this application, the programming requirements are as follows:

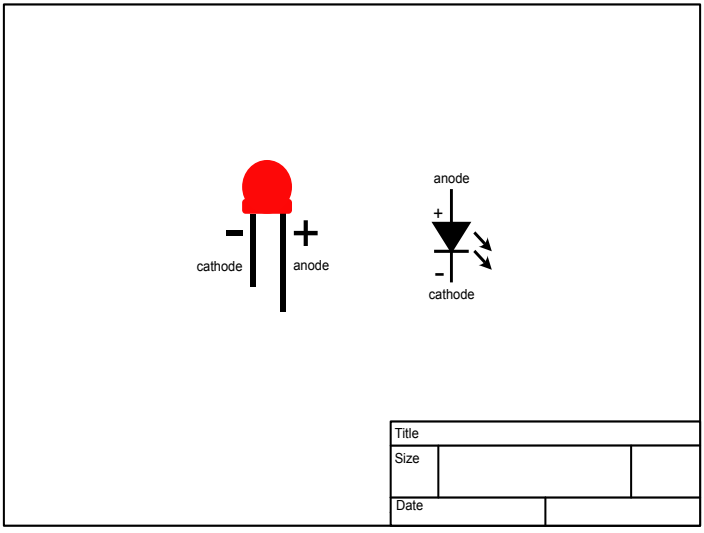
- Blink the Arduino UNO R2 built in LED first and blink an external LED next.

- The Serial Monitor is invoked but not used, with calls to Serial.print( ) commented out, so not to influence the timing of the blink sequence. Try it both ways if you have a scope to see how the calls to the serial methods slow things down.

- The code contains a #define guard to easily switch between the Arduino IDE and PlatformIO. If you want to use the Arduino IDE, remove the comment symbol //. That will block the code from pulling in the <Arduino.h> file.

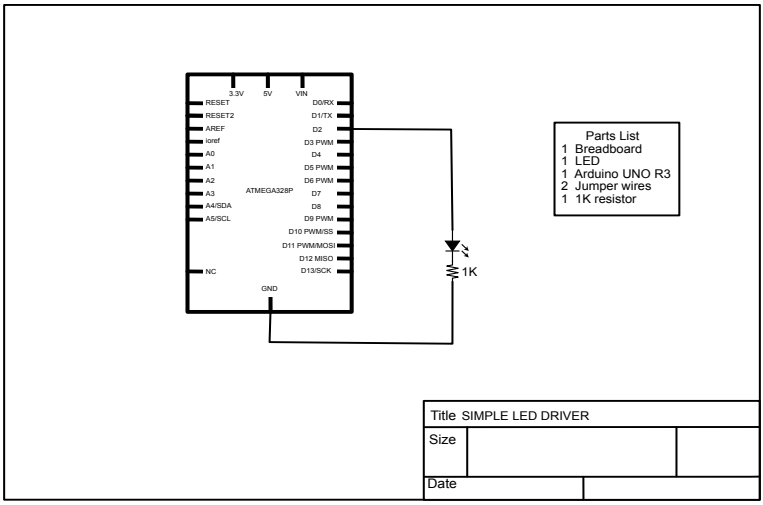
## The Circuit

Here's a drawing showing LEDs in a graphical and schematic form. We will use the schematic symbol for electronic components for the lion's share of the drawings in this book.

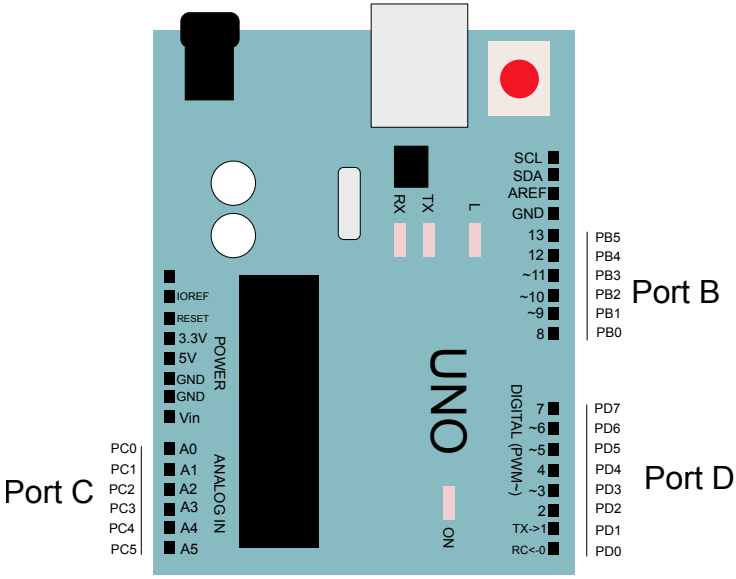



The circuit below for this project is as simple. The anode (positive side) of the LED is connected to pin 2 of the Arduino. The cathode (negative side) of the LED is connected to a 1K resistor. The opposite end of the resistor is connected to ground. Use a jumper to connect pin 2 of the Arduino to the LED anode on a breadboard.

LEDs typically utilize a forward voltage of 1.2 to 3.6 volts, and draw around 10 to 30mA.




Schematic diagrams will be provided for the projects and after working through a number of examples you should not have to rely on breadboard layouts. In a real working environments breadboard layouts are rarely provided, if ever. Also, the pinout of the ATmega328P is used instead of an UNO board. The pin names map directly to an Arduino UNO and is more in keeping with an actual schematic diagram.



## ATmega 328P to UNO R# Pinout Map

Here's a pinout map from the ATmega 328P to the Arduino UNO R3.

**ATmega328P ↔ Arduino Uno R3 Pin Mapping**

| ATmega328P Pin # | Port / Bit | Arduino Pin | Function / Notes |
|---|---|---|---|
| 1 | PC6 / RESET | RESET | Reset pin |
| 2 | PD0 | D0 (RX) | Serial RX |
| 3 | PD1 | D1 (TX) | Serial TX |
| 4 | PD2 | D2 | Digital I/O, INT0 |
| 5 | PD3 | D3 | PWM, INT1 |
| 6 | PD4 | D4 | Digital I/O |
| 7 | VCC | 5V | Power |
| 8 | GND | GND | Ground |
| 9 | PB6 | — | Crystal (XTAL1) |
| 10 | PB7 | — | Crystal (XTAL2) |
| 11 | PD5 | D5 | PWM |
| 12 | PD6 | D6 | PWM |
| 13 | PD7 | D7 | Digital I/O |
| 14 | PB0 | D8 | Digital I/O |
| 15 | PB1 | D9 | PWM |
| 16 | PB2 | D10 | PWM, SPI SS |
| 17 | PB3 | D11 | PWM, SPI MOSI |
| 18 | PB4 | D12 | SPI MISO |
| 19 | PB5 | D13 | SPI SCK, LED |
| 20 | AVCC | 5V | Analog power |
| 21 | AREF | AREF | Analog reference |
| 22 | GND | GND | Ground |
| 23 | PC0 | A0 | Analog input |
| 24 | PC1 | A1 | Analog input |
| 25 | PC2 | A2 | Analog input |
| 26 | PC3 | A3 | Analog input |
| 27 | PC4 | A4 | SDA ($I^2C$) |
| 28 | PC5 | A5 | SCL ($I^2C$) |

Pin D2 on the 328P corresponds to pin PD2 on the UNO and that's where to connect the anode of the LED.

## Breadboard Layouts

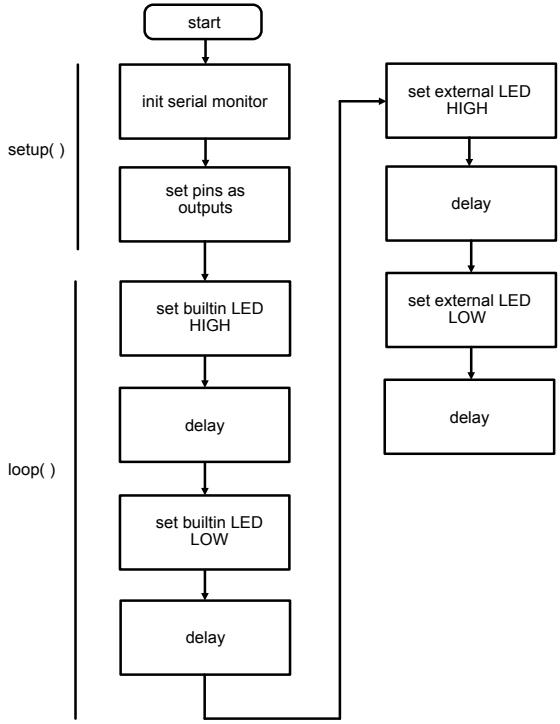
Again, explicit breadboard layouts will not be shown for the example circuits, for several reasons. They are redundant for many circuits and

tend to limit readers from making the leap from reading a schematic to building an actual circuit. It also gets you used to thinking in terms schematics and will aid you in efficiently laying out your own circuits. Reading and understanding schematics and how to create them is a springboard to more advanced designs and projects. Think of a schematic as the raw data for a breadboard layout. The data doesn't change, but it can be interpreted in a multitude of ways on a breadboard.

## Flowchart/State Machine Diagram

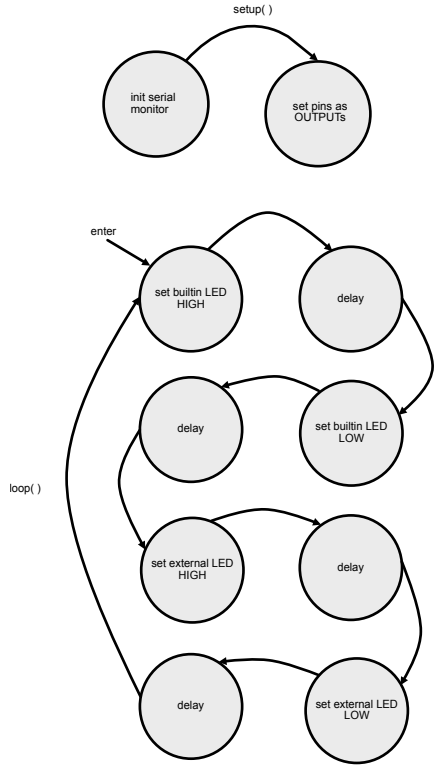
Flowcharts and state machine diagrams are used throughout the book where appropriate. If an application is best expressed as a flowchart, then a flowchart will be used. If better expressed as a state diagram, then that will be used. Although flowcharts have been around since the dawn of computers, they are still an effective tool in providing program control flow. Understanding a flowchart for a given application makes following the code much easier and makes it easier to spot potential mistakes.

The flowchart for the LED sketch is given below.



The flowcharts and state diagrams used in this book are modified to indicate what happens in setup( ) and loop( ).

State charts depict how an entity transitions from state to state. For example, a light switch is either on or off. State 1 could be OFF. When the light is turned on, the state transitions from OFF to ON. State machines are a useful and efficient way to model and describe system behavior. Below is a state machine for this sketch.




Like the flow charts, the state machines are broken into setup( ) and loop( ). After setup( ) completes, the state machine is entered at the arrow labeled “enter”. The bubbles indicate that states and the arrows indicate transitions between the states. In this sketch, the state transitions as the program progresses. In many applications, state transitions occur when a variable, input or output changes state. The lines with arrows between the states may be labeled. To follow a state machine, all that needs to be done is follow the flow of the lines with arrows.

Notice that both the flowchart and the state machine do not contain the particulars of the sketch, such as #defines or variable definitions. A top level flow chart or state machine hides these implementation details. If required, more detailed charts can be developed, and are sometimes necessary depending on the application. Most designs are conceived from the top down, meaning that the most general description is at the top, and the most detailed is at the bottom. This is where state machines really shine. For example, the set builtin LED HIGH state could be detailed in separate diagrams down to the machine code level, if desired.

## The Code

Please note if you're coming from Python and not familiar with C or C++ you must put a semicolon after each line of code. This is the delimiter that the compiler expects, otherwise an error will be generated and the error description may seem cryptic. If you get a compiler error always check for missing semicolons first.

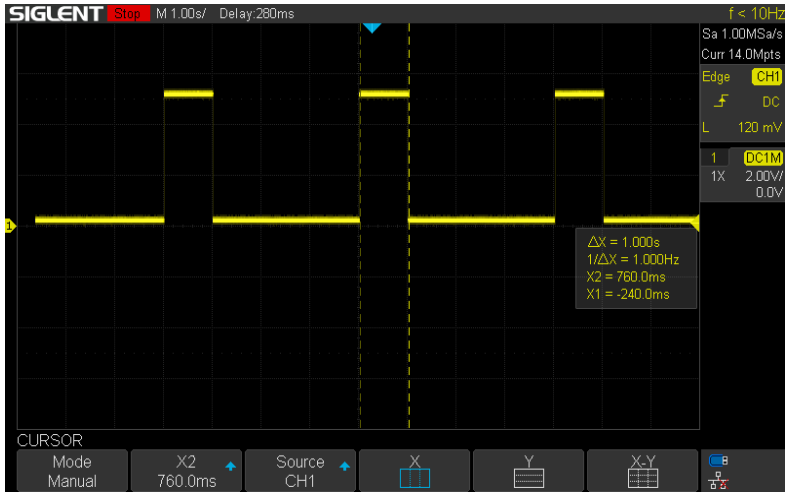
```
/*
Project: LED_Blink
Requirements:
Blink the Arduino UNO R3 builtin LED and an external LED.
Code for Serial.print() is commented out for the purpose
of not affecting timing for the scope traces.
This code uses a #define guard to easily switch between
compiling and running on the Arduino IDE or VSCode/PlatformIO
*/

// #define ARDUINO_IDE   // comment this out if using VSCode/PlatformIO
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define DELAY_TIME 1000   // use this to vary time delay
#define EXT_LED 2         // external LED on pin 2

void setup() {
// set up serial com
Serial.begin(9600);
Serial.print(" ** init **");
// set UNO LED pin 13 as output
pinMode(LED_BUILTIN,OUTPUT);
// init external LED
pinMode(EXT_LED,OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN,HIGH);
  //Serial.println("INTERNAL LED OFF");
  delay(DELAY_TIME);
  digitalWrite(LED_BUILTIN,LOW);
  //Serial.println("INTERNAL LED ON");
  delay(DELAY_TIME);

  // blink external LED
  digitalWrite(EXT_LED,HIGH);
 // Serial.println("EXTERNAL LED ON");
  delay(DELAY_TIME);
  digitalWrite(EXT_LED,LOW);
  //Serial.println("EXTERNAL LED OFF");
  delay(DELAY_TIME);
}
```

## Code Walk Through

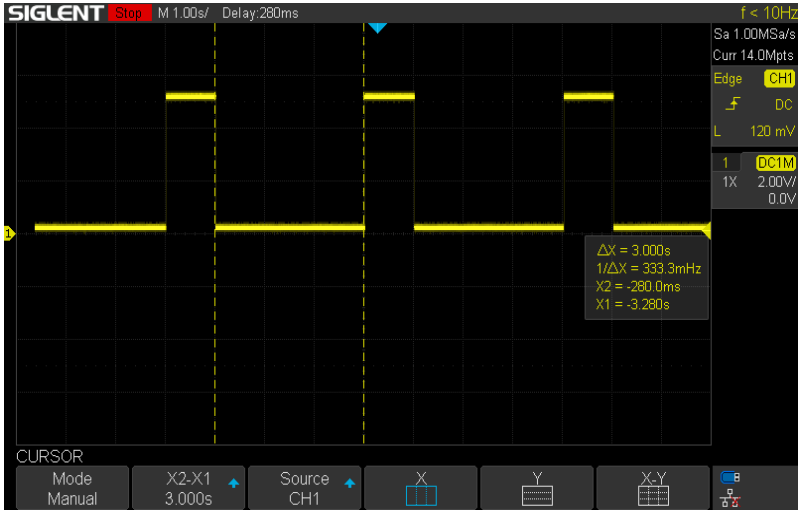
- Beyond the comments, the first thing done is to include a #define guard to use the same code using the Arduino IDE and PlatformIO. I developed this code in PlatformIO so there is no revision history, but it could easily be added in the comments.
- If the Arduino IDE is used, the comment lines // are removed from #define ARDUINO_IDE. This forces the #ifndef ARDUINO_IDE to be false, and the #include <Arduino.h> line is skipped. In the code above, #define ARDUINO_IDE is commented out therefore #ifndef condition is true and the header file <Arduino.h> is included.
- The time delay between turning the LEDs on and off is determined by #define DELAY_TIME value of 1000. This value is in milliseconds as required by the delay( ) function, so 1000ms is equal to 1 second. The external LED output, EXT_LED from the Arduino is on pin 2.
- In the setup( ) function the serial monitor is initialized via Serial.begin(9600). 9600 is the baud rate, or speed at which the Arduino outputs serial data. An ** init ** prompt is printed and the Arduino internal led pin is initialized as an output via pinMode(LED_BUILTIN,OUTPUT). The external LED, EXT_LED is initialized in the same fashion.
- In the loop( ) function the builtin LED is turned on via digitalWrite(LED_BUILTIN,HIGH). A delay of 1 second occurs via delay(DELAY_TIME) and the LED is turned off with digitalWrite(LED_BUILTIN,LOW). Another 1 second delay occurs with delay(DELAY_TIME).
- The next code sequence blinks the external LED. The external LED is turned on with digitalWrite(EXT_LED, HIGH) and then a 1 second delay occurs with delay(DELAY_TIME). The call to Serial.println( ) in the next line is commented out.

## Going Further

Let's take a look at an oscilloscope trace for the external LED.




The time division is set to 1 second per division on the horizontal and 2 volt increments on the vertical. The probe is clipped on the positive side of the 1K resistor and the ground clip of the probe of course is clipped to ground. The traced shows 3 seconds off and 1 second on. Does this jive with the code?



Let's start with the internal LED. The internal LED is energized for one second via the first 1 second delay. The internal LED is then turned off and another delay of one second occurs. That's a running total of 2 seconds. The external LED is energized for one second vial the next delay, giving a total of 3 seconds. The external LED is turned off and a delay occurs until the internal LED is energized as the loop repeats. So for the external LED, there's a 3 second delay before it energizes for 1 second, which corresponds to the scope trace.

Maybe this isn't the expected output so see on the oscilloscope, since the code for the external LED indicates that the external LED blinks on for 1 second and off for 1 second. This is an example that shows the great value of using an oscilloscope to troubleshoot and verify embedded system code. The intended code behavior is not necessary the way the code behaves.

If you have or have access to an oscilloscope I suggest trying out this code and varying the delay times and verify what you expect compared to your scope traces. This helps to sharpen your thinking and designs when writing code and will help you learn how to use an oscilloscope.

This is the basic form the project descriptions take. This of course is a very simple example, and they get more complex depending on the sensors that we will connect and interact with. This is where solid embedded engineering methods shine.

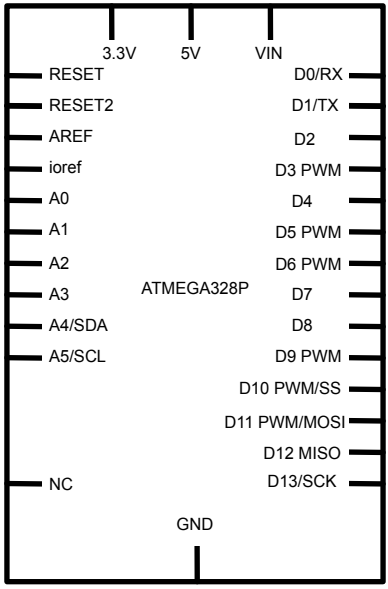
# CHAPTER TWO

## The ATmega 328p Microcontroller

At the heart of many Arduino boards, most notably the Arduino Uno, is the ATmega328P. The Atmega328P is a low cost but powerful 8 bit micro controller. It's built on a RISC (Reduced Instruction Set Code) architecture, contains 131 instructions and most instructions execute in one clock cycle. The ATmega328P is designed to perform a single task reliably and efficiently, making it ideal for embedded systems. The ATmega328P microcontroller features a total of 32 KB of flash memory for code, 2 KB of SRAM for data and 1 KB of EEPROM for non-volatile data storage. On Arduino Uno boards, roughly 0.5 to 2 KB of the Flash memory is used by the bootloader. All in all that's pretty good for a small, low cost microcontroller.

The 328P reads inputs from sensors, processes the information, and controls outputs such as LEDs, motors, displays and other actuators.. Despite its simplicity, the ATmega328P is remarkably versatile. It includes digital and analog input/output pins, timers, communication interfaces, and built-in features that allow it to interact with the physical world. By working with the ATmega328P, you gain insight into how real-world devices are controlled at the hardware level.

Here's a schematic of the ATmega328P microcontroller.

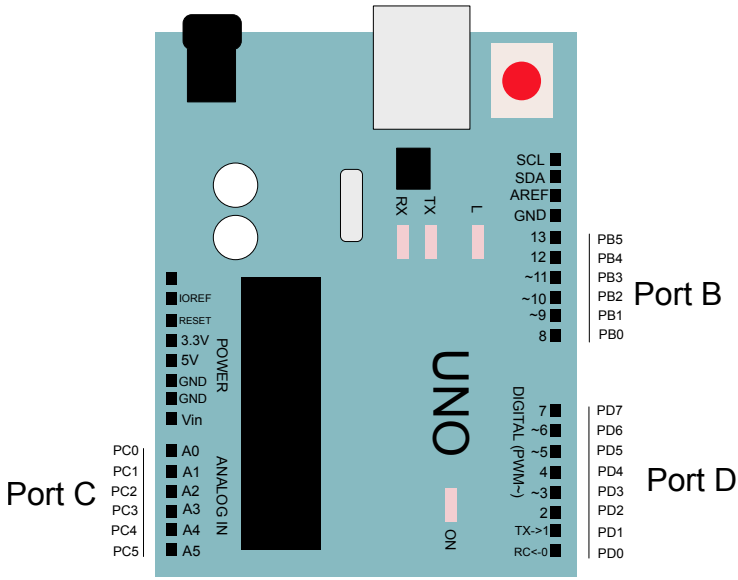



The ATmega328P supports 131 instructions with 32 general purpose registers which are directly linked to the ALU (Arithmetic Logic Unit). Most instructions execute in 1 clock cycle. Again, it contains 32Kbytes of programmable flash with read-while-write capabilities. It also contains 1KByte of EEPROM, 2KBytes of SRAM and 23 GPIO (General Purpose Input Output) lines.

The ATmega328P contains three timer/counters, 1 serial programmable USART, a 1 byte 2-wire I2C serial interface, a 6-channel 10-bit ADC (Analog to Digital Converter), a SPI (Serial Peripheral Interface) serial port and a watchdog timer with an internal oscillator, among many other features. Its on-board data retention is 20 years at 85 degrees C, 100 years at 25 degrees C and supports 10,000 flash write and erase cycles, and 100,000 EEPROM flash and erase cycles. The Atemga328P is powered by 2.7 to 5.5 volts. This microcontroller has been around for many years and is relatively simple, efficient and reliable.

## The Arduino UNO

The Arduino UNO was launched in 2010 and since then, over 10 million boards have been sold, making it one of the most, if not the most, popular microcontroller boards in the world. The Arduino UNO certainly broadened the reach of microcontrollers and makes the technology available to anyone at an extremely low cost. The UNO is popular with makers of all ages. The easy to use Arduino IDE (Integrated Development Environment) is free and provides a simple and reliable framework to program the UNO in C and C++.




## Architecture

The Atmega328P is a RISC processor, RISC stands for Reduced Instruction Set Computer. This essentially means that the Atmega328P executes one instruction on one clock cycle. CISC processors execute more then one operation and take several clock cycles to do so. CISC processors are generally slower than RISC processors but can perform many more complex and specialized instructions.

The Atmega328p, contains 32 x 8bit general purpose registers. The registers are:

- I/O registers that control the state of the digital inputs and outputs

- GPRs, or General Purpose Registers that are used for data manipulation and storage.
- Memory mapped registers that are located at specific memory addresses and control the 328p's memory and program execution.
- SFRs, or Special Function Registers that configure and control peripherals and functions, such as timers, counters, analog to digital converters, USARTs and the like.

Why so many registers? Registers are easy to access and are lightning fast in execution.

Regarding peripherals, the Atmega329p supports:

- 2 eight bit timer/counters.
- 1 sixteen bit timer counter.
- 1 USART (Universal Synchronous Asynchronous Receiver Transmitter) with a fractional baud rate generator and start-of-frame detection.
- 1 SPI (Serial Peripheral Interface).
- 1 I2C controller.
- 1 analog comparator with a scalable reference input.
- A Watchdog Timer with a separate on-chip oscillator.
- 6 PWM channels.
- Interrupt wakeup in a pin change.

Even though the Atmega328p is an 8bit processor it is by no means obsolete and is used in microcontroller applications all over the world. It's a mighty little chip.

## Damaging Your Arduino

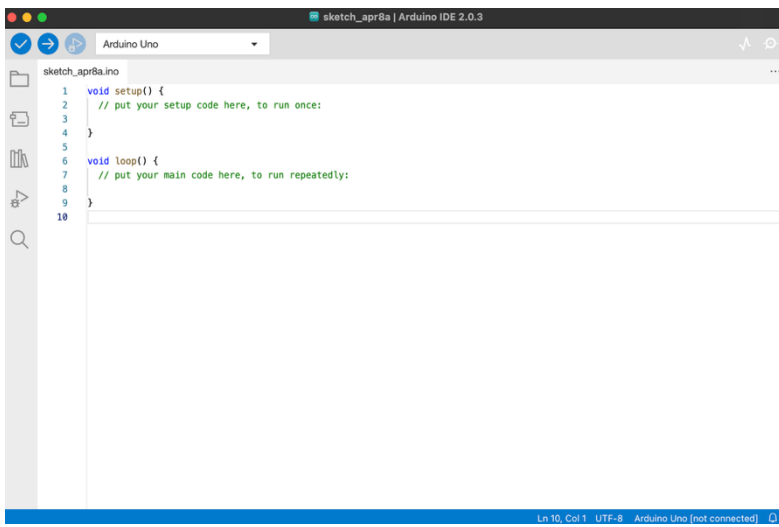
Your Arduino board is not invincible and is subject to damage, by YOU! There are several ways this can happen, but most damage occurs via an over-current or over-voltage error. Just be careful!

# CHAPTER THREE

## Development Environments

The primary development environment for Arduino hardware is the Arduino IDE. IDE stands for Integrated Development Environment. IDEs contain editors, access to compilers and, in many instances, access to debuggers. This makes it convenient for developers to concentrate on application development instead of fussing with separate development tools. The Arduino IDE is, at this time of writing, over version 2.3.8 and will certainly change rapidly over time.

### Arduino ID




The Arduino IDE is the environment of choice for countless developers,

especially those who are getting their feet wet in the world of embedded systems and microcontrollers.

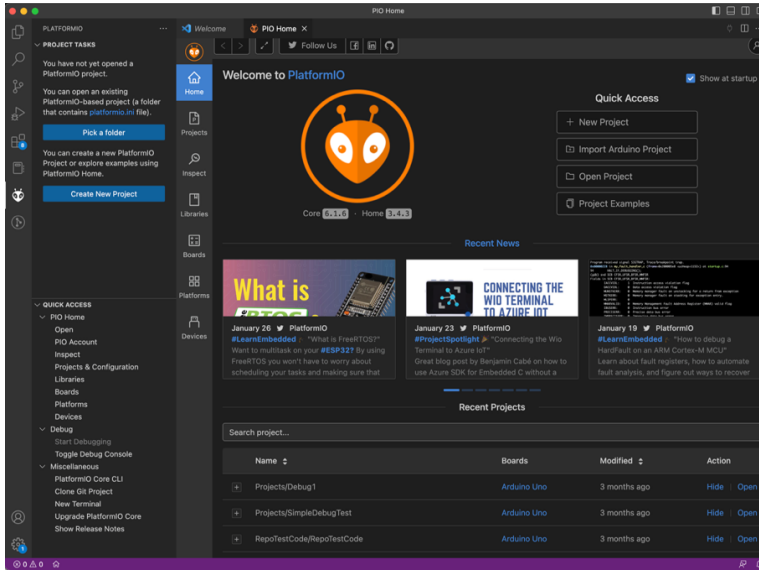
## VSCode/PlatformIO

Visual Studio Code (VSCode) is a free, easy-to-use code editor developed by Microsoft. It runs on Windows, macOS, and Linux. VSCode has become very popular because it is fast, simple to get started with, and can be expanded with extensions as your skills grow.

If you are new to programming, VSCode provides a clean environment where you can write and organize your code without feeling overwhelmed. Helpful features such as code suggestions, syntax highlighting, and an integrated terminal make it easier to learn and experiment.

For Arduino development, VSCode can be enhanced with tools like the Arduino extension, PlatformIO. PlatformIO is a powerful development environment that runs inside VSCode and adds features such as project management, library handling, source control and more advanced build tools. While the standard Arduino IDE is excellent for tons of work, PlatformIO offers a more professional workflow as your projects become more complex.

In this book, VSCode is used alongside the Arduino IDE to give you the best of both worlds: the simplicity of Arduino's native tools combined with the flexibility and power of a modern development environment.




So why use VSCode for Arduino? Visual Studio Code is not required for Arduino—but it can make your workflow much smoother as your projects grow. Here's why many developers use it.

- Better code editing: it provides clear formatting, color highlighting and helpful suggestions make code easier to read and write.
- Everything in one place: edit code, use the terminal, and manage files without switching between multiple programs.
- Works with PlatformIO: PlatformIO adds powerful tools for managing libraries, boards, and larger projects.
- Professional workflow: as your skills grow VSCode scales with you—from simple sketches to complex systems.
- Git integration built in: Easily track changes and back up your work (perfect for projects hosted on GitHub).

I suggest you start with the Arduino IDE to learn the basics, then move to VSCode when you're ready for more control and flexibility.

# CHAPTER FOUR

## Sensor Technologies

This chapter gives a general overview of varying sensor technologies. Sensors react to events in the analog world. These events can be temperature, pressure, chemical changes, sound, density, and other forms of mechanical or electrical energy. This list goes on and on. Sensors detect analog events from the outside world and transform them into another analog or digital output and sent to a computer for further processing.

### Transducers and Sensors

Sometimes you will hear the term sensor and transducer used interchangeably. This is not entirely correct. Sensors react to

environmental inputs and output information in the same form. Sensors convert physical signals to electrical signals. For example, a photo-resistor's resistance is altered as it reacts to light. It's not converting one form of energy into another- it's still resistance. A capacitive sensor may start with an initial capacitance value and change the value in reaction to an external force, such as touch or pressure. It's still capacitance. The output of a sensor is generally a measurable change in voltage.

Transducers convert one form of energy into another, usually in direct proportion. For example, a light bulb is a transducer. It converts a flow of electrons to visible light. An electric motor is a transducer that converts a flow of electrons to rotational motion. With a transducer, a change in the input quantity causes the output to change predictably. If more current is supplied to an electric motor, the motor will spin faster. We will use the terms pretty much interchangeably, but I wanted to point out the distinction.

## Passive and Active Sensors

Sensors boil down to two types, passive and active. Passive sensors do not require any external energy to do their job. Examples of passive sensors are photodiodes and thermocouples. Thermocouples converter temperature into voltage and photodiodes convert photons into voltage. Active sensors require external energy, called an excitation signal, to do their job. A classic example of an active sensor is a thermistor. A thermistor does not generate any voltage or current on its own, but when an excitation signal is applied (voltage) the thermistor's resistance changes with temperature.

## Sensor Properties

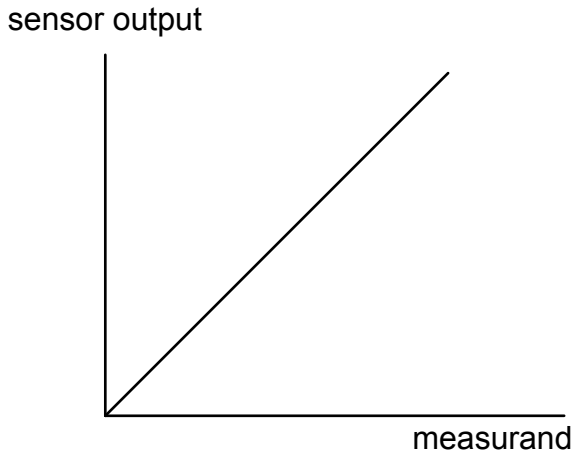
Some sensor properties to be aware of are listed below, in the form of a question. The characteristics apply to almost all sensors. These are things to inquire about any particular sensor.

- **Accuracy** - how accurate is the sensor and what is the sensor's resolution?
- **Stability** - how stable is the sensor? Is it overly sensitive to outside stimuli that influences the sensor's behavior?
- **Response** - is there a delay in the sensor responding to an external stimulus? If so, how much?
- **Hysteresis** - is there a big difference from one stimulus approach from another. For example, for a pressure sensor, does the sensor react differently when the pressure is increased and

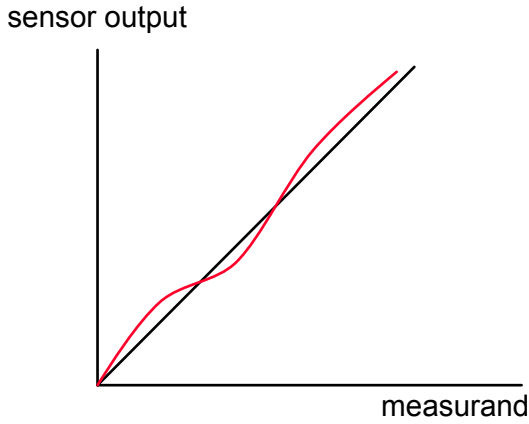
decreased?

- **Overload** - how does the sensor behave when it is overloaded, such as an overpressure or over temperature condition?
- **Linearity** - how linear is the sensor output? For example, thermistors are not linear and require some complex math to determinate accurate temperature.

So what would ideal sensor input look like? It would be perfectly linear, like the figure below.



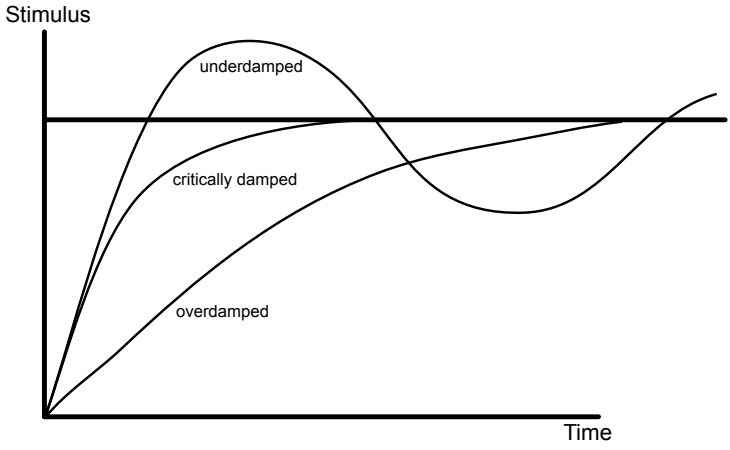

The Y axis is the sensor output value, which could be voltage, resistance, capacitance or inductance. The X axis is the measurand, which is the measured quantity, such as pressure, weight, force, temperature and other events in the real world. We live in the real world and there are no ideal sensors. They are subject to noise, temperature and voltage drift, hysteresis, and many other factors that alter their output. Sensor output can be close to linear for some values, but not all. Sensor output is subject to leading edge decay, trailing edge decay, a combination of both and oscillation. A sensor may behave more like the figure below, where the red trace is the actual measured input:




There are sections of the curve that approach linearity, but overall the sensor output is not entirely linear. Also, some sensors output logarithmic values, such as force sensing resistors, or FSRs.

## Damping

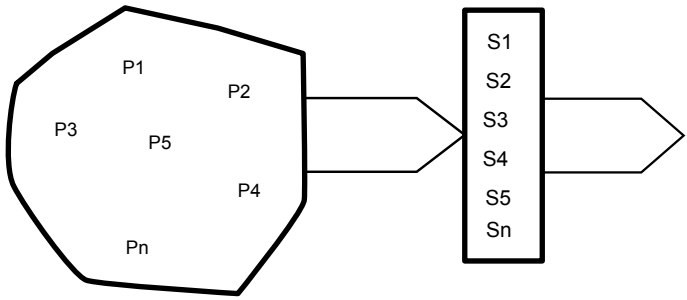
Damping is directly related how a sensor responds to an input stimulus, as the diagram below indicates. Damping reduces the oscillation of a sensor's response.




An underdamped signal overshoots and undershoots the target output before settling. An over-damped signal slowly responds before settling. A critically damped signal responds reasonably quickly before settling. A critically damped sensor, or course, is the most desirable.

## The Big View

Sensors react to phenomena in the physical, or analog world. This is represented in the figure below as P1 through Pn. Sensors and transducers, S1 through Sn, condition the detected phenomena and transforms the information into measurable electrical signals. The output of the sensors, S1 through Sn, are conditioned to be acceptable inputs to an external computer for further processing and display. This is called a data acquisition system.




There are discrete sensors, such as thermistors and photoresistors along composite system sensors which contain the sensing elements, including signal conditioning, as shown below.


Starting from left to right, the first entity in the chain or sequence is the environmental input to the sensors. The actual sensor that reacts to the input is generally a simple component. Sensors or sensing elements include resistors, capacitors, transistors, photodiodes, photoresistors and piezoelectric components. Other input components may be strain gauges, potentiometers, thermocouples, optical sensors, accelerometers- the list goes on and on. The conditioning block provides filtering, compensation, amplification, linearization and more. The signal then passes to an ADC, or analog to digital converter. This takes the conditioned signal and turns it into a binary signal- a one or zero. The binary output may be one or more bits, such as a byte or a word representing the state of the sensor input. The external interface may be, for example, a simple serial interface such as RS232, SPI, I2C, a bus interface, or a wireless interface such as Bluetooth, WiFi, Ethernet, a low

power 433MHz transmitter or many more forms of wireless communication.

## Sensor Technologies

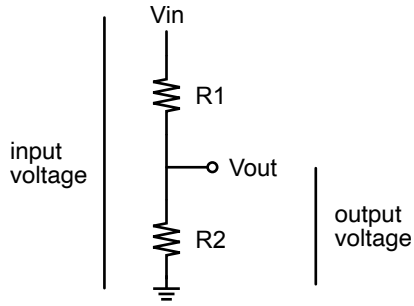
Sensors take input and measure or condition information from their environments. From these measurements electrical signals are produced and made available to external systems. There are analog sensors, digital sensors, and analog/digital sensors. Analog sensors return different values depending on the state of the sensors, where the output voltage is proportional to the sensor's measurement. Example analog sensors are Hall effect sensors and photo-resistors. Remember transducers? A microphone is a transducer, converting incoming acoustic energy into electrical impulses.

Digital sensors output a HIGH (true) or LOW (false) value. Example digital sensors are a tilt-switch, a shock sensor and a magnetic spring. Digital sensors, like analog sensors, have 3 pins and have the same configuration. Analog/Digital sensor modules, like many of those found in Arduino sensor kits, output analog and digital values. These sensors usually contain 4 pins, one for digital output, one for analog output, along with 5V and ground pins. Sensors of this type include metal touch, Hall effect and flame sensors.

## Resistive Sensors and Voltage Dividers

Resistive sensors are fairly simple and common. Physical displacement measured by change in resistance is also very common via potentiometers. Resistive sensors are used to measure temperature, force, pressure and may other applications. Microcontrollers, such as the Arduino UNO and most others do not read resistance directly. The answer to this is a voltage divider, one of the most basic and useful circuits in all of electronics. A voltage divider uses two resistors to scale a higher volage down to a lower voltage. The output of a voltage divider for an UNO R3 must be between 0 and 5V. Since the voltage may vary, the output of the voltage divider must be connected to an analog input which are pins A0 through A5. The UNO R3 accomodates this easily with 6 analog input pins connected to a 10 bit ADC (Analog to Digital Converter).

To build a voltage divider, all that needs to be done is to wire two resistors in series. A simple voltage divider is shown below.




$$Vout = Vin\left(\frac{R2}{R1 + R2}\right)$$

To find Vout, take the R2 value and divide it by the sum of R1 + R2, and then multiply by the input voltage. If both R1 and R2 are the same value, then Vout will be one half of Vin.

Voltage dividers are used in many resistive sensor applications. Here's an example, letting R1 = 1K Ohm and R2 = 2K Ohm. To find the total resistance, add the resistor values since they are in series, which is 3K Ohm.

$$Rt = R1 + R2 = 3K\ Ohm$$

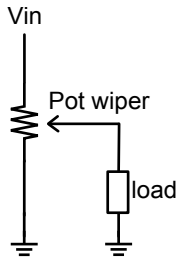
Now let the input voltage be equal to 5VDC. Since a voltage divider contains resistors in series, the total resistance is 3K Ohms, where Rt is the total resistance. The current is calculated using Ohm's Law.

$$I = \frac{Vin}{Rt}$$

Vin = 5VDC and Rt = 3K Ohm. This yields 1.67mA. Care should be taken and a few initial calculations like this should be made to make sure that the current is does not exceed the Arduino maximum allowable current for an input pin.
Let's say we want to monitor a 12VDC source, such as a car battery. We van let R1 = 10Kohms and R2 = 5KOhms, and Vin is 12V. The output of the divider is 4V, which is plenty safe for an UNO R3 input.

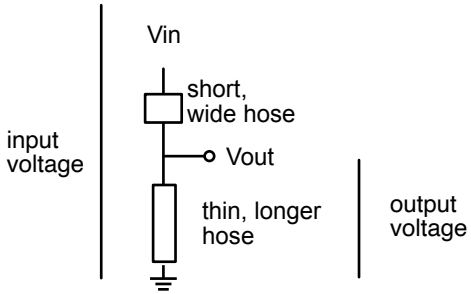
On of the simplest voltage dividers is a potentiometer. Pots are frequently used to measure distance and angle displacement and are also

used to drive circuits. The general form of a pot used as a voltage divider to drive a circuit is shown below.




The voltage on the load is controlled by the potentiometer wiper. The load is abstracted to be just about anything, but depending on the nature of the load and the current drawing capacity of the circuit, a transistor or other low to high voltage switching component may be needed.

One way to easily understand and visualize voltage dividers is to imagine each resistor is a hose of particular diameter, restricting the flow of electrons.



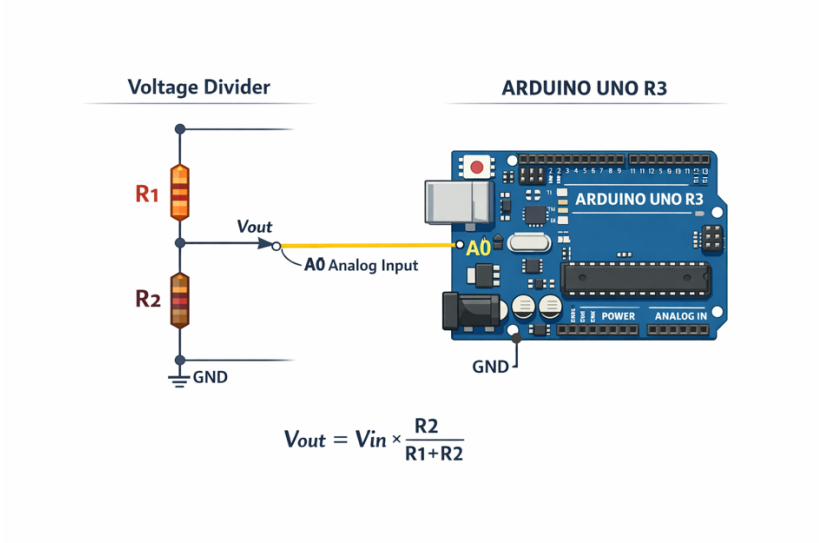

If a short, wide hose is used on the top and a thin, long hose is at the bottom, then it's clear that the short, wide hose will allow more electrons through that can be measured at Vout. This voltage will be a lot closer to the input voltage. If the hoses are reversed, then it's also clear that the voltage present at Vout will be tend more towards zero. This is a useful way to get insight into how voltage divider works.

Here's some tips when you build a voltage divider:

1. Use resistor values in the 1Kohm to 100Kohm range.
2. Avoid very low value resistors since they waste power.
3. Avoid very high values since they generate noise.
4. Add a 0.1microfarad cap to ground. This is optional but smooths noisy signals nicely.

5. Never, ever exceed 5V on the analog pins.

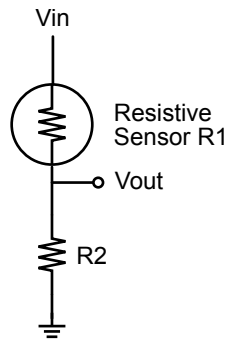
Here's a handy hookup for a voltage divider. The A0 pin is in the set of ANALOG IN pins in the lower right corner of the UNO R3.




Again, make absolutely sure the voltage does not exceed 5volts!

## Resistive Sensors

The general configuration of a resistive sensor is shown below. The resistive sensor is the upper component R1, and the lower resistor R2, is connected to ground.




There are several things to take into account when developing and/or creating your own resistive sensors. Thickness or the cross-sectional

area of the resistive material needs to be known. So does the length and conductivity of the material. Temperature also plays a factor. The length of the conductor is noted as $L$. $A$ is the area of the conductor and $p$ is the resistivity of the material, The resistance of the material can be calculated by the following formula:

$$R = p \cdot \frac{L}{A}$$

The resistance is equal to $p$ (resistivity of the material) times $L$ (the length of the material) divided by $A$ (the area of the material). Looking at the proportions, to double the resistance $R$, the length $L$ can be doubled. If the area is doubled, then the resistivity is cut in half.

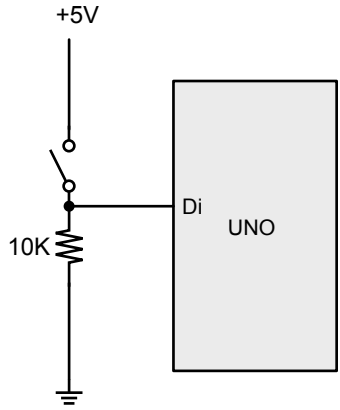
Understanding and building voltage dividers is one of the cornerstones of electronics and sensor interfacing.

## Capacitive Sensors

Capacitive sensors are fairly simple and reliable. They are used in all sorts of touch screen and button applications, force and moisture sensing, pressure sensing, positioning, displacement and many more applications. Capacitive sensors emit an electromagnetic field, and any disruption to this field is detected. Capacitive sensors are good at detecting metal, wood, plastic, glass, paper and many other materials. Capacitive sensors are widely used in industrial and commercial applications for proximity sensing and counting objects. The TTP223 capacitive touch sensor is cover later in this book.

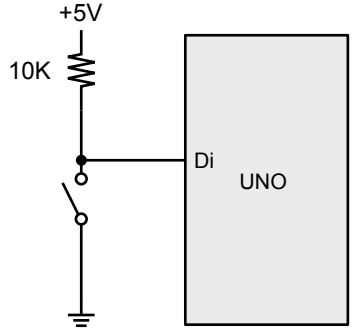
## Pull Up and Pull Down Circuits

Common to many inputs and outputs are pull-up and pull-down circuits. The first drawing below depicts a pull-down circuit going to an input pin, activated by a pushbutton.



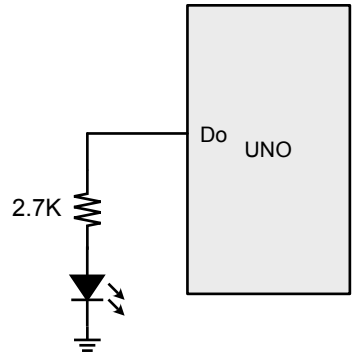

The input pin, Di, could be any digital input pin configured by pinMode(Di, INPUT). The lower case i stands for input. The input is pulled low with the switch open, and goes high when the switch is closed. This is called positive logic. When the input switch is activated a positive signal(+5V) is sent to the microcontroller via pin Di.

This drawing depicts a pull up circuit going to an input pin, activated by a pushbutton.



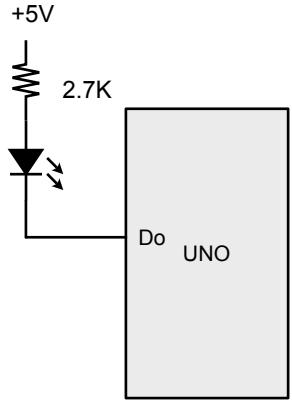

The input is pulled high by the 10K resistor and +5V, so positive volage is sitting on the input pin Dx. When the switch closes, the input is pulled low. This is called negative logic. When the input switch is activated negative signal (0V) is sent to the microcontroller via pin Dx.

The same positive and negative circuits also apply to outputs. The first drawing below shows an LED being driven by pin Do.




The output pin, Do, could be any digital output pin configured by pinMode(Do, OUTPUT). The lower case o stands for output.This is a positive logic circuit. When the output pin Do goes high, voltage is supplied to the 2.7K ohm resistor and the LED illuminates. Otherwise when Do is low, the LED is unlit. Depending on the characteristics of the LED, a current limiting resistor of 2.7K ohm or thereabouts works well.

The drawing below illustrates an LED being driven by an output pin, but in a different circuit configuration.



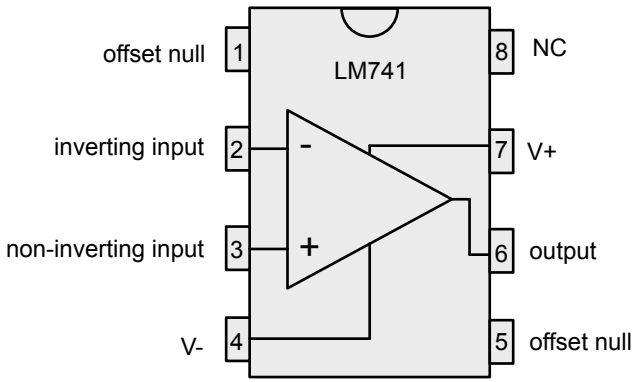

This is a negative logic circuit. When the input goes low, the LED illuminates, otherwise a high is maintained on output pin Do, turning the LED off. These are fundamental input and output circuit configurations, and care needs to be taken with respect to current. The maximum current that an Arduino IO pin can handle is 40ma. This is the absolute maximum- current in excess of 40ma will more than likely damage the board. Also, the voltage drops near this high-end current. Common

practice tries to keep the current on any give pin limited to 10 or 20ma. Anything outside or that makes it a good idea to use a transistor, usually a BJT or MOSFET or driver IC.

## Operational Amplifiers

One of the most useful components ever produced is the operational amplifier, or op amp. This unique chip enables tons of applications that would otherwise require unwieldy, complex electronic circuits and components. Operational amplifiers have three main features:
a) extremely high input impedance
b) low output impedance
c) extremely high gain
With the high impedance inputs, there is theoretically no current flow to or from the inputs. This means that, as far as inputs go, the op amp has no effect on the circuit that feeds it, such as a voltage divider. I said theoretically no current flows, but in the real world there is an infinitesimal amount of current flowing in and out of the inputs, but not enough to influence external circuits. What does an op amp fundamental do? Op amps take a small electrical signal and amplify it.

The most common op amp is the LM741, as shown below.



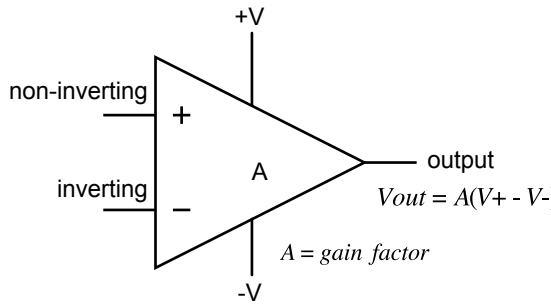

Here's the pinout:

- Pin 1 - offset null. Voltage is added to this pin to nullify any offset voltage. Often this pin is not used.
- Pin 2 - inverting input.
- Pin 3 - non-inverting input.

- Pin 4 - negative voltage supply
- Pin 5 - offset null. Voltage is added to this pin to nullify any offset voltage. Often this pin is not used.
- Pin 6 - output
- Pin 7 - positive voltage supply.
- Pin 8 - NC (no connection).

## Open Loop Differential Amplifier

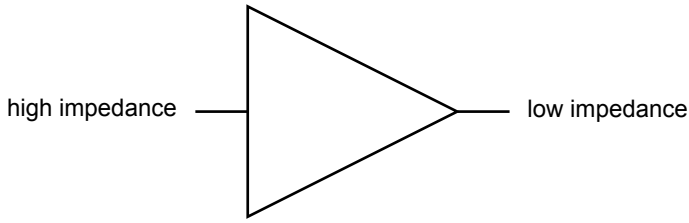
Open loop differential amplifiers are realized using an op amp in its simplest form, that is, without any feedback. For this we need to understand how an op amp functions on a fundamental level. Open loop op amps are rarely used in real applications but it's a good place to start when describing them. Looking at the figure below, an op amp has a non-inverting input and an inverting input, labeled positive and negative respectively. The voltages on the inverting and non-inverting are compared inside the op amp. Equal but opposite supply voltages are applied to the +V and -V pins, which are pin 7 for +V and pin 4 for -V. Let's say +10VDC is applied to +V and -10VDC is applied to -V. Looking at the input pins, if the voltage is higher on the non-inverting pin than the voltage on the inverting pin, then the op amp output will be +10VDC, which is known as the positive "rail" value. If the voltage is higher on the inverting pin than the non-inverting pin, the op amp will output -10VDC, which is the negative "rail" value. The rail values limit the output of the op amp, since we can consider the gain factor A is nearly infinite.



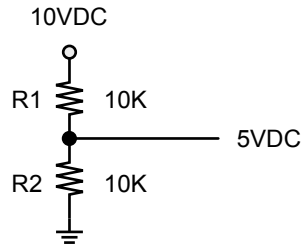

## Buffers

You may be thinking that what good is an open loop differential amplifier, and you are probably right, in and of itself. But this principle leads to a much more important application- buffers. A buffer takes a

weak input, both with respect to voltage and current, and outputs a low impedance signal that can drive other circuits. This is vastly important. The simplest symbol for a buffer is,



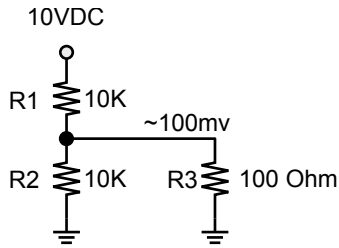

The value here is that the input is very high impedance and the output is very low impedance. No current will flow into or out of the input, acting like the buffer input isn't there. This is very useful when using voltage dividers in the input stage. Voltage dividers are good for providing a reference signal, but cannot be used with a load, which is another circuit or component that sinks current. For example, consider the voltage divider below:



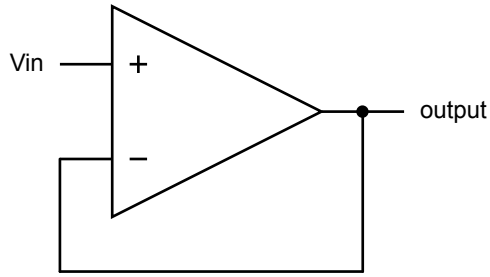

Repeated here is the formula for voltage dividers:

$$Vout = Vin\left(\frac{R2}{R1 + R2}\right)$$

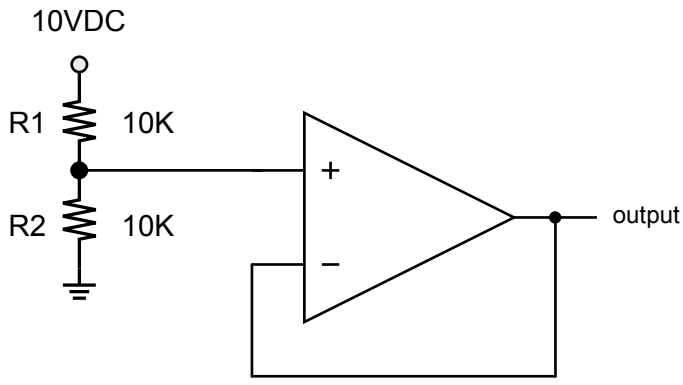
Using voltage dividers when utilizing or designing sensors is so common this formula needs to be completely understood and committed to memory. This simple voltage divider below divides the 10VDC into a 5VDC output at the junction of the two 10K resistors. This is with no load. Let's say a load of 100 Ohms is connected to the output. The output voltage drops to around 100mv.



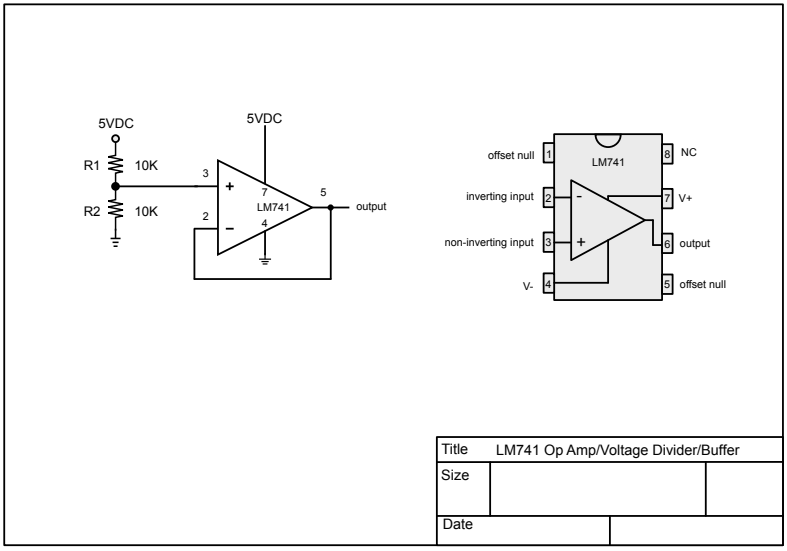

This is clearly not useful when a load is connected to a voltage divider. This is where buffers come in very handy, where the input impedance is so high that it has no effect on the output of the voltage divider. But what about the rail voltages? An open loop op amp will drive to the positive or negative rail depending on the inputs. This is mitigated by utilizing a negative feedback loop. This is simple to do with an op amp. Negative feedback forces the output voltage to be very close to the input voltage. Think of an object being pushed by a force. When the object moves, a physical negative feedback loop will push the object back to its original position. Below is a drawing of a fundamental negative feedback loop configuration.




The output is connected to the inverted input. This causes the op amp inverting input to match the non-inverting input, leaving the output equal to the input, Vin. The low impedance output can source generally 1 to 100ma. Here's a circuit connecting the voltage divider the non-inverting input pin.




The input to the op amp is 10VDC, and can range for the input values acceptable by the op amp. The negative feedback loop will keep the voltage at a stable 5VDC and is able to source current, unlike the voltage divider. This is obviously useful. Here's a complete schematic if the circuit, only with the input voltage at 5VDC. This circuit will output 2.5VDC at the output pin.

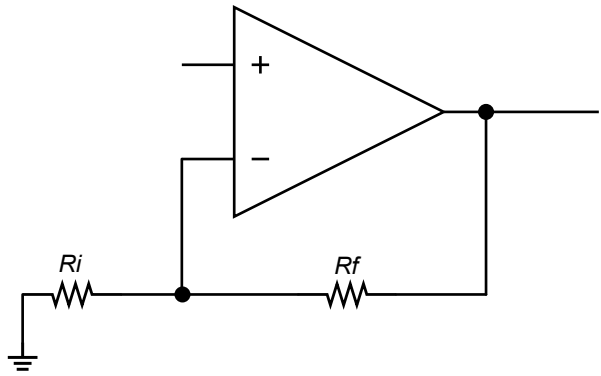



Driving a load with an op amp is reliable up to around 30-40mA, but not more. Some do more, and some do a lot less, so be careful to check the data sheet for the op amp you are using. The most common way to drive more current is to use a high-power transistor, with the op amp output connected to the transistor base. The op amp drives the base and the

current is moved through the collector and emitter.

## Gain and Non-Inverting Amplifiers

So far the buffer we've seen has a gain of 1:1. If one volt goes in, one volt goes out. The gain of an op amp can be easily changed by adding resistors to the feedback loop. The following simple formula is used to determine the gain change. First let's look at an op amp with negative feedback with a voltage divider in a different configuration. The output of the op amp is fed back into the inverting input via a voltage divider. This changes the gain of the amplifier and can force it to amplify an incoming signal on the non-inverting pin.




The gain formula is:

$$Av = \frac{Rf}{Ri} + 1$$

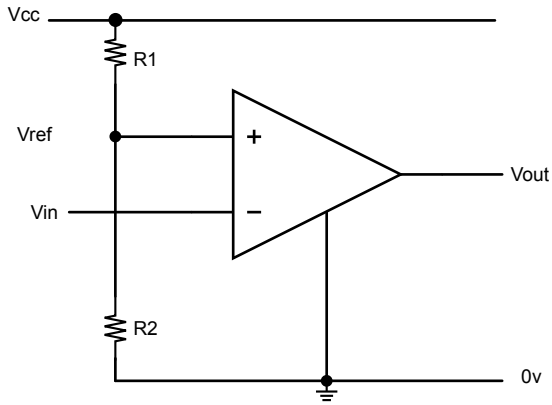
*Av* is the gain, *Rf* is the feedback resistor and *Ri* is the input resistor connected to ground.

## Comparators

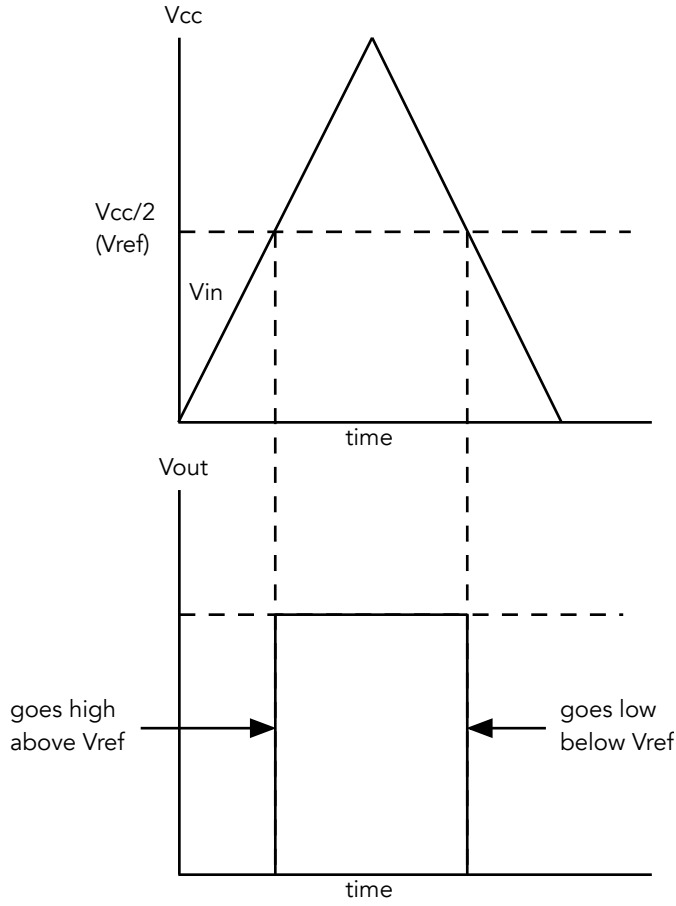
Comparators are the bridge between the analog and digital world. Comparators are used for analog to digital converters, null detectors, zero crossing detectors and many other applications. Comparators compare two input voltages and output a one or zero depending on the voltage relationship. Comparators act like operational amplifiers with no feedback. If one voltage is larger than the other being compared, the comparator switches all the way on, otherwise it stays at zero.

## Non-inverting Comparator

The figure below illustrates a fundamental, non-inverting comparator circuit.



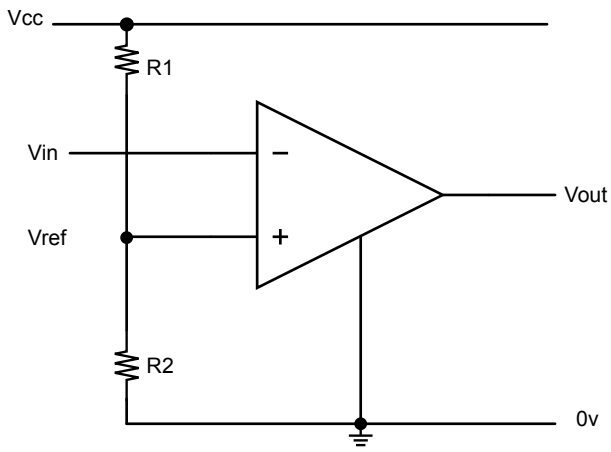

A voltage divider built from R1 and R2 provides a reference voltage on the negative pin of the comparator. This sets the threshold for the comparator to transition from a 0 to a 1 on Vout. The graph below shows how it works. Vin is the input voltage. R1 and R2 are selected for individual sensor applications.



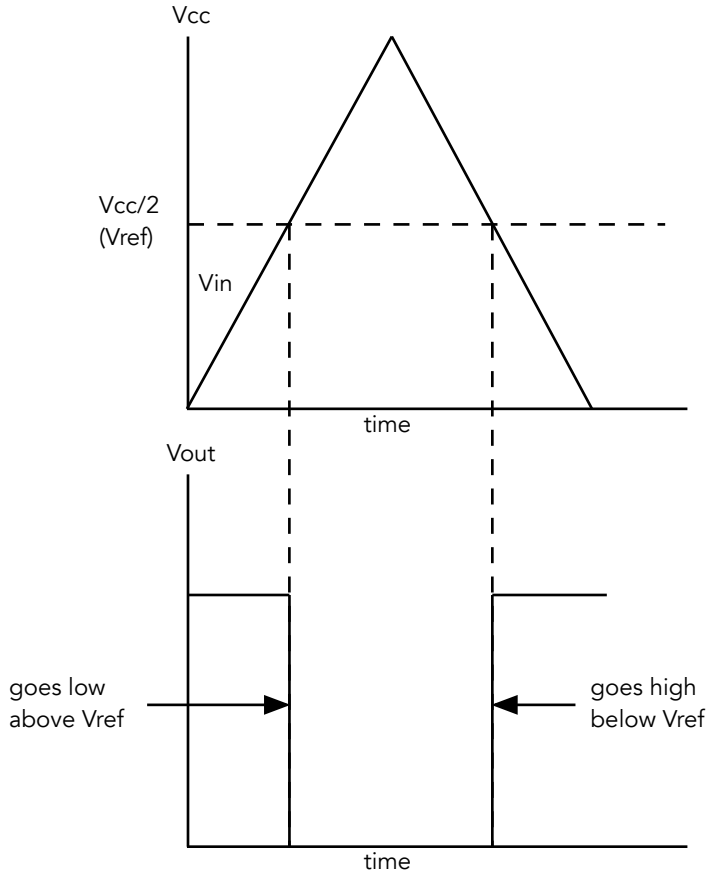

Looking at the top graph, the input voltage, Vin, is represented as a triangular trace. As Vin rises it matches the reference voltage Vref, which is the input from the voltage divider. At this point Vout goes high and stays high until Vin drops below Vref, forcing Vout to go low.

## Inverting Comparator

Now let's look at an inverting comparator, which is more effective for powering circuits, if need be. Vref is now on the plus pin and Vin is on the minus pin. The output will be high if Vin is less than Vref, and low when Vref is larger than Vref.



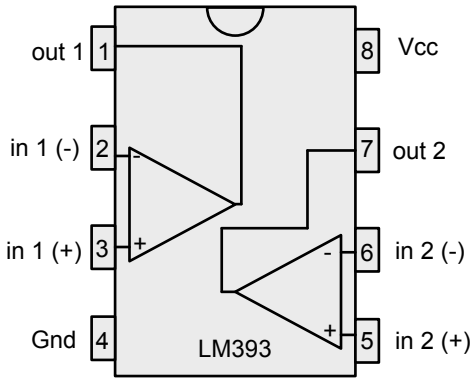

Like the non-inverting comparator, a voltage divider built from R1 and R2 provides a reference voltage on the negative pin of the comparator. This sets the threshold for the comparator to transition from a 0 to a 1 on Vout. Now, when the threshold voltage is hit Vout goes low. Vout returns to high when the threshold voltage drops below Vref.



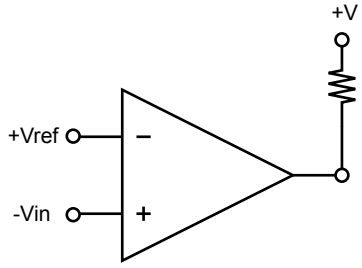

This awesome device is your gateway the measuring the analog world and building your own analog to digital converters.

## LM393 Comparator

At the heart of the 4 pin sensor modules is an LM393 dual comparator. The LM393 contains 2 precision comparators and is used to compare in input voltage to a reference voltage. Comparators take two voltages as input at +Vref and -Vin.

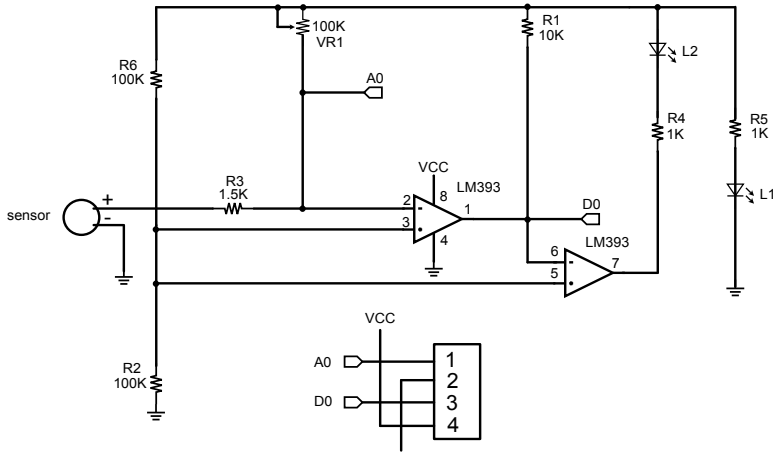



The comparator will output a logical 1 or HIGH if +Vref is greater than -Vin, and will output a logical 0 or LOW if -Vin is greater than +Vref. -Vin is called the inverting input or reference input. The +Vref input is typically connected to a 100K Ohm potentiometer to adjust the Vref voltage. The output typically utilizes a pull up resistor.



## General 4 Pin Sensor Module Schematic

Here's an example schematic of a standard Arduino 4 pin sensor board.




A0 is the raw analog output from the sensor and D0 is the digital output. The threshold level is set via the 100K pot VR1. The digital out D0 sources from pin 1 of the LM393. LED L1 illuminates when power is applied, and LED L2 flashes when D0 goes high. The comparator with pins 2, 3 and 1 do the real work. The comparator with pins 5,6 and 7 are used to illuminate R4, indicating the threshold crossing.

Thoughtfully, the 4 pin sensors are consistent and, if the sensors are of the same or similar type such as the big and small sound modules or the magnetic field detection modules, they are pretty much plug and play. All that's need to move from sensor to sensor is possibly a threshold voltage adjustment and some very minor code modifications.

## The Importance of Data Sheets

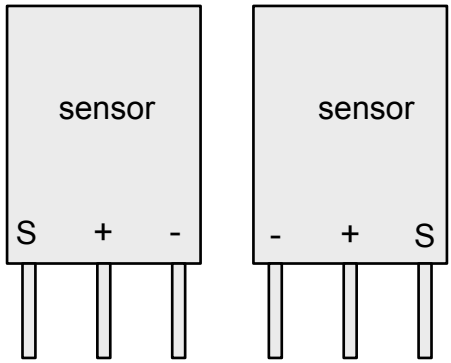
For every sensor there is a data sheet published by the sensor manufacturer. It's important and informative to study these.

## Sensor Configuration

Many if not most of the sensors available for Arduino are very similar in electrical configuration, and this is based on their pins. Aside from discrete components such as thermistors and photo-resistors, sensor modules for the Arduino are mostly partitioned into 3 and 4 pin configurations. The 4 pin sensors have more consistent, uniform pinouts than the 3 pin sensors. A typical 3 pin configuration is shown below.

## 3 Pin Configuration

Three pin sensors are plentiful for the Arduino.



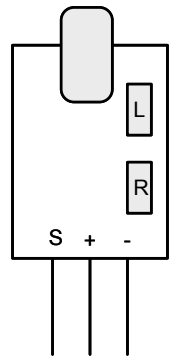

The sensor is located in most cases near the center of the circuit board. The sensor's output signal, S, is either the rightmost or leftmost pin. VCC, typically 5VDC, is the middle pin in almost all cases and Ground is on the rightmost or leftmost pin. Thankfully the pins are marked on the circuit boards, but care needs to be taken when swapping one 3 pin sensor for another.

Example 3 pin sensors for Arduino include photo-interruptors, IR emitters and receivers, laser emitters and many more.

# CHAPTER FIVE

## 2-3 Pin Input Sensors

Two pin sensors are almost all discrete components, such as resistors, capacitors and inductors. More sophisticated two pin sensors are thermistors and photo-resistors. Three pin sensors are a step up in sophistication, such as the DS18B20 digital thermometer which is covered in this chapter. These are of consistent design and make them easy to interface.




A generic sensor as shown above has three pins labeled S for the signal or output, + for Vcc input and - for ground. Unlike 4 pin sensors these are not adjustable in most cases and provide a digital versus analog output.

## Tilt Ball Switch

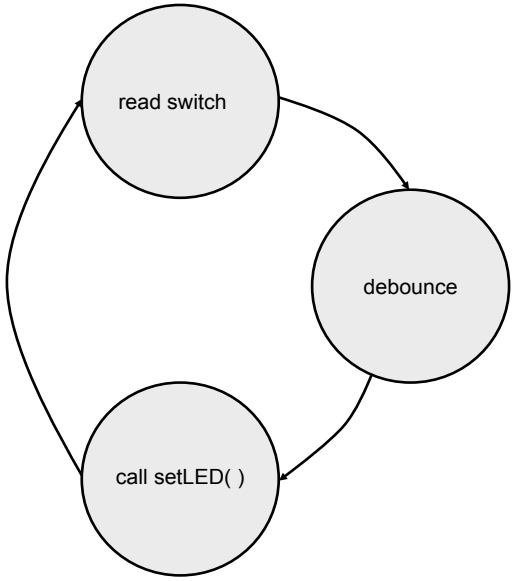
Tilt ball switches are useful for many applications. Some applications are cameras, robots, security systems, flight controls and many more. Tilt ball switches contain either a blob of mercury or a small metal ball that moves when the switch is tilted. Tilt switches are digital devices that contain a normally open contact. After the switch is tilted, the ball or mercury rolls by force of gravity causing the switch to short, or close. They are very easy to interface to an Arduino. These switches are sometimes called mercury switches or rolling ball sensors.

There are two tilt sensor types that come with sensor kits made for Arduino. This is the bare tilt sensor with two leads for the contact. This type of tilt switch looks like a capacitor.

The other type of sensor is a Keyes KY-020 tilt switch sensor. Like the two-wire sensor, the KY-020 contains a metal ball to close the normally open contact. The KY-020 has three pins, one for 5V, one for ground and one for signal, labeled S.

## State Diagram

The state diagram is simple and consists of three entities, read a switch, debounce the switch and turn on or turn off an LED.



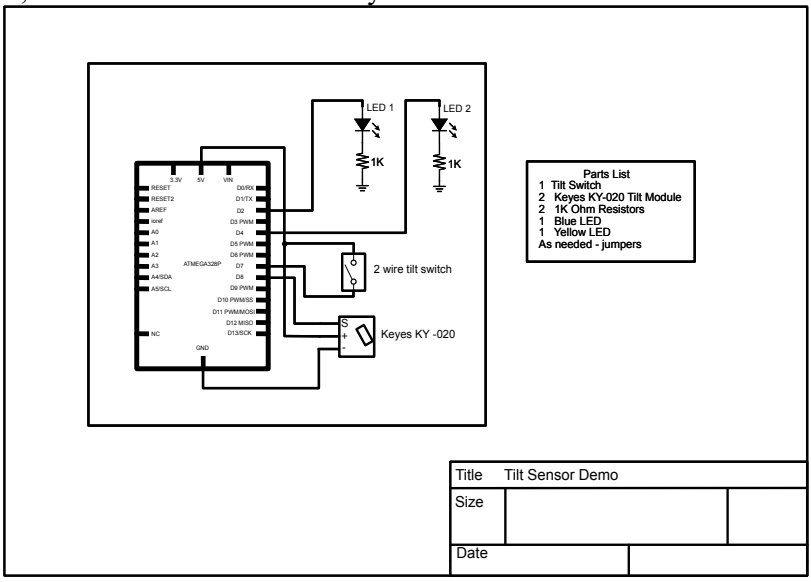
We will wire both tilt sensors to an Arduino UNO R3 as input and two LEDs as outputs. This being a mechanical switch, it is subject to bounce. We will use the debounce routine use in the TAC pushbutton demo.

The requirements are:

- Single tilt ball input on pin 7.
- KY-020 tilt ball module input on pin 7.
- LED_1 (blue) output on pin 2
- LED_2 (yellow) output on pin 4
- when the single tilt ball input is high, debounce input.
- light a blue LED (LED_1).
- when the KY-020 input is high, debounce input.
- light a yellow LED (LED_2).
- when the single tilt ball input is low, debounce input.
- turn off blue LED (LED_1).
- when the KY-020 is low, debounce input.
- turn off yellow LED (LED_2).

## The Circuit

The circuit consists of two LEDs, two resistors, one Arduino UNO R3, a 2-wire tilt switch and a Keyes KY-020 tilt switch module.



## Specifications

The bare tilt sensor has a sensitivity range of plus or minus fifteen degrees with a lifetime of over fifty thousand cycles. the plus or minus degrees sensitivity makes it useful for coarse applications, such as an object falling over or a sudden change in object orientation. It is not useful for applications such as electronic levels and the like.

## The Code

```
/*
File: TiltSwitchDemo1
Takes a 2 pin tilt switch and a Keyes YT-020 tilt
module as input. Fires LED_1 when pin 7 goes HIGH (switch is
closed). Fires LED_2 when pin 8 goes HIGH (switch is
closed). In this demo a blue LED was used for LED_1 and
a yellow LED for LED_2.
*/

// debounce function prototype
int debounce(int,int);
// LED driver prototype
void setLED(int,int);

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

// swith vars
const int switch1 = 7;  // 2 pin tilt switch
const int switch2 = 8;  // Keyes tilt module

// LED outputs
#define LED_1 2   // blue LED
#define LED_2 4   // yellow LED

// tilt switch inputs
#define TILT_1 7  // 2 wire tilt switch
#define TILT_2 8  // Keyes tilt module

void setup() {
   // init pins for sensors
   pinMode(TILT_1,INPUT); // 2 wire tilt switch
   pinMode(TILT_2, INPUT); // Keyes tilt module
   // init LED pins
   pinMode(LED_1,OUTPUT);  // blue LED
   pinMode(LED_2,OUTPUT);  // yellow LED
 }

void loop() {
  // light blue LED if tilted
  int state = digitalRead(switch1);
  state = debounce(state,switch1);
  setLED(state,LED_1);

  // light up yellow LED if tilted
  int state2 = digitalRead(switch2);
  state2 = debounce(state,switch2);
  setLED(state2,LED_2);
}

// debounce: debounces incoing switch state
int debounce(int state,int pin) {
  int curState = digitalRead(pin);
  delay(10); // a little time for stabilzation
  if(state != curState) {
    curState = digitalRead(pin);
  }
  return curState;
}

// turns on or turns off LED
void setLED(int inState,int pin) {
  if(inState == HIGH) {
    digitalWrite(pin,HIGH);
  } else {
    digitalWrite(pin,LOW);
  }
}
```

## Code Walk Through

- The function debounce( ) and setLED( ) prototypes are declared. This is required and standard practice in C/C++ programming.
- The #define ARDUINO_IDE switch is used to determine whether the Arduino IDE is used or PlatformIO running under VSCode.
- The global variables the monitor the state of the switches are declared, switch1 on pin7 and switch2 on pin8 on the Arduino. These are declared as const variables. These variables are passed to functions, but are not changed. The const declaration before them will generate a compiler error if the variables are attempted to be changed. These variables are used instead of #define declarations. Occasionally compilers balk at #defines being passed as parameters. Using a const int declarations ensure that the code will run correctly when those variables are encountered.
- The LED output #defines are declared for the blue LED_1 pin 2 and the yellow LED_2 pin 4 on the Arduino..
- Next the #defines for TILT_1 and TILT_2 for tilt switch 1 and tilt switch 2 are declared. Note there are two places where the switches are identified: in the variable declarations and in the #defines. The #defines are used setup( ) to initialize the pin modes and the variables are used as parameters to be passed to the declared functions.
- In setup( ) the tilt switch inputs are set with the pinMode(TILT_1,INPUT) and pinMode(TILT_2,INPUT) functions.
- The LED pins are set as output via the pinMode(LED_1, OUTPUT) and pinMode(LED_2, OUTPUT) functions.
- In loop( ) three functions are called for each switch and LED. A return variable, state, is declared, int state, which captures the return value for digitalRead(switch1). The state and const switch variables are passed to the debounce( ) function and the return value is stored in the state variable that was passed as a parameter. This is an efficient technique to save memory.
- The last function called is setLED, which takes the switch state and led pin.
- The debounce( ) function is the same as in the TAC pushbutton sketch, but with the addition of a 10ms delay. Tilt sensors

bounce like crazy and this short delay helps with a stable and accurate switch reading. The debounce( ) function returns the state of the tilt switch.

- the setLED( ) functions takes the state of the tilt switch and LED number as input. The LED is turned on if the switch states is HIGH or turned off if the switch state is LOW.

## Going Further

Tilt switches are subject to bounce, and you will notice that the sensitivity of the two tilt switches, the 2 pin and Keyes KY-020 behave differently. This is a good place to create and test debounce functions and compare their behavior. Use two of the same tilt switches and put them on a breadboard side-by-side. Write two separate debounce( ) functions or grab some from the web. As said in the TAC Pushbutton section, dealing with switch bounce is a fact of life in embedded systems programming.

## DHT11 Temperature and Humidity Module

The DHT11 is a simple and easy to interface temperature and relative humidity sensor that is made for an Arduino. Outside of being useful in itself, the DHT11 comes in handy when using it with the HC-SR04 ultrasonic sensor. The speed of sound is affected by temperature and humidity, and taking both into account when determining distance makes the final ultrasonic measurements more accurate.

What is relative humidity, exactly? Relative humidity is the measurement of water vapor in the air. Relative humidity is a ratio of how much water vapor is in the air relative the maximum amount of water the air can hold, which is the saturation humidity. Why is this important? One good reason is for monitoring storage of products and goods. Often goods are affected by humidity and may be rendered unusable if exposed to excess humidity over time. Monitoring relative humidity along with range checks that establish upper and lower humidity boundaries are a good way to mitigate potential damage.

## Requirements and Comments

The DHT11 is a simple sensor circuit that provides relative humidity and temperature readings. The output is on pin S, which is on the left when facing forward, 5DC is supplied to the middle pin and Gnd on the far right pin. The DHT11 is supported by the DHT sensor library created and supported by Adafruit available on GitHub.

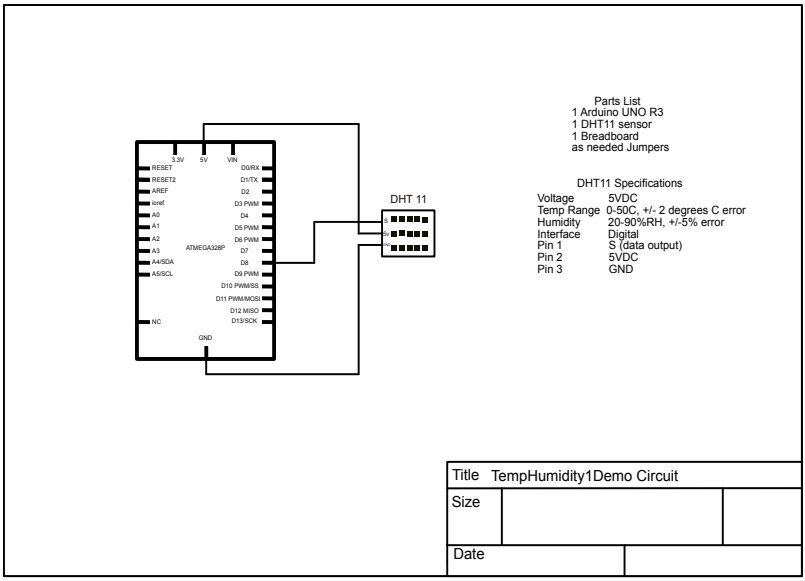
The sketch is required to:

- read the relative humidity and temperature.
- convert to degrees F. The temperature by default comes back in degrees C.
- print the humidity and temperature values via Serial.print.
- delay 2 seconds between readings.

More details, including where to get the DHT library are given below.

## The Circuit

Here's a simple circuit we'll begin with. The DHT11 reads temperature and humidity and outputs the values on pin S on pin 1. 5VDC goes to the adjacent pin, Pin 2. Ground is connected to the next pin, Pin 3.



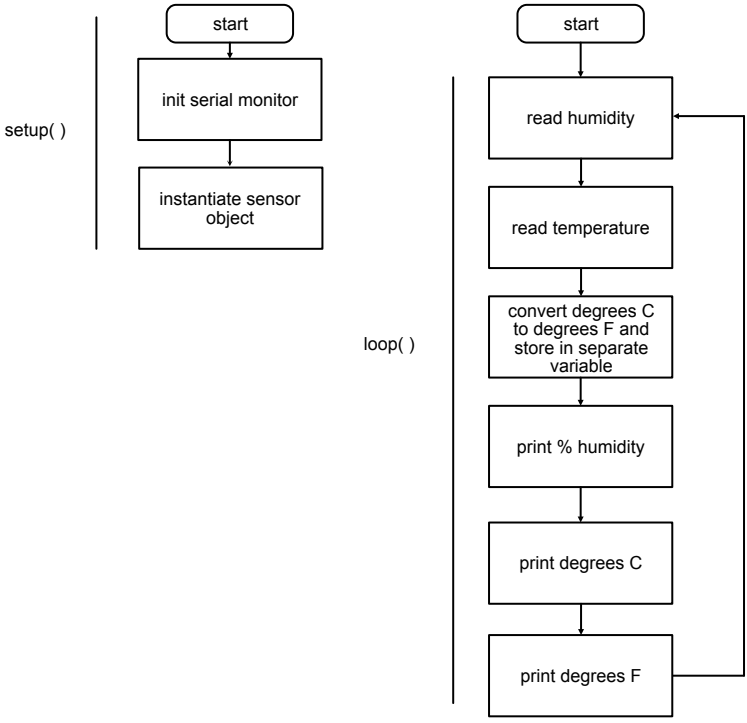

The DHT11 sensor takes 5VDC as power. The one and only data pin is in the middle, and Gnd is on the third and last pin. The sensor has a tolerance of +/-2 degrees Centigrade, and the relative humidity tolerance it +/- 5%. It's not the most sensitive sensor regarding temperature and humidity, but it will do the job in many monitoring situations.

Pin 8 on the UNO is connected to Pin 1 on the sensor and is marked blue. 5V goes to Pin 2 and is marked red, and Ground goes to Pin 3, marked black. It's a good idea to make a habit out of using the bus strips on a breadboard for supply voltage (5VDC) and ground. Placing power on the bus allows us to easily supply power to additional components. Plus it's neater in appearance, especially when new components are added.

## Library Functions

A commonly used library is the Adafruit/DHT-sensor-library accessible on GitHub, as also noted below.

## Flowchart




## The Code

The sketch is designed to read temperature and relative humidity, delay and repeat the readings. The delay time is controlled by a constant so it can easily be changed. As stated, the sketch also uses the DHT library developed by Adafruit Industries and is available here: https://github.com/adafruit/DHT-sensor-library The DHT code is governed by the permissive MIT open source license.

To install the library, select Manage Libraries from the Tools menu on the Arduino IDE and then select “Manage Libraries…”. Search for the “DHT sensor library” and install it. The DHT library is written in C++. Take a look at the include file DHT.h. Of main interest are the Public methods listed in class DHT, since these are easily used in code you wish to develop.

Here’s the baseline sketch that can be built upon using the DHT11 sensor.

```
/*
File TempHumidity1Demo.cpp
This sketch reads humidity and temperature from
a DHT11 humidity and temperature sensor. The code
utilizes the DHT sensor Adafruit library.
Uncomment #define DEBUG so see direct sensor input from
sensor method calls.
*/
//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#include <Adafruit_Sensor.h>
#endif
// DHT sensor lib header file
#include <DHT.h>

//#define DEBUG
#define DELAY_TIME  2000    // 2 second delay between reads

#define DATA_PIN    8       // incoming data

#define DHTTYPE     DHT11   // sensor type

DHT dht(DATA_PIN,DHTTYPE);  // instantiate sensor

void setup() {
    Serial.begin(9600);     // init serial monitor
    dht.begin();            // init sensor
}

void loop() {
  // get relative humidity
  float h = dht.readHumidity();
#ifdef DEBUG
  Serial.print("h is: ");
  Serial.println(h);
#endif

  // get temperature
  float t = dht.readTemperature();
  #ifdef DEBUG
  Serial.print("t is: ");
  Serial.println(t);
#endif

  // default is metric - convert to F
  float tf = (t * 1.8) + 32;
#ifdef DEBUG
  Serial.print("tf is: ");
  Serial.println(tf);
#endif

  // display humidity
  Serial.print("Humidity: ");
  Serial.print(h);
  Serial.println("%");

  // display temperature
  Serial.print("Temperature: ");
  Serial.print(tf);
  Serial.println("F");
  delay(DELAY_TIME);
}
```

## Code Walkthrough

Let's walkthrough the code.

- If the Arduino IDE is used, then the #define ARDUINO_IDE needs to be uncommented. This bypasses having to include the <Arduino.h> and <Adrafruit_Sensor.h> files. These are not required for the Arduino IDE but are required when using PlatformIO. Since this code was developed using PlatformIO, #define ARDUINO_IDE is commented out.
- The DHT library include file <DHT.h> header file is included. This is needed for the Arduino IDE and PlatformIO.
- A #define DEBUG is included. Using a #define to include debug statements is a good idea since the #define is a preprocessor directive and does not take any space in the executable program. This way you can instrument your code as you debug it with Serial.print( ) statements.
- The next two statements #define DATA_PIN 8 (where the data is received on the Arduino) and the sensor data type, DHT11 which is required by the DHT library.
- A data structure for the sensor is created by the line DHT dht(DATA_PIN,DHTTYPE). This is required by the DHT library.
- In setup( ) serial communications are initiated by Serial.begin(9600) and the sensor code is instantiated via dht.begin( ).
- In loop( ) we declare a float variable h and use that to store the value returned by the method call to dht.readHumidity().
- The value for h will be printed to the serial monitor if #DEBUG is not commented out. Using #DEBUG directives like this make for cleaner, more readable code.
- The variable t is used to store the value returned by the method call to dht.readTemperature( ). Like the humidity reading, the value of t will be printed to the serial monitor if the #define is not commented out. Note that the value returned by the call to dht.readTemperature( ) is in degrees C.
- The last step is to convert the t value into F units. This is performed by the declaring a new float variable, tf, and using the conversion formula tf = (t * 1.8)/2.
- The last two blocks of code display the humidity and temperature via the serial monitor.

Consider using #define DEBUG as a debugging tool to debug your code. You are not restricted to using just one #define DEBUG. You can have many debug #defines for areas that you are developing and debugging. For example, if you were developing a complex servo application, you could have #define DEBUG_SERVO_FN to debug a servo function. Be expansive and creative in your debug #defines.

## Going Further

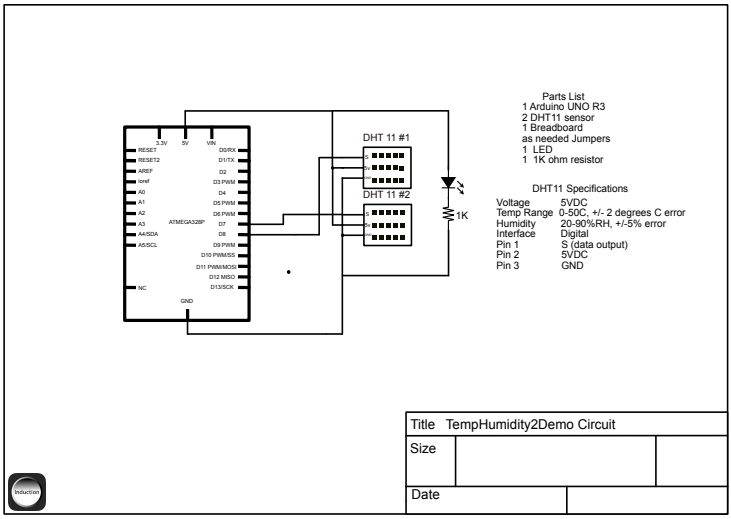
The DHT library contains additional public functions. Our temperature conversion calculation was simple by merely manipulating the temperature reading returned from calling dht.readTemperature( ). The value returned in in degrees C. To convert the temperature to F, the line of code to do this is float tf = (t * 1.8) + 32. As a little experiment you can call the DHT library function dht.convertCtoF( ) and compare the results to our in-line code to see if it is different. I did and the results were identical.

We can also clean up the code by creating a void disp_dat( ) function to display the Serial Monitor data. All we need to do is declare the function prototype at the top of the code: void disp_dat(float, float, float). We need three floats as parameters for humidity, the Celsius value and the Fahrenheit value for each sensor.

Another useful feature we can add is a heartbeat LED that flashes every time a reading is processed. This can be considered a “heartbeat” circuit that lets you at a glance if your sketch is running. So let’s add the additional hardware and code to incorporate these features.

## The Circuit

The circuit is still very simple since there are only three wires for each sensor. The LED uses the same time delay as the sensor read, so this gives a quick-glance look at how the circuit is behaving. To check the difference between one sensor reading and another, place the DHT11 sensors closely together when you breadboard the project.




We added the other DHT11 to the right of the first. The data pin (S pin) goes to the Arduino Digital Pin 7. This is where we will read the relative humidity and temperature values. The LED uses a 1K pulldown resistor and its energizing output is on digital Pin 2.

## The Code

The sketch now reads two DHT11 sensors and output a signal to a heartbeat LED that is in sync with the time delay value.

```
// TempHumidDemo2.cpp
//#define ARDUINO_IDE  // uncomment if using the Arduino
IDE
#ifndef ARDUINO_IDE
#include <arduino.h>
// DHT11 sensor library
#include <Adafruit_Sensor.h>
#endif
#include <DHT.h>

// function prototypes
void dispDat(float,float);

// constants
#define DELAY_TIME 3000 // delay time between readings
#define DATA_PIN_1 8    // data from first sensor
#define DATA_PIN_2 7    // data from second sensor
#define DHTTYPE DHT11   // sensor type requried by DHT
library
#define LED_PIN 2       // LED output pin

// sensor data structures
DHT dht1(DATA_PIN_1,DHTTYPE);
DHT dht2(DATA_PIN_2,DHTTYPE);

void setup() {
  // init serial port for serial monitor
  Serial.begin(9600);
  Serial.print("** sensor init **");
  // init DHT11 sensors
  dht1.begin();
  dht2.begin();
  // init LED pin
  pinMode(LED_PIN,OUTPUT);
}

void loop() {
// read temp and humidity via local variables
float h1 = dht1.readHumidity();
float t1 = dht1.readTemperature();
float h2 = dht2.readHumidity();
float t2 = dht2.readTemperature();

// toggle the heartbeat LED
digitalWrite(LED_PIN,LOW);
delay(DELAY_TIME);
digitalWrite(LED_PIN,HIGH);

// process sensor data
// convert sensor 1 data to F
// manual conversion - uncomment and recomile
// to compare results to library function
//float tf1 = (t1 * 1.8) + 32; // temperature from sensor
1

float dht1_f = dht1.convertCtoF(t1);  // temperature via
library method

// convert sensor 2 data to F
//float tf2 = (t2 * 1.8) + 32;
float dht2_f = dht2.convertCtoF(t2);

// write to serial monitor with manual conversion
//dispDat(h1,tf1);
//dispDat(h2,tf2);
// write to serial monitir with lib conversion methods
dispDat(h1,dht1_f);
dispDat(h2,dht2_f);
}

void dispDat(float h,float t){
Serial.print("Humidity: ");
Serial.print(h);
Serial.println("% ");
Serial.print("Temperature: ");
Serial.print(t);
Serial.println("F");
}
```

## Code Walkthrough

This sketch, based on the previous code, has been changed and refactored to an extent. Here's a summary of the changes.

- Instead of using in-line code to print the results we used the voice disp_dat( ) function that takes temperature and humidity values as parameters and prints them to the serial monitor.
- #defines DATA_PIN_1 and DATA_PIN_2 were added for the data output pins of each sensor. DATA_PIN_1 is pin 8 on the Arduino, and DATA_PIN_2 is pin 7 on the Arduino.
- DHT dht1(DATA_PIN_1, DHTTYPE) was dht, now is dht1 to correspond with sensor 1.
- DHT dht2(DATA_PIN_1, DHTTYPE) dht2 was added to correspond with sensor 2.
- #define LED_PIN 2 is the output pin to drive the heartbeat LED.
- In setup( ) the serial monitor is setup, a simple init message is printed, dh1 (sensor 1) and dh2 (sensor 2) are activated via the two dh1.Begin( ) and dh2.Begin( ) methods.
- pinMode(LED_PIN,OUTPUT) assigns the LED pin as an output.
- In loop( ) we read the humidity and temperature in sequence for each sensor via the readHumidity( ) and readTemperature( ) methods, storing them in local floats h1, t1, h2 and t2.
- The LED is turned off then turned on during sensor processing.
- Unlike the original sketch, we call the DHT library method convertCtoF( ) to convert from the default degrees C to F. To use the original code, comment out the convertCtoF( ) functions and uncomment out the manual conversion code.
- The humidity and temperature readings are displayed by successive calls to dispDat( ), where the humidity, degrees C and degrees F are displayed via parameters h, and t.

The DHT11 temperature sensor has a fairly loose tolerance of +/-2 degrees C. I found this to be true when I tested the circuit and compared the output between the two sensors. One way to mitigate this is to take a series of reading and average them. Try this yourself and see what you get.

At the time of this writing you can get a DHT11 for around six dollars and a ten pack for around twelve dollars. This is fairly inexpensive for a reliable and easy to interface sensor that is useful in many applications

Another good use for the DHT11 is to use it along with an ultrasonic sensor, such as the HC-SR04 that is used in this book. The speed of sound is affected by temperature and humidity, and taking the current temperature and humidity into account when calculating the distance to

an object will make the distance measurement more accurate.

## 5V Relay

THE UNO R3 Project Super Starter Kit comes with a bare bones 5VDC relay. The relay is a SRD-05DC-SL-C model made by SONGLE. It comes with one contact.

## Requirements and Comments

The circuit we will build and write a sketch for is a timed output signal from the Arduino to energize the coil for a brief period, close the NO (normally open) contact and light up an LED. Relays are switches that are controlled by applying voltage to a coil.

When the coil is energized, the contact changes state, allowing voltage or current to pass through. One advantage of using a relay is that a much larger voltage or current can be passed through the contacts compare to the small voltage required to energize the coil.

In the SONGLE relay, the coil is represented by the white box as shown in the figure above. On one side is ground, and on the other is an input marked S for Signal. The single contact in the unenergized state is NC, which stands for Normally Closed on one side and the other is NO, which stands for Normally Open. Connected to the contact is 5VDC in our demonstration. This does not have to be restricted to 5VDC. It could be a much different voltage, such as 120VAC, 48VDC and the like. That's one of the advantages of a relay. A much higher voltage can pass through the contacts while the coil can be energized from voltages as low a 3VDC. One way to think of a relay is a mechanical switching transistor.

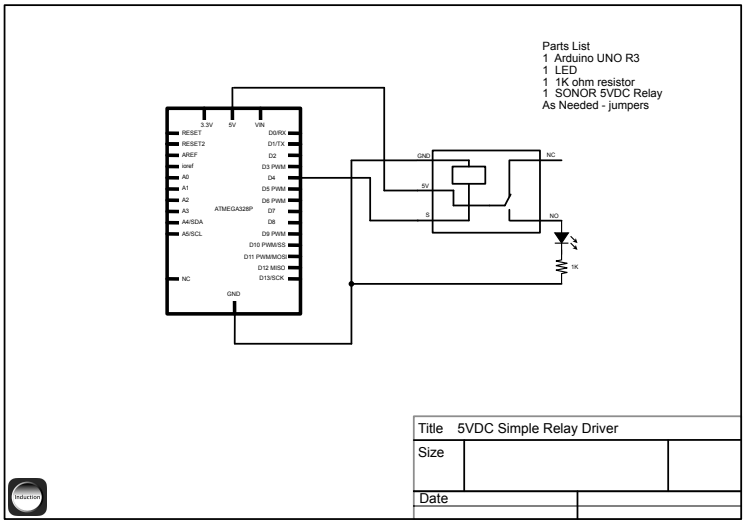
When S is pulled high, the coil energized and the contact switches from the NC rest position to the NC position. The NC contact opens and breaks any circuit it's connected to. The NO position now allows the 5VDC to pass through the contact to act on any circuit it's connected to.

## Relay With Terminal Board

To make life easier when prototyping relay circuits, many SRD-05DC-SL-C relays are mounted on a circuit board that includes screw terminals for the contacts, an LED that turns on when the coil is energized and three pins that align with breadboard pins. I've found that connecting the relay with Dupont connectors make for better contact.

## The Circuit

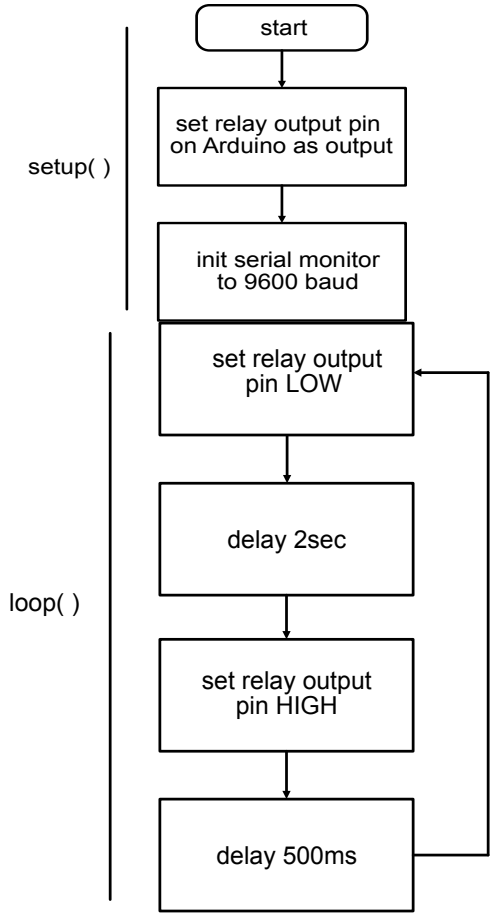
Here's the schematic circuit we will use. We are looking at the top view of the 5VDC relay.




Looking at the relay from the top, Ground is the first pin at the top, the voltage to be passed through the NO or NC contact is on the middle pin, and S is on the bottom pin. The input to S should be 5VDC to energize the coil. When the coil energized the NO contact closes allow the middle pin voltage, in this case 5VDC, to be passed through to act on an external circuit. In our circuit, an LED is energized that is pulled down by a 1K resistor.

## Flowchart

The code is simple and so is the flowchart. In setup( ) an output pin is set on the Arduino UNO and the serial monitor is initialized at 9600 baud. In loop( ) the relay pin is set low to turn it off, a delay of 2 seconds occurs, the relay is turned on, and a delay of 500ms occurs.




## The Code

The sketch for this in 5VoltSingleRelay.cpp and is very simple. This sketch energized the relay coil for 5 seconds and turns off the coil for 2 seconds. During the coil on-time the NO contact closes and the LED illuminates.

```
/*
File: 5VDCRelay.cpp
This demo energizes the 5VDC relay coil or 500ms and de-energizes it
for 2 seconds. The NO contact is connected to an LED and energizes
when the NO contact is closed. The ON/OFF state of the coil is reported

via Serial.println(). The code was prototyped in the Arduino IDE 2.0
and completed in VSCode/PlatformIO.
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define RELAY_PIN 2    // signal output for relay
#define ON_DELAY 500    // 500ms for coil on-time
#define OFF_DELAY 2000  // 2sec for coil off time

void setup() {
  // set the relay pin for output
  pinMode(RELAY_PIN,OUTPUT);
  // set up serial monitor
  Serial.begin(9600);
}

void loop() {
  // turn off relay
  digitalWrite(RELAY_PIN,LOW);
  Serial.println("RELAY OFF");
  delay(OFF_DELAY);
  // turn on relay
  digitalWrite(RELAY_PIN,HIGH);
  Serial.println("RELAY ON");
  delay(ON_DELAY);
}
```

## Code Walk Through

- The relay output pin, which is the S input for the relay, is on pin 2 of the Arduino as indicated by #define RELAY_PIN 2. The on and off times for the relay coil are determined by #define ON_DELAY 500 and OFF_DELAY 2000. The pin mode for pin 2 is set to OUPUT in setup( ) as pinMode(RELAY_PIN,OUTPUT). The serial monitor is also initialized via Serial.begin(9600).
- In the loop( ) function, the relay is turned off by the call to digitalWrite(RELAY_PIN,LOW), a “RELAY OFF” message is printed and a 2 second delay occurs via delay(OFF_DELAY). The relay coil then energizes via digitalWrite(RELAY_PIN,HIGH) and stays on for 1/2 second by the call to delay(ON_DELAY). There is no code for the LED since this is directly controlled by the relay contacts.

## Going Further

It’s not a good idea to drive a relay coil directly from a microcontroller

output. Even with small coils there is back EMF when a coil is de-energized that feeds back to the microcontroller output. According to the specifications, the relay coil pulls 89.3mA and Arduino output pins are rated for 40mA max and down to 20mA for continuous output, so this makes it a bad idea for a long-term, stable relay application. We can verify the coil current by using Ohm's law. We know that power is equal to voltage times current. To verify the coil current the specs say the relay's power consumption is about 0.45W or 450mW. So we can solve for I using **I = P/V**, so I = .45W/5VDC = 0.09 or 90mA. This is higher than the rating for an Arduino output pin, so that should raise an alarm.

A better design is to energize relay coils via a transistor. A common 2N2222 transistor works just fine to energize low coil voltage relays. The figure below is a graphic of the SONGLE relay mounted on a board that accommodates a three-pin breadboard compatible along with three terminals for the relay contacts. This configuration comes in many of the sensor kits for Arduino.

The schematic for the transistor-driven relay circuit is given below. We use the 2N2222 transistor as a switch. When an output becomes high from D2 on the Arduino, current passes into the base B of the transistor causing current to flow through the transistor through the collector C and out the emitter E. This causes the load, which is the relay coil, to energize. The NO contact from the relay closes allowing current to flow through the LED, lighting it. The current passing through the LED also goes through a 1K resistor, limiting it so the LED doesn't burn out.

A 3K resistor was chosen since the relay coil in the Arduino relay project draws around .90mA. How do we know this? The spec sheet for the relay states that the coil power is .45W. So using Ohm's Law, power P = I * V, so I (current) = P/V. This yields .45W/5VDC = 90mA, as we've already seen. The transistor we use must be able to handle this amount of current. So now, we need to determine what current we need to drive the base B of the transistor to turn it on.

We need to size the resistor for the base B to allow 90mv to pass through it. The Beta value for a transistor is also called hfe.This is the formula we use:

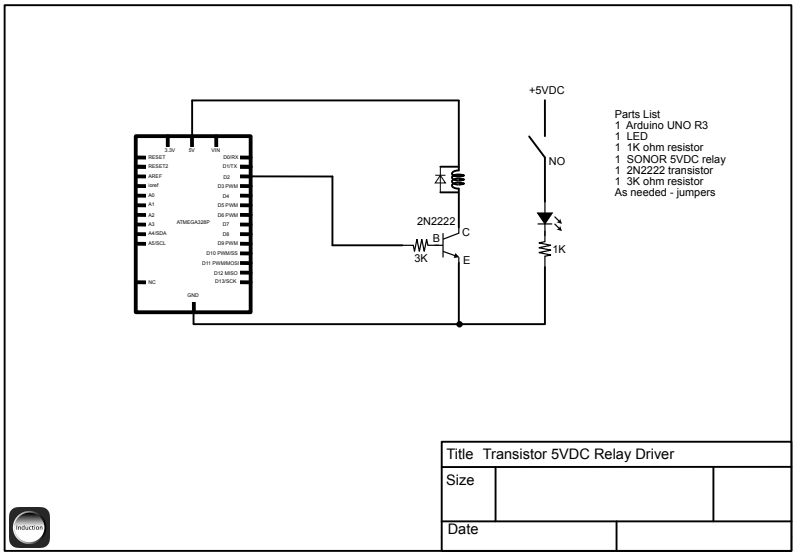
R = ((V - 0.6) * hfe) / I, where V is the control voltage that enters the base and I is the coil current. hfe is the Beta factor for the 2N2222 transistor and for our case it's 50.

So for our circuit we have R = ((5.0 - 0.6) * 50) /0.09 = 2,444 ohms, so we can use a 2-3K ohm resistor. I chose a 3K ohm. Why? To make sure the transistor fires correctly, just bump up the current value slightly. I chose 1.5ma.

The diode across the coil blocks excess discharge from the coil to back feed into the transistor. This is a very common design when using relays. The diode is called (you guessed it), a blocking diode. Sometimes you will see a capacitor in series with the collector and the coil as additional electrical spike protection.

## The Circuit

Here's the schematic of the new circuit.




At this point in the book you should have no problem breadboarding this circuit yourself. You can use a separate 5VDC supply or other voltage across the NO contact, but it doesn't have to be for 5VDC. The point of a relay is to switch a higher voltage and current through the contacts, so using a separate power supply is required for that. Just make sure the grounds are connected for the external supply and the Arduino.

If you are new to transistors, build and play with the circuits in the Transistors section before going any further.

## The Code

Here's the code for the transistor/relay project.

```
// File: RelayBoardTransistorDemo.cpp
//#define ARDUINO_IED
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define DEBUG             // comment out to block serial output
#define RELAY_PIN 2       // output pin to drive relay coil
#define ON_DELAY 500      // change these to change coil on/off time

#define OFF_DELAY 10000

// function prototype to drive 2N2222 base
void relayOnOff(int on_time, int off_time, bool state);

void setup() {
  // set the relay pin for output
  pinMode(RELAY_PIN,OUTPUT);
  pinMode(LED_BUILTIN,OUTPUT); // use builtin LED for debug
  // set up serial monitor
  Serial.begin(9600);
}

void loop() {
  relayOnOff(ON_DELAY,OFF_DELAY,false);
  relayOnOff(ON_DELAY,OFF_DELAY,true);
}

// function to control relay coil
void relayOnOff(int on_time, int off_time, bool state) {
  if (state == true) { // turn on coil
    digitalWrite(RELAY_PIN,HIGH);
    delay(on_time);
    #ifdef DEBUG
    digitalWrite(LED_BUILTIN,HIGH);
    Serial.print(" on delay time is ");
    Serial.println(on_time);
    #endif
  } else {  // turn off coil
      digitalWrite(RELAY_PIN,LOW);
      delay(off_time);
      #ifdef DEBUG
      digitalWrite(LED_BUILTIN,LOW);
      Serial.print(" off delay time is ");
      Serial.println(off_time);
      #endif
  }
}
```

## Code Walkthrough

- This code is more sophisticated than the previous version. This code was developed using PlatformIO. In this code a #define DEBUG is added to facilitate debugging throughout the code. The relay output is on pin 2 of the Arduino Uno and the on-delay and off-delay times are defined. A function prototype,

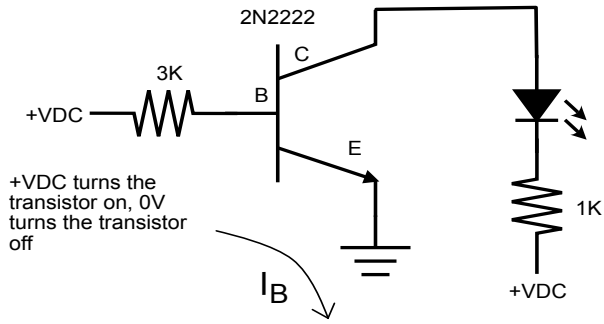
relayOnOff( ) is given with two int parameters and one bool. on-time, off-time are declared as ints and will hold the on time and off time for the coils. The bool variable state is passed as true or false and is used to prioritize switching the coil on first or off first.

- In setup( ) the pin modes are set to OUTPUT and the serial monitor is set up. The serial monitor and the built in LED are used for debugging.
- The contents of loop( ) is simply two successive calls to relayOnOff( ) with the time delay #defines and false as the first state. Within relayOnOff( ) the first call with state = false the code will branch to the else statement and turn off the relay coil via digitalWrite(RELAY_PIN,LOW) and then delay for the off_time. If #define DEBUG is not commented out in the top of the sketch, then the "on delay time" message is printed with the delay time and the built in LED is turn off.
- When the function ends control is returned to loop( ) and relayOnOff( ) is called again, only with state = true. This executes the code under the if(state == true) block and turns on the relay coil via digitalWrite(RELAY_PIN, HIGH) then delays for the on_time.

Moving the coil control code, delay times and the debugging statements to a function makes for more readable, better engineered and easy to maintain code.

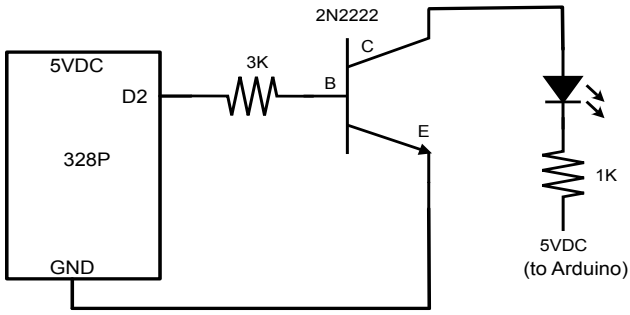
## Transistors

In this application we use a common 2N2222 NPN transistor. Here's a simple circuit you can build and play with to help you understand how transistors can act as switches.



It's a partial circuit from our Arduino relay project, only without the relay. the LED and 1K resistor are configured a little differently. When voltage is applied at +VDC (on the left) it passes through the 3K resistor into E, the emitter, and the transistor is turned on. The current flow direction is indicated by the curved arrow below the base B. This causes the LED to light up. Breadboard this and use a jumper to turn supply current to the emitter E by touching the jumper to 5VDC. The LED will light. Play around with different resistor values and maybe take some current and voltage measurements around the circuit. This is the best way to learn, by doing and experimenting.

Now let's add an Arduino output to drive the transistor and turn on the LED.




Here we connect pin 2 of the Arduino to the 3K resistor that is the input to the 2N2222 base B. The end of the 1K resistor is connected to the 5VDC output of the Arduino. At this point we can write the sketch to drive the emitter to turn on the LED. Again, I don't like driving coils from raw Arduino outputs. If somethings bad happens, I would rather fry a transistor that costs pennies than ruining an Arduino board.

We've just briefly touched on the utility of transistors and have used it merely as a switch. Transistors are also used to amplify signals in millions of circuits. Transistors are as fundamental as resistors and capacitors in my opinion and it's well worth the time to get to know them and work with them.

## KY-001 Temperature Module

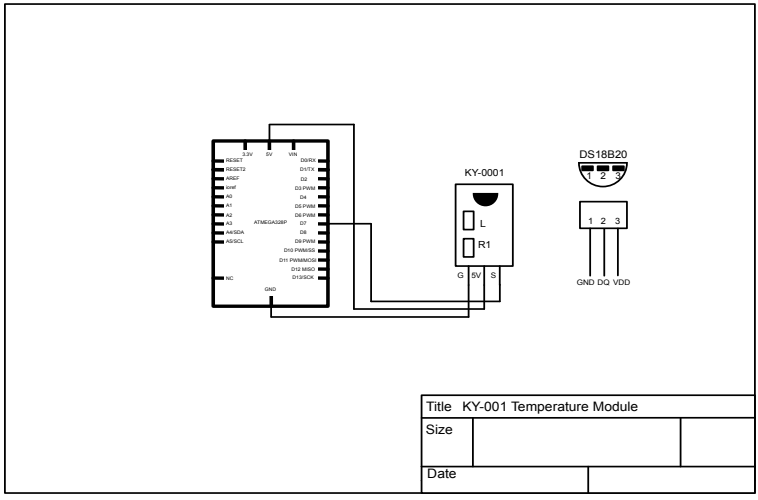
The KY-001 Temperature module reads temperature in default Celsius units. It's compact and easy to use when utilizing the OneWire communication library and DallasTemperature library.

## Requirements and Comments

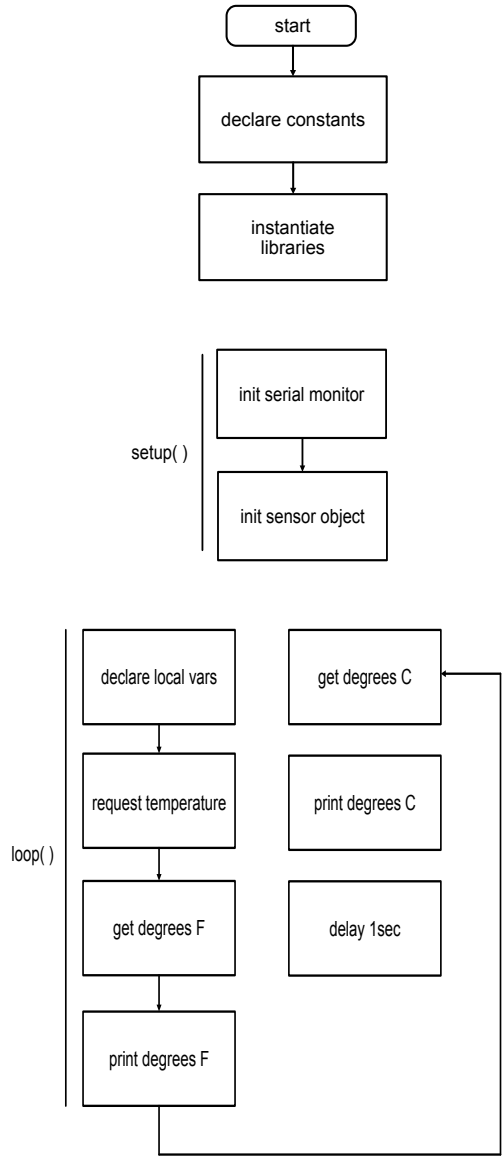
Read and display the current temperature in degrees F and C.

## The Circuit

The circuit consists of an KY-001 digital thermometer with the DS18B20 digital thermometer. The KY-002 contains an LED that flashes when communicating and a 4.7K pull up resistor connected internally to the DQ or S line.



## Flowchart


start
declare constants
instantiate libraries
setup( )
init serial monitor
init sensor object
loop( )
declare local vars
request temperature
get degrees F
print degrees F
get degrees C
print degrees C
delay 1sec

## The Code

The code is short and utilizes the OneWire library and protocol along with the DallasTemperature library.

```
/*
Project: KY-001TempModule
Demo code for the KY-001 temperature module that
uses the OneWire protocol and DallasSemiconductor method calls.

*/

//#define IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif
#include <OneWire.h>
#include <DallasTemperature.h>

const int ONE_WIRE_BUS = 7;
const int TIME_DELAY = 1000;

OneWire oneWireTSensor(ONE_WIRE_BUS);
DallasTemperature tSensor(&oneWireTSensor);

void setup() {
  Serial.begin(9600);
  tSensor.begin();
  while(!Serial)
    ;
}

void loop() {
  float temp_F = 0.0;
  float temp_C = 0.0;
  tSensor.requestTemperatures();
  temp_F = tSensor.getTempFByIndex(0);
  temp_C = tSensor.getTempCByIndex(0);
  Serial.print("Current Temp F: ");
  Serial.println(temp_F);
  Serial.print("Current Temp C: ");
  Serial.println(temp_C);
  delay(TIME_DELAY);
}
```

## Code Walk Through

- First the #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is un-commented.

- The OneWire and the DallasTemperature header files are include. These libraries are loaded into the Arduino IDE and PlatformIO in the usual manner.
- A constant int is defined, ONE_WIRE_BUS, for the input pin and the one wire bus, which is pin 7 on the Arduino UNO.
- A one wire object is declared of the OneWire class as oneWireTSensor with the ONE_WIRE_BUS as a parameter.
- A time delay is declared.
- A sensor object is declared of the DallasTemperature class as tSensor with the address of oneWireTSensor object (&oneWireTSensor) as a parameter.
- In setup( ) the serial monitor is initialized at 9600 baud along with the tSensor object. The code waits for serial communications to become available.
- In loop( ) two float variables, tempF and tempC, are declared for readings in degrees F and C.
- The method, tSensor.requestTemperatures( ) is called to request temperatures from the DS18B20 digital thermometer.
- The next two lines get the current degrees F and C and stores them in floats temp_F and temp_C.
- The next few lines print the F and C temperature readings.
- A one second time is used to not make the display continuously flash by.

## Going Further

At the heart of the KY-001 temperature module is based on the Dallas Semiconductor DS18B20 digital thermometer that utilizes the OneWire protocol. The One Wire protocol is a half-duplex, single wire serial data, low power protocol. It requires actually two wires: one for data and one for ground. One Wire is a master/slave protocol design, but unlike I2C it can only have one master, but up to 100 slaves per master. One Wire is a low speed protocol, with the data speed being typically 15.4kbps. One wire protocol is used in temperature sensors, EEPROMs, timers, clocks and other devices.

The OneWire library supports the methods used to work with the protocol. Below is a OneWire quick reference.

- **OneWire myWire(pin)** - creates a OneWire object with a PIN number as a parameter. In our case it's an Arduino UNO pin. Multiple myWire objects can be create for individual pins.
- **search(addrArray)** - searches for the next device. addArray is

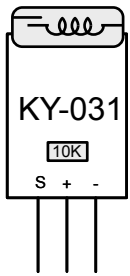
an 8 bit byte array and if an address is found, the address is filled in the array and true is returned. If no devices are found, false is returned.

- **reset_search( )** - starts a new search. The next search will start at the first device.
- **reset( )** - resets the OneWire bus. This is used before any attempt is made at communicating with devices.
- **select(addrArray)** - selects a device based on its address. This call is required after a reset and will persist the connection until the next reset.
- **skip( )** - used to immediately access a device and is only used when there is a single device.
- **write(number)** - writes a byte.
- **write(number, 1)** -writes a byte and leaves power supplied to the bus.
- **read( )** - reads a byte.
- **crc8(dataArray,length)** - performs a CRC check on a data array.

The OneWire methods are easy to understand and use, which mask the complexity of the software under the hood that makes it happen.

## KY-031 Percussion Knock Sensor

The KY-031 percussion knock sensor is also known as a tap module. It's a sensor that reacts to vibration, particularly rapid or abrupt vibration. The KY-031 sensor element is a spring held in place at one end and open at the other. IF the sensor receives enough mechanical energy, the spring will vibrate and the untethered side of the spring will touch a metal contact which is present on the S pin. There is no sensitivity adjustment. From the raw input from S to an Arduino UNO, duration of the knock can be measured and knocks may be counted and used as a code to operate or actuate another device. The KY-031 knock sensor consists of a spring, fixed on one end and open on the other as shown below.

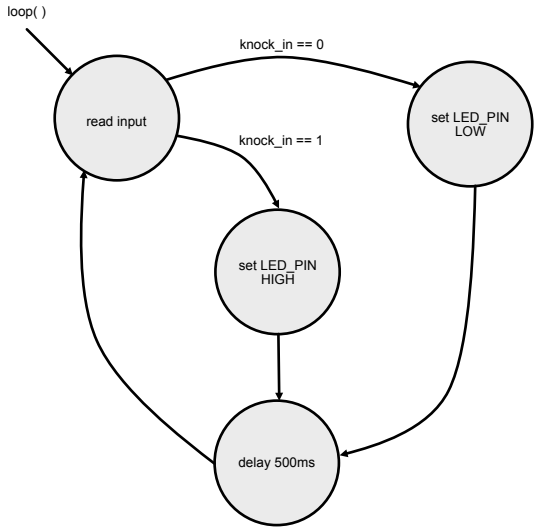



When the sensor is tapped or attached to a surface and the surface tapped with sufficient force, the spring will make momentary contact and send voltage out of pin S. The 10K resistor pulls the open input to ground, and S goes HIGH when contact is made.

## Requirements and Comments

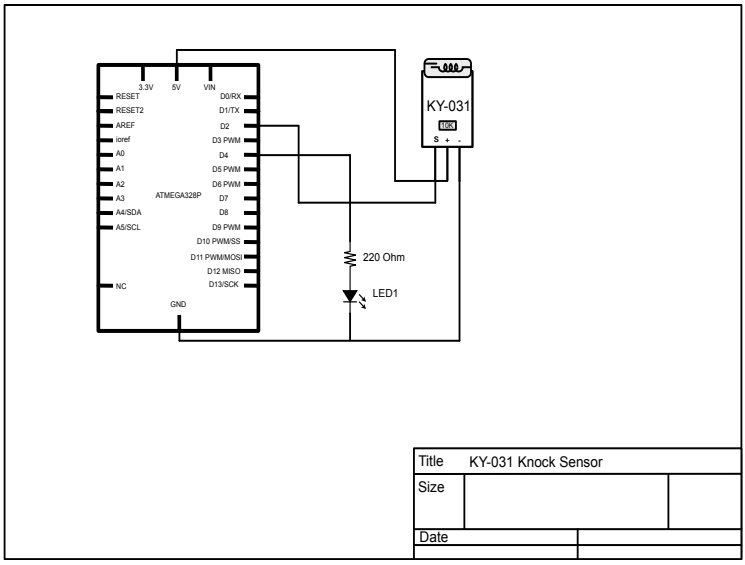
The requirements are simple: detect a knock and flash an LED when the knock is detected.

## State Chart




## The Circuit

The circuit is simple. The knock sensor input connected to pin 2 of the Arduino the LED output is connected to pin 4 of the Arduino. The KY-031 knock sensor should be connected with DuPont cables, and not directly placed on a breadboard.




## Specifications

The KY-031 module contains 3 pins. We will call the leftmost pin 1, the middle pin 2, and the rightmost pin 3. Pin 1 is the S or signal output, the middle pin is 5VDC or 3.3VDC, and the rightmost pin is ground.

## The Code

```
/*
Project: K-031KnockSensor
Demos the K-031 knock sensor (tap sensor)
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define DELAY_TIME 500
const int KNOCK_PIN = 2;
const int LED_PIN = 4;

void setup() {
  pinMode(KNOCK_PIN,INPUT);
  pinMode(LED_PIN,OUTPUT);
}

void loop() {
  int knock_in;
  // get knock sensor input - input is normally LOW
  knock_in =  digitalRead(KNOCK_PIN);
   if(knock_in == 0) {
      digitalWrite(LED_PIN,LOW);
      } else {
         if(knock_in == 1) {
          digitalWrite(LED_PIN,HIGH);
          delay(DELAY_TIME);
        }
      }
}
```

## Code Walk Through

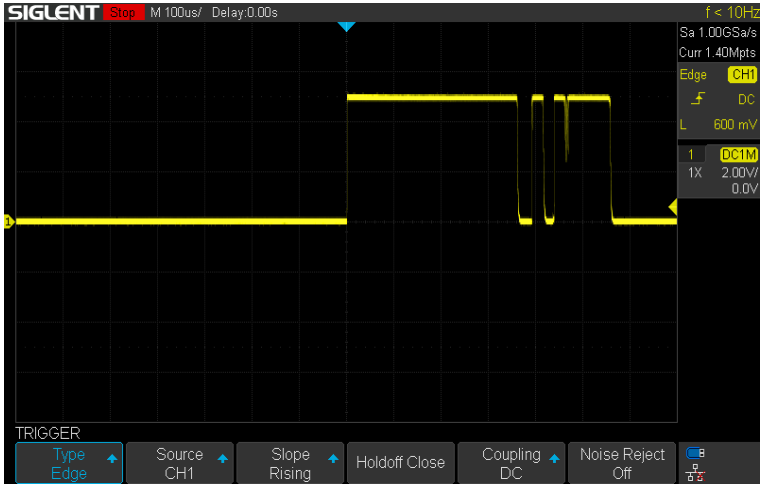
The code is simple and short.

- First the #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is uncommented.
- A delay time is defined for a 500ms delay to keep the LED on if a knock is detected.
- const ints are declared for KNOCK_PIN on pin 2 of the Arduino UNO and LED_PIN on pin 4 of the Arduino UNO.

- In setup( ) the pin mode for the KNOCK_PIN is set as INPUT and the pin mode for the LED_PIN is set as OUTPUT.
- In setup( ) the pin mode for the KNOCK_PIN is set as INPUT and the pin mode for the LED_PIN is set as OUTPUT.
- In loop( ) an integer variable knock_in is declared for the knock sensor input.
- The input from the knock_sensor a read by the call to digitalRead(KNOCK_PIN). The input from the pin is LOW if the sensor is not active and HIGH when the sensor is “knocked”.
- If knock_pin is LOW then the LED is turned off, else if knock_pin is HIGH the LED is illuminated. A 500ms delay is executed to keep the LED visible for a short duration.

## Going Further

One would expect some bounce on the input, since the sensor is a vibrating spring. Just as suspected, there is a lot of bounce under certain conditions. In the scope trace below the knock sensor was held about five inches from the surface of a workbench and dropped onto a static mat, which is essentially rubber. The knock sensor was connected with three Dupont cables, enabling it to have a reasonable degree of freedom.

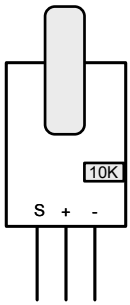



There’s a considerable amount of bounce with the spring making and breaking contact from this very gentle drop. If the contact with the sensors is more forceful, less bounce was experienced as shown in the trace below.


This is a reasonably clean signal that can be counted. Also, if a HIGH input is desired, all that needs to be done is to reverse polarity of the inputs.

## KY-002 Shock Module

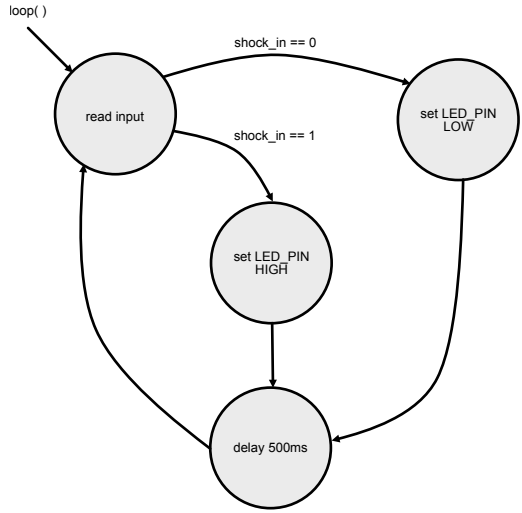
The KY-002 shock sensor is sometimes referred to a “shake switch”. At the heart of the switch is a Gaoxin SQ-18010P vibration switch which consists of a metal wire, labeled Terminal A, in the center of a coiled wire, labeled Terminal B. When the switch is shaken, the spring touches the metal wire making contact. The shock module is very well suited as a “wake up” input.



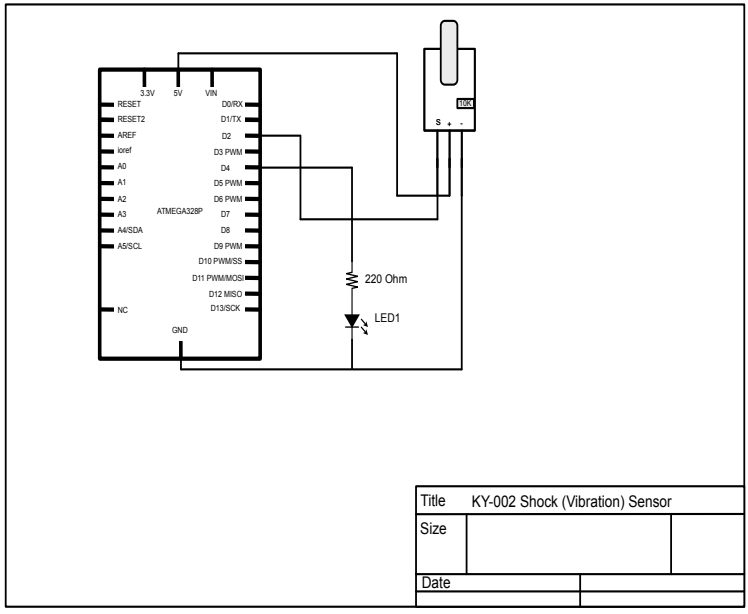

## Requirements and Comments

The requirements are simple: detect a vibration and flash an LED when the vibration is detected.

## State Chart




## The Circuit



## Specifications

The KY-002 module contains 3 pins. We will call the leftmost pin 1, the middle pin 2, and the rightmost pin 3. Pin 1 is the S or signal output, the middle pin is 5VDC or 3.3VDC, and the rightmost pin is ground.

## The Code

The code is based on that for the KY-002 shock sensor, since both have similar pinout and functionality.

```
/*
Project: K-002ShockSensor
Demos the K-002 shock sensor (vibration sensor).
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define DELAY_TIME 500
const int SHOCK_PIN = 2;
const int LED_PIN = 4;

void setup() {
  pinMode(SHOCK_PIN,INPUT);
  pinMode(LED_PIN,OUTPUT);
}

void loop() {
  int shock_in;
  // get shock (vibration) sensor input - input is normally LOW

  shock_in =  digitalRead(SHOCK_PIN);
   if(shock_in == 0) {
      digitalWrite(LED_PIN,LOW);
      } else {
         if(shock_in == 1) {
          digitalWrite(LED_PIN,HIGH);
          delay(DELAY_TIME);
        }
      }
}
```

## Code Walk Through

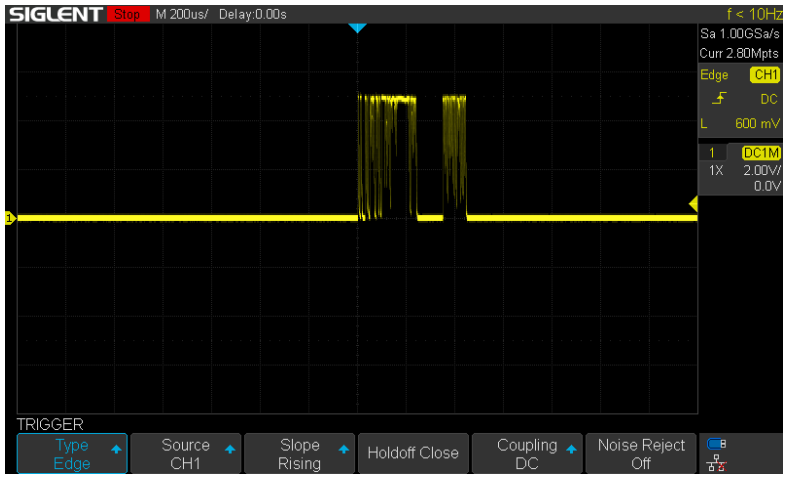
The code is simple and short and is based on the code for the KY-031 knock sensor.

- First the #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is uncommented.

- A delay time is defined for a 500ms delay to keep the LED on if movement or vibration is detected.
- const ints are declared for SHOCK_PIN on pin 2 of the Arduino UNO and LED_PIN on pin 4 of the Arduino UNO.
- In setup( ) the pin mode for the SHOCK_PIN is set as INPUT and the pin mode for the LED_PIN is set as OUTPUT.
- In loop( ) an integer variable shock_in is declared for the shock sensor input.
- The input from the shock_sensor a read by the call to digitalRead(SHOCK_PIN). The input from the pin is LOW if the sensor is not active and HIGH when the sensor is touched or vibrated.
- If shock_pin is LOW then the LED is turned off, else if shock_pin is HIGH the LED is illuminated. A 500ms delay is executed to keep the LED visible for a short duration.

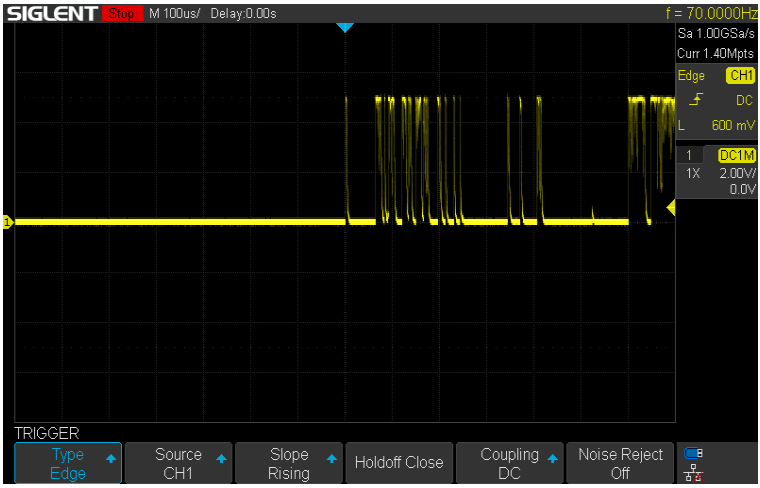
## Going Further

The KY-002 is very sensitive and is subject to bounce. A few scope traces are shown below. In the first trace, the sensor was gently tapped, indicating how sensitive the Gaoxin SQ-18010P vibration switch is. The horizontal scale is set to 200microseconds per division, and the vertical scale is 2 volts per division.

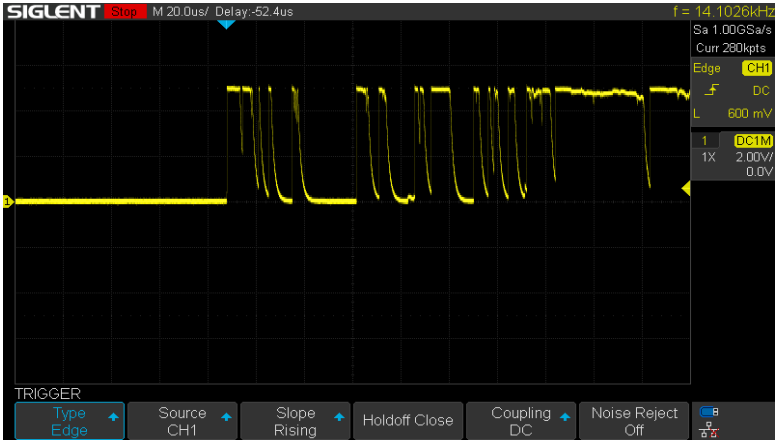



The sensor was tapped with more force as shown in the trace below. There is much more bounce. The horizontal scale is set to 100

microseconds per division and the vertical scale is 2 volts per division.



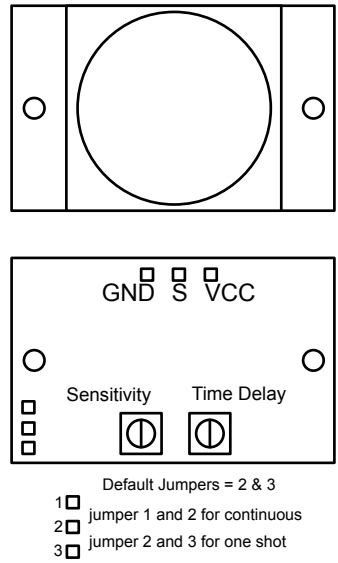

This last trace is the same trace as above, but the horizontal scale set to 20 microseconds per division. The input from the switch is very ragged and care must be exercised in managing the input, especially if the input is used as an interrupt source. Some form of debouncing is useful.




All in all, the KY-002 Shock Sensor is very useful as a wakeup device or for static object motion detection.

## HC-SR501 Motion Sensor

The HC-SR501 is a passive infrared sensor that detects motion. The facing surface of the sensor contains a hemispherical opaque Fresnel lens. Underneath the lens is the PIR sensor, which stands for Passive InfraRed. The PIR sensor responds to heat, such as that given off by humans or animals.




There are three pins on the back of the sensor, ground (GND), signal (S) and VCC (5VDC). Jumper pins on the left of the circuit board are used to output data as a pulse or one shot when pins 2 an 3 are jumpered (no-repeatable trigger), or continuous (repeatable trigger) when pins 1 and 2 are jumpers. There are two potentiometers. The pot on the left is for range sensitivity and the pot on the right sets the delay of the output. The range sensitivity can be set to 3 meters (fully counterclockwise), which is 9.8 feet up to 7 meters (fully clockwise), which is around 23 feet. The time delay pot is for the duration of the high signal output of the sensor. Fully counterclockwise causes the output to persist high for 3 seconds. Fully clockwise the output persists for 300 seconds, or five minutes.

## Specifications

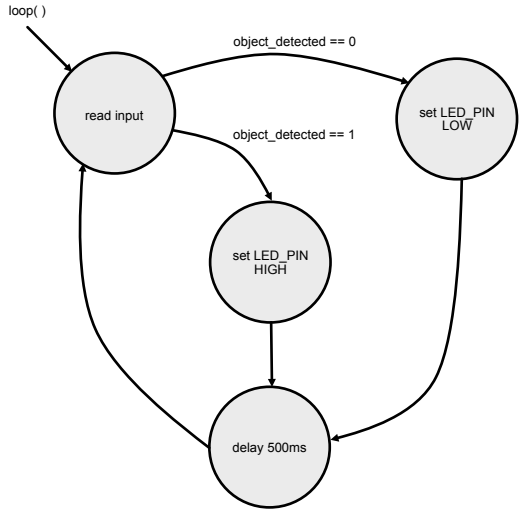
The HC-SR501 IR voltage range is 5-20VDC. Power consumption is 65mA. TTL output is 3.3-0V. Two trigger methods are used, L for non-repeatable trigger and H for repeatable trigger, which are jumper selectable. The sensing range is less than 120 degrees within 7 meters (approximately 23 feet). The HC-SR501 can operate between -15 and +70 degrees C (5 and 158 degrees F).

When the mode pins 2 and 3 are jumpered, placing the sensor in L or non-repeatable mode, the output goes high for the duration of the delay period set by the time delay potentiometer. If pins 1 and 2 are jumpered, the output of the sensor remains high until an object is detected, then goes low after the delay period determined by the time delay potentiometer.

## Requirements and Comments

The object of this sketch is to detect an object in no-repeatable trigger mode and energize an active buzzer when an object that radiates IR energy.

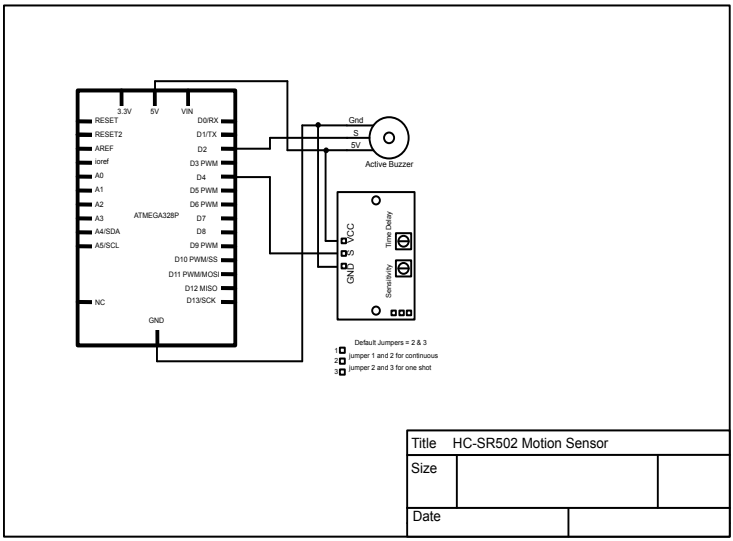
## State Chart




## The Circuit

The circuit is comprised of a HC-SR501 sensor and an active buzzer. Pin 2 outputs a high to activate the buzzer. The signal output S from the

sensor is connected to input pin 4 on the Arduino UNO.



## The Code

```
/*
Project: HC-SR501InfraredSensor
This code demonstrates the SR501 IR sensor.
When an objec is detected the active buzzer chirps
for one second.
The active buxzzer output is on pin 2 of the Arduino UNO.

The IR sensor input is on pin 4 of the Arduino UNO.
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

const int IR_INPUT_PIN   = 4;
const int BUZZER_OUT_PIN = 2;
#define TIME_DELAY 25

void setup() {
  pinMode(IR_INPUT_PIN,INPUT);
  pinMode(BUZZER_OUT_PIN,OUTPUT);
}

void loop() {
 int object_detect = digitalRead(IR_INPUT_PIN);
 if(object_detect == 1) {
  digitalWrite(BUZZER_OUT_PIN,HIGH);
  delay(TIME_DELAY);
  digitalWrite(BUZZER_OUT_PIN,LOW);
 }
}
```

## Code Walk Through

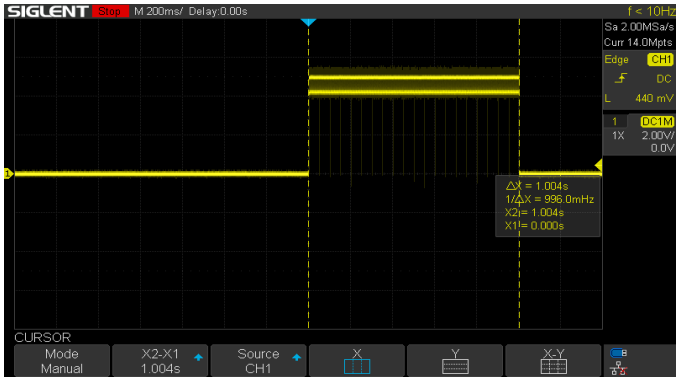
- First the #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is un-commented.
- const ints are used to define Arduino interface pins. The IR input is on pin 4 and the buzzer output is on pin 2.
- A time delay is used to toggle the buzzer output.
- In setup( ) the pin modes are set to input for the IR sensor and output for the buzzer.
- In loop( ) an int is declared, object_collect, to store the value of the IR output. If an object is detected, the input is high and the buzzer is toggled via the digitalWrite( ) calls.

A few things to watch for using this device is the startup time. The sensor can take 30 to 60 seconds to become acclimated to the IR energy where it's located. There is also a 5 to 6 second delay where the sensor resets after making a reading. During this period no object or motion will be

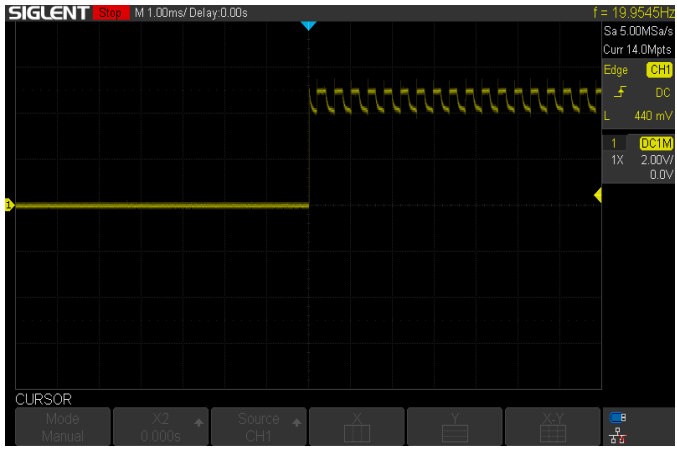
detected. Also, the sensor should not be located near light or other sources of optical interference

## Going Further

Here are two oscilloscope traces showing the output of the HCSR501 sensor. The first trace shows a 1 second pulse. Notice the high frequency noise when the signal goes high. The pulse time duration is 1.0004 seconds with the horizontal axis set to 200ms per division. The vertical axis is set to 2 volts per division.



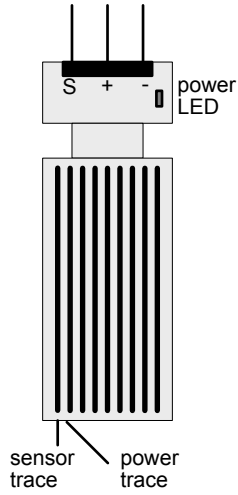

Here's a trace of the output with the horizontal axis extended to 1ms per division. The transition from 0 to 5 volts is clean.



The periodic noise that rides with the signal is benign and is not detrimental.

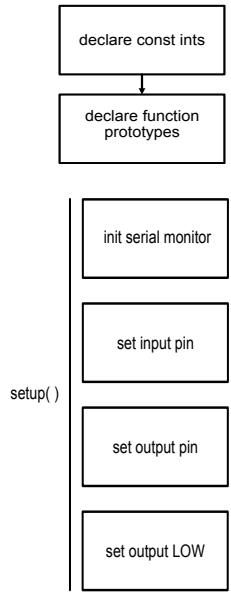
## K-0135 Water Level Sensor

The K-0135 water level sensor is a simple device and acts like a potentiometer. As the water level depth increases across the sensor, the resistance drops. We can adapt the code used for the YL-69 resistive moisture sensor and the HW-390 capacitive soil sensor.

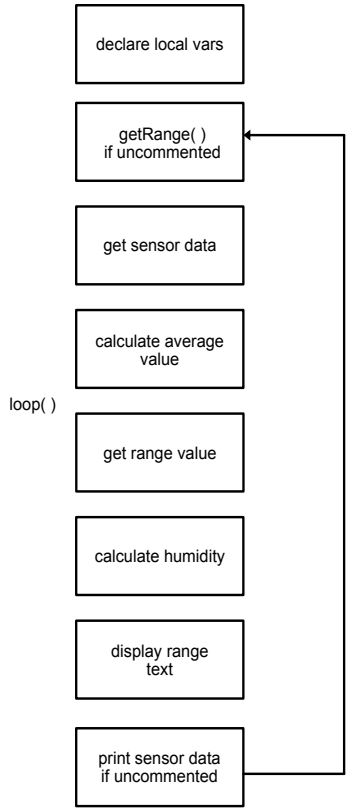



There are 9 copper traces on the sensor. The traces alternate between sensor traces and power traces. In dry conditions there is no connection between the power and sensor traces, but as soon as the sensor is exposed to water, conduction occurs, creating resistance. As the sensor is more immersed in water, the resistance increases. This is a simple yet effective sensor.

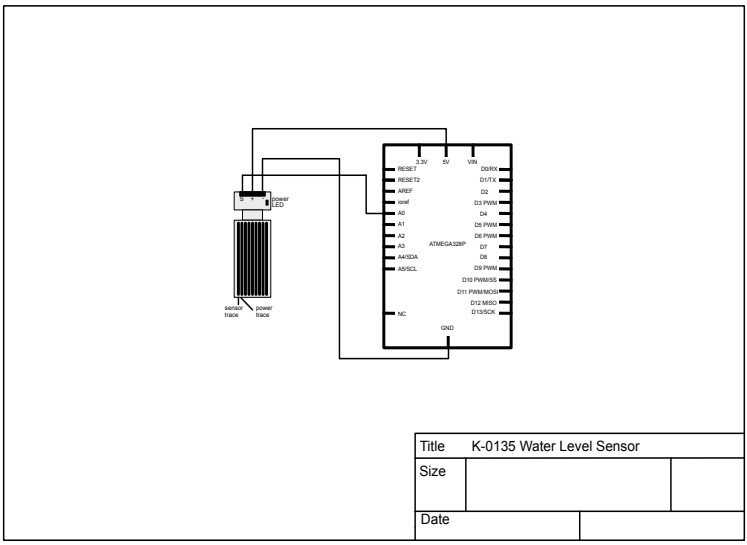
## Flowchart




Continued on next page…


declare local vars
getRange( )
if uncommented
get sensor data
calculate average
value
loop( )
get range value
calculate humidity
display range
text
print sensor data
if uncommented

## The Circuit




The circuit is simple with 5VDC connected to the + terminal, ground connected to the - terminal and S connected to A0 on the Arduino UNO. The power to the sensor can range from 3.3 to 5VDC. Like the YL-69 resistive moisture sensor, the K-0135 has a relatively short life span when exposed to moisture, even worse when the sensor is continuously energized. One way to increase the life span of the sensor is to energize it only when needed or periodically, with a reasonable duration between sensor readings. We will accomplish this in the code.

## The Code

```
/*
Project: K-0125WaterLevelSensor
Water level sensor demo using the
K-0135 water level sensor.
*/

//#define ARDUNIO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif
// defines for getting initial data
// uncomment get range and display
// average reading data
//#define GET_RANGE
//#define SHOW_DATA
#define DELAY_TIME 1000
#define SENSOR_DELAY 10
const int A_IN_PIN = A0;
const int POWER_SENSOR = 2;
const int ARRAY_SIZE = 5;
// high and low sensor range
const int LOW_VAL = 0;
const int HIGH_VAL = 172;
const int PERCENT_VAL_TOP = 100;
const int PERCENT_VAL_BOTTOM = 0;
// range for sensor output partitions
// change these for particular sensor
const int DRY_RANGE = 2;
const int WET_RANGE = 86;

// function prototype
void getSensorData(int[]);
int average(int[]);
int partitionValues(int);
#ifdef GET_RANGE
int getRange(void);
#endif
void calcHumidity(int);

void setup() {
  Serial.begin(9600);
  pinMode(A_IN_PIN,INPUT);
  // pin 2 powers the senssr on demand
  pinMode(POWER_SENSOR,OUTPUT);
  digitalWrite(POWER_SENSOR,LOW);
}
```

```
void loop() {
  int dat_array[ARRAY_SIZE];
  int average_val = 0;
  int range_val = 0;

  #ifdef GET_RANGE
    getRange();
  #endif
  getSensorData(dat_array);
  average_val = average(dat_array);
  range_val = partitionValues(average_val);
  calcHumidity(average_val);

  // display moisture status
  switch(range_val) {
    case 0:
      Serial.println("Dry");
      break;

    case 1:
      Serial.println("Wet - low water level");
      break;

    case 2:
      Serial.println("Wet - high water level");
      break;

    default:
      break;
  }

  #ifdef SHOW_DATA
  Serial.print("Average moisture value is ");
  Serial.println(average_val);
  Serial.print("range_val is ");
  Serial.println(range_val);
  #endif
}
```

```
/ getSensorData - gets the sensor data
void getSensorData(int val_array[]) {
  digitalWrite(POWER_SENSOR,HIGH);
  delay(SENSOR_DELAY);
  for( int i = 0; i < ARRAY_SIZE; i++) {
    val_array[i] = analogRead(A_IN_PIN);
    delay(DELAY_TIME);
  }
  digitalWrite(POWER_SENSOR,LOW);
}

// average - averages and returns the array values
int average(int val_array[]) {
  int ave = 0;
  for(int i = 0; i<ARRAY_SIZE; i++) {
    ave+=val_array[i];
  }
  ave = ave/ARRAY_SIZE;
  return ave;
}

int partitionValues(int val) {
  if(val <= DRY_RANGE) // dry
    return 0;
  if(val > DRY_RANGE && val < WET_RANGE) // moist
    return 1;
  if(val > WET_RANGE) // wet/moist
    return 2;
  // return -1 if bad param
  return -1;
}

// calcHumidity - calculates and displays percent humidity
// using the map function
void calcHumidity(int ave_val) {
  int percentHumidity = map(ave_val,LOW_VAL,HIGH_VAL,
      PERCENT_VAL_BOTTOM, PERCENT_VAL_TOP);
  percentHumidity = abs(percentHumidity);
  Serial.print("Wetness is ");
  Serial.print(percentHumidity);
  Serial.println("%");
}

// getRange - get range of sensor
// This is a manual process that allows
// a user to get the min and max values
// of a moisture sensor by dipping the
// sensor in water to get a full wet reading
// and getting a value when the sensor
// is in dry air
#ifdef GET_RANGE
int getRange(void) {
  int result = 0;
  digitalWrite(POWER_SENSOR,HIGH);
  result = analogRead(A_IN_PIN);
  Serial.print("sensor reads ");
  Serial.println(result);
}
#endif
```

## Code Walk Through

The water level sensor code is based on the code used for the YL-69 resistive moisture sensor and the HW-390 capacitive moisture sensor.

- First the #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is un-commented.
- Two #defines are commented out, GET_RANGE and SHOW_DATA. These #defines control the function getRange(void) and a block of code that prints sensor data.
- There are two delays defined, DELAY_TIME and SENSOR_DELAY. DELAY_TIME is used for a delay between readings and SENSOR_DELAY is a 10ms delay after the output pin 2, which is used to power the sensor, goes HIGH.
- The output of the sensor goes to analog pin A0, which is declared as the const int A_IN_PIN.
- The Arduino UNO supplies power to the sensor vial pin 2, which is defined as POWER_SENSOR.
- An array is used to store the sensor readings. The #define ARRAY_SIZE 5 is used when declaring the array of size 5.
- Two const ints, LOW_VAL and HIGH_VAL represent the range of the sensor. This is determined by uncommenting the function getRange( ) and recording the values when the sensor is completely dry and completely immersed in water. The sensor used for this demo ranged from 0 (dry) to 172 (wet).
- Two const ints, PERCENT_VAL_TOP and PERCENT_VAL_BOTTOM are declared to be used in a map( ) function to map the sensor readings to a percentage in the calcHumidity( ) function.
- Two const ints, DRY_RANGE and WET_RANGE are declared. These values are used in the partitionValues( ) function to return one of three values. These values are used to indicate dry, wet - low water level or wet - high water level.
- Several function prototypes are declared The first is void getSensorData(int [ ]) which takes an array of integers are a parameter. This function acquires the sensor data and fills a global integer array of size ARRAY_SIZE with the sensor values.
- The next function prototype is int average(int [ ]) which takes an array of integers as a parameter. The function takes the average of the five array readings produced by getSensorData(int [ ]).
- The next function, int partitionValues(int) takes a singular integer as a parameter. This function takes the average input and

uses that value to return one of three values. These values are used to determine the degree of moisture or wetness the sensor is exposed to.

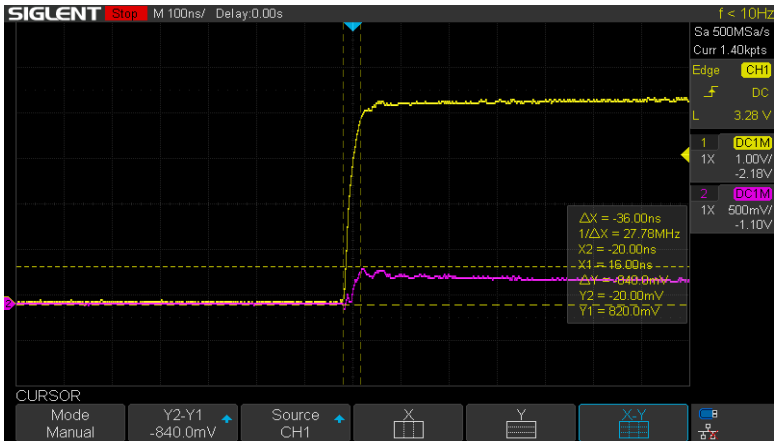
- If GET_RANGE is defined, meaning not commented out, the int getRange(void) prototype is declared. This function is used early on to determine a specific sensor's range and is commented out after the determination.
- The next function is void calcHumidity(int), which takes a single integer as a parameter. calcHumidity(int) calculates the percent humidity of the sensor value.
- In setup( ) the serial monitor is started at 9600 baud, the pinMode( ) method is called to set pin A0 as an input via pinMode(A_INPUT_PIN, INPUT).
- Pin 2 is set as an output via pinMode(POWER_SENSOR, OUTPUT) and then a call to digitalWrite(POWER_SENSOR,LOW) pulls pin 2 low.
- In loop( ) int dat_array[ARRY_SIZE] is declared. This holds a sequence of five sensor readings to be averaged.
- Two int variables, average_val and range_val hold the average value of the 5 readings and a range value that will be passed to the partitionValues( ) function.
- The getRange( ) function is called if GET_RANGE is defined. This function is used to find the maximum and minimum sensor range values.
- The function getSensorData(dat_array) is called using the five member dat_array[ ] as a parameter. Notice that the brackets are missing in the function argument. This is the way arrays are passed in a function. By omitting the brackets the address is passed and not the entire array, which is what we want.
- The next line of code calls the function average(dat_array) and returns the average value of the five readings in dat_array[ ].
- The function partitionValues(average_val) is called. This function returns one of three numbers, zero if the sensor is dry, 2 if the sensor moist and 2 if the sensor is wet.
- The function calcHumidity(average_val) is called. This takes the average value of the sensor readings and calculates and prints the percentage humidity or moisture percentage.
- The switch statement takes the range_val and prints to the serial monitor "Dry", "Wet - low water level" or "Wet -high water level" depending on the range_val value.

- If the #define SHOW_DATA is uncommented, the variables are printed to the serial monitor. This is useful when working with a new or replacement sensor.
- The function getSensorData(int val_array[ ]) turns on the sensor via digitalWrite(POWER_SENSOR,HIGH), delays for SENSOR_DELAY time which is 10ms, uses a for loop to read the sensor values via analogRead(A_IN_PIN) then turns the power off to the sensor via digitalWrite(POWER_SENSOR,LOW). Resistive sensors tend to corrode easily so selectively supplying power to a sensor of this type makes sense, but you need to be ever mindful of current draw as not to overload or damage the Arduino output pin. The val_array[ ] is just a placeholder name in the function parameter declaration- this is a different name than dat_array[ ] but it is in fact dat_array[ ]. You can use any name with a variable is passed into a function.
- The function average(int val_array[ ]) sums the values in the array and then divides by the array size, yielding the mean or average. This value is returned.
- The function partitionValues(int val) partitions the values int ranges which is used by the switch statement in loop( ).
- The function calcHumidity( ) calculates the humidity or moisture percentage of the sensor reading. It uses the map( ) function to map the sensor range to zero or one, making it easy to calculate the percentage. The absolute value of the percentHumidity variable is taken to endure that the value is positive. The percentHumidity is then printed to the serial monitor.
- The getRange(void) function is used to get the range of the sensor, from completely dry to completely wet. These values are recorded and placed into the LOW_VAL and HIGH_VAL #defines. This is required to calibrate the code to a particular sensor.

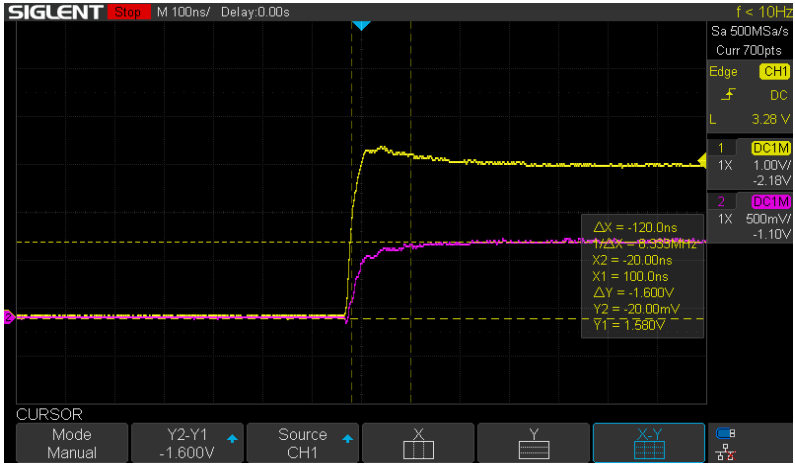
## Going Further

Here are two scope traces at near opposite ends of the range values. The trace below was taken at 4% water level where the sensor was placed in a damp paper towel, The yellow trace is the output of pin 2 which powers the sensor. The purple trace is the output of the sensor. The Y axis yellow trace is 1 volt per division. The Y axis purple trace is

at 500mv per division. As you can see, the response time for this sensor is very fast, with the delta X measurement at 36 nanoseconds. The voltage from the sensor output is 840mv. This is a high resistance reading with little current draw.




The trace below is at 98% water level. The amplitude of the output has increased to 1.6 volts since the resistance decreased. The response time has also decreased from 36 nanoseconds to 120 nanoseconds. The amplitude of pin 2 of the Arduino UNO has decreased, from around 4.5 volts to 1.6 volts, which makes sense since the resistance is lower. If the resistance is lower and the voltage is lower, the current draw will be larger, which is a risk directly powering a sensor from an Arduino output pin. Driving the sensor with a small transistor, such as a NP2222 may be a good design choice.




The water level sensor is simple and highly responsive, but like resistive moisture sensors, they are prone to corrosion and consequently, failure over time.

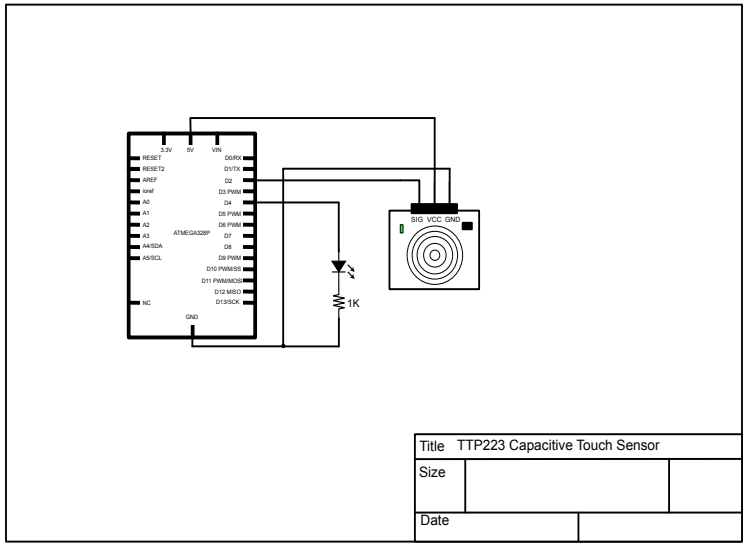
## TTP 223 Capacitive Touch Sensor

The TTP233 capacitive touch sensor is a reliable an effective sensor that can be used in place of discrete switches or pushbuttons. The module debounces the touches producing solid and responsive signals. The module consists of a touchpad, a few discrete components and a TTP223 touch pad detector IC. Capacitive sensors emit a small electric field on the sensing surface. An object that disrupts this field can be easily detected, such as human finger. The electric field reacts to the conductivity of the human body which provides conductivity and changes the electric field when coming into contact with a finger.

Our circuit comprises a capacitive touch sensor, an LED and a 1K resistor. When the capacitive sensor is touched, the LED illuminates.

## The Circuit

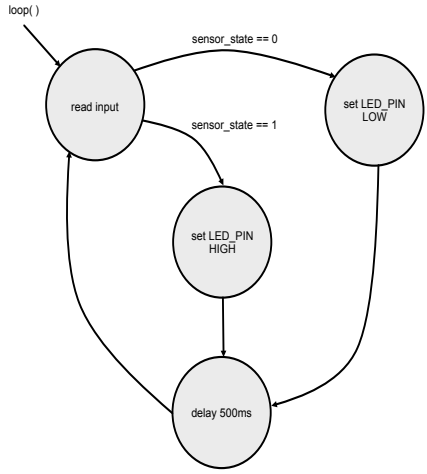
The output of the capacitive touch sensor, SIG, is connected to the Arduino UNO pin 2, configured as an input. VCC is connected to 5V and GND is connected to GND on the Arduino UNO. Pin D4 on the Arduino UNO is configured as an output is and connected to the anode of the LED.




**Specifications**

The operating voltage of the capacitive touch sensor ranges fro 2.0 to 5.5VDC. The operating current at 3V is 1.5uA to 3.0uA. The response time ranges fro 60 to 220ms.

State Chart



## The Code

```
/*
Project: CapacitiveTouchSensor
Demos the TTP223 capacitive touch sensor
*/
// #define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define DELAY_TIME 100
const int SENSOR_IN = 2;
const int LED_OUT   = 4;

void setup() {
  pinMode(SENSOR_IN,INPUT);
  pinMode(LED_OUT,OUTPUT);
}

void loop() {
  int sensor_state = 0;
  // read the senor and light the LED if
  // sensor is touched.
  sensor_state = digitalRead(SENSOR_IN);
    if(sensor_state == 1) {
    digitalWrite(LED_OUT,HIGH);
    delay(DELAY_TIME);
  } else {
    digitalWrite(LED_OUT,LOW);
  }
}
```

## Code Walk Through

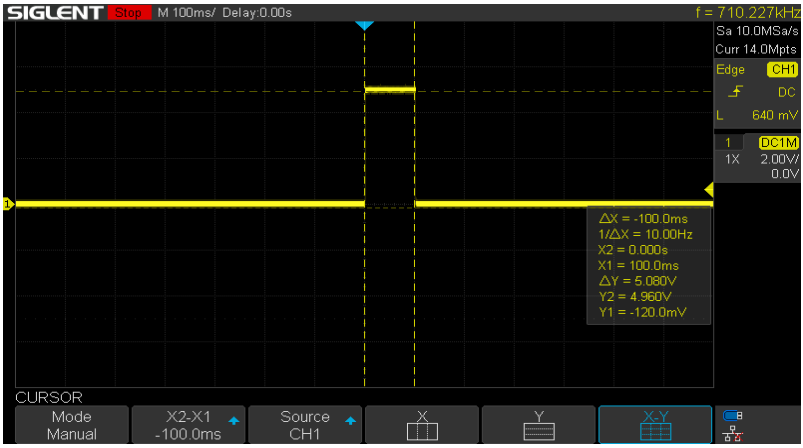
The code is short and simple.

- First the #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is un-commented.
- A 100 millisecond time delay is declared via #define DELAY_TIME 100. This is used as a delay between sensor readings..
- Two const int variables are declared, const int SENSOR_IN for sensor input on pin 2 and LED_OUT for LED output on pin 4.
- In setup ( ) the SENSOR_IN as declared as an input and LED_OUT as an output via the called to pinMode( ).
- In loop( ) int sensor_state = 0 is declared. The variable sensor_state contains the result of the call to digitalRead(SENSOR_IN). The output of the capacitive sensor is normally low and goes high when touched.
- The if statement turns on the LED if the sensor is touched,

delays for 100ms, then turns the LED off.

## Going Further

One good attribute of this sensor is the internal switch bouncing. There is no jitter or bouncing when the switch is activated and deactivated as shown in the oscilloscope trace below. This was measured with the delay(DELAY_TIME) commented out. The pulse width varies with the duration of the switch being contacted.




This is an inexpensive and reliable switch that is immune to mechanical fatigues, such as a pushbutton slide switch. The only small downside is that the switch does not function with non-human contact.

# CHAPTER SIX

## 4+ Pin Input Sensors

This section covers several 4 pin input sensors, which, in many cases, are more sophisticated than the 3 pin.

### TAC PushButton

The most popular buttons for Arduino applications are the 6x6mm momentary tactile (TAC) pushbuttons. These switches are found almost everywhere in electronic applications. Although they seem to be, dealing with buttons is not trivial. Buttons and switches rarely, if ever, make a completely clean transition from low to high or vice versa. Switches are mechanical devices that jitter or "bounce" when pressed and released, and we need to effectively deal with that.

#### Requirements and Comments

We will follow the classic pushbutton application driving an LED, but with a twist. Instead of turning on an LED with a pushbutton, we'll start the application with the LED on and use the switch to turn it off. As simple as it seems, this is more difficult to get right do to the nature of the loop( ) function and switch de-bouncing.

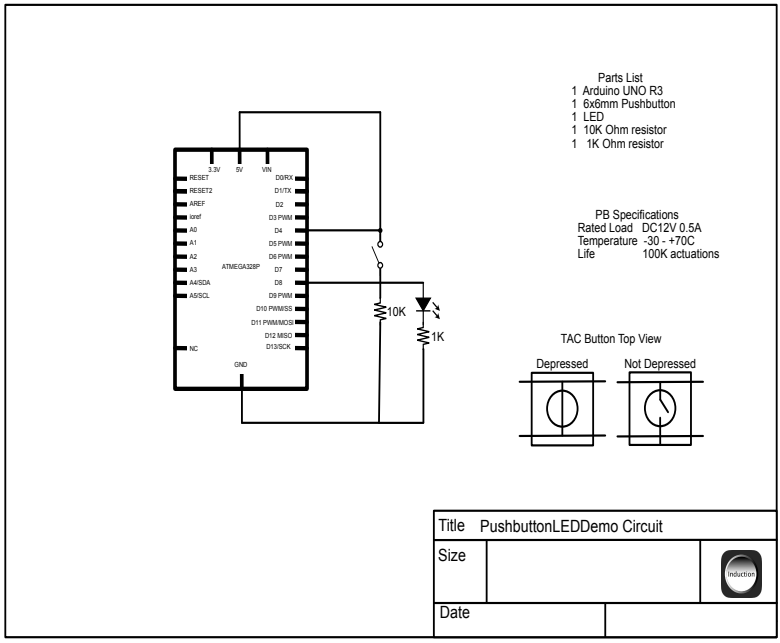
The requirements are:

- Initialize the LED_PIN pin for OUTPUT on pin 4.
- Initialize the BUTTON_PIN for INPUT on pin 2.
- Initialize the Serial Monitor at 9600 baud.
- In the loop( ) function:
- turn on the LED.
- print "button not pressed"

- read the button state.
- while the button is pressed:
- print “button pressed”.
- turn off the LED.
- delay until the button is released.

## The Circuit

The circuit is simple and can be breadboarded in a matter of minutes.




One side of the pushbutton is connected to 5VDC and pin 4 of the Arduino Uno R3. The other side of the pushbutton is connected to ground through a 10K Ohm resistor. The anode (+ side) of the LED is connected to pin 8 of the Arduino and the cathode (- side) is connected to ground through a 1K Ohm resistor.

## The Code

```
/*
File: PushbuttonLEDDemo.cpp
This demo de-energizes an LED when a pushbutton is pressed.
To debounce the pushbutton, the pushbutton state is continuously
read while the pushbutton is depressed. This is one of many methods
to debounce a momentary pushbutton input. The caution here is while
the pushbutton is depressed, the code stalls in the while loop.
*/
//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif
#define BUTTON_PIN  2
#define LED_PIN     4
int bp = 0;   // button pressed input variable

void setup() {
Serial.begin(9600);
// init LED pin
pinMode(LED_PIN,OUTPUT);
// init button pin
pinMode(BUTTON_PIN,INPUT);
}

void loop() {
// turn in LED
digitalWrite(LED_PIN,HIGH);
// print and get button status
Serial.println(" button not pressed");
bp = digitalRead(BUTTON_PIN);
if(bp) {
  Serial.println(" button pressed");
  // turn off LED
  digitalWrite(LED_PIN,LOW);
}
// stall while button is pressed
while(bp = digitalRead(BUTTON_PIN))
  ;
}
```

## Code Walk Through

- First the #define ARDUINO_IDE is examined. If not defined then PlatformIO is the development environment and the file <Arduino.h> is included.
- The BUTTON_PIN 2 and LED_PIN 4 are defined as pin 2 and pin 4 on the Arduino Uno R3.
- A global variable bp (standing for button pushed) is declared as type int and set to 0.
- In setup( ) the Serial Monitor is initialized and the pin modes for the LED and pushbutton are initialized.
- In loop( ) the LED is turned on, set by digitalWrite(LED_PIN,HIGH) and the message “button not pressed” is printed.
- The button state is read in bp = digitalRead(BUTTON_PIN).

- iIf the button is pressed, then bp is non-zero, therefore true, and the LED is de-energized.
- The code spins (stays in the while loop) as long as bp is true, that is, as long as the button is pressed. This is not the best way to debounce a switch, but I wanted to bring it to your attention because you may see it here and there in embedded code. On the positive side, it makes for a very stable button press input. On the negative side, the code suspends operation until the button is released. The only code that will be able to operate under this condition is an interrupt service routine. A different way to debounce is to delay a few milliseconds using delay(10) or the like.

Build the circuit and run the code. When you press the button, notice that that the TX LED on the Arduino turns off. This is because, as we know, the only code executing is the digitalRead(BUTTON_PIN) in the while loop. If you remove the while loop, something weird happens. The LED briefly flashes, even if you continuously hold the button down. Why? After the button is pressed the code within the if(bp) statement is executed, turning off the LED. After the if(bp) block of code is executed, the loop returns to the top and digitalWrite(LED_PIN,HIGH) is executed, turning on the LED.

## Going Further- Switch Debouncing

One of the challenges in embedded system design and coding is the debounce problem. Buttons and switches tend to toggle a little, or bounce, when turned off and on. If the bounce issue is not stabilized, the system could interpret a button or switch state incorrectly and execute code that it shouldn't. Here we will modify the code by adding a debounce function.

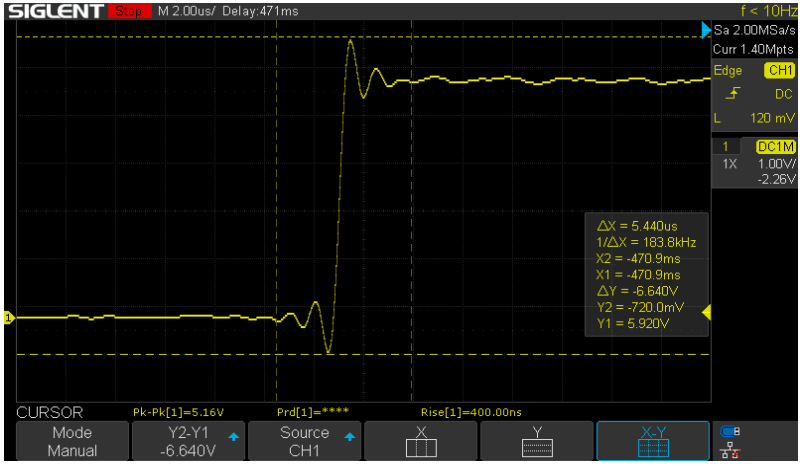
## The Code

```
/*
File: PushbuttonLEDDemo2.cpp
This demo de-energizes an LED when a pushbutton is pressed.

A debounce function is used to debounce button presses.
Uncomment #define DEBUG, recompile and download to see
the Serial.print( ) output.
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

//#define DEBUG
#define BUTTON_PIN  2
#define LED_PIN     4

int b_state = LOW;   // button pressed input variable

// debounce function prototype
int debounce(int state);

void setup() {
Serial.begin(9600);
  // init LED pin
  pinMode(LED_PIN,OUTPUT);
  // init button pin
  pinMode(BUTTON_PIN,INPUT);
}

void loop() {

 // turn on LED if button not pressed
  if(!digitalRead(BUTTON_PIN)) {
    digitalWrite(LED_PIN,HIGH);
    #ifdef DEBUG
    Serial.println(" button not pressed");
    #endif
  }
  // get button press
  int b_state = digitalRead(BUTTON_PIN);

  // call debounce function
  b_state = debounce(b_state);

  // de-energize LED
  if(b_state == 1) {
      digitalWrite(BUTTON_PIN,LOW);
      digitalWrite(LED_PIN,LOW);
      #ifdef DEBUG
      Serial.println(" button pressed");
      #endif
  }
}

// debounce( ) - takes the incomning state of the button
// and the state of the button (pressed or unpressed) and
// compares it to the current button presss state. If the
// states are different, the button state is read again.
// This provides a simple debouncing technique.
int debounce(int state) {
  int curState = digitalRead(BUTTON_PIN);
  if (state != curState) {
    curState = digitalRead(BUTTON_PIN);
  }
  return curState;
}
```

## Code Walk Through

- A differently name variable int b_state is used and initially set to LOW. This holds the state of the button (pushed or un-pushed) in loop( ).
- The function prototype int debounce(int state) is declared. This is the function that debounces the button press.
- The statement if(!digitalRead(BUTTON_PIN) reads the state of the pin and executes the follow block of code if the button is not pressed, setting the LED to HIGH. This is an easy and efficient way to test function return values while not wasting storage using return-value variables. When the button is pressed, the block of code is not executed.
- A check for a button pressed is tested again by int b_state = digitalRead(BUTTON_PIN).
- b_state is passed to debounce as a parameter and is debounced and updated by the debounce( ) function. The same variable, b_state, is used to reduce the clutter of adding a separate return value variable.
- The debounce( ) function takes the incoming button state as a parameter and returns an int representing the current state of the button press. A local int variable curState holds the value of a new call to digitalRead( ).
- The state parameter is compared to the current value. If they do not match, the state curState is read again and updated. This is used to smooth out any bouncing. For an especially bouncy switch, a short time delay of around 10ms can be used between the calls to digitalRead( ).
- Finally the curent state is return and stored in the b_state variable.
- if the button is pressed (b_state contains 1), then the LED is turned off.

Switch debouncing is a fact of life in embedded system programming and it will serve you well to learn as many different debouncing methods as you can, both in hardware and software. Here's screen shot of the TAC pushbutton in action.




You can see although the reaction time is fast, the switch input is very bouncy. Depending on the quality of switch, bounce can be more or less dramatic.

## HC-SR04 Ultrasonic Sensor

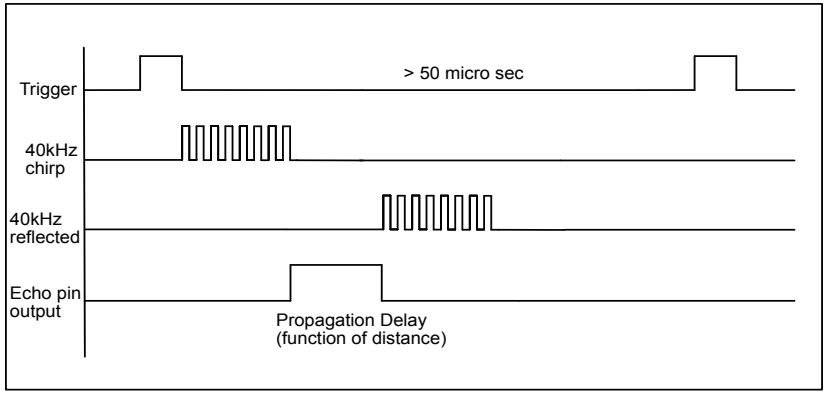
Ultrasonic sensors have a wide variety of uses. They are good for general depth and level measurement, distance sensing, proximity sensing among many other applications. Ultrasonic sensors are found in drone and parking apps, trash level monitoring, bottle counting, tank and parts bin levels all the way to car washes. In this section we'll utilize a HC-SR04 ultrasonic sensor, which is widely used in Arduino apps.

### How it Works

The HC-SR04 ultrasonic sensor sends out bursts of ultrasonic sound waves starts a timer is used to determine the time it takes for a reflected sound wave return to the sensor. The sensor has 4 pins that plug directly into a breadboard. Looking at the sensor from the front the pins are labeled 1-4 from left to right. Vcc (5VDC input), the next pin is labeled Trig.

The sensor sends out 8 bursts of 40KHz ultrasonic waves. This is called a "chirp". The HC-SR04 ultrasonic sensor sends the chirp when the Trig pin is set high for 10 microseconds. The waves travel and are reflected by any object they encounter, up to a limit.

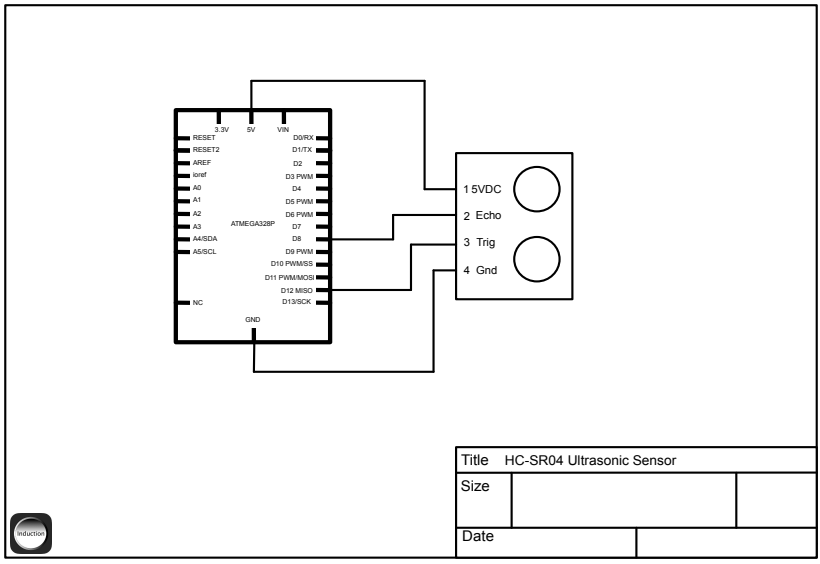
Looking at the logic diagram below, the Trigger pin is set high for 10 micro seconds. On the trailing edge of the trigger the 40kHz 8 burst chirp is emitted. A timer is started and the elapsed time is measured and output on the Echo pin. This is the propagation delay. The timer stops when the reflected sound is returned. Note that the propagation delay is for the sound wave round trip, so to measure the distance to the reflected object, the propagation delay must be divided by 2.



The HC-SR04 operates on 5VDC and the nominal current is 15mA. The maximum range is 4 meters and the minimum range is 2cm. The default units are centimeters. The measuring angle is 15 degrees and the input trigger signal is 10 micro seconds. The output, or propagation delay, is a TTL level is proportional to the distance.

## The Circuit

The circuit is simple with 4 connections. 5VDC is connected to pin 1 on the HC-SR04. Pin 2 is the Echo pin where the returned signal duration emanates. The Echo pin is connected to pin 8 on the Arduino. Pin 3 is the Trig pin which triggers the eight 40kHz chirp and is connected to the Arduino Pin 12. Pin 4 is ground and is connected to Gnd on the Arduino.



## The Code

Here's the code for the first version, named UltrasonicDemo1.cpp.

```
// UltrasonicDemo1.cpp
//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif
//#define NEW_PING_LIB // uncomment this to use NewPing lib
#ifdef NEW_PING_LIB
#include "NewPing.h"
#endif
// HC-SR04 control and data pins
#define TRIG_PIN 12
#define ECHO_PIN 8

// constant for metric velocity of sound
#define METRIC_VEL_SOUND 0.0343;
// delay time between pings
#define DELAY_TIME 1000  // default to 1sec

float elapsed_time, dist, inch_dist;  // measurement vars

void setup() {
  Serial.begin(9600);
  pinMode(TRIG_PIN,OUTPUT);  // send the sound out of pin 12
  pinMode(ECHO_PIN,INPUT);      // receive the sound back on pin 8

  // turn off output and delay to let the sensor settle a little
  digitalWrite(TRIG_PIN,LOW);
  delay(100);
}
void loop() {
  // write a 10ms pulse to the HC-SR04
  float outer_limit = 400.0; // outer range var
  float inner_limit = 2.0;   // inner range var
  digitalWrite(TRIG_PIN,HIGH);
  delayMicroseconds(10);
  digitalWrite(TRIG_PIN,LOW);

  // get the incoming sound pulse - read when the pulse is HIGH
  elapsed_time = pulseIn(ECHO_PIN,HIGH);
  // determine the distance
  dist = (elapsed_time/2) * METRIC_VEL_SOUND;
  inch_dist = dist/2.54;  // 2.54in per cm
  // print the output
  if(dist <= outer_limit || dist <= inner_limit) {
    Serial.print("Distance is: ");
    Serial.print(dist);
    Serial.println(" cm");
    Serial.print("Distance is : ");
    Serial.print(inch_dist);
    Serial.println(" in");
  } else {
      Serial.println("* no objects detected *");
  }
  delay(DELAY_TIME); // delay for next sensor read
}
```

## Code Walkthrough

- The #define ARDUINO_IDE is used to switch between using the Arduino IDE or PlatformIO. If the Arduino IDE is used, then the file <ardruino.h> is not included.
- The NewPing library is used in the sketch. To check out what NewPing does and to downloaded it, go here:https://www.arduino.cc/reference/en/libraries/newping/. The NewPing library is not used, but I wanted to include it just in case you what to play with it.
- The Trigger and Echo pins are assigned to Arduino pins 12 and 8 respectively.
- Constants are defined for the speed of sound in #define METRIC_VEL_SOUND and the delay time between pings in DELAY_TIME. The default units of the HC-SR04 are in metric, so the velocity of sound constant must be in metric units also.
- The global float variables are defined for elapsed_time, dist, and inch_dist, where inch_dist is the dist variable converted to inches.
- In setup( ), the serial monitor is initialized via Serial.begin(9600) and the modes for the trigger and echo pins are set. The trigger pin is set to OUTPUT, and the echo pin to INPUT. The trigger pin is initially set to LOW to enter the loop( ) in a known, stable state.
- In loop( ) two float variables are declared, outer_limit and inner_limit and are assigned values 400.0 and 2.0 respectively. They represent the upper and lower ranges of the HC-SR04. These are declared as float variables and will be used for comparisons later. Some compilers complain when #defines are used for these, so play it safe and use the same type variables when doing comparisons.
- A 10 microsecond pulse or “ping” is sent out by setting the Trigger pin HIGH, waiting 10 microseconds, the forcing the Trigger pin LOW.
- The round trip of the ping is captured by in elapsed_time via pulseIn( ).
- The distance is determined by dividing the elapsed_time by 2 and multiplying by the metric velocity of sound constant, METRIC_VEL_SOUND.
- The distance in cm is then converted into inches by the line

inch_dist = dist/2.54.

- The distance value is checked if it is in-range or out-of-range. If it is, the Serial.print( ) statements are bypassed and a * no object detected * message is issued. If the distance is in range, the distance is issued in cm and inch units.

## Going Further

C is a language designed around functions and variables. The above code is written in-line, meaning that the code executes step-by-step in the loop( ) function. This is not the best way to write code. A better way is to divide the important parts into individual functions. In our code, three functions were added, float getData(void) that returns the HC-SR04 echo data, float calcDistance(float elapsed_time) that returns the distance, and void dispDat(float distance, float et, int units, float outer, float inner) that displays the data if the data is within range. The int units parameters tells the function to display the units in cm or inches. The filename for the updated code is UltrasonicDemo2.cpp.

## The Code

```
// UlrasonicDemo2.cpp
//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

//#define NEW_PING_LIB // uncomment this to use NewPing lib
#ifdef NEW_PING_LIB
#include "NewPing.h"
#endif

// HC-SR04 control and data pins
#define TRIG_PIN 12
#define ECHO_PIN 8

// constant for metric velocity of sound
#define METRIC_VEL_SOUND 0.0343;
// delay time between pings
#define DELAY_TIME 1000  // default to 1sec

// function prototypes - see header for description
float getData(void);
float calcDist(float elaped_time);
void dispDat(float distance,float et, int units,float outer,float inner);

void setup() {
  Serial.begin(9600);
  pinMode(TRIG_PIN,OUTPUT);  // send the sound out of pin 12
  pinMode(ECHO_PIN,INPUT);      // receive the sound back on pin 8
  // turn off output and delay to let the sensor settle a little
  digitalWrite(TRIG_PIN,LOW);
  delay(100);
}

void loop() {
  float elapsed_time, dist;  // measurement vars
  float outer_limit = 400.0; // outer range var
  float inner_limit = 2.0;   // inner range var

  // get the elapsed time to calc distance
  elapsed_time = getData();
  dist = calcDist(elapsed_time);
  // calculate the distance
  dispDat(dist,elapsed_time,0,outer_limit,inner_limit);
  delay(DELAY_TIME);
}

// functions
// getData: actives the trigger pin and waits for the echo result.
// Returns: elapsed time / 2.
float getData(void) {
  float et;
  digitalWrite(TRIG_PIN,HIGH);
  delayMicroseconds(10);
  digitalWrite(TRIG_PIN,LOW);
  // get the incoming sound pulse - read when the pulse is HIGH
  et = pulseIn(ECHO_PIN,HIGH);
  return et;
}

// calcDist: calculate the distance to a pinged object using
// the elapsed time returne by getData().
float calcDist(float et) {
  float d;
  d = (et/2) * METRIC_VEL_SOUND;
  return d;
}

// displays the distance data in cm or inches. Takes the distance
// value returned by calcDist() as input.
void dispDat(float distance,float et,int units, float outer, float inner){
  if(distance >= outer || distance <= inner){
    Serial.println(" ** OUT OF RANGE **");
    return;
  }

  Serial.print("Distance is: ");
  if(units == 0) {
    Serial.print(distance);
    Serial.println(" cm");
  }

  if(units == 1) {
    distance = distance/2.54;
    Serial.print("Distance is : ");
    Serial.println(" in");
  }
  Serial.print("Time is: ");
  Serial.print(et);
  Serial.println(" us");
}
```

## Code Walk Through

Most of the code is the same but just rearranged. We will start at the function prototypes.

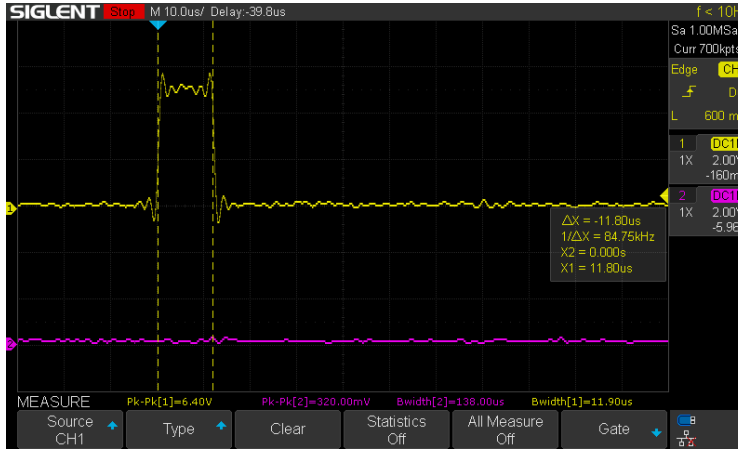
- Three function prototypes are declared for getData( ), calcDistance( ) and dispData( ). The parameters are not included for brevity. Declaring these before the functions are called inform the compiler that the functions will be used in the sketch.
- In loop( ), the measurement variables are declared along with the in and out of range variables.
- Four functions are called in order: getData( ), calcDist( ), dispData( ) and the built-in function delay( ). This simplifies the loop( ) function.
- The getData( ) function controls the ping and gets the incoming elapsed time. The elapsed time is returned in the local variable et.
- The calcDist( ) function returns the distance based on the elapsed_time value returned by getData( ). The distance value returned is in cm.
- The dispDat( ) function issues the data via Serial.print( ) if the data is in range. dispDat( ) also looks at the value of units to determine whether to issue the data in cm or in inches.

Ultrasonic sensors are capable and useful sensors that are used in a wide variety of applications. Use your imagination to create a few.

## Oscilloscope Traces

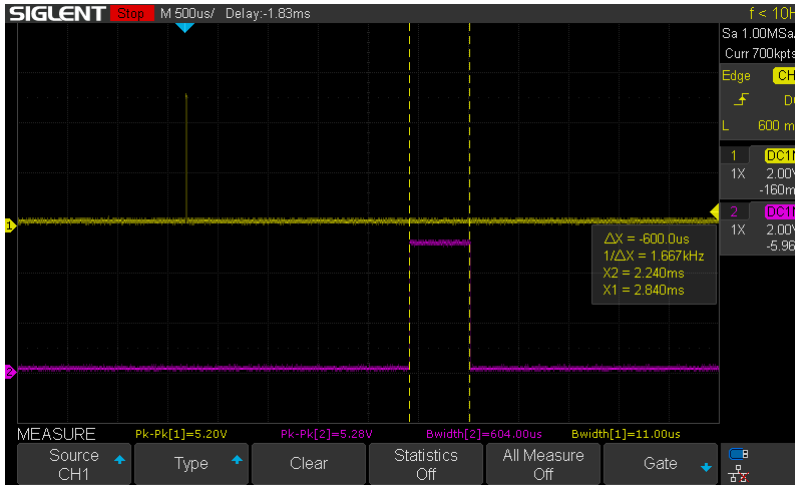
What do the Echo and Trigger pins look like on a scope? The code is written to output a 10 microsecond pulse on the Arduino pin 12. Channel 1 is the yellow trace and is connected to the trigger pin output. and channel 2 is the magenta trace and connected to the echo pin input. The scope trigger was set to 40mV on channel 1.

Here's the trace for the 10 microsecond pulse.




The time on the horizontal axis is set to 10 microseconds per division. I used the scope's internal measurement feature to measure the trigger pulse width and also the cursors to manually measure the pulse. Looking at the measurement box on the right, the cursor measurement, delta X, is 11.80 microseconds, so it's a little longer than 10 microseconds. The scope's automatic measurement value displayed in the lower right of the screen as Bwidth[1]=11.90 microseconds. Also notice the bounce on the pulse.

Here's the echo output:

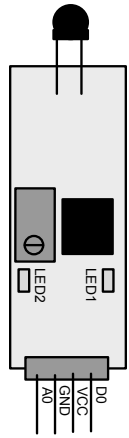



The time on the horizontal display is set to 500 microseconds per

division. In this view it's easy to see how small the trigger pulse is compared to the echo response. I had the distance set to approximately 10cm and the duration value returned was 600 microseconds, give or take a few microseconds. This corresponds with the measurements displayed on the scope, where the manual cursor measurement came out to 600.0 microseconds and the internal scope measurement calculated 604.00 microseconds. The pulse increases and decreases with distance.

## KY-026 Flame Sensor

The KY-036 flame sensor uses and infrared LED to detect heat. The more heat an object gives off, the more infrared radiation is emitted. The KY-036 uses the infrared LED to detect IR thermal radiation sourcing from a flame.



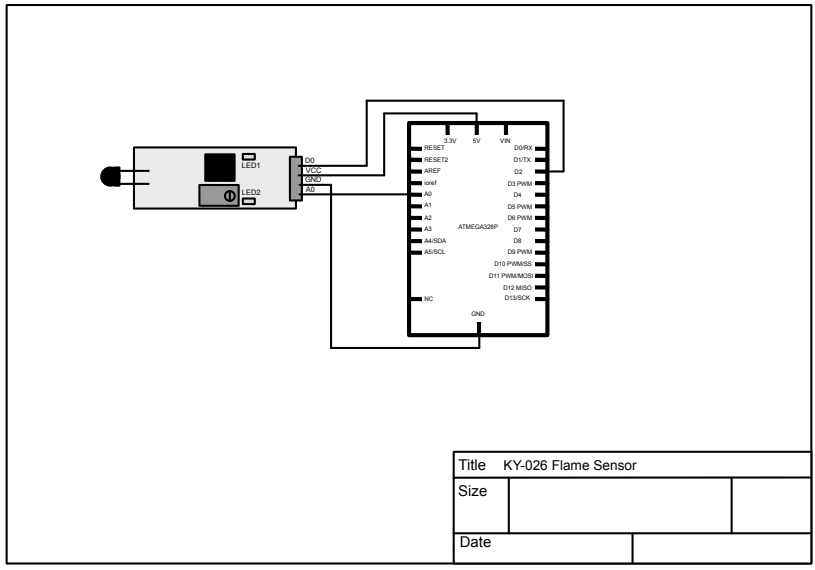

The KY-036 flame sensor utilizes the familiar 4 pin circuit board with a 292 comparator. The potentiometer is used to set the D0 digital output threshold. LED1 is the power LED and LED2 illuminates when the DO output threshold is met. GND is the ground pin and A0 is the analog output pin.

## Specifications

The KY-036 flame sensor's IR LED can detect a light source wavelength range of 760nm to 1100nm, with a distance of 20cm at 4.8VDC to around 100cm at 1VDC. The range is adjustable via the 100K pot on the circuit board. The detection angle is approximately 60 degrees. The KY-036 operating voltage ranges for 3.3VDC to 5VDC.

## The Circuit

The circuit uses both the digital and analog outputs of the sensor. The digital output, D0, is connected to D2 on the Arduino UNO. The analog output A0 is connected to A0 on the Arduino UNO. VCC is connected to 5V and GND is connected to GND on the Arduino UNO.


LED1
LED2
D0
VCC
GND
A0
3.3V
5V
VIN
RESET
RESET2
AREF
ioref
A0
A1
A2
A3
A4/SDA
A5/SCL
NC
ATMEGA328P
D0/RX
D1/TX
D2
D3 PWM
D4
D5 PWM
D6 PWM
D7
D8
D9 PWM
D10 PWM/SS
D11 PWM/MOSI
D12 MISO
D13/SCK
GND
Title KY-026 Flame Sensor
Size
Date

## The Code

```
/*
Project: K-026FlameSensorDemo
Demos the K-026 IR LED flame sensor.
*/

//#define ARDUINO_ide
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif
#define SENSOR_DELAY 10
// const ints for input pins and
// data array size
const int D_IN_PIN  = 2;
const int A_IN_PIN  = A0;
const int ARRAY_SIZE = 5;

// function prototypes
void getSensorData(int[]);
int average(int[]);

void setup() {
  Serial.begin(9600);
  pinMode(D_IN_PIN,INPUT);
  pinMode(A_IN_PIN,INPUT);
}

void loop() {
  int analog_val = 0;
  int digital_val = 0;
  int analog_vals[ARRAY_SIZE];

  // get analog value averaged over 5 readings
  getSensorData(analog_vals);
  analog_val = average(analog_vals);
  // read digital pin- if 1 then flame is detected

  digital_val = digitalRead(D_IN_PIN);
  if(digital_val == 1) {
    Serial.println("Flame Detected");
  }
  // uncomment to see analog values
  //Serial.print("Analog value is ");
  //Serial.println(analog_val);
}

  // getSensorData - gets 5 sensor analog redings
  // and loads them into an array
  void getSensorData(int val_array[]) {
    for(int i = 0; i < ARRAY_SIZE; i++) {
      val_array[i] = analogRead(A_IN_PIN);
      delay(SENSOR_DELAY);
    }
  }

  // average - averages the array sensor readings
  int average(int a_vals[]) {
    int a_ave = 0;
    // get analog vals
    for(int i = 0; i < ARRAY_SIZE; i++) {
      a_ave+=a_vals[i];
    }
    a_ave = a_ave/ARRAY_SIZE;
    return a_ave;
  }
```

## Code Walk Through

- If the Arduino IDE is used, then the #define ARDUINO_IDE needs to be uncommented. This bypasses including the <Arduino.h> include file. This is not required for the Arduino IDE but is required when using PlatformIO. Since this code was developed using PlatformIO, #define ARDUINO_IDE is commented out.
- A #define SENSOR_DELAY 10 is declared. This is used for a 10ms delay between sensor readings.
- The next three lines declare const ints for the input pins and array size are declared. The digital pin, D_IN_PIN is on pin 2 of the Arduino UNO and the analog pin, A_IN_PIN is on pin A0 of the Arduino UNO. ARRAY_SIZE 5 is the size of the array that holds the raw analog sensor readings.
- Two function prototypes are declared, void getSensorData(int[ ]) and int average(int[ ]). The first function takes 5 analog readings and places them in an array, and the second functions takes the 5 readings and averages them. This smooths the data.
- In setup( ) the serial monitor is initialized at 9600 baud, and the pin modes are set for the digital and analog input pins. Both are set to inputs.
- In loop( ) three local variables are declared int analog_val, digital_val and the array analog_vals[ARRAY_SIZE]. The analog_val holds the average of the values in the analog_vals array and digital_val holds the state of the digital input from the sensor. The analog_vals array holds the five raw sensor readings.
- The next line of code reads the digital input from the sensor via the call to digital_val = digitalRead(D_IN_PIN).
- If the digital value is 1, then a flame is detected and the message "Flame Detected" is printed on the serial monitor.
- The next two lines of code print the analog values if uncommented. This is useful to see the range of the flame sensor output.
- The void getSensorData(int val_array[ ]) uses a for loop making five successive calls to analogRead(A_IN_PIN). This loads the array with five back to back sensor readings. A 10ms delay, delay(SENSOR_DELAY), is called to induce a short delay between sensor readings.

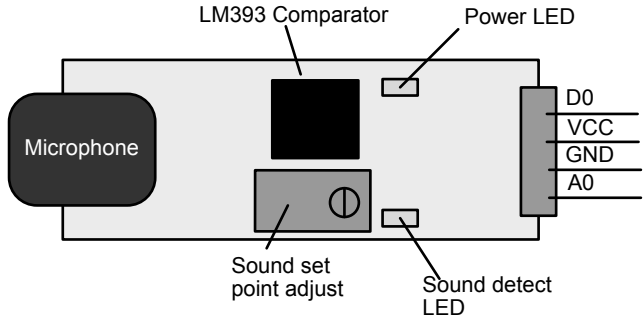
- The int average(int a_vals[ ]) function averages the values in the incoming array and returns the average in the int variable a_ave.
-

The KY-026 is an effective and fast responding sensor and is worthy of a project or two.

## KY-038 Sound Modules

There are two modules that are supplied with Arduino sensor kits, a KY-038 Big Sound and a Small Sound module. Both contain microphones that take acoustic energy as input to an LM393 comparator. A sound threshold, or set point, is controlled by a potentiometer that is also an input to the LM393. This sets the sound sensitivity level.

The "Big Sound" module detects loud sounds, such as a hand clap. The big sound sensor is used for intrusion alert, such as breaking glass or other the presence of other substantial noises.




The Big Sound and Small Sound modules have the same pinout.

- D0 is the digital output based on the output of the LM393 comparator.
- VCC is 5VDC.
- GND is ground.
- A0 is the direct output from the microphone and an input to the LM393 comparator.

Sound energy enters into the microphone and is amplified by an LM393 comparator. The sound level detection, or set point, is controlled by a multi-turn 100K potentiometer. The Power LED indicates whether power is applied to the module, and the sound detect LED flashes when sound is detected.

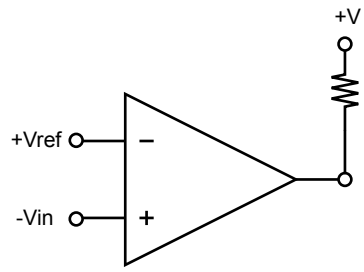
The 100K potentiometer adjusts the sensitivity level of the microphone input. One side of the comparator is wired to the microphone and the other to the potentiometer. Turning the pot counterclockwise makes the unit less sensitive to sound. Turning the pot to the right makes it more sensitive to sound. Turning the pot counterclockwise causes the LED to go off. Turing the pot clockwise causes the LED to go on. The most common way to adjust the KY-038

modules is to turn the pot clockwise until the sound detect LED comes on, then slowly turn the pot counter clockwise until the sound detect LED turns off. This is where the sound module is the most sensitive.

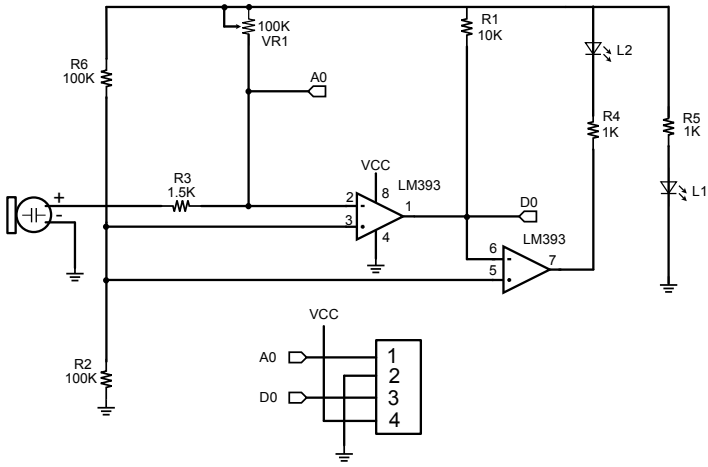
The pinout is labeled on the sensor, making it easy to connect. A0 is the analog output from the microphone which is an input to the LM393 comparator. D0 is the digital output signal the goes high when the sound set point is exceeded. VCC is 5VDC and GND is ground. We will use pin A0 on the Arduino UNO as an analog read pin for the A0 output pin from the sensor. Pin 2 on the Arduino UNO will be used for the digital D0 output pin from the sensor.

## Requirements and Comments

Our requirements are to place a big sound and a small sound sensor side by side, light a different color LED for each and examine some scope traces. The application is simple, but a more important component than the module itself is the on board LM393 dual comparator. The LM393 is a high precision dual comparator commonly used in used in simple analog to digital converters, time delay generators, multivibrators and many more applications. Comparators are very valuable and are used to convert analog signals to digital signals. The figure below shows a basic comparator.



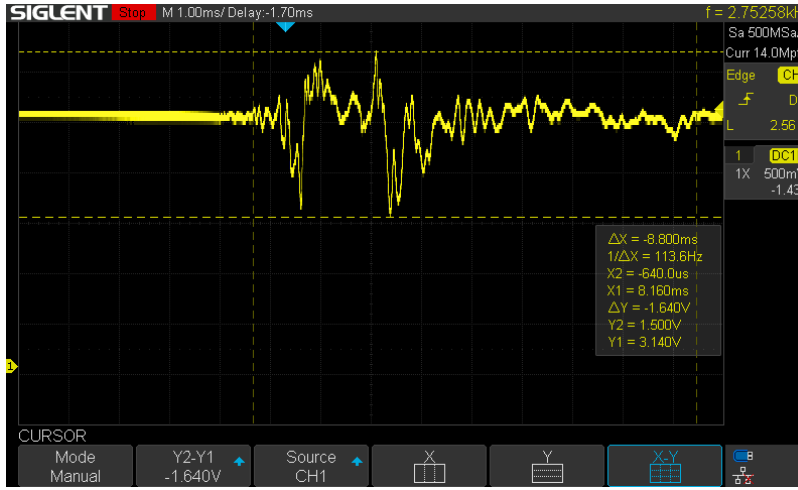

It's instructive to examine and understand the KY-038 sound module schematic. The microphone input is connected to pin 2 negative input of the LM393 via 1.5K Ohm resistor.



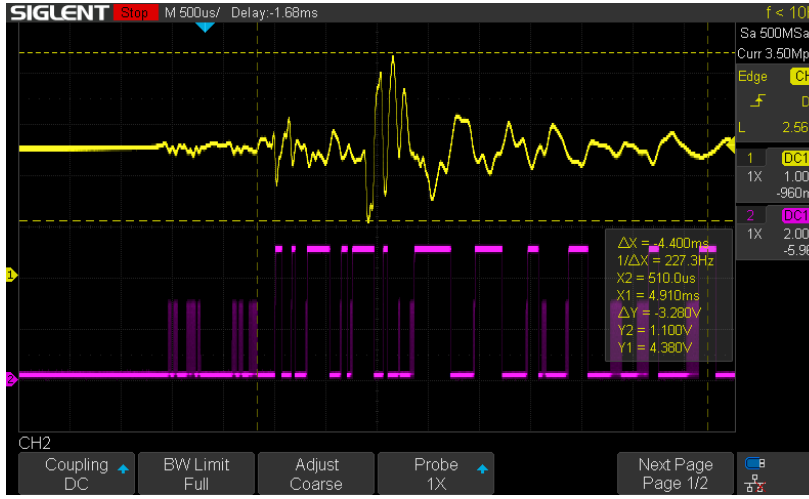

Output pin A0 is the un-amplified microphone input and is influenced by resistor R3 and VR1, the 100K potentiometer. The 100K potentiometer ultimately sets the threshold value to the LM393 comparator. When the threshold value is exceeded, an output is present on pin 1 of the LM393 comparator, and consequently pin D0. LED L1 illuminates when power is applied to the module. LED L2 illuminates for the duration that pin 1 of the LM393 comparator and D0 is 1. See the Sensor Utilization for more details regarding the ML303 comparator.

Before diving into the sketch let's take a look at a couple of scope traces for the big sound module without an Arduino sketch running. Just connect 5V from the Arduino to VCC on the big sound module as well as GND. I used a couple of long jumpers attached to the A0 and D0 outputs to connect the scope probes. The analog output A0 is connected to Channel 1 of the oscilloscope and Channel 2 is connected to the digital output D0.

This trace monitored the A0 pin only. The sound set point was adjusted to right before where the sound detect LED would stay permanently on. First adjust the pot clockwise where the LED stays on continuously, then back it off counter clockwise until the LED goes off. This is supposed to be the maximum sensitivity setting. Make a sound noise, such as a hand clap, and the sound detect LED should illuminate.




This is the response from a loud handclap close to the microphone. The signal duration from the microphone is 8.8ms, and the voltage swing is 1.640V peak to peak. The signal is spiked and takes a while to settle down. This has consequences for the digital output, D0, as shown in the next trace.




The D0 output bounces a lot, following the microphone input, calling for some kind of debounce routine in the sketch. This is usually implemented as a simple time delay.

Different materials of course produce different sounds when struck

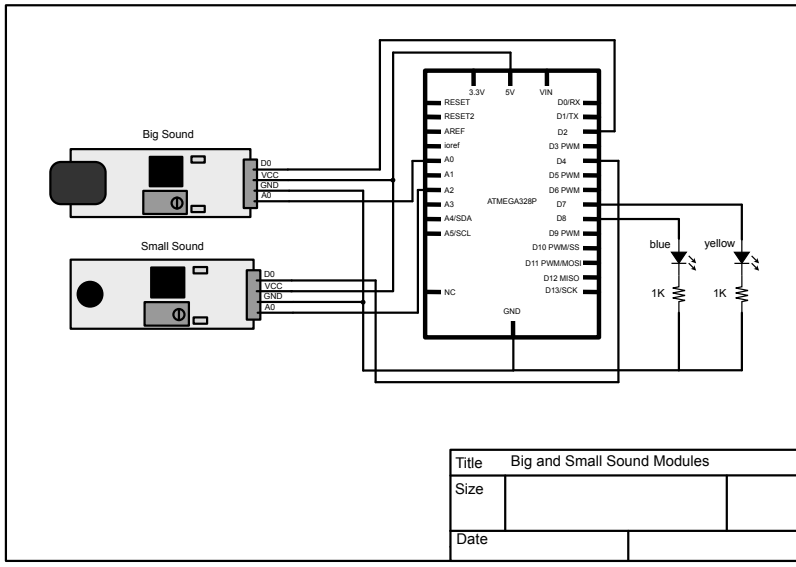
together. The first trace below uses a small sound sensor set at the max sensitivity setting. The scope trace captures a loud hand clap. The top yellow trace is the output from the microphone at pin A0 and the purple trace is the output at D0.


The yellow hand clap trace is fairly distinct and measures end to end at 5.250ms. I measured worst case for this, where D0 starts to output, even on the low voltage, grassy spikes. Compare this to the trace below.

This trace was produced by tapping a small screwdriver against an empty coffee cup. This trace is radically different than the hand clap trace with a duration of 6.190ms. It's an interesting and enlightening experience to see the different acoustic characteristics that different materials produce. An interesting project, possibly using a more powerful microcontroller than an Atmega 928p would be to learn and characterize the input signals to detect and classify the materials used to produce the sound, possibly with a neural network. That's a topic for another book!

## The Circuit

Here's the circuit we will use for both sensors.




The Big Sound sensor contains 4 pins, D0 which is the digital output, VCC is connected to 5VDC, Gnd is connected ground and A0 is the direct analog output from the microphone.

## Specifications

The code is required to read the Small and Big Sound sensors back to back. The D0 output of the big sound sensor triggers a blue LED connected to Arduino UNO output D8. The D0 output of the small sound sensor triggers a yellow LED connected to Arduino UNO output D7. The A0 inputs are connected to Arduino UNO analog inputs A0 and A2,

but are not used. The LEDs blink for the duration of the acoustic energy input.

## The Code

```
// FIle: BigSmallSoundDemo.cpp
//#define ARDUINO_IDE

#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

// define output pins for Big Sound
#define BIGS_AIN A0
#define BIGS_DIN 2

// define pins for Small Sound
#define SMLS_AIN A2
#define SMLS_DIN 4

// defines for LEDs
#define BLUE_BIG_SOUND_LED 8
#define YELLOW_SMALL_SOUND_LED 7
#define LED_ON  1
#define LED_OFF 0

// function prototypes
void lightLED(int,int);

void setup() {
// Big sound sensor
pinMode(BIGS_AIN, INPUT);
pinMode(BIGS_DIN, INPUT);
pinMode(BLUE_BIG_SOUND_LED,OUTPUT);
// Small sound sensor
pinMode(SMLS_AIN,INPUT);
pinMode(SMLS_DIN,INPUT);
pinMode(YELLOW_SMALL_SOUND_LED,OUTPUT);
}

void loop() {
    // flash blue LED on Big Sound detection
    int d_big = digitalRead(BIGS_DIN);
    if(d_big == 1) {
    lightLED(BLUE_BIG_SOUND_LED,1);
    } else {
      lightLED(BLUE_BIG_SOUND_LED,0);
    }

    // flash blue LED on Small Sound detection
    int d_small = digitalRead(SMLS_DIN);
    if(d_small == 1) {
    lightLED(YELLOW_SMALL_SOUND_LED,1);
    } else {
      lightLED(YELLOW_SMALL_SOUND_LED,0);
    }
  }

// lightLED – illuminates LED based on led input
void lightLED(int led,int state) {
  if(state == 1) {
    digitalWrite(led,HIGH);
  } else {
    digitalWrite(led,LOW);
  }
}
```

## Code Walk Through

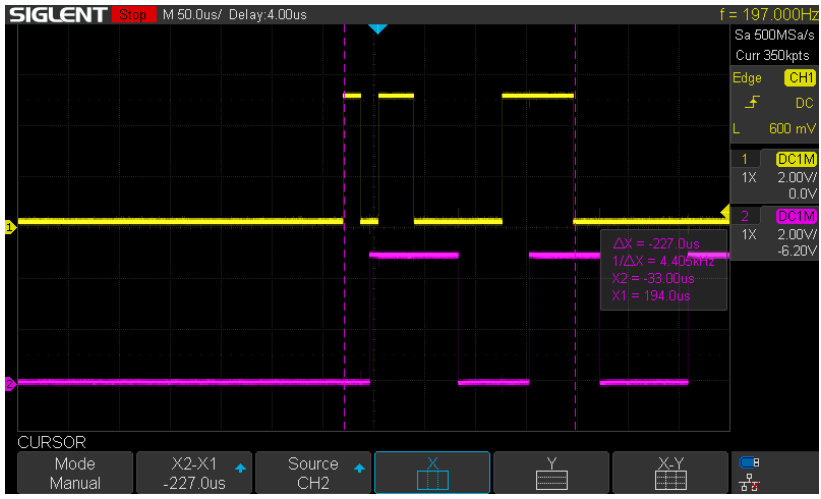
Let's walkthrough the code.

- If the Arduino IDE is used, then the #define ARDUINO_IDE needs to be uncommented. This bypasses including the <Arduino.h> include file. This is not required for the Arduino IDE but is required when using PlatformIO. Since this code was developed using PlatformIO, #define ARDUINO_IDE is commented out.
- The next section of code define the output pins for the big and small sound sensors, along with the LED on of off #defines.
- There is one function used, lightLED( ), that is called to illuminate the LEDs. lightLED has two int parameters, one to identify the LED and the other to signal whether to turn it on of off, via a 1 or 0.
- In setup( ) the input pins are set for the big and small sound modules, along with the LED outputs. The blue LED is the big sound sensor LED and the yellow the small sound sensor LED.
- In loop( ) the big sound input is read first and stored in the int variable d_big. If d_big is a 1 then lightLED is called. The 1 parameter is used to tell the lightLED function to turn on the LED. If false, the LED is turned off.
- The next block of code for the small sound sensor is identical that of the big sound sensor.
- The lightLED( ) function turns on theLED or turns if off, depending on the state variable.

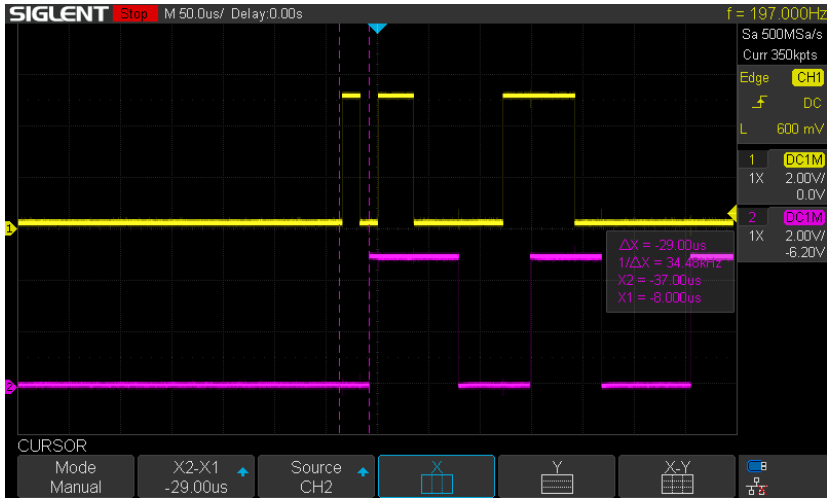
There are no delays in the code to debounce inputs. I intentionally kept it this way to take a look at the collective sensor's raw output via scope traces.

## Going Further

There are a couple of items of interest here. First is the delay time between the readings from the big sound and small sound sensors. On the surface you would think that since the big sensor code executes first, but this is not the constant case. Here are scope traces for a hand clap. The small sound sensor is on channel 1 which is the yellow trace, and the big bound sensor is on channel 2 which is the purple trace. I triggered on the small sensor output from the Arduino, pin D0 on the sensor, which is the D2 input on the Arduino. The small sound sensor trace leads the big sound sensor,

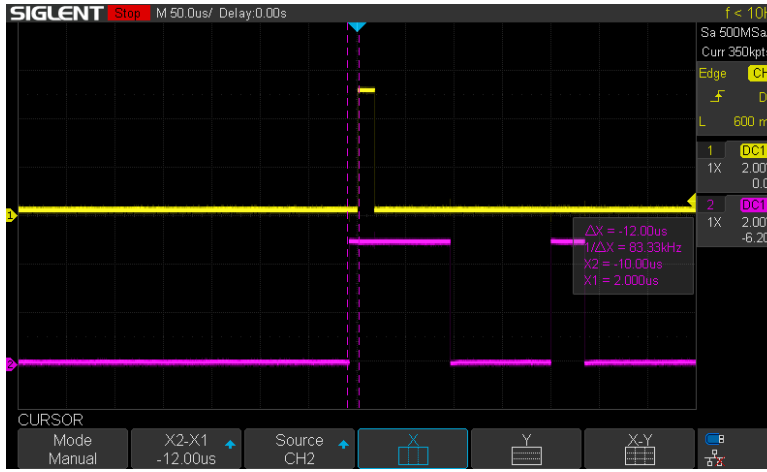



The pulse width of the big sound sensor output is larger and more consistent than the small sensor output.

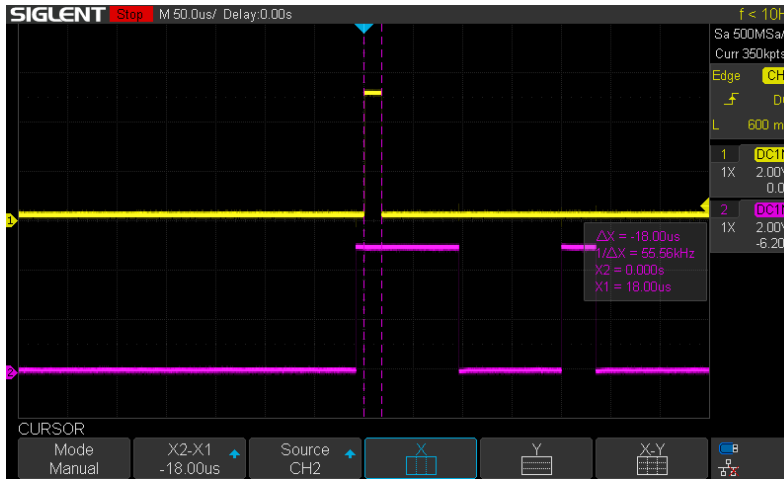



The time between detecting the small sound sensor and big sound sensor is around 29 microseconds.

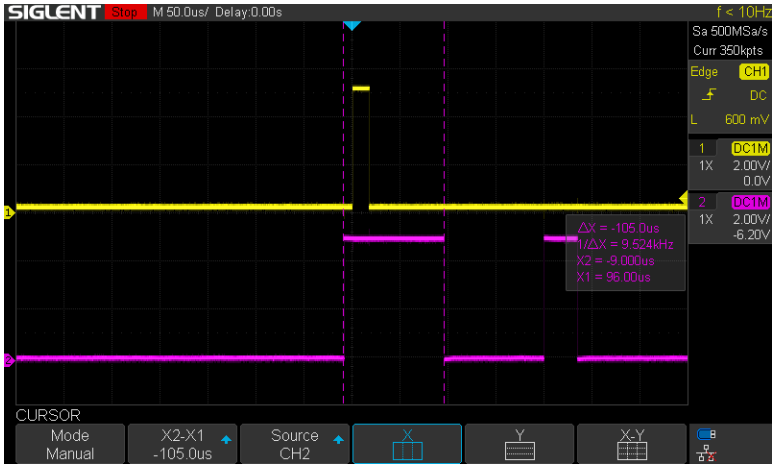
Here's another trace where the big sound sensor leads the small sound sensor for another hand clap. The big sound sensor leads the small by 12 microseconds.



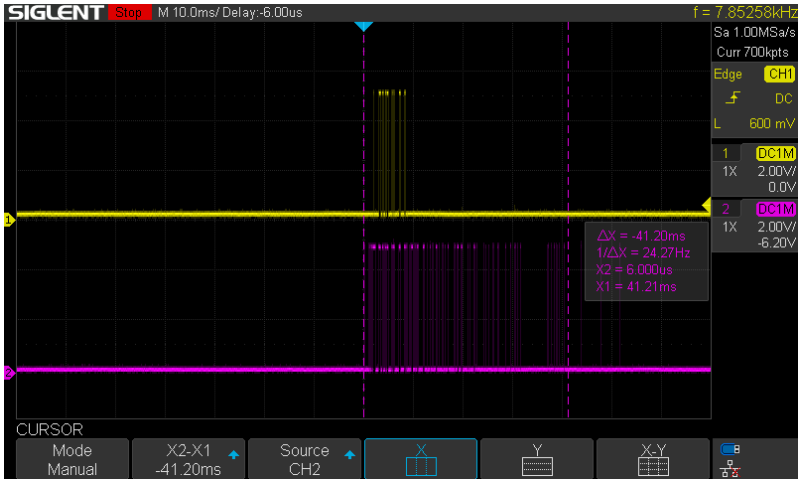

The traces below measure the pulse durations for the small sound and big sound sensors. The small sound sensor pulse width is 18 microseconds.




The pulse width for the big sound sensor is 105 microseconds, which is a little under six times the width of the small sound sensor.




An interesting trace shows the response from a tap of a coffee cup with the shaft of a screwdriver. The horizontal width is set to 10ms per division. The pulse duration of big sound module measure at 41.20ms, which is much larger than the small sound sensor trace. Both are incredibly bouncy.



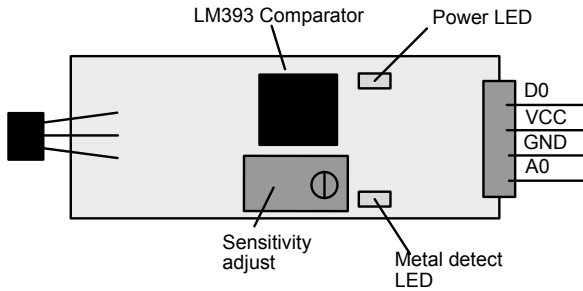

A delay of around 50ms should act as a reasonable debounce delay. Again the big takeaway from this section is not so much the big and sound modules, but the usefulness of the LM293 comparator. This component is very useful and a good foundation for your own sensor design.

s

## Linear Hall

Hall effect sensors detect the presence of a magnetic field. The Hall effect can be described as the way a magnetic field interacts with the flow of current through a conductor. A Hall effect sensor senses both the presence and magnitude of a magnetic field. A Hall effect sensor is simple in concept. A small, continuous current is passed through a semiconductor with a constant voltage. When a magnetized object

The Hall effect sensor we will use is a KY-024 that is a common sensor in Arduino sensor kits. The analog output A0 increases linearly in voltage as a magnet gets closer to the sensor.




## Requirements and Comments

This requirements for this demo are:

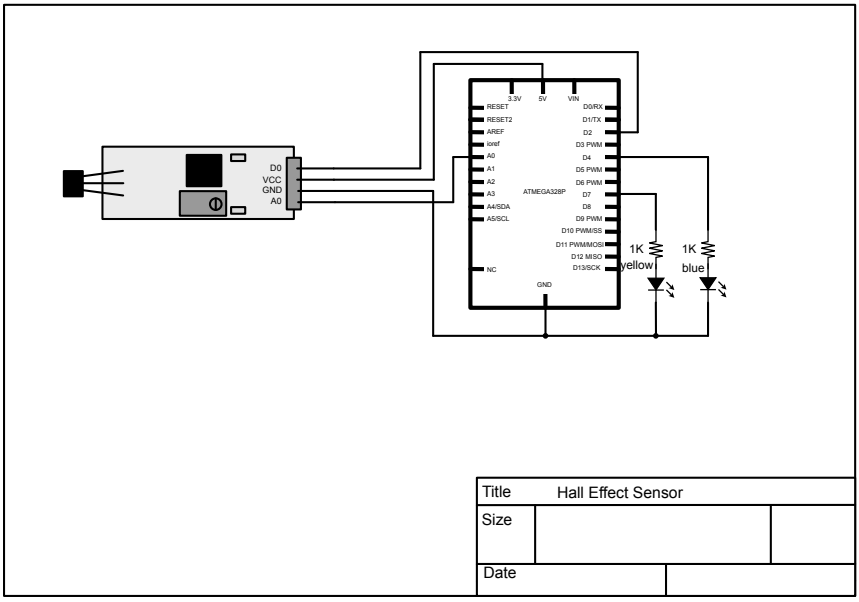
- Initially detect a magnetic object at an outer range and turn on a red LED.
- Detect a magnetic object at an inner range, turn off the red LED and turn on a yellow LED.
- Turn off both LEDs if the magnetic object is out of range.

The sensor reads the magnetic field when a magnet is in proximity of the sensor. The analog output value increases the close the magnet gets to the sensor. The sensitivity is set to around 512 by reading the analog value on the sensor pin A0. This is a common sensitivity setting for the KY-024 and is attained by running a small sketch that reads the analog pin A0 value. The map( ) function is used to map the incoming values to a smaller number range, namely 0 to 50. If a magnet is detected, a red LED is illuminated if the map value reading is greater than 40. If the map value reading is less than or equal to 40, then the red LED turns off and the yellow LED turns on.

This is useful for staged deceleration and distance measurement.

Using a multi-range set of values an object's deceleration can be controlled. For example, when a controllable object is initially detected within an outer limit range value, it could be directed to decelerate. When it approaches the inner range, the object can be commanded to stop.

## The Circuit




## Specifications

- Pin 1 or D0 outputs a 1 when a magnet is in proximity of the Hall effect sensor.
- Pin 2 is VCC which can range from 3.3V to 5VDC.
- Pin 3 or GND is ground.
- Pin 4 or A0 is the analog output from the LM393 comparator.

## The Code

```
/*
File: HallEffectDemo.cpp
Description: Reads a magnet in proximity of
the hall effect sensor. Turns on a red LED
if the mapped analog output value of the sensor is
less than or equal to 40. Turns off the red LED
and a yellow LED if the mapped analog value is
greater than 40.

Wiring:
AO Sensor   -> A0 Arduino
D0 Sensor   -> pin 2 Arduino
Red LED     -> pin 4 Arduino
Yellow LED  -> pin 7 Arduino
Uses the serial monitor to dispay values from the sensor.
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>

#enif

#define DIGITAL_DATA  2
#define ANALOG_DATA   A0
#define RED_LED       4
#define YELLOW_LED    7
#define TIME_DELAY    50

void setup() {
  Serial.begin(9600);
  pinMode(DIGITAL_DATA,INPUT);
  pinMode(RED_LED,OUTPUT);
  pinMode(YELLOW_LED,OUTPUT);
}

void loop() {
  // vars used to read and map data
  int analog_data = 0;
  int digital_data = 0;
  int map_val = 0;

  // read and display analog and digital outputs from sensor.

  analog_data = analogRead(ANALOG_DATA);
  map_val = map(analog_data,210,512,0,50);
  digital_data = digitalRead(DIGITAL_DATA);
  Serial.println(ANALOG_DATA);
  Serial.print(" ");
  Serial.println("map val is: ");
  Serial.print(map_val);

  // switch LEDs depending on analog input (distance)
  if(digital_data == 1 && map_val > 40) {
    digitalWrite(RED_LED,HIGH);
    digitalWrite(YELLOW_LED,LOW);
  } else if (digital_data== 1 && map_val <= 40) {
    digitalWrite(RED_LED,LOW);
    digitalWrite(YELLOW_LED,HIGH);
  }

    // delay 300ms for settling time
    delay(TIME_DELAY);
    digitalWrite(RED_LED,LOW);
    digitalWrite(YELLOW_LED,LOW);
}
```

## Code Walk Through

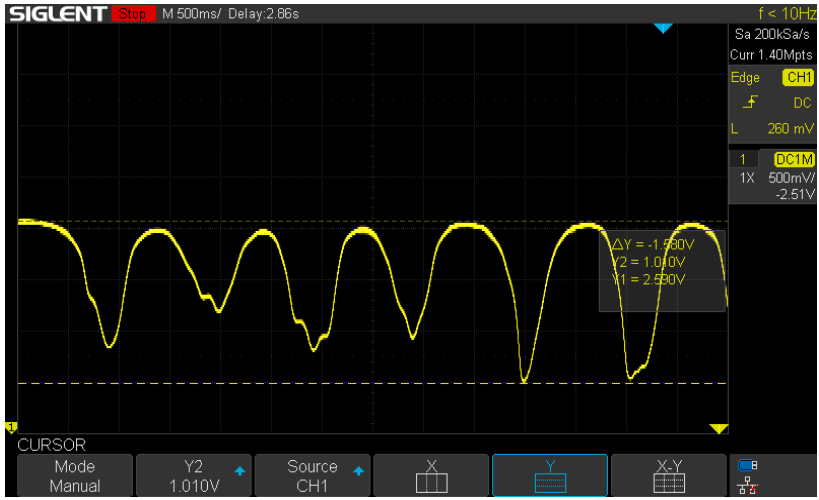
- The code starts out with the familiar ARDUINO_IDE guard. This line #define ARDUINO_IDE should be un-commented if the Arduino IDE is used.
- #defines are declared for the analog and data input pins on the Arduino. A time delay of 50ms is also defined. Note that these could have been declared as const int instead of #define.
- in setup( ), the serial monitor is initialized to 9600 baud and the

Arduino pin 2, the DATA_PIN and the LED pin modes are set.

- In loop( ) the local variables are initialized. The analog data is read from the sensor. The sensitivity of the sensor was set to 512 via the 10K pot on the sensor, which is the max or default value. When a magnetic field is detected by the Hall effect sensor the output of the analog pin A0 drops. When the magnet is very close to the sensor, the output drops to 210. This is the analog range we are dealing with. To make matters easier, the map( ) function is used. The prototype for the map( ) function is *map(value, fromLow, fromHigh, toLow, toHigh)*. The parameters are *value*, the number to map, *fromLow*, the lower bound of value range, *fromHigh*, the upper bound of the value range, *toLow*, the target lower range, and *toHigh*, the target upper range. The map( ) function returns an integer. Map is used here to make the numerical range of inputs more intuitive. It's easier to understand and deal with a range from 0 to 50 than a range of 210 to 512
- The analog data and map value is displayed on the serial monitor.
- The next block of code triggers the red LED if map_val > 40 and the output of D0 on the sensor is high. This lights the LED when the magnet is first detected by the Hall effect sensor. If the map_val <= 40 then the red LED turns off and the yellow LED turns on. This is when the magnet is close to the Hall effect sensor.
- A delay of 50ms is used to persist the LED and the LED are turned off if they are out of range.

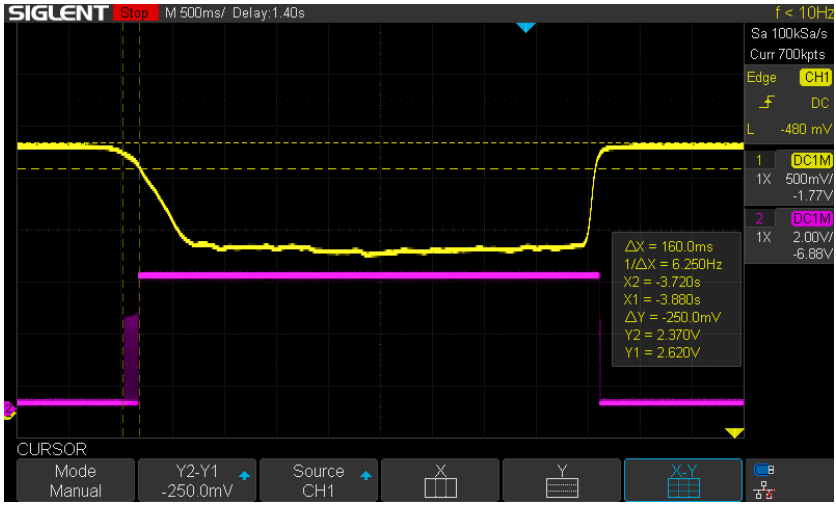
## Going Further

Here's a scope trace of the A0 output of the KY-024 Hall effect sensor. I took a small magnet and moved it toward and away from the sensor, having the red and yellow LEDs fire in sequence.

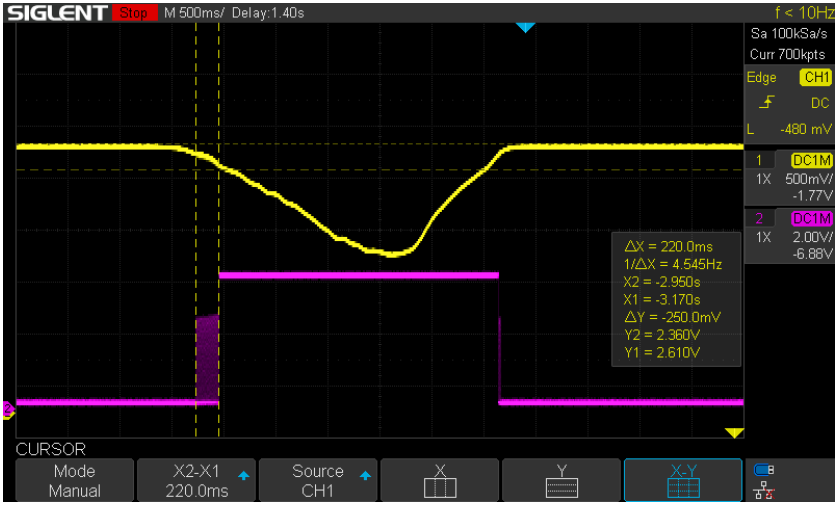



The closer the magnet was placed near the sensor, the wider the voltage range. The voltage swing in the trace is 1.580 volts. Below are three traces of the A0 analog output and the D0 digital output. The same method was used to capture the traces as above. The A0 analog output is yellow and is on channel 1 and the D0 digital output is purple and is on channel 2. The analog threshold attained via the potentiometer is at 512.

The trace below shows a fairly rapid entry and withdrawal with the magnet that energized the hall effect sensor. Notice the “grass” on the D0 output before the D0 output became high. Looking at the A0 trace, the magnet was inserted at a slower rate than it was taken out. The grassy signal before the D0 input became high measured at 160ms, which is significant. The magnet was withdrawn at a higher rate as indicated by the sharp rising voltage and the right-hand side of the scope display. There is also less grass or noise on the D0 transition to low.



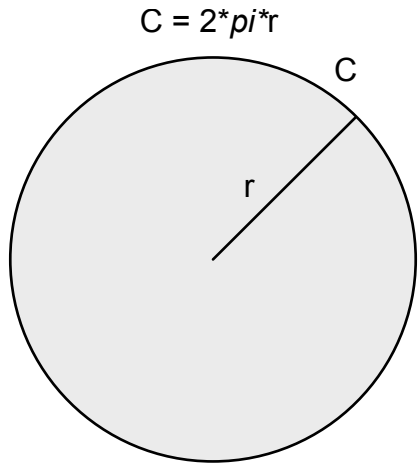

The trace below reflects the a much slower insertion and withdrawn rate of the magnet. Bear in mind, I was holding the magnet by hand so the signals will close to worse case, as opposed to being precisely inserted and withdrawn on some kind of test fixture. I would initially like to see the worst case before getting more precise in testing.




The grass or noise leading the D0 transition to high measured at 220.0ms, which again is long. Like the trace above, the magnet was

withdrawn at a faster rate. So what does this tell us? It says that Hall effect sensors, at least the KY-024 might not be the best sensor to use when trying to accurately position an object or mechanism, such as machine tool slide. What they are really good for are non-contact counting, such as an RPM meter and the like.

To use a Hall effect sensor to measure distance and rotation, we need a little basic geometry. The distance a wheel will travel in one rotation is equal to the circumference of the wheel. To determine the circumference of a wheel, measure the radius (or the diameter and divide by 2), then multiply the radius by 2 * pi, or 2 * 3.14.




The simplest way to measure rotation is to fasten a magnet at the end or the rim of the radius. If the magnet is placed at a shorter distance, then the value of r needs to be altered. As the wheel rotates, detect and count the pulses from the Hall effect sensor module output D0. Each pulse from the Hall effect sensor covers the distance of one rotation of the wheel. To get the distance, multiply the pulse count by the circumference value. The distance units will be the same as the units designated from the radius. For example, if the radius is 20in, the units will be in inches. If the radius is 50.8cm, then the units will be in centimeters.

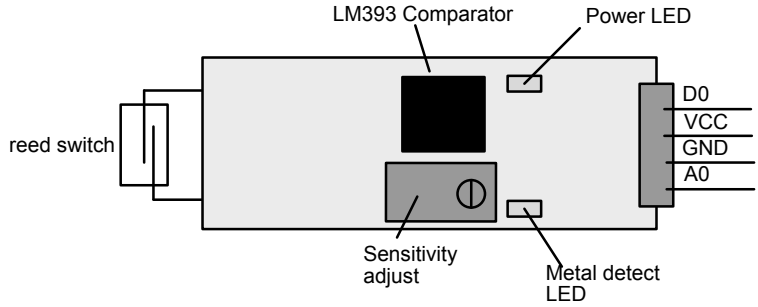
As an example, let's say a wheel has a radius of 5 inches. C = 2 times 3.14 times 5in = 31.4in. So if the wheel rotates 8 times, this yields 31.4in times 8 = 251.2 inches which is 20.93ft. As you probably suspect, over time error will be accumulated. The calculation for the distance really yields 20.933333… ft. Using 3.14 for pi introduces a small inaccuracy into the calculation. Also, we run the risk of overflowing the count

variable over time. Bear these factors in mind when you are using a Hall effect or other counting sensor when converting rotation to linear distance.

## Magnetic Spring Switch

A magnetic spring sensor, or reed switch, is a normally open switch that closes when it comes in proximity of a magnetic field. Each end of the reed switch becomes polarized the switch closes sue to attraction. Think of it as a kind of a mechanical Hall effect sensor.

The spring sensor we will use is a KY-025 reed switch module. The board is the same as the KY-024 Hall effect sensor and many more 4 input sensors, making a common electrical interface easy and manageable.

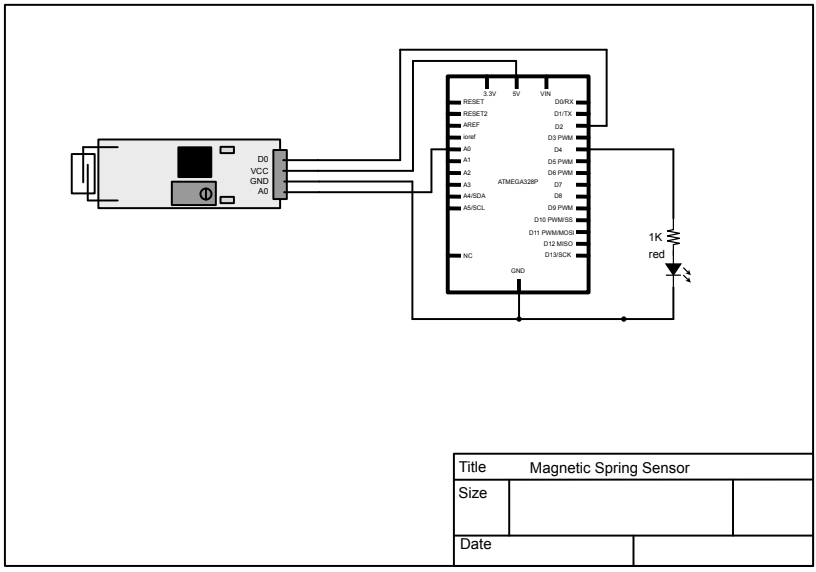



## Requirements and Comments

The requirements for this demo are:

- initially detect a magnetic object and turn on a red LED.
- turn off the red LED if the magnetic object is not detected.

## The Circuit




## Specifications

- Pin 1 or D0 outputs a 1 when a magnet is in proximity of the magnetic spring sensor.
- Pin 2 is VCC which can range from 3.3V to 5VDC.
- Pin 3 or GND is ground.
- Pin 4 or A0 is the analog output from the LM393 comparator.

## The Code

The code is almost identical to the Linear Hall sensor module.

```
/*
File: MagSpringDemo.cpp
Description: Reads a magnet in proximity of
the spring sensor or reed sensor. Turns on a red LED
if output D0 is high from the spring sensor module.
Wiring:
D0 Sensor   -> pin 2 Arduino
Red LED     -> pin 4 Arduino
Uses the serial monitor to print the output of D0.
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define DIGITAL_DATA  2
#define RED_LED       4
#define TIME_DELAY    50

void setup() {
  Serial.begin(9600);
  pinMode(DIGITAL_DATA,INPUT);
  pinMode(RED_LED,OUTPUT);
}

void loop() {
  // vars used to read and map data
  int digital_data = 0;

  // read and display analog and digital outputs from sensor.

  digital_data = digitalRead(DIGITAL_DATA);
  Serial.print("Output D0 is: ");
  Serial.println(digital_data);

  // turn red LED on or off depending on D0.
  if(digital_data == 1) {
    digitalWrite(RED_LED,HIGH);
  } else if (digital_data== 0) {
    digitalWrite(RED_LED,LOW);
  }
    // add short delay so LED stays on.
    delay(TIME_DELAY);
    digitalWrite(RED_LED,LOW);
}
```

## Code Walk Through

- Like most of the examples in this book, the code starts out with the familiar ARDUINO_IDE guard. This line #define ARDUINO_IDE should be un-commented if the Arduino IDE is used.
- #defines are declared for D0, the spring switch output as an input on pin 2 of the Arduino UNO. The red LED is an output on pin4 of the Arduino, and a time delay of 50ms is defined that is used to persist the LEDs until they are turned off by the code.
- In setup( ) the serial monitor is initialized and the pin modes are set.
- In loop( ) one local variable, digital_data is declared that is used to store the D0 input from the sensor. The D0 state is read via digitalRead(DIGITAL_DATA) and printed to the serial monitor.

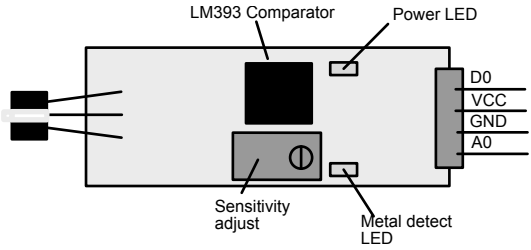
- If DIGITAL_DATA == 1 then the LED is turned on, else it's turned off.
- A 50ms delay is executed to allow the LED to stay on and visible, then turned off.

## Going Further

The magnetic spring switch works in the same contactless fashion as the Hall effect sensor, but with one drawback- it's a mechanical device that is bouncy and will wear out over time. Here's a scope trace of the switch in action.

## Metal Touch

The metal touch sensor module, sometimes referred to and the "human touch" module or touch sensitive switch, reacts to a touch, driving digital pin D0 high. The sensitivity, like many sensors, is adjusted via the 10K pot for more or less sensitivity.

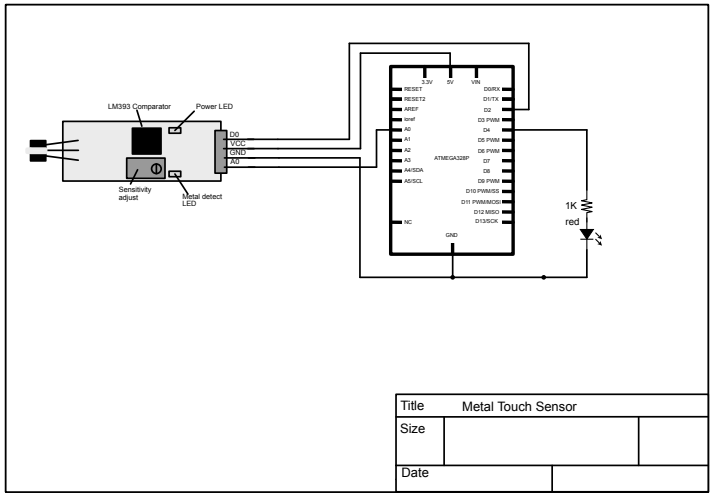



## Requirements and Comments

The requirements are identical to that of the magnetic spring switch sensor:

- initially detect a magnetic object and turn on a red LED.
- turn off the red LED if the magnetic object is not detected.

## The Circuit




## Specifications

- Pin 1 or D0 outputs a 1 when a magnet is in proximity of the metal touch sensor.
- Pin 2 is VCC which can range from 3.3V to 5VDC.
- Pin 3 or GND is ground.
- Pin 4 or A0 is the analog output from the LM393 comparator.

## The Code

```
/*
File: MetalTouchDemo.cpp
Description: Reads a magnet in proximity of
the metat touch or reed sensor. Turns on a red LED
if output D0 is high from the metal touch sensor module.
Wiring:
D0 Sensor   -> pin 2 Arduino
Red LED     -> pin 4 Arduino

Uses the serial monitor to dispay values from the sensor.
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define DIGITAL_DATA  2
#define RED_LED       4
#define TIME_DELAY    50

void setup() {
  Serial.begin(9600);
  pinMode(DIGITAL_DATA,INPUT);
  pinMode(RED_LED,OUTPUT);
}

void loop() {
  // var used to read pin D0
  int digital_data = 1;

  // read and display digital outputs from sensor.
  digital_data = digitalRead(DIGITAL_DATA);

  Serial.print("Output D0 is: ");
  Serial.println(digital_data);

  // turn red LED on or off depending on D0.
  if(digital_data == 1) {
    digitalWrite(RED_LED,HIGH);
  } else if (digital_data == 0) {
    digitalWrite(RED_LED,LOW);
  }
}
```

## Code Walk Through

Like most of the examples in this book, the code starts out with the familiar ARDUINO_IDE guard. The line #define ARDUINO_IDE should be un-commented if the Arduino IDE is used.

- #defines are declared for D0, the spring switch output as an input on pin 2 of the Arduino UNO. The red LED is an output on pin4 of the Arduino.
- In setup( ) the serial monitor is initialized and the pin modes are set.
- In loop( ) one local variable, digital_data is declared that is used to store the D0 input from the sensor. The D0 state is read via digitalRead(DIGITAL_DATA) and printed to the serial monitor.

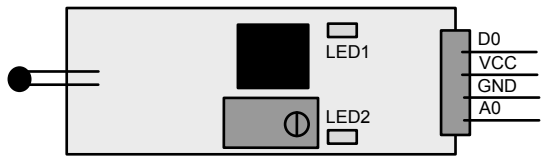
- If DIGITAL_DATA == 1 then the LED is turned on, else it's turned off.

## Going Further

There have been demo applications using this form of switch as a mechanism to stop machinery, in particular machinery that can cause injury. This I do not recommend, since I do not believe this is the most reliable and robust approach for this application. That being said, touch sensors are very common in cell phones, remote control interfaces and many IoT applications.

## KY-028 Digital Temperature Sensor

The KY-028 measures temperature via a thermistor and provides both digital and analog outputs. The thermistor input is fed into one input of a LM393 comparator, with the other connected to a potentiometer. The potentiometer is used to set a voltage threshold for the comparator to output a one or zero which is output on the D0 pin. The A0 pin is an analog output. This board follows the form factor of the typical 4 pin configuration. See the Sensor Utilization chapter section Sensor Configuration for details.

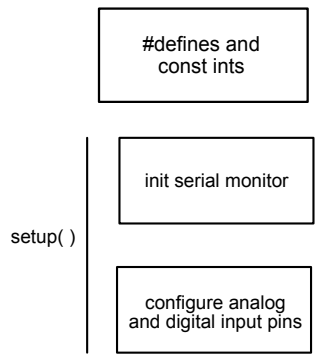


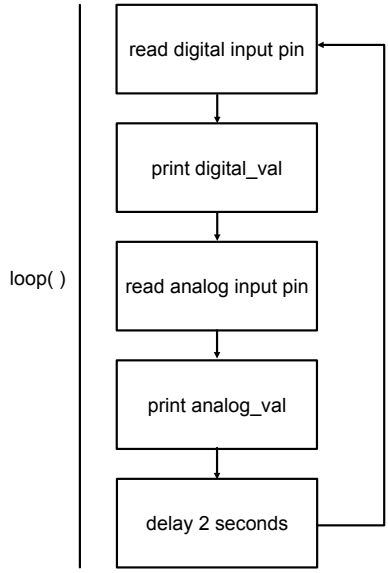

The sensor is designed for threshold measurement. When a temperature value exceeds a threshold as determined by the thermistor output and comparator voltage level, it will output a 1 or HIGH when the threshold is met. The analog output is not suitable for use with the Steinhart-Hart equation.

## Requirements and Comments

The requirements for this sketch is to read the current temperature and output a HIGH when the temperature exceeds 72 degrees F.

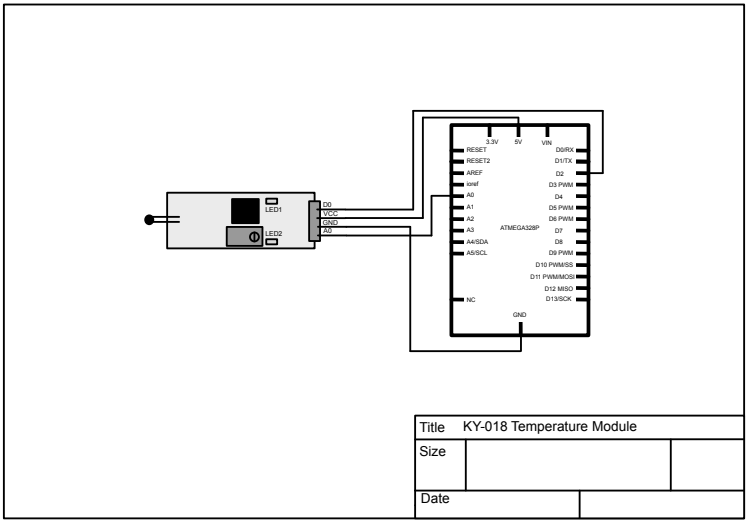
## Flowchart






## The Circuit

The circuit is similar to the other 4 pin circuit, with a power LED, LED1 and a threshold LED, LED2. A potentiometer



## The Code

```
/*
Project: KY-028DigitalTemp
Demo code for the KY-028 Digital Temperature Module
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define TIME_DELAY 2000
// define pins
const int ANALOG_PIN_028 = A0;
const int DIGITAL_PIN_028 = 2;

void setup() {
Serial.begin(9600);
while(!Serial)
  ;
pinMode(ANALOG_PIN_028,INPUT);
pinMode(DIGITAL_PIN_028,INPUT);
}

void loop() {
  int digital_val;
  int analog_val;

  // read digital input
  digital_val = digitalRead(DIGITAL_PIN_028);
  Serial.print("digital in = ");
  Serial.println(digital_val);

  // read analog input
  analog_val = analogRead(ANALOG_PIN_028);
  Serial.print("analog in = ");
  Serial.println(analog_val);
  delay(TIME_DELAY);
}
```

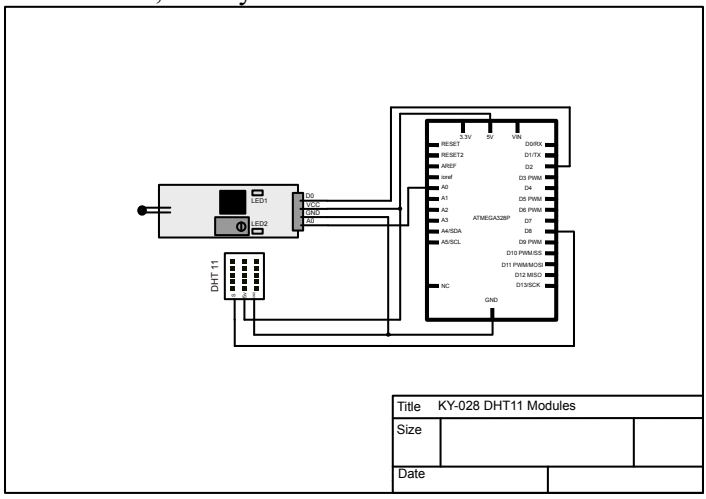
## Code Walk Through

- Like most of the examples in this book, the code starts out with the familiar ARDUINO_IDE guard. This line #define ARDUINO_IDE should be un-commented if the Arduino IDE is used.
- A time delay of 2 seconds is declared to be used as a delay between temperature readings.
- Two constant ints are declared for the analog input and the digital input from the sensor, named ANALOG_PIN_028 and DIGITAL_PIN_028, with 028 being based on the KY-028 numbers. This naming convention is useful when multiple sensors of different types are used.
- In setup( ) the serial monitor is initialized and the code waits until the serial port is available.

- The pin modes are setup with both as inputs.
- In loop( ) two int variables are declared, one for the digital input and the other for analog.
- The digital value is read from and stored in digital_val.
- The digital value is printed to the serial monitor.
- The analog value is read and stored in analog_val.
- The analog value is printed to the serial monitor.

## Going Further

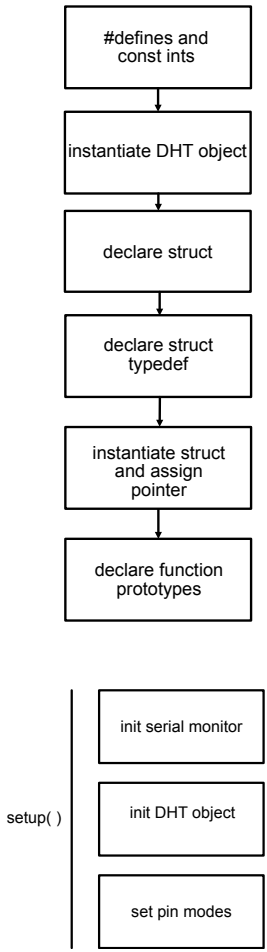
One way to correlate sensor readings is to place them side by side and determine a relationship between values. This circuit and code places the KY-018 temperature sensor and also uses the DHT11 humidity and temperature sensor, side by side.

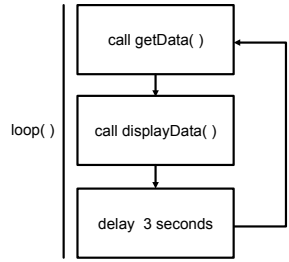



It's interesting and informative to see the readings from both sensors. It's also useful to set the digital threshold on the KY-028 sensor at a specific temperature value.

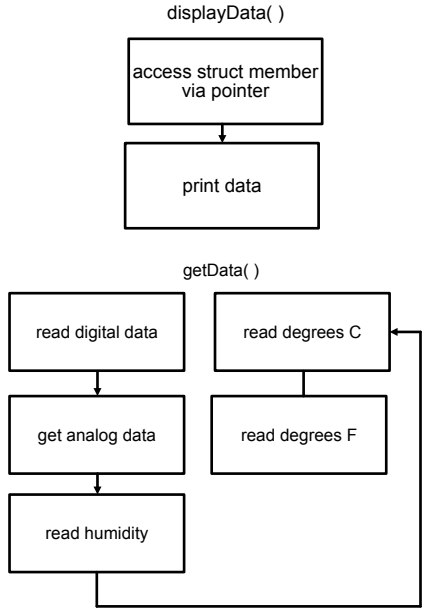
A feature of the code below demonstrates how to use a structure and structure pointers passed as arguments to functions. This is useful and memory efficient. If pointers are not used when dealing with structures in a function, an entire copy of the structure is passed to the function, making it highly memory inefficient. When a pointer is passed, all the function sees is the address of the structure, making it highly memory efficient.

## Flowchart


#defines and const ints
instantiate DHT object
declare struct
declare struct typedef
instantiate struct and assign pointer
declare function prototypes
setup( )
init serial monitor
init DHT object
set pin modes


call getData( )
loop( )
call displayData( )
delay 3 seconds

## Functions


displayData( )
access struct member via pointer
print data
getData( )
read digital data
read degrees C
get analog data
read degrees F
read humidity

## The Code

```
/*
Project: KY-028DHT11Demo
Comparison of the KY-028 temperature sensor and the
DHT11 humidity and temperature sensor outputs.
*/
//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#include <Adafruit_Sensor.h>
#endif
// DHT sensor lib header
#include <DHT.h>

#define TIME_DELAY 3000
// define pins
const int ANALOG_PIN_028  = A0;
const int DIGITAL_PIN_028 = 2;
const int DHT_DATA_PIN    = 8;

#define DHTTYPE DHT11
DHT dht(DHT_DATA_PIN,DHTTYPE);

struct sensordata {
  int KY_digital_val;
  int KY_analog_val;
  float DHT11_humidity;
  float DHT11_temp_c;
  float DHT11_temp_f;
};

// global typedef for sensor struct
typedef struct sensordata SensorData;
// declare instantiation and pointer to structure
SensorData s;
SensorData *sptr = &s;

// fn prototype to display data
void displayData(SensorData *);
// fn prototype to assign stucture values
// note th argument is a pointer *
void getData(SensorData *);

void setup() {
  Serial.begin(9600);
  while(!Serial)
    ;
  dht.begin();
  pinMode(ANALOG_PIN_028,INPUT);
  pinMode(DIGITAL_PIN_028,INPUT);
}

void loop() {
  getData(sptr);
  displayData(sptr);
  delay(TIME_DELAY);
}
```

```
void displayData(SensorData *s) {
  Serial.print("KY-028 digital val = ");
  Serial.println(s->KY_digital_val);
  Serial.print("KY-028 analog val = ");
  Serial.println(s->KY_analog_val);
  Serial.print("DHT11 temp C = ");
  Serial.println(s->DHT11_temp_c);
  Serial.print("DHT11 temp F = ");
  Serial.println(s->DHT11_temp_f);
}

void getData(SensorData *s) {
  // read digital input from KY-28 sensor
  s->KY_digital_val = digitalRead(DIGITAL_PIN_028);
  // read analog input from KY-28 sensor
  s->KY_analog_val = analogRead(ANALOG_PIN_028);
  // uncomment to read read DHT11 humidity
  // s->DHT11_humidity = dht.readHumidity();
  // read DHT11 temp C
  s->DHT11_temp_c = dht.readTemperature();
  // read DHT11 temp f
  s->DHT11_temp_f = (s->DHT11_temp_c * 1.8) + 32;
}
```

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples

; https://docs.platformio.org/page/projectconf.html
[env:uno]
platform = atmelavr
board = uno
framework = arduino
lib_deps = adafruit/DHT sensor library@^1.4.4
```

## Code Walkthrough

This code employs structures and pointers to structures passed as function parameters. The discussion will be brief, but in the section below, Structure/Pointer Template, bare-bones code describing how structures and pointers to structures are given a detailed explanation. The code given in the Structure/Pointer Template can be the foundation for you to use when deciding to use structures to organize your variables.

- First the #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is un-commented. The Adafruit_Sensor.h header file is also included here.
- The DHT.h header file is included. This contains the methods used to access the DHT11 sensor.
- A time delay of 3 seconds is declared. This is the time between sensor readings.
- The pins are declared for the KY-28 and the DHT11. The

variables with KY as a prefix are for the KY-38 sensor, and the DHT11 prefixes are used for the DHT11 sensor.

- A struct sensordata is declared to hold the values received from the sensors. The first two are for the KY-28 sensor and the last three are for the DHT11.
- A typedef is declared for the sensordata named SensorData. This is not required but makes the code more manageable and easy to read.
- The structure is instantiated by this line, SensorData s;. The structure is now ready for use.
- This lines declares a pointer, *sptr, to the address of s. The &before a variable means that the address will be acted on and not the data the address holds.
- A function prototype, void getData(SensorData *); is declared. The argument contains a pointer to the SensorData type.
- In setup( ) the serial monitor is initialized and the code waits until the serial port is available via the while statement. The DNT11 object is instantiated via dht.begin( ). This line of code is easy to forget, and the DHT11 will not work without it, so be careful to include it.
- The pin modes are next, where the pin modes for the KY-28 analog and digital pins are set for inputs.
- In loop( ), the getData(sptr) function is called, with sptr passing the address of the structure as a parameter. Note there is not a * or & preceding the parameter sptr. This is particular to functions that pass pointers as addresses.
- the displayData(sptr) function is called, and like the getData(sptr) function, the pointer to the structure is passed as a parameter.
- A delay of 3 seconds is used to introduce a delay between sensor readings.
- The function displayData(SensorData *s) takes the pointer, or address, of the structure s as a parameter. Inside the function the -> operator is used to assign values to the variables inside the structure s.
- The function getData, like displayData, takes the structure pointer as a parameter. The data is read from the KY-28 and the DHT11 sensors in sequence using the -> operator. The temperature value returned from the DHT11 sensor via the DHT library defaults to degrees Celsius. The next line converts

degrees C to degrees F.

As said before, the code below is walked through very carefully, explaining structures, pointers to structures, and how they are used.

## Structure/Pointer Template

Here's a code template that you can use for your projects. We will carefully walk through the code.

```
/*
Project: StructPtrDemo
This code demonstrates how to use structures
and pointers to structures passed as function
arguments. This demo uses global vars that can
be localized in main.
Steps:
1 - declare global structure
2 - declare global typedef
3 - declare function prototype
4 - in main() declare local struct of typedef
5 - declare pointer of structure type
6 - call function. The parameter(s) has a pointer *
    and the function uses the -> construct to assign values.
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define DELAY_TIME 1000

// 1 - declare global structure
struct sensordata {
  int sval_1;
  int sval_2;
};

// 2 - declare global typedef
typedef struct sensordata SensorData;
// 3 - declare function prototype
// Note the argument is a pointer *
void setData(SensorData *s);
 // 4 - declare local struct of type SensorData
SensorData s;
// 5 - declare pointer to type SensorData
// Note the * and & operators
SensorData *sptr = &s;

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

void loop() {
  //6 - call function. The parameter(s) has a pointer *
  //    and the function uses the -> construct to assign values.

  setData(sptr);
  Serial.print("s.val_1 = ");
  Serial.println(s.sval_1);
  Serial.print("s.val_2 = ");
  Serial.println(s.sval_2);
  delay(DELAY_TIME);
}

// setData - uses a structure pointer (address)
// to access and change structure member values
void setData(SensorData *s) {
  s->sval_1 += 1;
  s->sval_2 += 2;
  // reset when number gets to 100
  if(s->sval_1 >= 100)
    s->sval_1 = 1;
  if(s->sval_2 > 100)
    s->sval_2 = 2;
}
```

## Code Walkthrough

- First the #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is un-commented.

- The comments at the head of the code describes a 6 step process when using structures, typedefs and pointers. In Step 1 a global structure is declared. This can be changed to local structures that are declared in loop( ) but many times a in small sketches, global variables, such as structs, are fine.
- In Step 2 a global typedef is declared. Typedefs allow you to declare your own data type. This makes it easier to read code.
- In Step 3 function prototypes are declared for the functions that will act on the structure.
- In Step 4 inside loop( ) (or main( ) in a regular C program) a local structure is declared using the typedef.
- In Step 5 a pointer is declared to point to the typedef.
- In Step 6 the functions are called using a pointer to the typedef or struct.
- Getting back to the code, a delay of 1 second is declared. The only reason for this is to not have the structure values fly by on the screen in the serial monitor.
- The next line of code is Step 1, declaring the structure. The struct, sensor data, contains two integer variables. These can be of any type that is conducive to any particular sensor.
- The next line of code is Step 2. This is where the typedef is declared. All this code does is define a data type, SensorData, based on the structure sensordata. The main reason for this is to just refer to the structure as SensorData, which is easy to understand and uncluttered.
- The next line of code is Step 3. This is where the function prototypes are declared that access the structure. There of course can be many function prototypes declared. Note the * operator in the function parameter. This tells the compiler that a pointer, or the address of a variable, is going to be passed into the function. This is called “call by reference”. If a star wasn’t present, the value of the variable would be passed into the function. This is called “call by value.”
- The next line of code is Step 4. This is where the SensorData object is declared. In our case, it’s called s.
- The nedt line of code is Step 5 which is critically important. A pointer variable, *sptr is assigned the address of s via the operator &. Pointer variables hold addresses and not data. The line SensorData *sptr = &s first says that *sptr is a pointer to the typedef SensorData, and the = &s makes the pointer variable

point to the address of s, which we declared of type SensorData in the previous line. *sptr now allows us direct access to s and what's inside.

- Inside setup( ) the serial monitor is initialized to 9600 baud.
- Inside loop( ) the first line of code encountered is Step 6, which is the call to the function setData(sptr). The pointer variable *sptr is used, but without the star. This is confusing in C but remember that this is the way to pass pointer variables to functions.
- The next flew lines print the value of the structure, set by the call to setData(sptr).
- A delay of DELAY_TIME, which is 2 seconds, is used to not make the values flash by on the screen.
- In setData( ) the function expects a pointer to the typedef SensorData to be passed in (SensorData *s).
- The next two lines of code sets the values of the structure member variables, sval_1 and sval_2, to what they are plus one and two respectively. Remember we are working with pointers, which are addresses, so the dot operator cannot be used. The -> is used to set structure member values when pointer are used. If the values of the variables are equal to or exceed 100, the variables reset to their initial values.

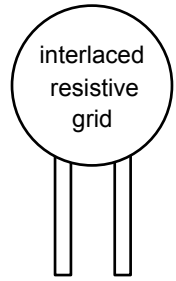
Structure with pointers take some getting used to, but use this code as a starting point for your own sketches and move on from there. Using pointer with structures are, as said before, very memory efficient, since it takes up much less space on the stack to store an address rather than a complete copy of a structure.

# CHAPTER SEVEN
## Specialized Sensors

This section covers a variety of specialized sensors.

### FSR Pressure Sensors

FSR stands for Force Sensitive Resistor. The essence of an FSR is that the resistance drops as pressure is applied to the sensor pad. Since an Arduino cannot directly read resistance, a voltage divider is used to create a voltage that varies with resistance. FSR sensors come in many shapes and sizes, but the most common are circular. All have an internal interlaced resistive grid. The illustration below depicts an RP-C pressure sensor. These are circular with two internal interlaced traces.

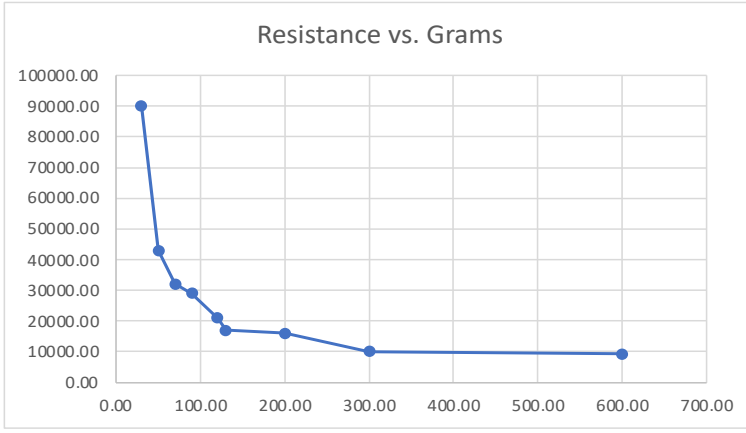



FSR sensors are transducers, where a physical quantity is the input and an electrical quantity is the output. With an FSR, the input is a mechanical force (weight, pressure) and the output is a variable resistance. They consist of three components, a flexible substrate that's printed with a semiconductor material, a spacer containing adhesive, and a flexible substrate that printed with interlacing traces or electrodes.

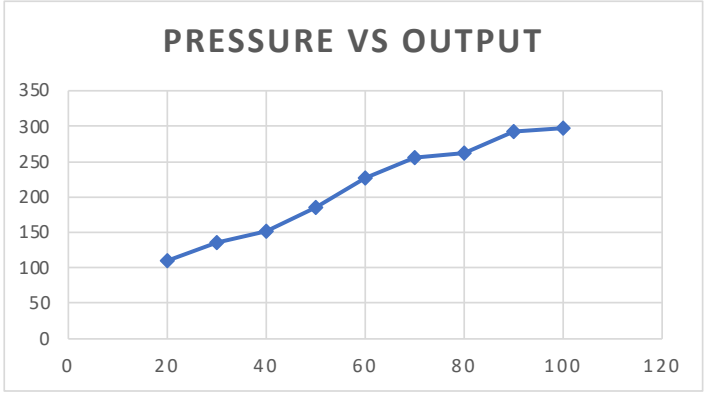
FSR sensors are used in robotic end effector applications, detecting

localized forces on a localized surface, such as patient movement on a hospital bed, automotive seat occupancy detection among many other applications that require force or pressure measurement. FSR sensors are inexpensive, but are not precise. Also, FSR sensors vary from sensor to sensor. If an application requires low resolution force measurement, then an FSR sensor will do the trick. For more precise and consistent measurements, a more expensive transducer will be the better choice.

The FSR sensor used for this demonstration is an RP-C type, in particular an RP-C10-ST. To determine the resistive characteristics versus pressure a small precision scale was used. A small kitchen scale works well. In the graph below, the Y axis is in K-Ohms and the X axis is force applied in grams.




The curve is logarithmic, as resistance drops radically as pressure is applied. FSR sensors often use the term “trigger” to describe the weight that causes the sensor to react. For the RP-C10-ST a trigger weight of 20g produces a resistance. A non-triggering resistance, which generally means no force is applied to the sensor, is greater than 10M-Ohms. The pressure range for the RP-C10-ST is from 20g to around 2Kg. The response time is less than 10ms and the measurement value drifts less than 5 percent. The next step is to connect the sensor




## Mass, Weight and Inertia

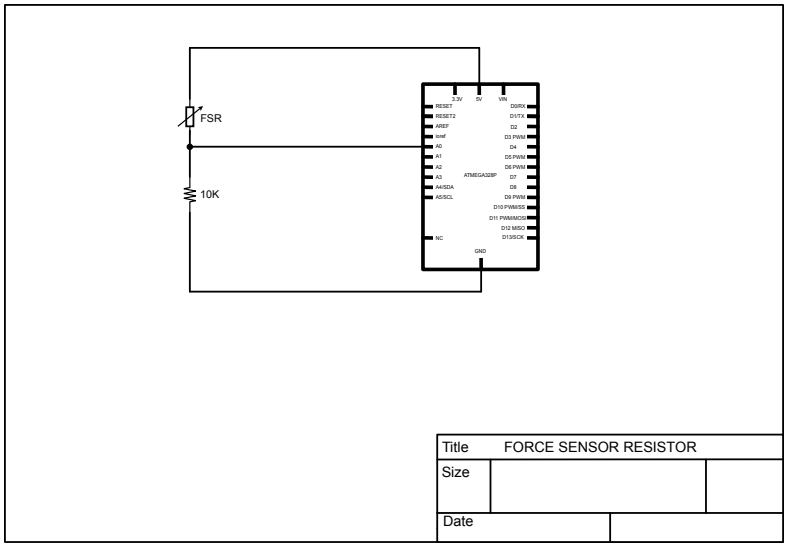
To be technically accurate, grams are unit of mass and not force. Mass and weight are not the same. Mass is also not "bulk" or how much space an object takes up or its density. Mass describes the inertial properties of a body. The greater the mass, the more work it takes to overcome inertia. Weight is a force that is exerted on a body, either by gravity or other means. Force in the metric systems is described by Newtons. One Newton is the quantity of net force that yields an acceleration of one meter per second squared to a body of a mass of 1Kg. To convert grams to Newtons (abbreviated N), the conversion factor is:

**1 gram = 0.0098 Newtons**

For example, in the chart above, 300g is equal to 300 x 0.0098 = 2.94N. Twenty grams, the weight threshold of the RP-C10-ST sensor is 20g is equal to 20 x 0.0098 = 0.96N.

## The Circuit

The Arduino UNO does not directly read resistance, so this is where voltage dividers come in handy. The circuit consists of an RP-C10-ST configured in a voltage divider.



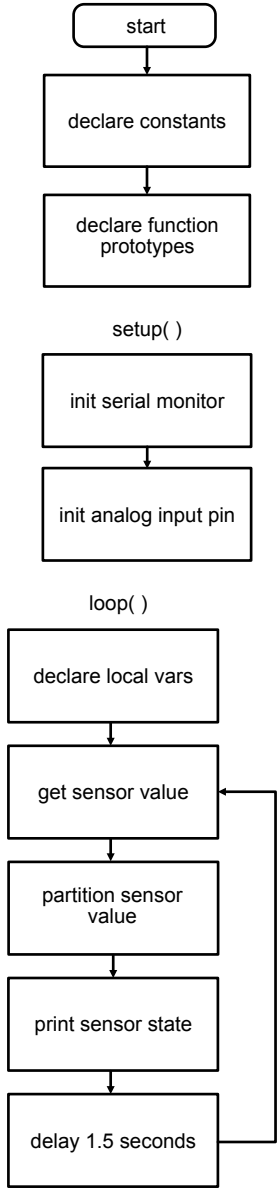

One of the big issues with a voltage divider is that it cannot drive a load, that is, it cannot provide current to an external circuit. The voltage divider formula,

$$Vout = Vin\left(\frac{R2}{(R1 + R2)}\right)$$

fails when additional circuits are added to the *Vout*. So how can we get away with wiring Vout to an Arduino UNO analog input pin and not seeing an adverse effect? Thankfully, the analog input pins have a very high impedance of about 100 megohms. Like connecting a voltage divider to an op amp, the high impedance of the input pin will have little or no effect on the voltage divider since it will not be sinking current. This is immensely helpful when directly using voltage dividers with an Arduino UNO.

**Flowchart**

Here's flowchart for the example code.



## The Code

```
/*
Project: FSRDemo
Demonstrates FRS resistive sensors.
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

// uncomment to get sensor data to
// manually get range values
//#define GET_RANGE_VALUES
#define DELAY_TIME 1500
const int A_IN_PIN = A0;

// high and low sensor range
const int LOW_VAL = 0;
const int HIGH_VAL = 800;
// range values
const int LOW_PRESSURE = 266;
const int MID_PRESSURE = 532;
const int HIGH_PRESSURE = 798;

// function prototypes
int readSensor(void);
int partitionValues(int);

void setup() {
    Serial.begin(9600);
    pinMode(A_IN_PIN,INPUT);
}
```

Continued on next page…

```
    void loop() {
      int sensor_value = 0;
      int partition_val = 0;

      sensor_value = readSensor();
      partition_val = partitionValues(sensor_value);
      switch(partition_val) {
        case 0: // high pressure
          Serial.println("HIGH PRESSURE");
        break;

        case 1:
          Serial.println("MID PRESSURE");
        break;

        case 2:
          Serial.println("LOW PRESSURE");
        break;

        default:
        break;
      }
      delay(DELAY_TIME);

    #ifdef GET_RANGE_VALUES
      int s_val = 0;
      s_val = analogRead(A_IN_PIN);
      Serial.println(s_val);
      delay(DELAY_TIME);
    #endif
    }


// readSensor - reads the raw analog sensor input
int readSensor(void) {
  int sensor_value =  analogRead(A_IN_PIN);
  return(sensor_value);
}

// partitionValues -
// returns one of three pressure states
// 0 = low pressure
// 1 = mid pressure
// 2 = high pressure
int partitionValues(int val) {
  if(val>= MID_PRESSURE) {
    return 0;
  }
  if(val < HIGH_PRESSURE && val > LOW_PRESSURE) {

    return 1;
  }
  if(val < MID_PRESSURE) {  // low pressure
    return 2;
  }
  // return -1 if bad param
    return -1;
}
```

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples

; https://docs.platformio.org/page/projectconf.html

[env:uno]
platform = atmelavr
board = uno
framework = arduino
```

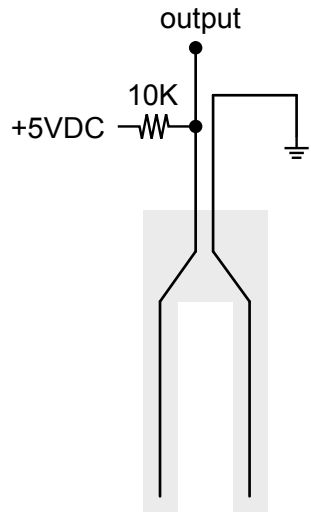
## Code Walk Through

- The #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is un-commented.
- The #define GET_RANGE_VALUES is commented out for the normal execution of the program. When uncommented, this is used to get the range values of a particular FSR, like in the cases of the resistive and capacitive moisture sensors.
- #define DELAY_TIME 1500 provides a 1.5 second time value. A const int declaration, A_IN_PIN is declared for pin A0 on the Arduino UNO for the analog input from the FSR.
- Two const ints, LOW_VAL and HIGH_VAL are declared to hold the low and high values of the FSR sensor used. These values are determined by experimentation.
- Three const ints, LOW_PRESSURE, MOD_PRESSUE and HIGH_PRESSURE partition the range of the FSR into three partitions. For the particular RP-C10-ST sensor used in this example, the range was from 0 to 800. A partition value of 266 was used as a divisor, resulting in the values of 266, 532 and 798.
- Function prototypes are declared for int readSensor(void) and int partitionValues(int). These functions are used to read the sensor data and to partition the sensor data into three categories.
- In setup( ), the serial monitor is initiated and the pin modes are set for the analog and digital inputs.
- In loop( ), two integer variables are declared, int sensor_value and partition values. These are used to store the raw sensor value reading and the resulting partition values.

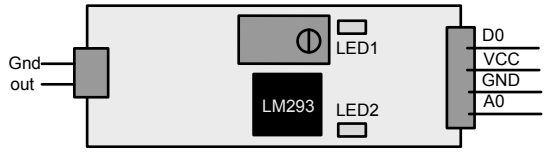
- The function readSensor( ) is called and the return value is stored in the int variable sensor_value.
- The sensor_value variable is passed as an argument to the function partitionValues( ). This function returns one of three integer values, 0, 1, or 2, representing low, medium and high pressure sensed from the FSR. This value is stored in the int variable partition_val.
- The variable partition_val is used in a switch statement. One of the three integer values determines which case statement is executed, resulting in HIGH_PRESSURE, MID_PRESSURE or LOW_PRESSURE being printed to the serial monitor.
- The function delay(DELAY_TIME) is called to delay the next iteration through the loop by 1.5 seconds.
- If the #define GET_RANGE_VALUES is uncommented, the next block of code is executed. This reads the incoming value on the Arduino UNO pin A0 and prints it to the serial monitor.
- The function int readSensors(void) reads the value on the analog pin A0, stores and returns the value in the int variable sensor_value.
- The function int partitionValues(int val) takes the partition value as input and through a series of if statements returns one of three constant values, 0,1, or 2, depending on the values of the pressure constants.

## YL-69 Resistive Moisture Sensor

The YL-69 hygrometer is based on a simple voltage divider fed into an LM293 comparator and is commonly called a soil moisture sensor. The sensor schematic is below, but note that the 10K resistor is on the module that accompanies the sensor. The YL-69 is a simple and accurate sensor but has a few caveats. It corrodes over time when left in soil and reacts as an electrolysis device, reacting with the elements within the soil.



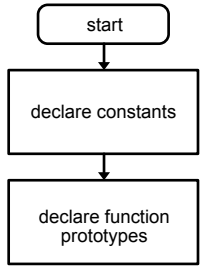

Along with the sensor comes the general purpose 4 pin HC-38 interface board that's described in the Sensor Configuration section. The 10K pot is used to adjust the digital output threshold. LED1 indicates power and LED indicates the state of the digital output.



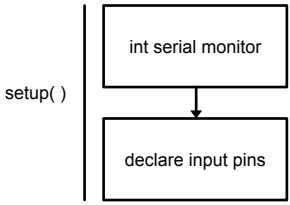

D0 is the digital output, VCC is 5VDC usually powered by the Arduino, GND is ground and A0 is the analog output value. This is what we are interested in.

## Flowchart

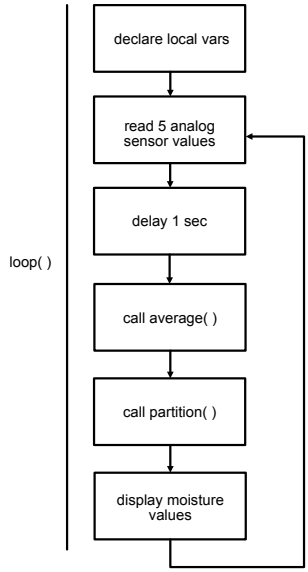
Const ints are declared before entering setup( ) and loop( ).




The serial monitor is initialized and the input pins are declared in setup( ).

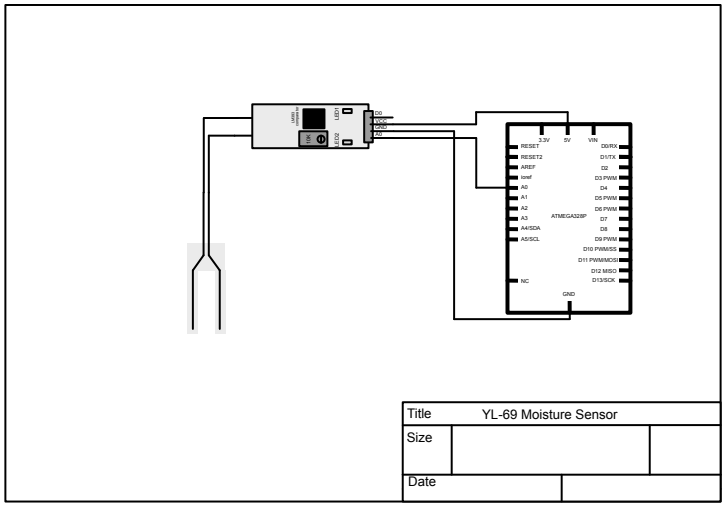



Local vars are declared, 5 sensor values are read and averaged, and the value is is partitioned and displayed into three categories, “dry”, “moist”, or “wet moist”.




## The Circuit

The circuit consists of an Arduino UNO, an HC-38 4 pin sensor interface card and the YL-69 moisture sensor. The HC-38 outputs a value from 0 to 1023. A value of 1023 represents dry or open resistance and a value of zero represents full wet or near zero resistance. From this range partitions can be calculated. In the code presented here three partitions are used. Dupont cables are used to connect the sensor to the HC-38 board.




## The Code

```
/*
Project: RMoistureSenor
Resistive moisture sensor demo using the YL-69 hygrometer.

Takes a sample of 5 readings and averages
the readings.
*/
//#define ARDUNIO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define DELAY_TIME 1000
const int D_IN_PIN = 2;
const int A_IN_PIN = A0;
const int ARRAY_SIZE = 5;
// high and low sensor range
const int LOW_VAL = 0;
const int HIGH_VAL = 1023;
const int PERCENT_VAL = 100;
// range for sensor output partitions
// change these for particular sensor
const int DRY_RANGE = 683;
const int MOIST_RANGE = 341;

// average function prototype
int average(int[]);
int partitionValues(int);

void setup() {
  Serial.begin(9600);
  pinMode(D_IN_PIN,INPUT);
  pinMode(A_IN_PIN,INPUT);
}

void loop() {
  int dat_array[ARRAY_SIZE];
  int average_val = 0;
  int range_val = 0;
  int percentHumidity = 0;
  // uncomment to utilize digital value
 // int dig_val = 0;
```

```
      // read sensor
       for( int i = 0; i < ARRAY_SIZE; i++) {
         dat_array[i] = analogRead(A_IN_PIN);
         // uncomment to read digital value from sensor
         //dig_val = digitalRead(D_IN_PIN);
         delay(DELAY_TIME);
       }
       average_val = average(dat_array);
       range_val = partitionValues(average_val);


   // display moisture status
     switch(range_val) {
       case 0:
         Serial.println("Dry");
         break;

       case 1:
         Serial.println("Moist");
         break;

       case 2:
         Serial.println("Wet/Moist");
         break;

       default:
         break;
     }
   /* remove comments to see sensor values
     Serial.print("Average moisture value is ");
     Serial.println(average_val);
     Serial.print("range_val is ");
     Serial.println(range_val);
   */
   }

// average - averages and returns the array values

int average(int val_array[]) {
  int ave = 0;
  for(int i = 0; i<ARRAY_SIZE; i++) {
    ave+=val_array[i];
  }
  ave = ave/ARRAY_SIZE;
  return ave;
}

   // partition values - returns one of 3 hydro states
   // 0 = dry       < 682
   // 1 = moist     342 - 682
   // 2 = wet/moist 0 - 341
   int partitionValues(int val) {
     if(val >= DRY_RANGE) // dry
       return 0;
     if(val < DRY_RANGE && val > MOIST_RANGE) // moist
       return 1;
     if(val < MOIST_RANGE) // wet/moist
       return 2;
     // return -1 if bad param
     return -1;
   }
```

## Code Walk Through

- The #define ARDUINO_IDE is used to switch between using the Arduino IDE or PlatformIO. If the Arduino IDE is used, then the file <ardruino.h> is not included.
- A delay time of 2 seconds is defined. This delays the readings for no other reason than make the readings less jumpy on the serial monitor.
- A delay tine of 2 seconds is defined. This delays the readings for no other reason than make the readings less jumpy on the serial monitor.
- An array of size 5 is declared to hold five successive readings to be averaged.
- The next five const ints are designed around particular sensors. For the YL-69 sensor, the output values range from 0 to 1023. The first two const ints define the lower and upper boundaries of the sensor range. Other sensors, like capacitive moisture sensors, output a different range. These values should reflect the low to high values output by any particular sensor. The next const int is for the top percentage value, which is 100. A const int here was used just for good form not to use a discrete value in the body of the code. The next two const ints are used as partition values to discriminate between dry, moist and wet/most, which change from sensor type to sensor type. Changing these values make substituting sensor types, such as resistive and capacitive moist sensors painless.
- An int function prototype, average(int[]) is declared with an array as a parameter. This function is used to take the 5 successive readings, average them, then return the average.
- An int function partitionValues(int) is declared with an int parameter.
- In setup( ) the serial monitor is initialized at 9600 baud. The pin modes are set for INPUT for the digital and analog pins.
- In loop( ) the array, dat_array[ARRAY_SIZE] is declared. This is the array that holds the sensor readings. Two int variables are declared, dig_val to hold the digital value from the sensor and average_val, which holds the average of the 5 successive sensor readings.
- The int dig_val variable is commented out. If an application needs to utilize the digital output of the sensor, then uncomment this.

- A for loop iterates over the sensor readings that are directly loaded into the array. The iteration terminates at i < ARRAY_SIZE, which is 4, and not 5. In C, arrays start at index 0 and not 1. If the iteration was i <= ARRAY_SIZE then the array would exceed its boundary value. In this application, the digital value from the sensor is not used, but if need be, the digitalRead( ) statement can be uncommented.
- A 2 second delay is utilized between the readings.
- Outside of the for loop the average(dat_array) function is called. The return value is stored in the variable average_val. Note that the brackets for the array are not used in the parameter parentheses. This is used to pass the address of the array to the function.
- The function partitionValues(average_val) is called using the previously calculated average_val as a parameter. The partitionValues( ) returns one of three values, 0, 1 or 2. If 0 is returned, the sensor is dry, if 1 the sensor is moist, and if 2 the sensor is wet/moist. The value is stored in the int variable range_val.
- The precent humidity is calculated using the map( ) function. The map( ) function "maps" one set of numerical values to another. The first parameter is the value that is within the range of values. First the map function is called with the previously calculated average value, the full range of the sensor, and the range we want the full range to map to, which is 0 to 100. Since the resistance lowers as the moisture level rises, the percent value is in reverse so it needs to be subtracted from 100 to get the correct percentage. The abs( ) function is called to get rid of the minus sign.
- The percent moisture value is then printed to the serial monitor.
- The range value is used is a switch statement. Depending on the range value, the state of the sensor is printed in the cases. The default case is not really required but it is good form to include a default case in any switch statement.
- The next lines print the average value of the 5 successive readings and the range value. Uncomment this to see the values if desired.
- The function int average(int val_array[ ]) contains a local int variable, ave, the holds the average value to be returned. It is initialized to zero every time the function is called.

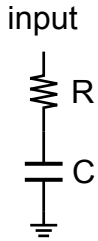
- A for loop reads the array values and sums them via the line of code ave+=val_array[I].
- Outside of the for loop the average is calculated and returned.
- The function partitionValues(int val) partitions the average value returned by the average( ) function into three values, 0 , 1 and 2. The YL-69 resistive sensor returns a value between 0 and 1023, where 1023 is dry (open resistance) and 0 is completely saturated (0 resistance). The range is divided by 3 to get three numerical ranges in a series of if statements. If a value lies in one of the ranges, the function returns one of the values. If an out of range parameter is passed, -1 is returned. Note that the const int values are used so not to alter the body of the code with discrete numerical values.

## Going Further

Adding an LCD display with pushbuttons. The SELECT button clears the LCD and takes a reading and displays ‘wet/moist’, ‘moist’ or ‘dry’. The LEFT button shows a bar graph indicating wet/moist, moist or dry, where wet moist is 16 cells, moist is 8 cells, and dry is 5 cells.

## HW-390 Capacitive Moisture Sensor

In contrast to resistive moisture sensors which are basically a voltage divider, a capacitive moisture sensor is fundamentally an RC network.

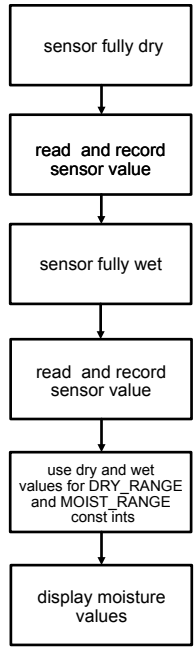



With capacitive moisture sensors, the capacitor is comprised of two conductive plates that act as a capacitor. This is called parasitic capacitance. If the conductive plates are sufficiently large, the parasitic capacitance charging time will change in response to environmental factors. Immersed in water, the capacitance will increase which in turn takes more time to charge. In a dry environment, the capacitance will decrease and take less time to charge.

The capacitive sensor receives a pulse train from a 555 timer near the sensor's connector. The connector has pins for Gnd, 5VDC, and the analog output, AOUT. The pulse train from the 555 time provides the energy to the RC network which allows a rise time measurement of the RC network. The result of this is provided on the analog output pin, AOUT.

## Flowchart

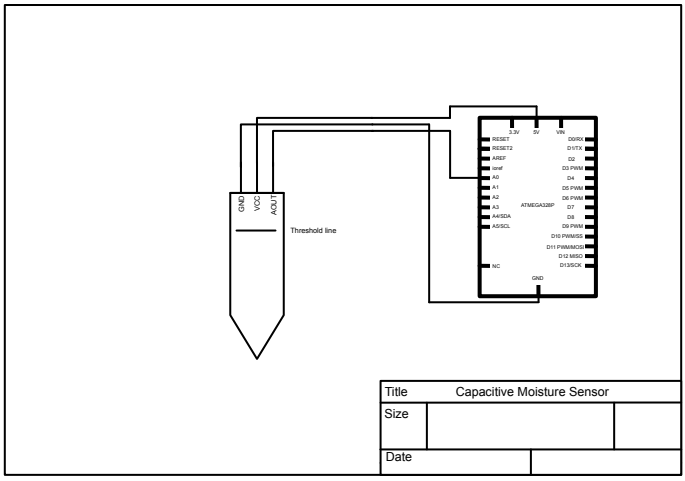
The flow chart is essentially the same as the resistive moisture sensor. The flowchart shown here describes the manual range function.




The sensor value is read fully dry and the value is recorded. The sensor is read fully wet and that value is recorded. These values define the range of the sensor used and need to be stored in the const ints DRY_RANGE and WET_RANGE as these values are used by the map( ) function to determine the humidity level.

## The Circuit

The circuit is simple with a three wire connector. To connect the sensor to an Arduino UNO, three male-to-male Dupont wires are required.




## The Code

```
/*
Project: CMoistureSenor
Capacitive moisture sensor demo using the HW-390
Takes a sample of 5 readings and averages
the readings.
*/
//#define ARDUNIO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define DELAY_TIME 1000
//const int D_IN_PIN = 2;
const int A_IN_PIN = A0;
const int ARRAY_SIZE = 5;
// high and low sensor range
const int LOW_VAL = 198;
const int HIGH_VAL = 510;
const int PERCENT_VAL_TOP = 100;
const int PERCENT_VAL_BOTTOM = 0;
// range for sensor output partitions
// change these for particular sensor
const int DRY_RANGE = 312;
const int MOIST_RANGE = 208;

// function prototype
int average(int[]);
int partitionValues(int);
int getRange(void);
void calcHumidity(int);

void setup() {
  Serial.begin(9600);
 // pinMode(D_IN_PIN,INPUT);
  pinMode(A_IN_PIN,INPUT);
}
```

```
void loop() {
  int dat_array[ARRAY_SIZE];
  int average_val = 0;
  int range_val = 0;
  // uncomment to utilize digital value
  // int dig_val = 0;
  // use this function to get initial wet and dry
  // readings for a particular sensor. Uncomment
  // to use the function.
 // getRange();
  // read sensor
  for( int i = 0; i < ARRAY_SIZE; i++) {
    dat_array[i] = analogRead(A_IN_PIN);
    // uncomment to read digital value from sensor

    //dig_val = digitalRead(D_IN_PIN);
    delay(DELAY_TIME);
  }
  average_val = average(dat_array);
  range_val = partitionValues(average_val);
  calcHumidity(average_val);


// display moisture status
  switch(range_val) {
    case 0:
      Serial.println("Dry");
      break;

    case 1:
      Serial.println("Moist");
      break;

    case 2:
      Serial.println("Wet/Moist");
      break;

    default:
      break;
  }

 //remove comments to see sensor values
 /*
  Serial.print("Average moisture value is ");
  Serial.println(average_val);
  Serial.print("range_val is ");
  Serial.println(range_val);
  */
}

// average - averages and returns the array values

int average(int val_array[]) {
  int ave = 0;
  for(int i = 0; i<ARRAY_SIZE; i++) {
    ave+=val_array[i];
  }
  ave = ave/ARRAY_SIZE;
  return ave;
}
```

```
// partition values – returns one of 3 hydro states
int partitionValues(int val) {
  if(val >= DRY_RANGE) // dry
    return 0;
  if(val < DRY_RANGE && val > MOIST_RANGE) // moist
    return 1;
  if(val < MOIST_RANGE) // wet/moist
    return 2;
  // return -1 if bad param
  return -1;
}

// calcHumidity – calculates and displays percent humidity
// using the map function
void calcHumidity(int ave_val) {
  int percentHumidity = map(ave_val,LOW_VAL,HIGH_VAL,
                        PERCENT_VAL_BOTTOM, PERCENT_VAL_TOP);
  if(percentHumidity >= 0)
    percentHumidity-=100;
  else if(percentHumidity < 0)
    percentHumidity+=100;
     percentHumidity = abs(percentHumidity);
     Serial.print(percentHumidity);
     Serial.println("%");
}

// getRange – get range of sensor
// This is a manual process that allows
// a user to get the min and max values
// of a moisture sensor by dipping the
// sensor in water to get a full wet reading
// and getting a value when the sensor
// is in dry air.
int getRange(void) {
  int result = 0;
  result = analogRead(A_IN_PIN);
  Serial.print("sensor reads ");
  Serial.println(result);
}
```

## Code Walk Through

The code is directly similar to the YL-69 resistive moisture sensor with a few const int changes for the capacitive sensor. This is the advantage in writing modular, generalized code. It is easily adapted to sensors of relatively the same type.

- The #define ARDUINO_IDE is used to switch between using the Arduino IDE or PlatformIO. If the Arduino IDE is used, then the file <ardruino.h> is not included.
- A delay tine of 2 seconds is defined. This delays the readings for no other reason than make the readings less jumpy on the serial monitor.A delay tine of 2 seconds is defined. This delays the readings for no other reason than make the readings less jumpy on the serial monitor.
- A delay tine of 2 seconds is defined. This delays the readings for no other reason than make the readings less jumpy on the serial monitor.
- An array of size 5 is declared to hold five successive readings to

be averaged.

- The next six const ints are designed around particular sensors. For the HW-390 sensor, the output values range from 198 to 510, which is a much tighter range than the resistive sensor. The first two const ints define the lower and upper boundaries of the sensor range (510 and 198). These values should reflect the low to high values output by any particular sensor, and there is a function in this code that can be commented out to find these values. The next const int is for the top percentage value, which is 100. The next const int is for the lowest percentile value, which is zero. Const ints are used here just for good form and not to use discrete numerical values in the body of the code. The next two const ints are used as partition values to discriminate between dry, moist and wet/most, which change from sensor type to sensor type. Changing these values make substituting sensor types, such as resistive and capacitive moisture sensors painless.
- Four function prototypes are declared.
    - int average(int[ ]) calculates the mean (average) a set of five sensor readings and returns the average value.
    - int partitionValues(int) takes the average value returned by average(int[ ]) and determines where the value fits in the sensor range. The function returns a 0 for dry, 1 for moist, and 2 for wet/moist, just like in the resistive sensor code.
    - int getRange(void) is a function that is used to determine a given sensor's range. Uncomment this prototype and the function body and run the code with the sensor completely dry then completely wet. Record the values and use them in the const int LOW_VAL and HIGH_VAL. Comment the prototype and the function body when done.
    - void calcHumidity(int) takes the average value as the input parameter and calculates the percent moisture.

- In setup( ) the serial monitor is initialized at 9600 baud. There is no digital input for the capacitive moisture sensor so the pinMode assignment for a digital pin is commented out. The code was not removed since it may be useful with another sensor that provides a digital output.

- The pin mode for the analog pin is initialized to an INPUT. This is the analog output of the capacitive sensor that will go to the

analog input pin, in this case A0, on the Arduino Uno.

- In loop( ) the array, dat_array[ARRAY_SIZE] is declared. This is the array that holds the sensor readings. Two int variables are average_val, which holds the average of the 5 successive sensor readings and range_val which holds the return value of the function partitionValues(average_val).
- The int dig_val variable is commented out. If an application needs toe utilize the digital output of the sensor, then uncomment this.
- The getRange( ) function call is commented out. As described earlier, uncomment this to determine the range values of the particular sensor used.
- The for loop iterates over the sensor readings that are directly loaded into the array dat_array. The iteration terminates at i < ARRAY_SIZE, which is 4, and not 5. In C, arrays start at index 0 and not 1. If the iteration was i <= ARRAY_SIZE then the array would exceed its boundary value. In this application, the digital value from the sensor is not used, if need be, the digitalRead( ) statement can be uncommented.
- A delay of 1 second (1000ms) is used between the readings.
- Outside of the for loop the average value is returned bu the function cll to average(dat_array). Note that brackets are not used in the function call, but they are present in the function prototype. The function prototype tells the compiler what the parameter's data type is, which is an int array (int [ ]) and the actual call to the function takes the address of the array, which is just dat_array. Using data_array in the function call parameter is really a pointer to the array. This is the way to pass entire arrays passed to functions in C.
- Now that the average_val is returned, it's passed as a parameter to the partitionValues( ) function, which returns the range_val. The range_val is where the reading falls within the entire range partitions, which are 0,1,2. A value of 0 indicates "Dry', 1 which is "Moist" and 2 which is "Wet/Moist". How this partitioning works is described in the partitionValues( ) function description.
- The humidity (moisture) level is calculated by the calcHumidity( ) function. This calculates the percentage humidity and prints it to the serial monitor.
- The range_val variable is used in a switch statement. Each case prints the moisture range level depending on the contents of

range_val. A default case is not really needed, but is present for good form and completeness of the switch statement.

- The serial monitor print lines of code are commented out, but are useful to see the actual values output by the functions.
- the average(int val_array[ ]) function uses a for loop to average the readings in the array that holds the sensor readings. Each value is summed and stored in the int variable ave.
- Outside of the for look the average value, ave is divided by the ARRAY_SIZE to yield the mean of the readings. This value is returned.
- The partitionValues(int val) function uses a set of if statements to determine where the sensor value (val) fits in the entire range. The comparison values were declared at the top of the code and are specific to individual sensor ranges. If val is greater than or equal to DRY_RANGE, the 0 is returned, representing "Dry". If val is less than DRY_RANGE but greater than MOIST_RANGE then 1 is returned, representing "Moist". If val is less than MOIST_RANGE then 2 is returned, representing "Wet/Moist".
- The function calcHumidity(int ave_val) calculates the percentage of moisture detected by the sensor using the built-in Arduino map( ) function. The map functions is a very handy function that maps one set of values to another and places the first parameter, in this case ave_val, in the new range. This is a good function to use when a percentage value is needed. Mapping from one range to another ranging from 0 to 100 accomplishes this easily.
- An if-then-else statement is used to keep the percentage value scaled and positive. Finally the built-in abs( ) function is used to yield a positive percentage. The results are printed via the serial monitor.
- The getRange( ) function is used to determine the output range of a moisture sensor. Uncomment this function call earlier int he code to get the range values. Immerse a moisture sensor completely in liquid to get the upper sensor value, and completely dry to get the lower sensor value. Record these values and use them in the LOW_VAL and HIGH_VAL const int declarations. This customizes the code of a particular sensor or class of sensors**.**

# CHAPTER EIGHT

## Output Devices

Output devices make things happen in the real world. This is were the rubber truly meets the road. First we need to see things, so we'll start out with the LCM1602 LCD Display Module.

### LCM1602A LCD Display Module

The LCM16702A LCD display module comes with several Arduino sensor kits. It's a fundamental display device so we will look at this early on. The first thing to do is to understand the LCD pinout. In the description below H means 5VDC and L means 0V.

- **Pin 1** - VSS. This is the ground pin and is grounded to the any of the Gnd pins on the Arduino.
- **Pin 2** - VDD. This the 5VDC power input and is connected to the 5VDC pin on the Arduino.
- **Pin 3** - V0. This is the input to drive the LCD supply voltage, commonly connected to the wiper of a 10k potentiometer.
- **Pin 4** - RS. Takes H/L as input. RS is held low for writing and reading to/from the LCD internal RAM and held low otherwise. (Note: instructive to monitor this pin with an oscilloscope).
- **Pin 5** - RW. Takes H/L as input. Held high in Read mode, and held low for write mode. Typically held low for most applications. This pin is connected to any Gnd on the Arduino.
- **Pin 6** - E (Enable). - Takes H/L as input. The E pin is used to send data to the data pins of the LCD. Typically connected to Pin 12 on the Arduino. (Note: instructive to monitor this pin

with an oscilloscope).

- **Pin 7** - D0. Data line 0. Takes H/L as input. Bit 0 of the 8 bit bi-directional bus.
- **Pin 8** - D1. Data line 1. Takes a H/L as input. Bit 1 of the 8 bit bi-directional bus.
- **Pin 9** - D2. Data line 2. Takes H/L as input. Bit 2 of the 8 bit bi-directional bus.
- **Pin 10** - D3. Data line 3. Takes H/L as input. Bit 3 of the 8 bit bi-directional bus.
- **Pin 11** - D4. Data line 4. Takes H/L as input. Bit 4 of the 8 bit bi-directional bus.
- **Pin 12** - D5. Data line 5. Takes H/L as input. Bit 5 of the 8 bit bi-directional bus.
- **Pin 13** - D6. Data line 6. Takes H/L as input. Bit 6 of the 8 bit bi-directional bus.
- **Pin 14** - D7. Data line 7. Takes H/L as input. Bit 2 of the 8 bit bi-directional bus.
- **Pin 15** - +5VDC - backlight power.
- **Pin 16** - 0V - backlight ground.

## Requirements and Comments

The code demonstrates some of the useful methods that are provided with the LiquidCrystal library. All we will do is cycle through the methods, but they will be encapsulated in functions we created. These are generally called "wrapper" functions. Why do that when we can directly call the library method? One reason is to modularize the method calls, which adds a layer of abstraction with the small cost of some compiler overhead and memory space. Another reason is to add support code within the functions as required, therefore eliminating "glue code" in the main loop. This makes the code in loop( ) more modular and readable. This becomes very useful when implementing state machines.

The functions called are:

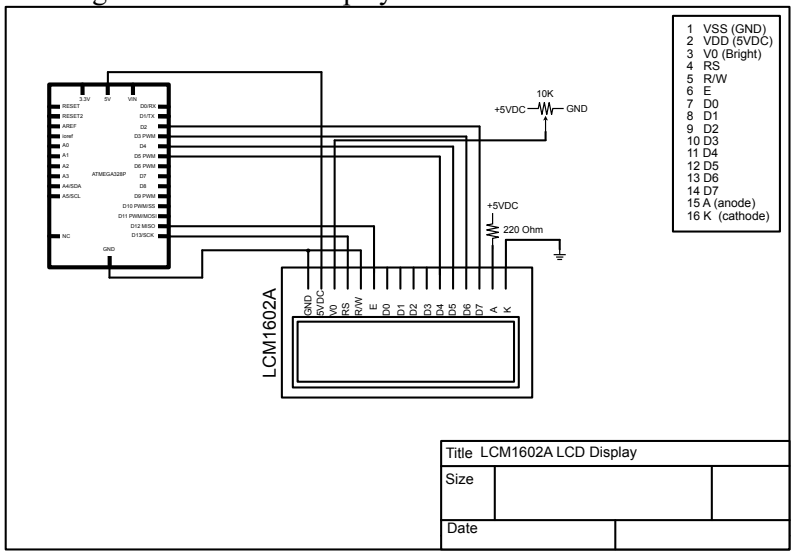
- clear( ) - clears the LCD display.
- home( ) - sends the cursor to the home position.
- noCursor( ) - turns off the cursor.
- display( ) - displays a string.
- cursor( ) - displays the cursor.
- blinkCursor( ) - blinks the cursor.
- noBlinkCursor( ) - turns off blinking

- scrollDisplayLeft( ) - scrolls the message on the display left one cell at a time.
- scrollDisplayRight( ) - scrolls the message on the display right one cell at a time.

On entry to loop( ) the LCD is cleared and the cursor is sent home. The cursor is turned off, then the string “Hello Mars! Is displayed. The cursor is then turned on and the cursor is blinked for a few seconds. The cursor stops blinking and the display is scrolled left for the length of the string, pausing for one half second for each cell until only the cursor is left. The display is then scrolled right in the same fashion until the entire string is displayed.

## The Circuit

This is the standard circuit that is shown in most examples when connecting the 1602A LCD display to an Arduino Uno.



## The Code

```
/*
File: LCDMethodDemo.cpp
Demonstrates class methods for the LCM1602A LCD.
Also demonstrates encapsulating method calls in order
to reduce "glue code" in loop().

LCD/Arduino Connections
LCD Pin 1   - VSS (GND)   - Arduino GND pin
LCD Pin 2   - VDD (%VDC)  - Arduino %VDC pin
LCD Pin 3   - VO (Bright) - 10K pot wiper
LCD Pin 4   - RS          - Arduino GND pin
LCD Pin 5   - R/W         - Aruino pin D13
LCD Pin 6   - E           - Arduino pin D12
LCD Pin 7   - D0          - n/c
LCD Pin 8   - D1          - n/c
LCD Pin 9   - D2          - n/c
LCD Pin 10  - D3          - n/c
LCD Pin 11  - D4          - Arduino pin 5
LCD Pin 12  - D5          - Arduino pin 4
LCD Pin 13  - D6          - Arduino pin 3
LCD Pin 14  - D7          - Arduino pin 2
LCD Pin 15  - A           - +5VDC through 220 Ohm resistor
LCD Pin 16  - K           - Arduino GND pin

LCD Methods
clear()               - clears the disply
home()                - sends the cursor home
cursor()              -
noCursor()            -
noDislay()            -
display()             -
blink()               - blinks cursor
noBlink()             - turns off blinking curo
scrollDisplayLeft()   - scrolls the display left
scrollDisplayRight()  - scrolls the disply right
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

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

// delay times
#define DELAY_1SEC    1000
#define DELAY_3SEC    3000
#define DELAY_HALFSEC 500
#define DELAY_5SEC    5000

// function prototypes
void demoClear(void);     // calls lcd.clear() to clear the LCD
void demoHome(void);      // calss lcd.home() to home the cursor
void demoCursor(void);    // shows cursor
void demoNoCursor(void);  // hides cursor
void demoDisplay();       // writes string to LCD
void demoNoDisplay();     // removes string from LCD
void demoBlinkCursor();   // blinks cursor
void demoNoBlinkCursor(); // turns of blink
void demoScrollDisplayLeft();   // scolls display left
void demoScrollDisplayRight();  // scrolls display right

// initialize the library by associating any needed LCD interface pin
// with the arduino pin number it is connected to.
// Note that using const int is a more modern way of assigning constants.

// Traditional C programs will use #define for this.
const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);

void setup() {
  Serial.begin(9600); // init serial port
  lcd.begin(16, 2);   // init LCD
  // Print ** init ** msg to the LCD.
  lcd.print("** init **");
  delay(DELAY_1SEC);
}

void loop() {
  demoClear();
  demoHome();
  demoNoCursor();
  demoDisplay();
  demoCursor();
  demoBlinkCursor();
  demoNoBlinkCursor();
  demoScrollDisplayLeft();
  demoScrollDisplayRight();
}
```

```
/* FUNCTIONS */
// demoClear() - clears the LCD.
void demoClear(void) {
  Serial.println("clearing display...");
  lcd.clear();
  delay(DELAY_1SEC);
}

// demoHome() - takes cursor to home position.

void demoHome(void) {

  Serial.println("sending cursor home...");

  lcd.home();

  delay(DELAY_1SEC);
}

// demoCursor() - shows cursor.
void demoCursor(void){
  Serial.println("turning on cursor...");
  lcd.cursor();
  delay(DELAY_1SEC);
}

// demoNoCursor() - hides cursor.
void demoNoCursor(void) {
  Serial.println("turning off cursor...");
  lcd.noCursor();
  delay(DELAY_1SEC);
}

// demoDisplay() - displays a string.
void demoDisplay(void) {
  Serial.println("displaying string...");
  lcd.print("Hello, Mars!");  // if println is used CR/LF is printed
  delay(DELAY_3SEC);
}

// demoBlinkCursor() - blinks cursor.
void demoBlinkCursor(void) {
  Serial.println("blinking cursor...");
  lcd.blink();
  delay(DELAY_3SEC);
}

// deoNoBlinkCursor() - stop blinking cursor.
void demoNoBlinkCursor(void) {
  Serial.println("no blinking cursor...");
  lcd.noBlink();
  delay(DELAY_3SEC);
}

// demoScrollDisplayLeft() - scrolls left 1 cell at a time,
// so need a for loop. Also need to add a short delay
// to make the scroll visible. Note that the var i is
// an unsigned int to be compatible with the return value
// of strlen().
void demoScrollDisplayLeft(void) {
  Serial.println("scrolling display left");
  for (unsigned int i = 0; i<=strlen("Hello Mars!");i++) {
    lcd.scrollDisplayLeft();
    delay(DELAY_HALFSEC);
  }
  delay(DELAY_3SEC);
}

// demoScrollDisplayLeft() - scrolls left 1 cell at a time,
// so need a for loop. Also need to add a short delay
// to make the scroll visible. Note that the var i is
// an unsigned int to be compatible with the return value
// of strlen().
void demoScrollDisplayRight(void) {
  Serial.println("scrolling display left");
  for (unsigned int i = 0; i<=strlen("Hello Mars!");i++) {
    lcd.scrollDisplayRight();
    delay(DELAY_HALFSEC);
  }
  delay(DELAY_3SEC);
}
```

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples

; https://docs.platformio.org/page/projectconf.html
[env:uno]
platform = atmelavr
board = uno
framework = arduino
lib_deps =
     Wire
     marcoschwartz/LiquidCrystal_I2C@^1.1.4
```

## Code Walk Through

Let's walk through the code.

- First the #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is un-commented.
- The Arduino library header LiquidCrystal.h file is included so we can use the library methods. To understand what methods are available, check this file out and look at the public methods.
- Several timer defines are declared for experimenting with delays between the function calls and scrolling times. A delay is required when scrolling the display, allowing time for the LCD to react. As an experiment substitute the various times into the functions where they are used and see how the LCD reacts.
- The wrapper function prototypes are declared along with a short description of what each function does.
- The data structure for the LCD code is initialized by first defining the pins on the LCD display, being rs, en, and the data pins d4 though d7. Half-byte addressing is used. (Explain how it works). The lcd data structure is initialized as of type LiquidCrystal.
- In setup ( ), the serial monitor is initialized along with the lcd via Serial.begin(9600) and lcd.begin(16,2). The parameters 16 and 2 denote the number of characters per row and the number of rows. An improvement to the code could be declaring each parameter as a #define or const int instead of hard-coded values. An ** init ** message is displayed on the LCE via the call to

the method lcd.print( ). Finally a one second delay is executed so the ** init ** message will persist on the screen.

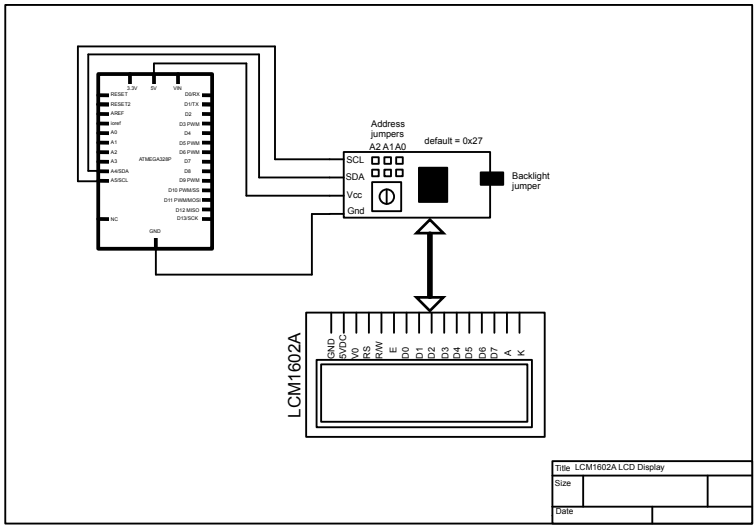
- in loop( ) each of the functions are called in sequence. There is no additional glue code that links the functions together. This is a very simple form where no data is passed from function to function. If there were, a global data structure could be allocated and manipulated, or local structures within the functions could be allocated, depending on what tasks needed to be performed. Using wrapper functions in this manner allow for state machines to be relatively easily constructed.
- The remainder of the code consists of the wrapper functions. These are in their simplest form since they are of type void (don't return any values) and no parameters are passed. They are completely self contained. Another advantage of wrapper functions other than the ones stated are any change made to a function does not effect other functions. This avoids hidden bugs and increases program reliability.

**Going Further**

A more efficient and common way of utilizing the LCD display is to use an I2C interface instead of using discrete wires to connect to the 1602A LCD. This saves pins on the Arduino and is easy to program using the available Arduino library Wire.h. I2C is explained in detail in the Serial Communications chapter. To take advantage of I2C when controlling and communicating with a 1602A LCD, an adaptor such as a 5V 1602 I2C adaptor needs to be used. They are directly connected to the power and data pins of an Arduino UNO R3 and are very inexpensive.

## The Circuit

The 5V 1602 I2C adaptor is pin-for-pin compatible with the 1602A LCD. Most LCD hardware configurations have the I2C adaptor directly soldered to the 1602A LCD.




The I2C adaptor contains a potentiometer that controls the contrast and a backlight jumper that turns the backlight on or off. If the jumper is connected, the backlight is on and off if disconnected. The default address of the I2C adaptor is 0x27. The table below shows the configuration for the 8 different available addresses.

| A0 | A1 | A2 | Address |
|---|---|---|---|
| Open | Open | Open | 0x27 |
| Jumper | Open | Open | 0x26 |
| Open | Jumper | Open | 0x25 |
| Jumper | Jumper | Open | 0x24 |
| Open | Open | Jumper | 0x23 |
| Jumper | Open | Jumper | 0x22 |
| Open | Jumper | Jumper | 0x21 |
| Jumper | Jumper | Jumper | 0x20 |

The connections need to be soldered with jumpers to force different addresses.

## The Code

Here's the code using I2C adaptor to communicate with a 1602A LCD. The address is at the default 0x27. Often the address can be determined by inspecting the address jumpers A0, A1 and A2. If this

isn't possible, the I2C device address must be scanned. The code for this is provided and described in the I2C section of the Serial Communications chapter.

```
/*
Project: LCD_I2C_Demo
Exercises some of the methods that are
listed in LiquidCrystal_I2C.h
Conslut LiquidCrystal_I2C.h for a full list of methods.

*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif
#define DELAY_TIME 1000
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
int addr = 0x27;
const int num_chars = 16;
const int num_rows = 2;

LiquidCrystal_I2C lcd(addr,num_chars,num_rows);

void setup() {

  Wre.begin();
  lcd.begin(num_chars,num_rows);
  Serial.begin(9600);
  while(!Serial)
    ;
  //  turn on LCD backlight
  lcd.backlight();
}

void loop() {
  // exercise a few LCD methods
  lcd.home();
  lcd.blink_on();
  lcd.print("LCD I2C Demo");
  delay(DELAY_TIME);
  lcd.home();
  lcd.blink_off();
  lcd.noCursor();
  // shift the output string all the way to right
  for(int i=0; i<16;i++) {
    lcd.scrollDisplayRight();
    delay(DELAY_TIME);
  }

  // shift the output string all the way to the left
  for(int i=0; i<16;i++) {
    lcd.scrollDisplayLeft();
    delay(DELAY_TIME);
  }
  // re-setup the LCD
  lcd.clear();
  lcd.blink_on();
  lcd.cursor();
}
```

## Platform.ini File

```
[env:uno]
platform = atmelavr
board = uno
framework = arduino
lib_deps =
     Wire
     marcoschwartz/LiquidCrystal_I2C@^1.1.4
```

## Code Walkthrough

All of the methods listed in LiquidCrystal_I2C.h are not implemented. Look at LiquidCrystal_I2C.h to find the methods you want to check out or use for your own applications. They are very similar to those found in LiquidCrystal.h. PlatformIO was used to

- First the #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is un-commented.
- a #define DELAY_TIME is used for a 1 second time delay between library method calls.
- The Wire library is called to utilize I2C commands.
- The LiquidCrystal_I2C library is used to access the 1602A LCD. Next, the default address of 0x27 is hard coded.
- The lcd object is instantiated next with the address, number of characters the LCD display has and the number of rows.
- The lcd object is instantiated with the address, number of chars in a row and the number of rows.
- In setup( ), the serial monitor is initialized via Serial.begin(9600) and the code waits until the serial port is available via the while loop.
- When the serial port is available the LCD backlight is illuminated.
- In loop( ) a few of the LCD methods defined in LiquidCrystal_I2C.h are exercised. The first method send the cursor com, turns blinking on and displays the string “LCD I2C Demo”.
- The code delays for 1 second to allow the LCD display to persist then the cursor is sent home. The blinking is turned off and the cursor is turned off.
- The for loop cycles through the display characters shifting the display to the right, one position at a time, including a 1 second

delay for display persistence.

- The next for loop works exactly the first for loop, only shifts the characters left one position at a time.
- The LCD is then cleared, the blink turned on and the cursor is restored in preparation for the next pass through loop( ).

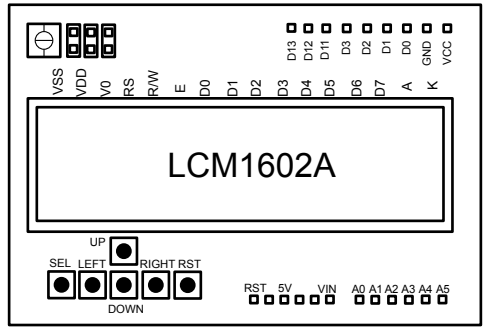
The I2C interface to the 1602A LCD is very handy, easy to use and greatly saves Arduino I/O pins.

## 1602 LCD Keypad Shield

The LCD keypad shield is handy for a minimal display system.

### The Circuit

The diagram below shows the LCD keypad shield. The shield, like other shields, attaches to the Arduino UNO's header pins.




### Specifications

The LCD is controlled by digital pins D4 through D9. Pin 9 is the enable pin E, and pin 8 is for RS (data or signal). The RS pin is the register select pin. If RS = 1, the data register is selected, allowing data to be displayed on the LCD. IF RS = 0, then the control register is selected, allowing for commands to be sent to the LCD, such as setting the cursor position, returning the cursor to home, and the like. Pin 10 is the LCD backlight control pin, and analog pin A0 is the button input pin. Different values are passed to A0 depending which button was pressed. For example, the LEFT button will output a different value than the SELECT value. The values input from the buttons may vary from module to module and there are two ways to accommodate this. One is to use the exact values issued from the buttons, or to use a range of approximate values. We will avoid using the exact values for a specific LCD keypad since it diminishes code portability.

## The Code

```
/*
Project: LCDKeypad2
Proesses button preses from LCD keypad.
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif
// This code uses the LiquidCrystal library and uses
// platformio.ini to include the library. The particluar
// library used is: ardruio-libraries/LiquidCrystal^@1.0.7
#include <LiquidCrystal.h>

// time delay to persist display
#define TIME_DELAY 20

// assigning int virtual values
// instead of using the keypad actual values.
const int noBtn     = 0;
const int selectBtn = 1;
const int leftBtn   = 2;
const int downBtn   = 3;
const int upBtn     = 4;
const int rightBtn  = 5;

// the button values for the specific LCD keypad
// used in this demo
#define  NOBTNVAL   1023
#define  SELBTNVAL  721
#define  LFTBTNVAL  482
#define  UPBTNVAL   133
#define  DNBTNVAL   310
#define  RTBTNVAL   0
// pad value to add to the actual values
#define  VALPAD  10

// const ints for pins and LCD class instance initialized
const int rs = 8, en = 9, d4 = 4, d5 = 5, d6 = 6, d7 = 7;
LiquidCrystal lcd(rs,en,d4,d5,d6,d7);

// function prototypes
int readBtn(void);
void processSelect(void);
void processLeft(void);
void processUp(void);
void processDown(void);
void processRight(void);

void setup() {
 lcd.begin(16,2);
 pinMode(A0,INPUT);
 Serial.begin(9600);
 while(!Serial)
  ;
 lcd.print("* init *");
 delay(500);
 lcd.clear();
 lcd.home();
 lcd.cursor();
}
```

```
Void loop() {
  int btnVal = readBtn();
  Serial.println(btnVal);

  // switch statement to process button presses
  switch(btnVal) {
    case 0: // no button pressed
    break;

    case 1: // Select button

     processSelect();
    break;

    case 2: // Left button
      processLeft();
    break;

    case 3: // Down button
      processDown();
    break;

    case 4: // Up button
      processUp();
    break;

    case 5: // Right button
      processRight();
    break;

    default:
    break;
  }
}

// reads button state
int readBtn(void) {
  // make sure the values are in desending order
  // the specific value for the keypad buttons are used

  // along with an extra numerical "pad"
  int buttonState = analogRead(A0);
  if(buttonState >  NOBTNVAL – VALPAD)
    return noBtn;
  if(buttonState < RTBTNVAL + VALPAD)
     return rightBtn;
  if(buttonState < UPBTNVAL + VALPAD)
    return upBtn;
  if(buttonState < DNBTNVAL + VALPAD)
     return downBtn;
  if(buttonState < LFTBTNVAL + VALPAD)
    return leftBtn;
   if(buttonState < SELBTNVAL + VALPAD)
    return selectBtn;
  return buttonState;
}
```

```
// processSelect = stub for Select button
void processSelect(void) {
  lcd.clear();
  lcd.home();
  lcd.print("in Select fn");
  delay(TIME_DELAY);
}

// processLeft – stub for Left button
void processLeft(void) {
  lcd.clear();
  lcd.home();
  lcd.print("in Left fn");
  delay(TIME_DELAY);
}

// processUP – stub for Up button
void processUp(void) {
  lcd.clear();
  lcd.home();
  lcd.print("in Up fn");
  delay(TIME_DELAY);
}

// processDown – stub for Down button
void processDown(void) {
  lcd.clear();
  lcd.home();
  lcd.print("in Down fn");
  delay(TIME_DELAY);
}

// processRight – stub for Right button
void processRight(void) {
  lcd.clear();
  lcd.home();
  lcd.print("in Right fn");
  delay(TIME_DELAY);
}
```

```
; PlatformIO Project Configuration File
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples

; https://docs.platformio.org/page/projectconf.html
[env:uno]
platform = atmelavr
board = uno
framework = arduino
lib_deps = arduino-libraries/LiquidCrystal@^1.0.7
```

## Code Walk Through

- The #define ARDUINO_IDE is used to switch between using the Arduino IDE or PlatformIO. If the Arduino IDE is used, then the file <ardruino.h> is not included.
- The LiquidCrystal library is included. The library used is the Arduino-libraries/LiquidCrystal@^1.0.7. This is the library dependency declared in Platformio.ini.
- A time delay #define is declared for 20ms. This is used to allow the display to persist and not jitter.
- Instead of using the actual values that are received from the

buttons, a set of const ints are used instead, one for no button pressed and the remainder for the rest of the buttons. A numerical sequence such as 0, 1,. . . allow for readability and abstraction of the input values.

- The actual button values for each of the buttons are abstracted here by using #defines.
- A define for a numerical "pad" is declared here. This value will be added to the actual const int values for each of the buttons, giving the button input a wider numerical range.
- const ints are defined for the LCD keypad pins, and an LCD object, lcd, is declared.
- Function prototypes are declared for the functions that respond to specific button presses.
- In setup( ) the lcd object is instantiated, analog pin A0 is set as an input, the serial monitor is initiated at 9600 baud. The code spins until the serial port is available, and an init message is printed on the serial monitor. A short 500ms delay is used to allow the init message to be seen. This code, along with all serial monitor code can be removed in a real application. The lcd is cleared, sent home, and the cursor turned on. Note that the button presses enter the Arduino UNO via analog pin A0. The different values input run through a voltage divider, which is a very good method to limit the use of input pins but requires external hardware.
- In loop( ) the function readBtn( ) is called and stores the return value in the int variable btnVal. The return value is printed to the serial monitor.
- The switch statement acts on the value of btnVal. Each of the cases call the specific functions based on the value of btnVal, such as processSelect( ), processLeft( ) and the like. Each of the functions clear the display, sends the cursor home and displays a short message. Using a switch statement and isolating the functions to be called within the cases is a very clean and efficient way to code.
- The readBtn( ) function reads the raw value of a button press. A call to analogRead(A0) reads the A0 pin value and, though a series of if statements, returns the appropriate button. The first if statement determines of no button has been pressed and immediately returns. The button value is compared to the sum or the actual value of the button (NOBTNVAL) subtracted from

the numerical pad VALPAD. If the results is greater than the actual button value, the function returns.

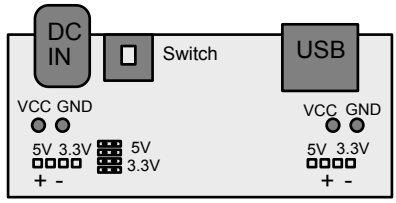
- The remainder of the function returns depending on the comparison of the actual button value and the numerical pad.
- If the button value is spurious or undetermined, the raw button value is returned.
- Five functions are written to act on the keypad button presses. These are just stubs that serve no real purpose but provide an entry point for specific application code. Each function clears the LCD, sends the cursor home and displays a message on the LCD. A time delay allows for the message to remain on the display.plat
- The Platformio.ini file is standard for the Arduino UNO except for the extra lib_deps line. This line pulls in the LiquidCrystal library and runtime and must be present.

Many if not most LCD keypad sketches use estimated values of the button presses and not the direct discrete button press values themselves. The only area of this code that needs to use the actual values from the buttons are the #defines for the actual key values. Other than that, the code does not need to be change except for the possibility of the VALPAD value. This value may need to be increased if the buttons on any particular LCD keypad is not within the range of the #defines for the specific values. Abstracting as much as you can makes your code more portable, and if it needs to be altered, the alterations are slight and concentrated in a single area.

## Power Supply Module

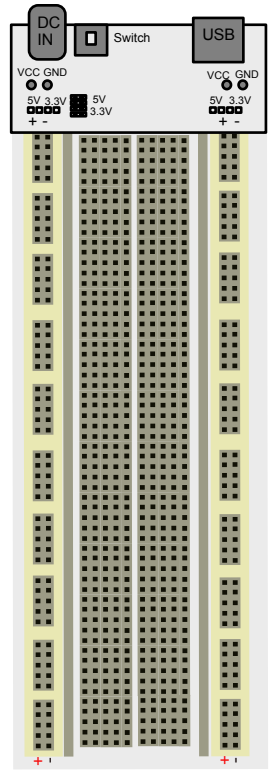
The power supply module that comes with numerous Arduino sensor kits is great for not only Arduino projects but is useful for electronics prototyping and learning. The power supply module is designed to be directly connected to a standard MB102 breadboard.

The power supply module is simple but useful. The specifications state that the input voltage can vary from 6.5-12VDC or 5V from a USB power supply USB. The output voltage is is 3.3V and 5V, which is convenient. The maximum output current is 700mA.




The DC IN connecter accepts a barrel connector from a common power supply. The power to the rails of a breadboard are selected by jumpers on the horizontal pins labeled 5V or 3.3V. Just jumper the pins for the voltage you need. It's convenient to have one set of rails at 5V and the other at 3.3V for components. Power is also accessible from the vertical pins. Use Dupont cables to tap power from these pins.

Plug the power supply module in at the end of the breadboard with the + (red) bus facing left to line up with the + symbol on the power supply module. Remember to turn on the power supply module using the switch! This sounds silly, but it's easy to forget.




This configuration with the power supply and breadboard is a very convenient way to experiment with electronics and circuits before ever connecting an Arduino. Be aware that the voltage drops a little under load.

# CHAPTER NINE

## Serial Communications

Serial communications take data and sends and receives it one bit at a time. In our microcontroller world, the data is typically parallel data to be transmitted, such as the state of a word or byte received or sent on I/O pins. Serial communication generally falls into two categories, synchronous and asynchronous. Synchronous communications requires a mutual clock to transfer data. Asynchronous does not. In the next few sections, we'll cover basic serial communication methods using UARTs, I2C, and SPI. First, we will look at a few general ideas, then move on to how a UART works.

### Duplex Modes

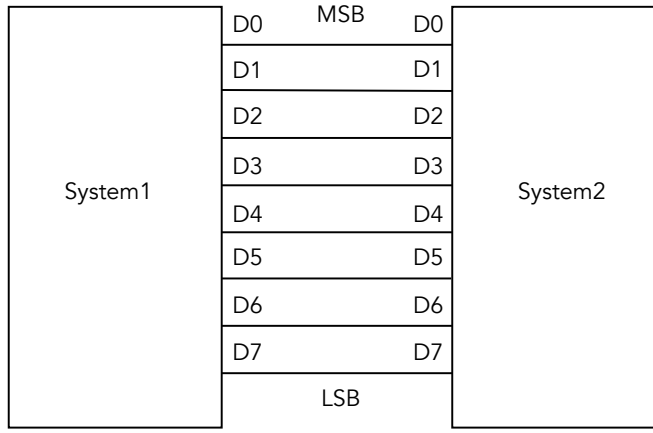
There are three basic duplex modes, Simple, Half and Full. Duplex means two or more entities in communication. The first is Simplex Mode. Simplex mode is unidirectional, meaning only one-way communication. An entity can either transmit or send data one way. For example, an over-the-air TV signal is an example of simplex communication. The TV station transmits the signals, and receivers receive them.

The next mode is Half-Duplex. Half-Duplex enables bidirectional communication and can send and receive, but only one entity at a time. Browsing the Internet and USB utilize Half-Duplex communication.

The next mode is Full-Duplex. Full Duplex is bidirectional and sends and transmits data at the same time. The classic example of Full Duplex communication is a phone, where users can talk and listen at the same time.

## Parallel vs. Serial Communication

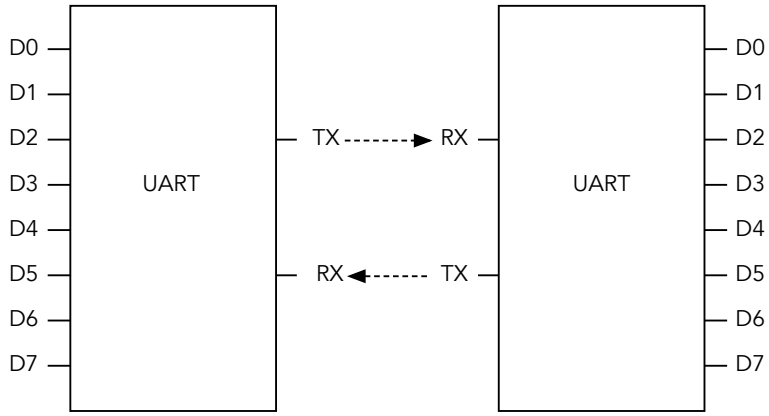
Parallel communication transfers all data in a byte at the same time. There is a direct connection between two systems communication, as shown below.



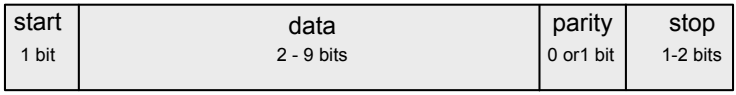

System1 may be a component on a common computer bus, or two entirely separate computers. It doesn't matter- the data transfer principle is the same. All data is transferred on one synchronous clock pulse. This is great for components communicating with one another in computer architectures and designs such as between CPU registers and memory. It becomes difficult and unfeasible when communicating with separate or distant systems. The alternative to parallel is serial communications, sending one bit at a time to a remote system, then reassembling the bits into the system's preferred format. This is where the UART comes into play.

## UART

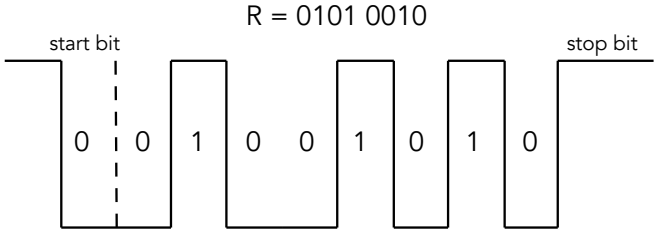
UART stands for Universal Asynchronous Receiver Transmitter. The UART accepts parallel data bits and transmits and receives them serially. After the parallel data is received, the UART adds a stop bit, an optional parity bit and a stop bit. The Arduino UNO default serial communications functions do not use a parity bit. The data is accurately sent and received when the bit rate is agreed upon. This is called the baudrate, which means how many bits per second are transmitted and received. The figure below illustrates two connected UARTs.




The parallel data assembled with a start bit, an optional parity bit and a stop bit is called a packet. The data within the packet is called the data frame. The drawing below illustrates a minimal packet.




Here's an example below of a data packet for the ASCII character upper case R, which is binary 0101 0010 or 0x52 hexadecimal. There is no parity bit included in this packet.




The UART transmit line will stay high for a duration indicating that no data is to be sent. When the transmit line drops low, this indicates to the receiving UART that a data packet is being sent. The next 8 bits are sent with the least significant bit or LSB sent first. The first bit after the stop bit is 0, then a 1 is transmitted, then two 0s. This makes up the least significant byte. 0100. Then a 1, 0, a 1 and 0 are sent. This makes up the most significant byte. The stop bit then goes high to indicate the end of the packet.

So how does the receiving end know when one bit is sent? That is where the baudrate comes into play, which is how many bits per second data is transmitted or received. If for the above example the baudrate is set to 9600, this means that 9600 bits per second are sent. To get the time period for one bit, take the inverse of 9600, which is 1/9600, which is 0.000104167 seconds, or 104 microseconds. To each bit state will be sampled in a 104 microsecond period. This is explained further in the Serial Trace Analysis section where we look at oscilloscope traces of an ASCII character transmission.

## Parity

Parity is a simple error checking technique. An additional bit is used to make the data packet an even or odd number of bits. For example, in the ASCII 'R' character transmission, there are three 1 bits. If we wanted even parity, we would use an additional 1 parity bit to making the number of 1 bits in the packet even. For odd parity, if the number of bits in the packet are even, then a 1 parity bit is added to make the total number of 1 bits odd.

## Errors

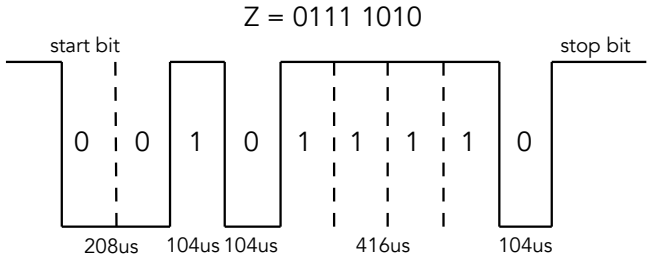
Serial communications are subject to many errors. They could happen internally when clocks overlap or externally when electrical noise is introduced during packet transmission. A framing error usually occurs when the receive does not detect a stop bit in time. A parity error (only active when the UART is in parity mode) when the parity is incorrect. An overrun error occurs does not process and delete a character from the input buffer in time.

Protocol stack designers need to be aware of these and possibly other errors to create, robust, bullet proof designs.

## Serial Trace Analysis

In this section we'll do an analysis of a set of scope traces when the z key is pressed on a PC keyboard and transmitted to an Arduino UNO. An oscilloscope probe was connected to the RX pin on the Arduino along with ground. To understand the traces we need to do a couple simple calculations.

Binary encoding for the character 'z' is 0111 1010. This is what the trace should look like, with a total of 8 data bits.

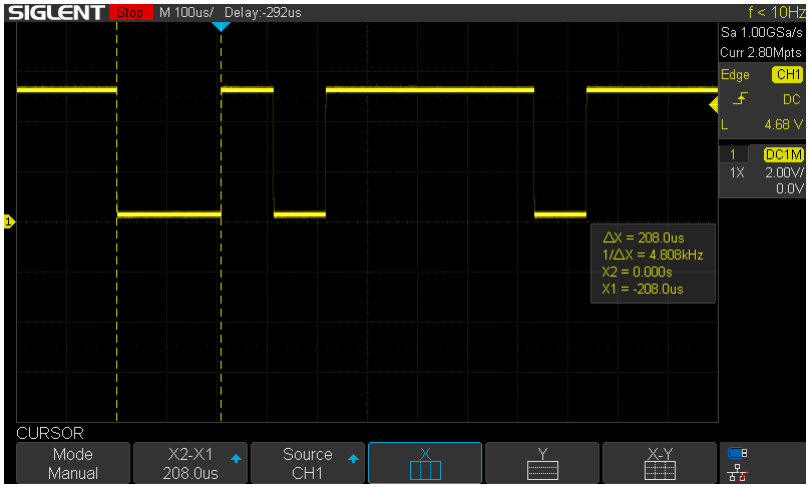



The baud rate was set to 9600. This means 9600 bits per second will be transmitted. We need to know the time period that one bit transmission will take. To figure this out we take the reciprocal of 9600, which is 1/9600 = 0.00010467 sec, which is 104.67microseconds. We will round down to 104microseconds for our measurements. Starting from the left, we have two 0 bits for a time of 208microseconds. The first 0 bit is the start bit and the next is the least significant bit, or LSB. The UART sends the data starting with the LSB, so as we measure, we will see the character encoding "backwards" from or usual notation with the LSB on the right and the MSB (most significant bit) on the left. The next bit is a 1 with a duration of 104microseconds, so that is one bit. The next bit is 0 for 104microseconds, so that is one bit also. The next duration where the signal is high is 416microseconds, which is equal to 4 bits. The last bit is zero with a duration of 104microseconds, the next signal goes high and stays high, indicating the end of the transmission.

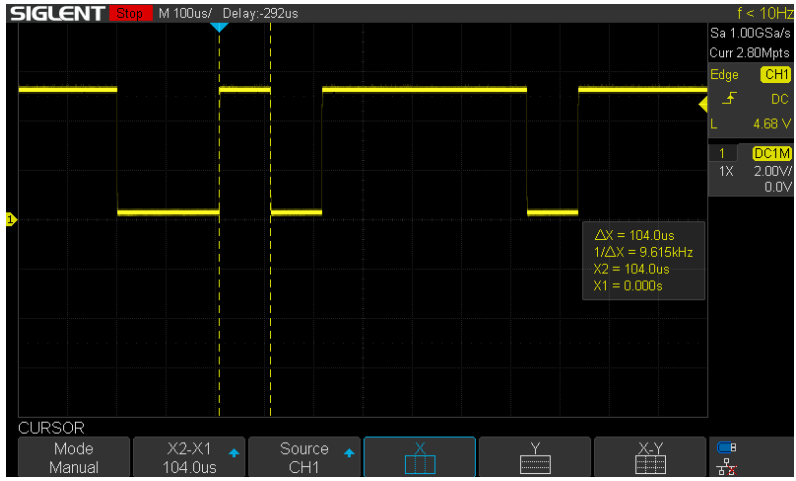
## Oscilloscope Traces

We now have a good idea of what we should see on an oscilloscope. Knowing the time duration to transmit one bit we can set the horizontal time value on an oscilloscope for 100 micro seconds per division. The voltage level is 5VDC so the vertical is set to 2V per division.

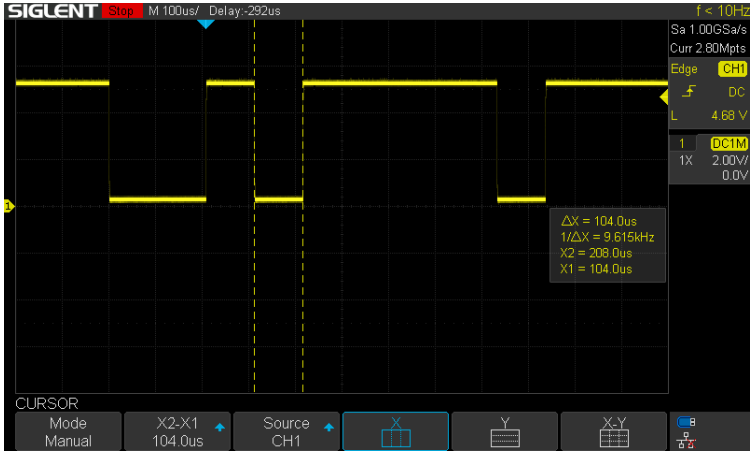
The scope trigger value was set to 4.88 volts and single trigger was used. The trigger value is set high, since the UART outputs a high or 5VDC when no serial activity occurs and we want to catch the start bit, which is low or z0 volts. Here is the first trace.



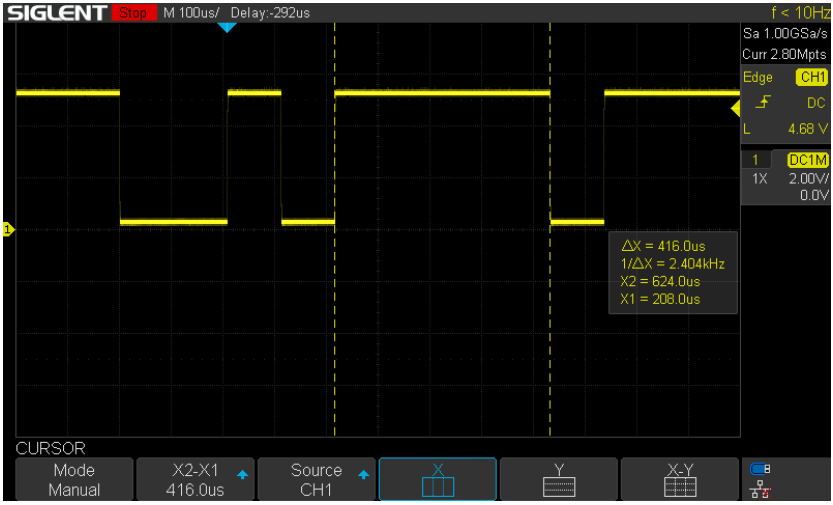

The first measurement starting from the left is when the signal first drops low. The time period measured is 208.0 micro seconds. Dividing 208 by 104 yields 2, or 2 zero bits. This makes sense, since the start bit is always 0. Our first bit of data is 0.



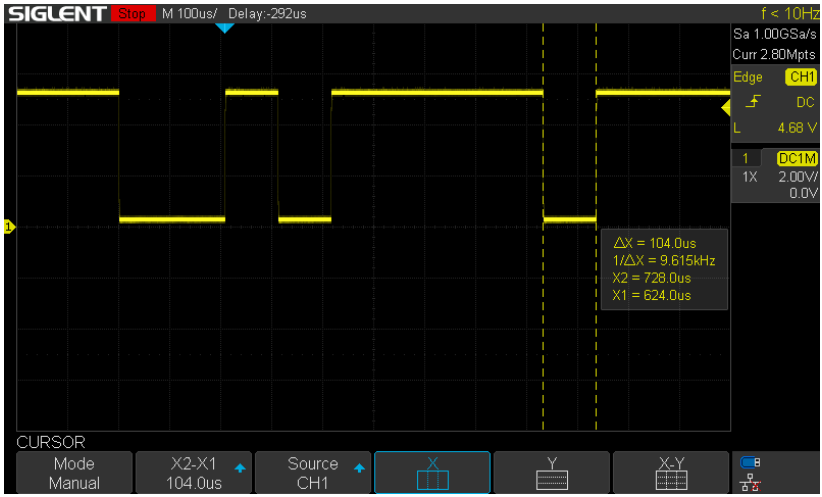

The next measurement duration is 104 microseconds, which is a single high bit.




The next measurement is 104microseconds, which is a single low bit.




The next measurement duration is 416microseconds. 416/104 = 4, so we have 4 high bits.

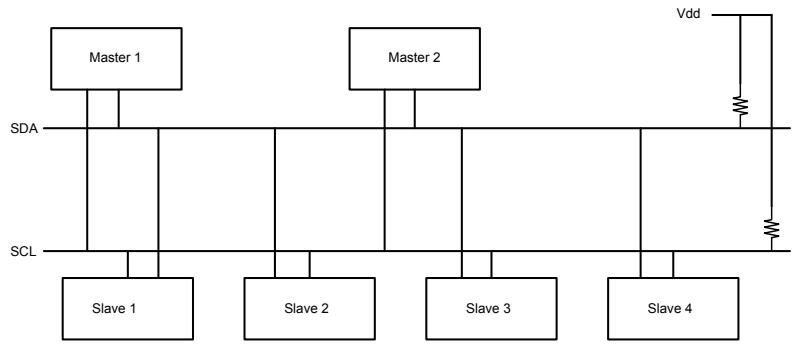



The last measurement duration is 104microseconds, so this is one low bit. The stop bit goes high and stays high.

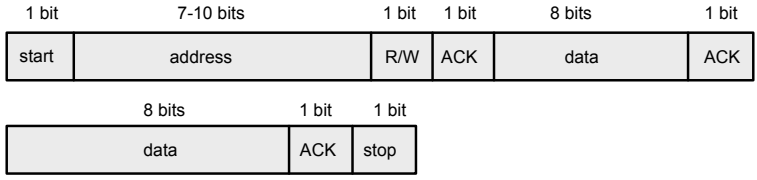
So, looking at the scope trace from left to right and ignoring the start bit, the bit order is 0101 1110, with the lower byte first and the high byte second. This is the mirror image of the byte ordering we are used to, so the bits need to be reordered, with the rightmost bit now occupying the MSB bit position. We then traverse through the bits putting them in their correct positions. After reordering, we get 0111 1010, which is 'z'.

## I2C

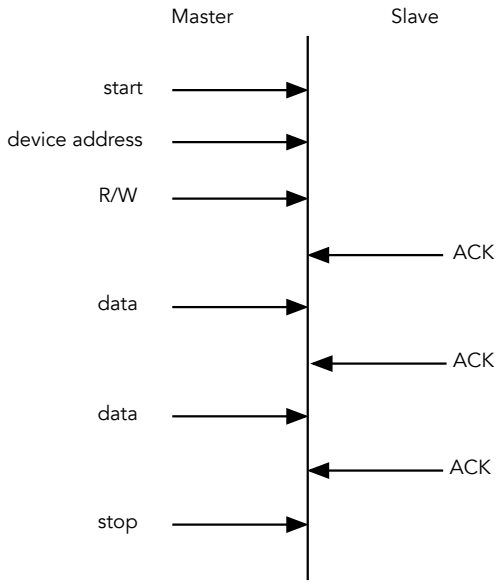
I2C stands for Inter-Integrated Circuit Bus. Developed by Philips in 1989, I2C is a half-duplex, synchronous, two wire communication mechanism for different devices over short distances. I2C can have multiple masters and multiple slaves and is primarily used for short or intra-board communications. Only two lines are required, labeled SCL and SDA. SDL is a serial clock line and SDA is a serial data line. I2C can communicate at different clock speeds. In Standard Mode, 8 bit bidirectional data may be sent up to 100Kbits per second. In Fast mode, bidirectional data may be sent up to 400Kbits per second. Fast Mode+ allows up to 1Mbit per second. Ultra Fast Mode is unidirectional and allows up to 5Mbits per second. We will be using standard mode for the Arduino UNO Every I2C device has a unique I2C address from 0 to 127 Address 0 is a reserved address and is the general call address. This addresses all of the I2C devices on the I2C bus at the same time. The figure below illustrates the basic I2C bus architecture.



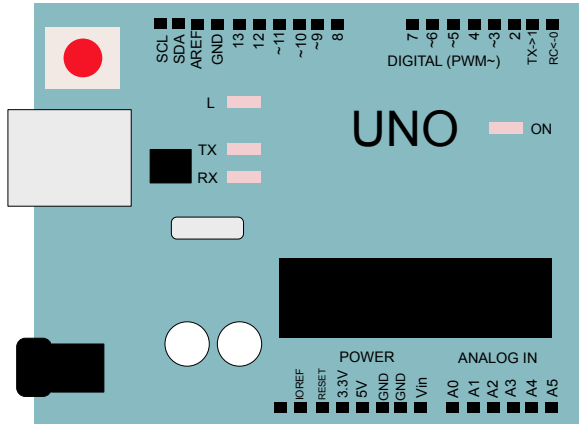

I2C employs a master-slave model of communications. Masters are called controllers and Slaves are generally peripherals. The SCL line provides a clock to synchronize communications, and data is written on the SDA line. Each devices has a unique one byte address, starting from 127 and counting down from there. Pull up resistors are required for the SDA and SLC lines. The pull-up resistors are generally in the range of 2K to 4K, with 4.7K being typical for many applications. The communication packet is shown below.




The packet describes two way transmission from a master to a slave. The master writes to the bus and the slave returns an ACK. The diagram below further illustrates this.




The I2C pins, SCL and SDA are built into the Arduino UNO and are the first two pins before the AREF pin, as shown below.




The Wire library is popular with Arduino users when working with I2C. The wire functions are:

- **begin( ), or begin(address)** - initializes the Wire library and is only called once. If no argument is given, the device calling begin( ) is the controller. If the address parameter is used, then the device at that address is a peripheral. The address is a 7 bit slave address. No return value.
- **end( )** - disables the Wire library. No return value.
- **requestFrom(address, quantity)** or **requestFrom(address ,quantity, stop)** - controller requests bytes from peripheral. Bytes are retrieved with the available( ) and read( ) functions. The parameters are address which is the address of the peripheral, quantity which is the quantity of bytes to be read, and stop, which is true or false. The stop parameter is binary. If false, a restart is sent after the request, persisting the connection.
- **beginTransmission(address)** - starts a transmission to the I2C devices at the address parameter. Bytes are queued by the write( ) function and transmitted by endTransmission( ). No return value.
- **endTransmission( ) or endTransmission(stop)** - ends transmission to a peripheral device started by beginTransmission( ) Transmits the bytes that are in the write( ) queue. If the stop parameter is true, then a stop message is sent after transmission and the I2C bus is released. If false, a restart message is sent and the I2C bus is not released. The return values are:
  - 0 - success

  - 1 - data too long to fit in transmit buffer
  - 2 - received NAK on address transmission
  - 3 - received NAK on data transmission
  - 4 - other error
  - 5 timeout
- **write(value), write(string), write(data, length)** - writes data from a peripheral device is commanded, or queues bytes for transmission to a peripheral device from a controller. write( ) is used in-between calls to beginTransmission( ) and endTransmission( ). The parameters are:
  - **value** - a single byte value
  - **string** - a string sent as a series of bytes
  - **data** - an array sent as bytes
  - **length** - the number of bytes to transmit
  - Returns the number of bytes written.
- **available( )** - returns the number of bytes available to retrieve with a call to read( ). read( ) is called from a controller after a call to requestFrom( ).
- r**ead( )** - readsa a byte from a peripheral to a controller after a call to requestFrom( ), or a byte that was transmitted from a controller to a peripheral. Returns the next byte received.
- **setClock(clockFrequency)** - modifies the clock frequency for I2C communication. 100KHz is the baseline frequency. The parameter clockFrequency is the frequency in Hz for the communication clock. Each of the I2C modes have agreed upon values:
  - 10000Hz - low speed mode
  - 100000Hz - standard mode
  - 400000Hz - fast mode
  - 1000000Hz - fast mode plus
  - 3400000Hz - high speed mode
- **onReceive(handler)** - registers a function to be called when a peripheral device receives a transmission from a controller device. The handler function is called when a peripheral device reviews data. The handler function is recommended to take a single int parameter, which represents the number of bytes read from the controller.
- **onRequest(handler)** - registers a function to be called when a controller requests data from a peripheral. The handler function to be called takes no parameters.

- setWireTimeout( ), setWireTimeout(timeout, reset_on_timeout) - sets the timeout for Wire transmissions. Timeouts occur when a problem exists with communication. The parameters are:
  - **timeout** - timeout value in microseconds. If zero, timeout checking is disabled.
  - **reset_on_timeout** - if true Wire hardware will reset on timeout.
- **clearWireTimeoutFlag( )** - clears the timeout flag.
- **getWireTimeoutFlag( )** - tests if a timeout has occurred since the last time the timeout flag was cleared.

## I2C Address Finder Code

One of the tasks using an I2C device is determining the device's address. Also, as common with I2C, more than one device may be connected. The simple code below show how to determine I2C addresses.

```
/*
Project: I2CAddressFinder
Finds addresses of connected I2C devices.
Make sure to add lib_deps = Wire in platformio.ini

*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

#define DELAY_TIME 1000

#include <Wire.h>

void setup() {
  Wire.begin();
  Serial.begin(9600);
  Serial.println("Looking for I2C devices...");
}

void loop() {
  byte ret_code, addr;
  int num_I2C_devices;

  // loop through addresses

  for(addr = 1; addr <= 127; addr++) {
    Wire.beginTransmission(addr);
    ret_code = Wire.endTransmission();
    if(ret_code == 0){
      Serial.print("address found at 0x");
      Serial.println(addr,HEX);
      delay(DELAY_TIME);
      num_I2C_devices++;
   } else
     if(num_I2C_devices == 0) {
       Serial.println("no I2C devices found");
      }
  }
}
```

### platform.ini

Two lines are added to the platformio.ini in the [env:uno] block.

```
[env:uno]
platform = atmelavr
board = uno
framework = arduino
lib_ldf_mode = chain+
lib_deps = Wire
```

The lines are lib_ldf_mode = chain+ and lib_deps = Wire. These utilize the Wire library that we need to communicate with the I2C devices.

## Code Walkthrough

- The #define ARDUINO_IDE is used to switch between using the Arduino IDE or PlatformIO. If the Arduino IDE is used, then the file <ardruino.h> is not included.
- A #define DELAY_TIME is declared to introduce a 1 second delay between scans.
- The Wire library is used to send and receive data to and from an I2C device. <Wire.h> is included to access the Wire library functions.
- in setup( ) the wire library is initialized via Wire.begin( ), as well as the serial port via Serial.begin(9600). A message is also displayed.
- In loop( ) two byte variables are declared, ret_code and addr. ret_code is the return code from the method call Wire.endTransmission( ). addr is the address of the I2C device to query. The int variable num_I2C_devices counts the number of I2C devices found.
- A for loop is used to index through the address 1 through 127. Wire.beginTransmission(addr) takes addr as a parameter, attempting to write to that address.
- ret_code = Wire.endTransmission( ) ends the attempted communication at the attempted I2C device address, addr.
- If ret_code is zero, then a device is found at addr. The address is printed, a delay of 1 second is introduced and the number of I2C devices found stored in num_I2C_devices is incremented.
- If the for loop completes and no I2C devices are found, then a “no I2C devices found” message is printed.

Use this code to scan for I2C devices you use in your projects.

# CHAPTER TEN

## Functions and Data Structures

This sections delves into functions and arrays. These are fundamental to C programing and are the foundation on which effective and efficient code is built. This book assumes that you are familiar with the Arduino IDE and the version of C it supports. The examples in this section are written in ANSI C and compiled using Xcode on an iMac. This code is portable and compiles on the GCC compiler. If you have an C compiler running under an IDE such as CodeBlocks, Visual Code Studio or other IDE, it should compiler and run just fine.

### Functions

Functions are an efficient way to perform specific tasks, such as searching, sorting, filling data structures and the like. Let's walk through the code below.

```
#include <stdio.h>

// global vars
int global_var = 0;

// function prototypes
// bare bones function
void f1(void);
// pass by value
int retValFn(int);

int main(int argc, const char * argv[]) {
    int val1 = 0;
    int val2 = 25;

    printf("** Function Demo **\n");
    // call void f1(void)
    printf("global_var before function vcall is %d\n",global_var);
    f1();
    printf("global_var after function call is %d\n",global_var);
    // call retValFn( )
    printf("val1 before function vcall is %d\n",val1);
    val1 = retValFn(val2);
    printf("val1 after function call is %d\n",val1);
    return 0;
}

// f1 - bare bones function
void f1(void) {
    printf("This function changes the value of a global var\n");
    global_var = 100;
}

// retValFn - returns an int
int retValFn(int in_val) {
    in_val += 200;
    return in_val;
}
```

## Code Walkthrough

- The line #include <stdio.h> is a standard input/output header file used in standard C console programs.
- One global variable is declared, int global_var = 0; This will be used by function f1 to change it's value.
- Two function prototypes are declared, void f1(void) and int retValFun(int). Function void f1(void) returns no value and takes no parameter. That's what void does- tells the compiler that it returns nothing and takes no parameters. The int retValFn(int) returns an integer and takes an integer as an argument.
- Inside main( ), two integer variables are declared, val1 and val2. The variable val1 is set to 0 and val2 is set to 25. These variables will be used in the functions.
- In the main body of the code the state of the variables used by the functions are printed before and after the function calls. The first function called is f1. It's called simply as f1( ).
- The second function called is retValFn. This is called as val1 =

retValFn(val2). The variable val1 stores the return value of the function. The parameter is the int variable val2.

- Outside of main( ) the function body are declared. The function void f1(void) is used to set global_var to 100. This is a way of manipulating global variables if they are used. It's generally not recommended to use global variables, but in short sketches they can be useful. If global variables are used, then a function it recommended to change their values if required instead of having the values changed sprinkled through the code.
- The function int retValFn(int in_val) takes an integer as a parameter and returns an integer. retValFn takes the input parameter and adds 200 to it. It then returns the new value.

This very simple overview of functions lay the foundation for more sophisticated function use, such as passing arrays, structures and pointers, as we will see in the next few sections

## Arrays

Arrays are data structures that contain like elements and are stored sequentially in a computer's memory. Arrays are must commonly accessed using for loops. A single array is a contiguous list of like elements. Array elements must be of the same type, such as integers, chars, floats and the like. A typical one dimensional array declaration is shown below.

```
#define MAX_ELEMENTS 5
   int z[MAX_ELEMENTS];
```

The array, named z, contains MAX_ELEMENTS, which in this case is 5.

Arrays can have multiple dimensions, meaning it can have multiple rows and columns. An effective way to visualize multidimensional arrays is by color. Consider the array below:

```
int a[ROW_SIZE][COL_SIZE] = {
      {0,1,2},
      {3,4,5}
   };
```

This is a two dimensional array that contains two rows and three columns. as

```
int a[ROW_SIZE][COL_SIZE] = {
      {0, 1, 2},
      {3, 4, 5}
};
```

The rows are blue and the columns are orange. The most common way to access multidimensional array elements is by row and column. First, the row is selected (blue), then the columns in that row are read (orange). In this example the first row is row 0 and is blue. The first orange element is in column 0. The next orange element is 1, and the last is 2. Once the columns are read, the row index is incremented to read the next row, containing the elements {3,4,5}. The columns are then read to access the column elements. The most common way to read arrays is with a for loop.

## For Loops

For loops are iterators, which means repeating a process a number of times, such as moving sequentially through an array or some other collection of variables. A for loop has the form:

```
for(initial_counter, terminating_condition, increment_counter) {
            do_something;
  }
```

There is no need for a colon following the closing bracket. The initial counter is usually an integer variable set to zero.

A for loop has the form of For i = 0…N, do something. In C, to sequentially load numbers into array z starting at zero, we could code it like this:

```
for(i=0; i < MAX_ELEMENTS;i++) {
   z[i] = i;
  }
```

The variable i is a counter and is usually initialized to zero. It is then compared to some terminal value. In this case, the value is MAX_ELEMENTS which was defined as 5. The line z[i] = i; takes the current value of i, and loads the array with the current value using i as the index. The resulting array is 0,1,2,3,4. Remember that array indexing starts at zero. This causes issues if one is not careful, since it is a bad thing to overrun the boundaries of an array. For example, if the for loop was coded as,

```
for(i=0; i <=MAX_ELEMENTS;i++) {
   z[i] = i;
  }
```

This would overrun the array boundary, trying to load sequential values of 0,1,2,3,4,5. This is where the counting variable i stops at 5, but it has really incremented 6 times. The value 5 will be placed in the next two bytes of memory, which may already be used by something else. This may cause quirky program behavior, if not destroy your code altogether. This has been a source of errors for C programmers since C was conceived. Compilers usual flag an array overrun, but this is not guaranteed, so just be careful.

So let's go back to the multidimensional array and how it's indexed. Our array is,

```
int a[ROW_SIZE][COL_SIZE] = {
    {0, 1, 2},
    {3, 4, 5}
};
```

So we want to index through the rows first them columns. We can visualize the indexing by following the colors.

```
for   outer row
    for   inner col
```

This is the code that starts with the outer row, then indexes through each of the columns.

```
for(int i = 0; i<ROW_SIZE;i++){ // row
      for(int j = 0;j<COL_SIZE;j++) {  // col
         printf("[%d][%d] %d\n",i,j,a[i][j]);
      }
   }
```

There are two for loops, with one within another. This is called a nested loop. The int variable i is the counter variable and is initialized to zero. It then executes the inner for loop, where j is the counter variable. The inner loop indexes through the columns then control is transferred back to the outer loop where the counter variable I is incremented. The process repeats until i<ROW_SIZE is true. Think of it this way, for every one of the row counters, i, is executed, all of the column counters are executed for that particular column.

Let's move on to a three dimensional array. We can again visualize the indexing by following the colors.

```
for   outer row
    for   outer row
        for   inner col
```

## Example Code

Here's an example of using multidimensional arrays. This code is written in ANSI C using Xcode's C compiler and should compile on any up to date C compiler such as GCC.

```
#include <stdio.h>
#define ROW_SIZE 2
#define COL_SIZE 3

int main(int argc, const char * argv[]) {

    int a[ROW_SIZE][COL_SIZE] = {
        {0,1,2},
        {3,4,5}
    };
    int b[ROW_SIZE][ROW_SIZE][COL_SIZE] = {
        {
            {0,1,2},
            {3,4,5}
        },
        {
            {6,7,8},
            {9,10,11}
        }
    };

    printf("** array demo **\n");
    printf("print 2x3 dim array\n");
    for(int i = 0; i<ROW_SIZE;i++){ // row
        printf("\n");
        for(int j = 0;j<COL_SIZE;j++) {  // col
            printf("[%d][%d] %d\n",i,j,a[i][j]);
        }
    }
    printf("\n");
    printf("print 2x2x2 dim array\n");
    for(int i=0;i<ROW_SIZE;i++) {
        printf("\n");
        for(int j=0;j<ROW_SIZE;j++){
            printf("\n");
            for(int k = 0;k<COL_SIZE;k++) {
                //printf("\n");
                printf("[%d][%d][%d] %d\n",i,j,k,b[i][j][k]);
            }
        }
    }

    // get element 10 (which contains the value 11) from b array
    int dat = 0;
    // the order is block - row - col
    dat  = b[1][1][2];
    printf("Element data 10 is %d\n",dat);

    return 0; // program exit
}
```

## Code Walk Through

- The code begins with #include <stdio.h>. This is the standard input/output include that accepts and displays data to a terminal.
- The two #defines define a ROW_SIZE of 3 and a COL_SIZE of 3. These define the size of the two dimensional arrays a and b.
- Inside main( ) two multidimensional arrays are declared, a and b. The first array, a, has two dimensions. The second array, b, has three dimensions. C and C++ are designed to easily accommodate multidimensional arrays.
- Array a contains two rows and two columns. The entire array is surrounded by curly brackets ending with a semicolon. Inside the curly brackets are two more sets of brackets and elements or values, {0,1,2} and {3,4,5}. The convenient thing about multidimensional arrays is that the array is visualized for you. There are two rows and three columns. To find any element in the array, all that needs to be done is to index into the array. For example, to find the element 4, then the index 1,1, since the array is in the second row and second column. Why 1,1? Arrays start at index 0, so in this case 0,0 are the coordinates of element 0. This is the only trick that you need to remember when indexing into an array.
- Array b contains two sets of values. All that was done here was to add two more rows of the same dimension. To find element 11, as shown in the code, we need to index into the second row which will take us to {6,7,8},{9,10,11}, then index into the second row again, which takes us to {9,10,11}. Element 11 is at index 2. So the position of element 11 in the array is b[1][1][2].
- The remainder of the code indexes through the arrays and prints their values using nested for loops. The most inner for loop indexes through the columns and the outer for loops index through the rows.

# CHAPTER ELEVEN

## Information Representation

Working with digital devices without an in-depth understanding of binary logic is like speaking a language without know how to read. Binary numbers and binary logic are the DNA of any digital system. Understanding binary numbers and logical operators are fundamental to computer and embedded systems engineering. From framework of embedded systems, we can classify information in two ways, analog and digital. Analog information is primarily physical. The heat of the sun on our skin is physical/analog information. The sun passing overhead from dawn to dusk is physical/analog information. Physical phenomena is information in its raw form- unmeasured and unquantified but perceived without instrumentation.

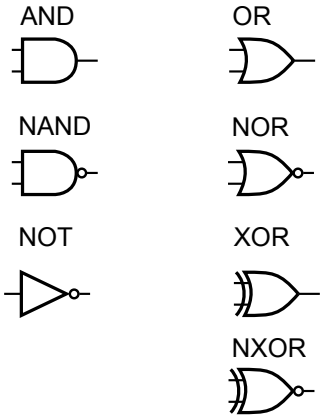
Digital information to be binary, either something is on or off, with no gray area. A room is light or it is dark. A switch is on or it is off. A car will start, or it won't. In order for us to make sense of the world in a consistent, repeatable way we need to quantify and classify the information.

### Digital Logic

In this section we will cover basic binary symbols and operations.

### Basic Binary Operations

Boolean operations are AND, OR, XOR, NOT and complements. Basic binary arithmetic operations are addition, subtraction, division and multiplication. Before we can do subtraction in binary we need to understand how complements work. Let's first look at the fundamental digital logic symbols.




These symbols are foundational and are the building blocks of all computers and microcontrollers. From a simple NAND gate almost any computer can be built, starting from a basic 1 bit memory circuit.

## Binary Conversions

Information representation is an abstract way of dealing with data. In physics, length, mass, volume, time, temperature, density- the list goes on and on. In computers where transistors have to states, on and off, information is represented using the binary numbering system. Binary number consist of two values, 1 and 0, or On and Off.

Binary numbers, like any numbering system, is weighted, as shown below.

$$2^n\ 2^3\ 2^2\ 2^1\ 2^0$$

Starting from the right, we raise 2 to the 0 power which is 1, then in the next position 2 to the 1st power which is 2, then in the next position 2 to the 2nd power which is 4, and so on. The sum of these number is the decimal equivalent. If there is a 1 in one of the positions we count it. If there is a 0 we don't. So what is 1011 in decimal? Using the scheme above we have, starting from the right we have 1, 2, 0, and 8, which added together is 11.

## Converting From Decimal to Binary and Back Again

It's a simple task to whip out a calculator to convert decimal to binary and vice versa. This how we all do it, but it's necessary to understand how binary works in relationship to decimal numbers. For example, where is the decimal point in a binary number, called the binary point? What is the fraction 11/16 in binary? This may seem like esoteric trivial but it isn't. This is fundamental knowledge of how numbering systems work.

The first step is to understand how the decimal system works. First, decimal numbers have a base of 10. Think of a base as a number limit. As each number in each digit position gets to 10, it spills over to the next digit position. If we wanted to be formal, we could denote the number 11 in decimal as $11_{10}$, that is 11 with the subscript 10 denoting the base. Binary number are base 2, so 11 in base 2 is $1011_2$. So how do we convert 11 decimal to 1011 binary? We do this with successive division by 2, as shown below.

```
11/2 = 5  R1
5/2  = 2  R1
2/2  = 1  R0
1/2  = 0 R1
```

At first it looks a little strange, but we'll go through it step by step.

- Divide 11 by 2, which yields 5.5. We keep the 5 before the decimal place, which is called the exponent, and we throw away the 5 after the decimal place, called the mantissa. But what we do remember is that the mantissa wasn't zero, so we denote that a remainder existed. If there is a remainder, we mark that as a 1. If there is not a remainder, we mark that as a 0, as shown as R1 in the figure above.
- Divide the exponent of the last division, which is 5, and divide by 2, which yields 2.5. We keep the 2 and get rid of the 5, but note that 5 was a remainder. As in the last step, we mark the remainder as a 1 if it exists and 0 if it doesn't. A remainder exists here, so we mark it R1. Note there is nothing special about the repeating remainder 5- it just worked out that way for this example.
- Divide the exponent of the last division, which is 2, and divide by 2. This yields 1. 2 goes into 2 evenly so there is no remainder. We mark it R0.
- divide the exponent of the last division, which is 1, by 2. 2 goes into 1 zero times but has a remainder of 5. Again we throw away the 5 but remember there was a remainder so we mark it R0.
- Since we now have an exponent of 0, we're done. We start at the bottom of the list. The bottom division remainder is the rightmost bit, or least significant bit, of the binary number. This is R1, which we denote as 1. We go up the list of remainders, finally coming to 1011, which is 11 binary.
-

Binary numbers, like any numbering system, is weighted, as shown below.

$$2^n\ 2^3\ 2^2\ 2^1\ 2^0$$

Starting from the right, we raise 2 to the 0 power which is 1, then in the

next position 2 to the 1st power which is 2, then in the next position 2 to the 2nd power which is 4, and so on. The sum of these number is the decimal equivalent. If there is a 1 in one of the positions we count it. If there is a 0 we don't. So what is 1011 in decimal?. Using the scheme above we have, starting from the right we have 1, 2, 0, and 8, which added together is 11.

The hardest part of all this is knowing which end is which. To convert from decimal to binary with successive division by 2, the last division remainder will be in the most significant position, which will be on the left. The first division remainder with be in the last, or least significant position.

## Bit Operation Code

When working with microcontrollers bit manipulation is a must-know skill to master. Setting and resetting bits in registers and memory efficiently require bit operations. Below is code showing bit operations in action. The code is broken up into three sections and a walkthrough discussion is provided for each. There are a few key functions and designed decisions made, and they will be explained. The code uses the serial monitor for getting input from a PC keyboard and receiving the result of the binary operations from an Arduino UNO. A simple menu is displayed when the sketch is run, which is described in the sketch header.

The bitwise operators demonstrated are AND, OR, XOR, ones complement, shift left one bit and shift right one bit. A simple menu is displayed corresponding to PC keyboard inputs, as shown in the keyboard input: header description. The sketch was developed in PlatformIO. Here's the header for the sketch.

```
/*
Project: BitOpDemo
File: main.cpp

Tests bitwise operators:
& AND operator
| OR operator
^ XOR operator
~ Ones Complement
<< shift left ex: a << 2 shifts left by 2 bits
>> shift right ex: a >> 2 shifts right by 2 bits

keyboard Input:
1 – 49 ASCII get first number
2 – 50 ASCII get second number
3 – 51 ASCII = AND
4 – 52 ASCII = OR
5 – 53 ASCII = XOR
6 – 54 ASCII = 1s complement
7 – 55 ASCII = shift left one position
8 – 56 ASCII = shift right one position
9 – 57 ASCII = redisplay menu
*/
```

Pressing the 1 key, which is ASCII 49, calls a function to get the first number. Pressing the 2 key, which is ASCII 50, calls a function to get the second number. The remainder of the functions, invoked by pressing keys 3-8 perform the logical operations. To re-display the menu, the 9 key is pressed.

The first code block sets up the variables, function prototypes and initializing the serial monitor.

```
//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

// declare struct and buffer input number storage
#define BUFFER_SIZE 30
char buffer[BUFFER_SIZE];
struct nums {
    unsigned int n1;
    unsigned int n2;
  };
  struct nums in_num;

  // strings needed for Serial.readString();
  String firstNumStr = "";
  String secondNumStr= "";

// function prototypes
int getInput(void);
void showSelection(void);
void getFirstNum(void);
void getSecondNum(void);
void doAND(void);
void doOR(void);
void doXOR(void);
void doONES(void);
void shiftLeft(void);
void shiftRight(void);

void setup() {
  while(!Serial) // wait until serial port is availble
    ;
  // init serial port and flush the buffer
  Serial.begin(9600);
  Serial.println("* Bitwise Operator Demo *");
  showSelection();
}
```

## Code Walkthrough

- The code starts out with the familiar ARDUINO_IDE guard. This line #define ARDUINO_IDE should be un-commented if the Arduino IDE is used.
- A buffer is declared to hold the incoming string data received from the PC keyboard that will be used by the function sprintf( ) function in the function bodies. A buffer size of 30 was selected, but is adjustable via the #define BUFFER SIZE 30. The buffer is an array of characters.
- A structure is declared to hold the numerical values that will be operated on. The structure sums holds two integer values, n1 and n2. The structure declaration defines a template for a new data type.
- In order to use the structure, a variable of the structure type needs to be declared. The declaration struct nums in_num;

accomplishes this. Now, when the struct variables are accessed, they will be in_num.n1 and in_num.n2.

- Two variables of type String are declared, firstNumStr and secondNumStr. These are both set to null via the two quotation marks together. It's important to set these strings to null when reading in data from the PC keyboard via the serial monitor.
- Next, the function prototypes are declared for getting input from the PC keyboard, displaying the menu and performing the logical operations on the input.
- In setup the first statement waits for the serial port to become available on the Arduino UNO. Usually the UNO's serial port is immediately available, but on other boards it is not. The while (!Serial) ; waits for the serial port to be available and is good practice to use this code in your sketches.
- The serial port is initialized to 9600 baud and showSelection( ) is called to display the menu.

The next block is the main body of the sketch.

```
void loop() {
  int toggle = 0;
  int numIn = 0;

  // get menu selection
  numIn = getInput();

  switch(numIn) {

   case 49:  // 1 key – get first number
      Serial.println("Input first number: ");
       // call get first number function
       getFirstNum();
       Serial.print("firstNumStr = ");
       Serial.println(firstNumStr);

       // convert to int
       in_num.n1 = firstNumStr.toInt();
       firstNumStr = "";
     break;

     case 50: // 2 key – get second number

       Serial.println("Input 2nd number: ");
       // call get first number function
       getSecondNum();
       Serral.print("secondNumStr = ");
       Serial.println(secondNumStr);

       // convert to int
       in_num.n2 = secondNumStr.toInt();
       secondNumStr = "";
     break;

     case 51:  // 3 key – AND first and second number

       Serial.println("* AND selected *");
       doAND();
     break;

     case 52:  // 4 key – OR first and second number
       Serial.println("* OR selected *");
       doOR();
     break;
     case 53:  // 5 key – XOR first and second number

       Serial.println("* XOR selected *");
       doXOR();
     break;

     case 54:  // 6 key – 1s complent first number
       Serial.println("1s complement selected");
       doONES();
     break;

     case 55:  // 7 key – shift left first number
       Serial.println("shift left selected");
       shiftLeft();
     break;

     case 56: // 8 key – shift right first number
       Serial.println("* shift right selected *");
       shiftRight();
     break;

       case 57: // 9 key – re-display menu
       if(toggle == 0) {
         showSelection();
         toggle = 1;
       }
     break;

     default:
       break;
  }
}
```

## Code Walkthrough

The code processes the menu selection and calls the appropriate bit functions via a switch( ) statement. Switch statements are efficient and a great way to code state machines.

- Two int variables are declared inside loop( ). toggle is used to stop the display from repeating constantly in the function.
- The menu selection is determined by a call to getInput( ) and stores the value in numIn, which is described in the code walkthrough below.
- Based on the value of numIn, the switch statement executes a case statement. Since the value of numIn is the ASDII value of the key that was pressed, that is what's used for each case. Switch statements are easy to follow and code. In case 49 (the 1 key) a call to getFirstNum( ) is made. The input from the keyboard is read as a string, printed, then converted to an integer via firstNumStr.toInt( ).The results are stored in the structure member in_num.n1. The identical process for the second number is executed in case 50.
- The remainder of the functions call their respective bit operation functions. If the menu selection input is not 1 though 9, the switch statement falls through to the default case, which breaks out of the switch statement.

The last block of code contains the functions.

```
// :- getInput - gets raw input from keyboard
int getInput(void) {
   int n;
   if(Serial.available()>=0) {
    n = Serial.read()
    delay(10);
    Serial.flush();
  }
  return n;
}

// :- getFirstNum - gets first number to operate on
void getFirstNum(void) {
  while(firstNumStr.equals("")) {
    firstNumStr = Serial.readString();
  }
}

//:-  getSecondNum - gets second number to operate on
void getSecondNum(void) {
  while(secondNumStr.equals("")) {
    secondNumStr = Serial.readString();
  }
}

// :- showSelection - displays menu
void showSelection(void) {
  Serial.println("Enter Selection: ");
  Serial.println("1 to enter first number");
  Serial.println("2 to enter second number");
  Serial.println("3 = AND");
  Serial.println("4 = OR");
  Serial.println("5 = XOR");
  Serial.println("6 = 1s complement");
  Serial.println("7 = Shift left");
  Serial.println("8 = Shift right");
  Serial.println("9 - show this menu");
}

// :- doAND - AND 2 numbers
void doAND(void) {
  int result = in_num.n1 & in_num.n2;
  sprintf(buffer,"0x0%x AND 0x%x is 0x0%x",in_num.n1,in_num.n2,result);
  Serial.println(buffer);
}

// :- doOR - OR 2 numbers
void doOR(void) {
  int result = in_num.n1 | in_num.n2;
  sprintf(buffer,"0x0%x OR 0x%x is 0x0%x",in_num.n1,in_num.n2,result);
  Serial.println(buffer);
}

// :- doXOR - XOR 2 numbers
void doXOR(void) {
  int result = in_num.n1 ^ in_num.n2;
  sprintf(buffer,"0x0%x XOR 0x%x is 0x0%x",in_num.n1,in_num.n2,result);
  Serial.println(buffer);
}

// :- doONES - perform 1s complement on the two input numbers
void doONES(void) {
  int result = ~in_num.n1;;
  sprintf(buffer,"1s Compliment for 0x%x is 0x%x",in_num.n1,result);
  Serial.print(buffer);
  Serial.println();
  result = ~in_num.n2;
  sprintf(buffer,"1s Compliment for 0x%x is 0x%x",in_num.n2,result);
  Serial.println(buffer);
}

// :- shiftLeft - shift n1 left 1 bit
void shiftLeft(void) {
  int temp = in_num.n1;
  int result = temp << 1;
  sprintf(buffer,"0x0%x shift left is 0x%x",temp,result);
  Serial.println(buffer);
}

// :- shiftRight - shift n1 right 1 bit
void shiftRight(void) {
  int temp = in_num.n2;
  int result = in_num.n1 >> 1;
  sprintf(buffer,"0x0%x shift right is 0x%x",temp,result);
  Serial.println(buffer);
}
```

## Code Walkthrough

This block of code contains the function bodies, which is outside of loop( ). Each function has a short header describing what it does. Notice

the :- characters. This unique character sequence is used to easily search for functions, especially in larger programs.

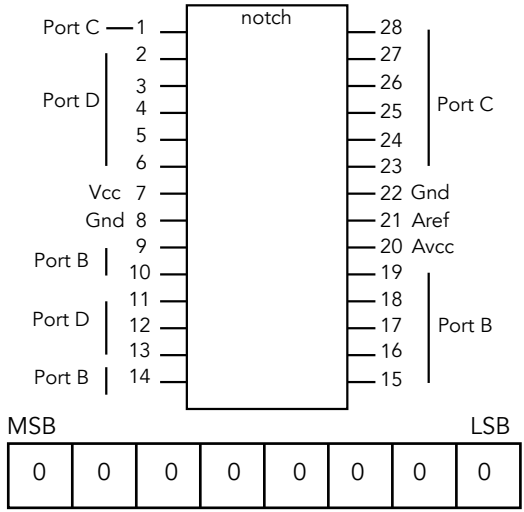
- getInput( ) gets input from the PC keyboard via the serial monitor. It tests whether the serial port has one or more characters available then a call to Serial.read( ) is made to store the character in local variable n. A slight delay of 10ms is used to let things settle and the serial port is emptied via the call to Serial.flush( ). The character read in from the serial port is returned via n.
- getFirstNum reads the first number represented as a string from the PC keyboard. It reads a string from the serial port via Serial.readString( ) while the String variable firstNumStr is equal to "" or null. This is why firstNumStr was set to num when first declared. Serial.readString( ) waits for 1000ms after the end of input. This is the default and can be changed via a call to Serial.timeout( ), where the parameter is a time value.
- getSecondNum is identical to getFirstNum, but reads the second input string.
- showSelection displays the menu in the serial monitor. The user inputs 1 though 9 to enter the numbers, the bit operations and redisplay the menu.
- doAND ands the two numbers, processes the results and displays the output on the serial monitor. The AND operator is a single &. A temporary variable results stores the results of the bitwise operation. The important function here is sprintf( ), which makes formatted output easy to manipulate. The arguments are buffer, which is the global buffer[BUFFER_SIZE] declared at the start of the code, the format specifiers, and the contents to be displayed. The format specifiers, 0x0%x represents the data as hexadecimal. There are three format specifiers for the first two number and the result of the AND operation. Note the actual format specifier for hexadecimal is %x. The 0x before the %x is added for readability. sprintf( ) processes the arguments and writes them to buffer.
- doOR ORs the two numbers and is the identical format and display code as doAND. The OR operator is a single |.
- doXOR exclusive ORs the two numbers and is the identical format and display code as doAND. The XOR operator is a single ^.

- doONES performs a one's complement on the first number and second number. The operator for ones complement is a single ~. A ones complement is performed on the first number, stored in result, formatted by sprintf( ) and printed to the serial buffer. The same sequence is performed for the second number.
- shiftLeft shifts the first number one place to the left and like the other functions, uses sprintf( ) to format the data.
- shiftRight shifts the first number one place to the right and is identical to shiftLeft for the remainder of processing.

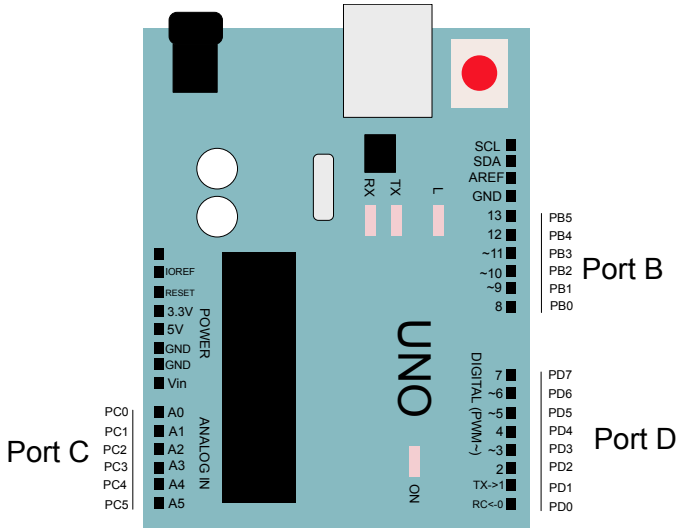
This is not the typical microcontroller sketch. There is no I/O and is designed more like a traditional C program. What's important is the usage of two-way serial communication, formatting output via sprintf( ), and the logical operators themselves.

## Bit Operations and Ports

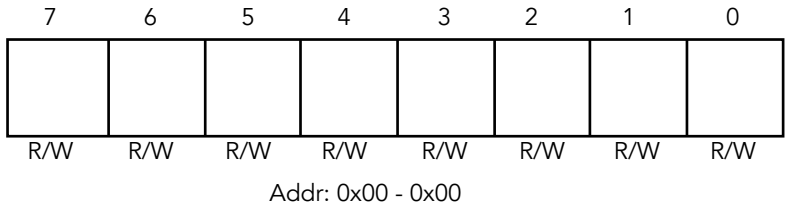
Here's an example of bit operations in action with an Arduino Uno. Four ports are available on an Arduino UNO, Port C, Port B and Port D. Each port is controlled by an eight bit register, with the most significant byte, or MSB, on the left and the least significant byte, or LSB, on the right. The ATmega 328p pinout is shown below. The Port registers are 8 bits wide.




The pins of the 328p do not map one-to-one to the Arduino Uno pins. The ports with respect to I/O pins on an Arduino Uno are shown below.




Not all of the bits in the ports are used as I/O pins. For example, in Port D, bits 0 and 1 (PD0 and PD1) are allotted for transmit and receive. This can cause communication and debugging issues when trying to utilize these pins for other purposes. The general form of the registers is shown below.




The most significant bit (MWB) is bit 7 and the least significant bit (LSB) is bit 0. R/W means Read or Write. Care must be taken when writing directly to these registers since R/W are different values for different registers. Below is a drawing of the Data Direction Register, or DDRD for Port D. What's explained below is the same for Port C and Port B. Values of 0 or 1 are written to this register to determine if a bit will be an input or an output.

DDRD Port D Data Direction Register

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Addr: 0x0A - 0x2A

If a 0 is written to a bit in the DDRD, then the bit will be an input. If a 1 is written to a bit in the DDRD, then the bit will be an output. For example, binary 11111100 is written to the data register, then bits 2-7 are inputs and bits 1-0 are outputs. Directly writing bits to a port can trigger unwanted results. In the case of Port D, we know that bits 0 and 1 are used for receive and transmit and we will want to leave these bits alone. This is explained in detail below.

PORTD Port D Data Register

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Addr: 0x0B - 0x2B

The Port D Data Register, PORTD, is where data is present to be written to or read from the outside world. A 1 on a bit is a HIGH, and a 0 is a low, as used in digitalWrite(pin, value), where value is HIGH or LOW.

The Port D Input Pin Addresses register, PIND, is a read only register that contains the state of the inputs. This is convenient to read all of the input bits at the same time, if the Port D is configured for inputs.

PIND Port D Input Pin Addresses

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Addr: 0x09 - 0x29

To write ones to a port registers DDRD and PORTD the OR operator is used, and to write zeros to a register the AND operator is used. The OR operator is |, and the and operator is &. A detailed example program demonstrating how to use the logical operators is given in the section entitled Bit Operation Code. In this section, code is provided that explicitly manipulates PORTD and PORTB.

PORT definitions are built-in to the Arduino framework and can be used freely in code. The definitions are listed below. Remember that DDR and PORT registers may be written to and read from, but the PIN registers can only be read from. The list below shows the definitions for the three ports, PORTD, PORTB and PORTC.

| Port | Description |
|---|---|
| Port D | Maps to Arduino UNO pins 0-7 |
| DDRD | Data direction register R/W |
| PORTD | data register R/W |
| PIND | input pin register Read Only |
| Port B | maps to Arduino Uno pins 8=13. Pins 6 and 7 are not usable. |
| PORTB | data register R/W |
| PINB | input pin register Read Only |
| Port C | maps to Arduino UNO pins A0 - A5 |
| DDRC | data direction register R/W |
| PORTC | data register R/W |
| PINC | input pin register Read Only |

Remember there are potential issued with PORTD pins 0 and 1 that are labeled for RX and TX. Again, these pins should not be used for general purpose I/O pins. This makes it more complicated to cleanly use a full 8 bit byte when manipulating registers.

To initialize bits in a register, it's proper to write directly to a register with discrete values. For example, to set a port, you may use PORTD = B11111111, where B is the binary type designator. The astute reader will quickly realize that the UNO pin 0 and pin 1 are set HIGH, potentially overwriting the bites set for RX and TX. This can be avoided by using the OR operator. The initialize the port using OR you may use PORTD |= B11111100. This leaves the last two bits in their current state. To write a zeros to a register, the AND operator is used. Following the same reasoning, PORTD &= B00000011 can be used. The last two bits will remain a 1 if they are already in that state.

For example, if bit 2 is to be set on PORTD, we can write PORTD |= 00000100. Regardless of the state of bit Port D bit 6, this code will write a 1. If bit 2 is to be reset or set to 0, we can write PORTD &= 11111011. If the ones in Port D are already set to 1, then they will persist since 1 AND 1 is 1.

So why bother to use registers? Primarily for speed. The method calls digitalWrite( ) and digitalRead( ) contain several instructions that take many clock cycles to execute.

So why not directly manipulate registers all all the time? The primary reason is portability. What runs on one Atmel controller with respect to registers may be different on another. Using digitalWrite( ) and digitalRead( ) obviates this. Another reason, but less important in my opinion, is code readability. Using oblique-looking boolean operators everywhere may make the code hard to read and debug. It's also easy to make mistakes using bit strings like Bxxxxxxxx. A misplaced bit may wreak havoc on your program.

The big takeaway is, if I/O speed is critical, consider using direct reads and writes to registers. If you do decide to use them, just be conscientious and careful.

## The Code

The code below illustrates how to initialize a register and write values to a register. The whole concept focuses on speed and we will toggle bit 2 of Port D to see how fast it runs. The code also uses digitalWrite( ) to toggle bit 2. The code is designed to show the minimum timing of an output turning on and off. For example, if a device requires a time-critical output pulse then digitalWrite( ) may be too slow and using direct register programming May be a solution.

```
/*
Project: RegisterBitTiming
Demonstrates back-to-back timing via writing
1s and 0s to bit 2 PORTD and doing the same
thing using digitalWrite( ) to compare and
contrast the difference.
To use this code, comment out the regisiter code
and uncomment pinMode() and digitalWrite() lines of code.
*/

//#define ARDUINO_IDE
#ifndef ARDUINO_IDE
#include <Arduino.h>
#endif

  // init Port D bit 2 as output
  DDRD = B00000100;

  //Equivalent code using pinMode()
  //for(int i = 2; i < 10; i++){
   // pinMode(i,OUTPUT);
  //}
}

void loop() {
  //write all zeros to PORTD.
  PORTD &= B00000011;
  // write a 1 to PORD bit 2
  PORTD |= B00000100;

  // using digitalWrite( )
 // digitalWrite(2,HIGH);
  //digitalWrite(2,LOW);
}
```

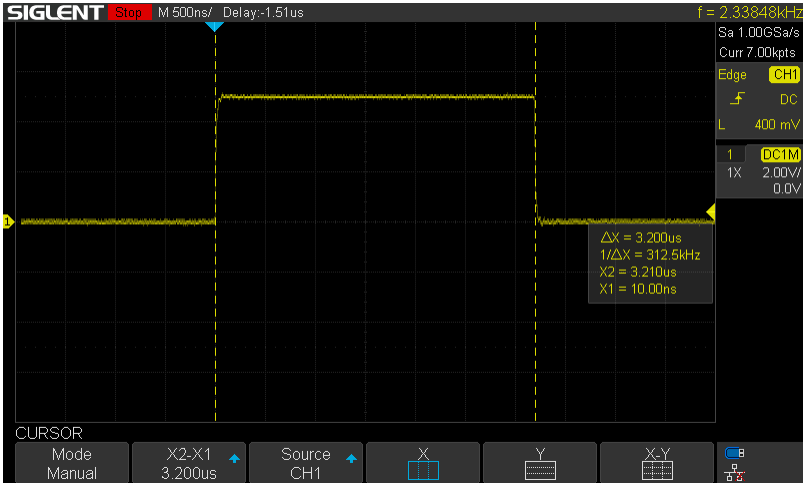
## Code Walkthrough

- First the #define ARDUINO_IDE is commented out since the code was written using PlatformIO. If the Arduino IDE is used, then the #define is uncommented.
- Port D data direction register is written to, setting bit 2 as an output via B00000100. Setting a 1 at bit position zero sets pin 2 as an output. Port D pin 2 maps directly to the Arduino UNO pin 2.
- The standard pin mode initialization is commented out. To compare and contrast the timing difference between the direct register programming, comment out the DDRD = B00000100 line and uncomment the for loop that iterates over pinMode( ).
- In loop( ) Port D is set, particularly bit 2, to zeros by using the AND & operator and the bit pattern B00000011. Bits 0 and 1 are left as they are.
- Port D bit 2 is set to a 1 via the OR | operator.
- The next two lines of code toggle pin 2 via calls to digitalWrite( ).To see the timing difference between the direct register

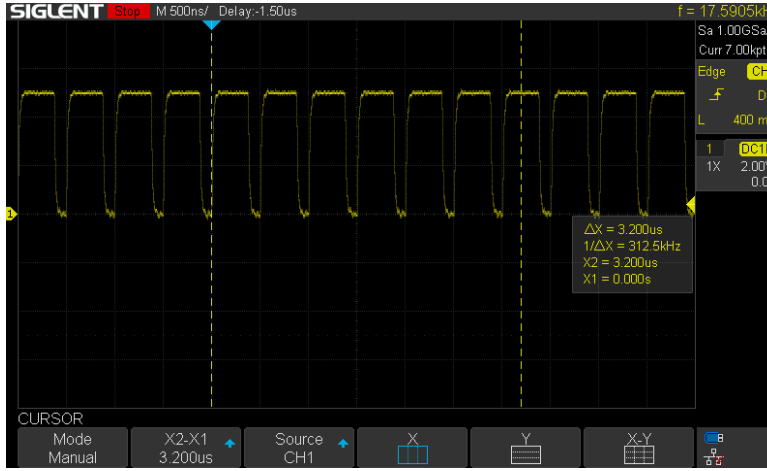
manipulation and the calls to digitalWrite( ), comment out the PORTD lines and uncomment the digitalWrite( ) lines.

## Going Further

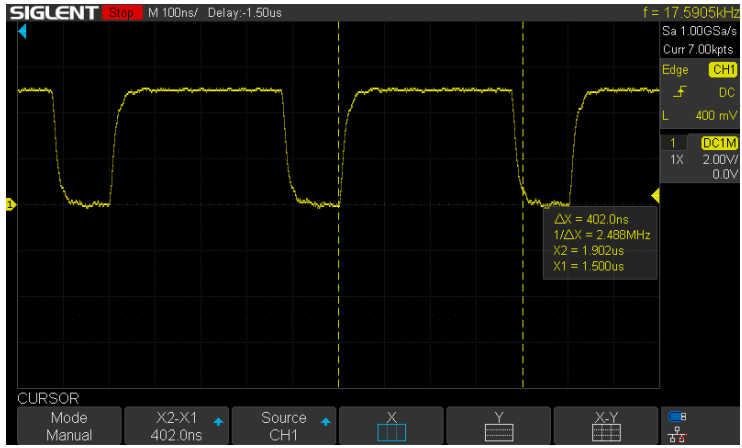
To illustrate the execution time difference between direct register programming and calls to digitalWrite( ) and oscilloscope is needed. Below is a trace using calls to digitalWrite( ).




The on-time for pin 2 is 3.2 micro seconds. The trace below illustrates the difference between direct register programming and calls to digitalWrite( ). The scope settings or measurement cursors were not changed.




Pin 2 toggles a little over six times as fast. An expanded trace is shown below.




The width of the pin 2 pulse is 402.0 nanoseconds, or 0.402 microseconds, which is a vast increase in speed over the calls to digitalWrite( ).

## ASCII Chart

The chart below is a quick reference for ASCII characters 0 through 127 for decimal, binary, hex and the associated ASCII code.

| Decimal / Binary / Hex / ASCII Chart | | | |
|---|---|---|---|
| Decimal | Binary | Hex | Ascii |
| 0 | 0000 0000 | 0x00 | NUL |
| 1 | 0000 0001 | 0x01 | SOH |
| 2 | 0000 0010 | 0x02 | STX |
| 3 | 0000 0011 | 0x03 | ETX |
| 4 | 0000 0100 | 0x04 | EOT |
| 5 | 0000 0101 | 0x05 | ENQ |
| 6 | 0000 0110 | 0x06 | ACK |
| 7 | 0000 0111 | 0x07 | BEL |
| 8 | 0000 1000 | 0x08 | BS |
| 9 | 0000 1001 | 0x09 | HT |
| 10 | 0000 1010 | 0x0A | LF |
| 11 | 0000 1011 | 0x0B | VT |
| 12 | 0000 1100 | 0x0C | FF |
| 13 | 0000 1101 | 0x0D | CR |
| 14 | 0000 1110 | 0x0E | SO |
| 15 | 0000 1111 | 0x0F | SI |
| 16 | 0001 0000 | 0x10 | DLE |
| 17 | 0001 0001 | 0x11 | DC1 |
| 18 | 0001 0010 | 0x12 | DC2 |
| 19 | 0001 0011 | 0x13 | DC3 |
| 20 | 0001 0100 | 0x14 | DC4 |
| 21 | 0001 0101 | 0x15 | NAK |
| 22 | 0001 0110 | 0x16 | SYN |
| 23 | 0001 0111 | 0x17 | ETB |
| 24 | 0001 1000 | 0x18 | CAN |
| 25 | 0001 1001 | 0x19 | EM |
| 26 | 0001 1010 | 0x1A | SUB |
| 27 | 0001 1011 | 0x1B | ESC |
| 28 | 0001 1100 | 0x1C | FS |
| 29 | 0001 1101 | 0x1D | GS |
| 30 | 0001 1110 | 0x1E | RS |
| 31 | 0001 1111 | 0x1F | US |

| Decimal / Binary / Hex / ASCII Chart | | | |
|---|---|---|---|
| Decimal | Binary | Hex | ASCII |
| 32 | 0010 0000 | 0x20 | SP |
| 33 | 0010 0001 | 0x21 | ! |
| 34 | 0010 0010 | 0x22 | " |
| 35 | 0010 0011 | 0x23 | # |
| 36 | 0010 0100 | 0x24 | $ |
| 37 | 0000 0101 | 0x25 | % |
| 38 | 0000 0110 | 0x26 | & |
| 39 | 0000 0111 | 0x27 | ' |
| 40 | 0000 1000 | 0x28 | ( |
| 41 | 0000 1001 | 0x29 | ) |
| 42 | 0000 1010 | 0x2A | * |
| 43 | 0000 1011 | 0x2B | + |
| 44 | 0000 1100 | 0x2C | , |
| 45 | 0000 1101 | 0x2D | - |
| 46 | 0000 1110 | 0x2E | . |
| 47 | 0000 1111 | 0x2F | / |
| 48 | 0011 0000 | 0x30 | 0 |
| 49 | 0011 0001 | 0x31 | 1 |
| 50 | 0011 0010 | 0x32 | 2 |
| 51 | 0011 0011 | 0x33 | 3 |
| 52 | 0011 0100 | 0x34 | 4 |
| 53 | 0011 0101 | 0x35 | 5 |
| 54 | 0011 0110 | 0x36 | 6 |
| 55 | 0011 0111 | 0x37 | 7 |
| 56 | 0011 1000 | 0x38 | 8 |
| 57 | 0011 1001 | 0x39 | 9 |
| 58 | 0011 1010 | 0x3A | : |
| 59 | 0011 1011 | 0x3B | ; |
| 60 | 0011 1100 | 0x3C | < |
| 61 | 0011 1101 | 0x3D | = |
| 62 | 0011 1110 | 0x3E | > |
| 63 | 0011 1111 | 0x3F | ? |

| Decimal / Binary / Hex / ASCII Chart | | | |
|---|---|---|---|
| Decimal | Binary | Hex | ASCII |
| 64 | 0100 0000 | 0x40 | @ |
| 65 | 0100 0001 | 0x41 | A |
| 66 | 0100 0010 | 0x42 | B |
| 67 | 0100 0011 | 0x43 | C |
| 68 | 0100 0100 | 0x44 | D |
| 69 | 0100 0101 | 0x45 | E |
| 70 | 0100 0110 | 0x46 | F |
| 71 | 0100 0111 | 0x47 | G |
| 72 | 0100 1000 | 0x48 | H |
| 73 | 0100 1001 | 0x49 | I |
| 74 | 0100 1010 | 0x4A | J |
| 75 | 0100 1011 | 0x4B | K |
| 76 | 0100 1100 | 0x4C | L |
| 77 | 0100 1101 | 0x4D | M |
| 78 | 0100 1110 | 0x4E | N |
| 79 | 0100 1111 | 0x4F | O |
| 80 | 0101 0000 | 0x50 | P |
| 81 | 0101 0001 | 0x51 | Q |
| 82 | 0101 0010 | 0x52 | R |
| 83 | 0101 0011 | 0x53 | S |
| 84 | 0101 0100 | 0x54 | T |
| 85 | 0101 0101 | 0x55 | U |
| 86 | 0101 0110 | 0x56 | V |
| 87 | 0101 0111 | 0x57 | W |
| 88 | 0101 1000 | 0x58 | X |
| 89 | 0101 1001 | 0x59 | Y |
| 90 | 0101 1010 | 0x5A | Z |
| 91 | 0101 1011 | 0x5B | [ |
| 92 | 0101 1100 | 0x5C | \ |
| 93 | 0101 1101 | 0x5D | ] |
| 94 | 0101 1110 | 0x5E | ^ |
| 95 | 0101 1111 | 0x5F | _ |

| Decimal / Binary / Hex / ASCII Chart | | | |
|---|---|---|---|
| Decimal | Binary | Hex | ASCII |
| 96 | 0110 0000 | 0x60 | ` |
| 97 | 0110 0001 | 0x61 | a |
| 98 | 0110 0010 | 0x62 | b |
| 99 | 0110 0011 | 0x63 | c |
| 100 | 0110 0100 | 0x64 | d |
| 101 | 0110 0101 | 0x65 | e |
| 102 | 0110 0110 | 0x66 | f |
| 103 | 0110 0111 | 0x67 | g |
| 104 | 0110 1000 | 0x68 | h |
| 105 | 0110 1001 | 0x69 | i |
| 106 | 0110 1010 | 0x6A | j |
| 107 | 0110 1011 | 0x6B | k |
| 108 | 0110 1100 | 0x6C | l |
| 109 | 0110 1101 | 0x6D | m |
| 110 | 0110 1110 | 0x6E | n |
| 111 | 0110 1111 | 0x6F | o |
| 112 | 0111 0000 | 0x70 | p |
| 113 | 0111 0001 | 0x71 | q |
| 114 | 0111 0010 | 0x72 | r |
| 115 | 0111 0011 | 0x73 | s |
| 116 | 0111 0100 | 0x74 | t |
| 117 | 0111 0101 | 0x75 | u |
| 118 | 0111 0110 | 0x76 | v |
| 119 | 0111 0111 | 0x77 | w |
| 120 | 0111 1000 | 0x78 | x |
| 121 | 0111 1001 | 0x79 | y |
| 122 | 0111 1010 | 0x7A | z |
| 123 | 0111 1011 | 0x7B | { |
| 124 | 0111 1100 | 0x7C | \| |
| 125 | 0111 1101 | 0x7D | } |
| 126 | 0111 1110 | 0x7E | ~ |
| 127 | 0111 1111 | 0x7F | DEL |

# CHAPTER TWELVE

## PlatformIO

Here's how to utilize git and GitHub with PlatformIO. It's convenient and fairly easy to keep code under source control within PlatformIO. There are several steps you need to take. Before committing real projects I suggest experimenting several times with the following steps so you can get used to them. It takes awhile. Just go slow and practice creating a few dummy repositories and PlatformIO projects, and you will be good to go. You will need a GitHub account either free or paid.

Here's a bulleted list of the GitHub sequence above, which is easier to refer to one you have created a few repositories and do not have to follow screen-by-screen.

- Create a new GitHub repository - make sure that you include a description and README file.
- Start VSCode/PlatformIO.
- From the side menu in PlatformIO select Clone Git Project.
- Go to the GitHub repository and click the Code button and copy the repository URL.
- Select a location. Put it in or create a directory where you intend to keep your code.
- From the File menu, create a new workspace using File->Save Workspace As… Put this in the same location as the repository.
- In PlatformIO select Projects and Configuration.
- Create a new project. Do not use the default location. Put the project in the same folder as the GitHub repo.
- Click Finish. Trust if asked (you are the trusted author).
- Select the source control icon in the left side menu.
- There will be a number on the right of the source control icon (a little branch). These are the number of files that need to be staged before being committed.
- Select + to stage the files to be committed. The files will now be ready to commit to the repository.
- Make sure to add a Commit Message in the top window before clicking Commit. Otherwise Commit will just spin.
- Click Commit to add to the GitHub repository. Trust if asked.
- Sync changes.
- Now check GitHub. Your project should appear in the

repository.

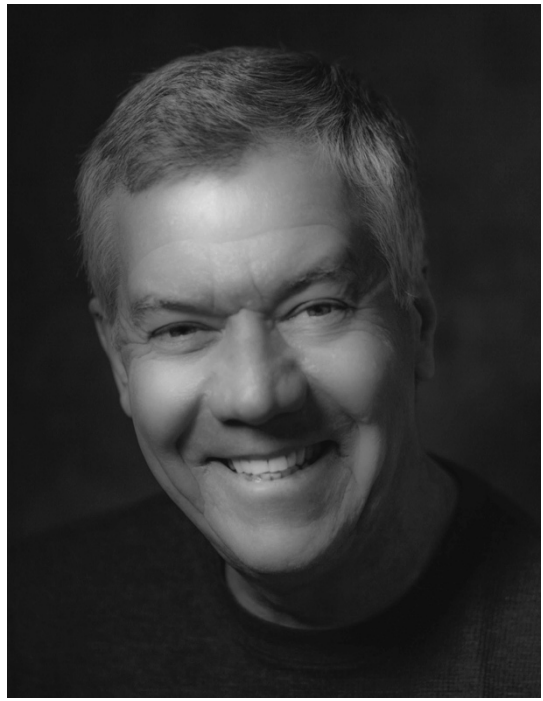
You can always delete your repositories at any time. If you are working on a valuable project, I suggest using source control, even if you are the sole programmer. Many individual programmers use comment headers in code files. This is okay for small prototypes but for any large project, using source control can keep you out if trouble.

# CHAPTER THIRTEEN

## The Embedded Systems World

Embedded systems are all around us—mostly invisible and powering everything from household devices and automobiles to medical equipment and industrial infrastructure. Unlike general-purpose computers, these systems are designed with a specific purpose in mind, where reliability, efficiency, and simplicity matter more than raw computing power. By learning to work with platforms like Arduino, you are stepping into a world where software meets the physical world—where code doesn't just run, but actually does something useful. I hope you found this book useful and more importantly, I hope it motivates you to go further into the world of embedded systems with microcontrollers as they evolve.

# About the Author



Jeff Stefan is a registered patent agent and the author of Patents and Inventive Thinking. Jeff managed intellectual property at a major advanced technology company. Jeff has a Master's Degree in Computer Science in Artificial Intelligence and is also an inventor on twenty nine U.S. utility patents. Earlier in his career Jeff spent years as a software engineer working on control software and embedded applications, including bare metal programming. For more, check out https://jmstefan.com.