Rapid Electronics Prototyping with Arduino

Introduction to Arduino

By

Benjamin Tyler, PE
PDH4Engineers

July 20, 2022

Contents

Introduction

This course is an introduction to Arduino, an ideal platform for rapid development of programmable electronics. The course will discuss what Arduino is, how to use it, who should use it, what kinds of things can be made with it, and the practical limitations of the platform. We will walk through a few sample projects to demonstrate core concepts of Arduino coding and interfacing.

Arduino has simplified the software and hardware aspects of electronics development. Writing code for Arduino has a lower learning curve than traditional embedded systems development. The low level software and hardware design work have mostly been done for users. There is a large community built around Arduino, with a lot of open source software and hardware that can be used "off the shelf." Consequently, Arduino enables an engineer to take an idea to prototype much quicker.

This course is for engineers with at least a rudimentary grasp of programming and electronics. You should be able to read a schematic diagram and recognize elements like resistors and capacitors. Arduino is very beginner friend, yet those with more software development experience will not be held back.

Author Introduction

This course has been produced by Benjamin Tyler, PE, who has over 15 years designing embedded systems, data acquisition and automation systems, and mobile apps.

Suggested Course Materials

While this course can be completed without an Arduino board, you will get much more out of this course if you follow along with one. I recommend the Arduino Uno, as it is inexpensive and very capable. Furthermore, I recommend you buy a kit that contains an Arduino board, sensors, and other components. You can use an Arduino board other than the Uno, but you might need to make minor changes to the project code. The Arduino Leonardo board is a great alternative to the Uno, and one advantage it has over the Uno is that it can function as a USB peripheral. There are a few places you can buy an Arduino board or kit that includes the board:

- ***www.arduino.cc***
- ***www.amazon.com***
- ***www.sparkfun.com***
- ***www.adafruit.com***
- ***www.microcenter.com***

Make sure to get the following components, or a kit that contains them:

- male-female and male-male jumper wires
- solderless breadboard
- 100k Ohm potentiometer
- 10k Ohm carbon film resistor
- 220 Ohm resistor
- momentary button
- LED
- 1602 LCD with I2C interface
- DHT11 temperature and humidity sensor

What is Arduino?

Arduino is an open source platform for developing programmable electronics. The Arduino boards have a microcontroller, which is a microprocessor with integrated memory and peripherals. The Arduino platform consists of three components: circuit boards, Arduino IDE (integrated development environment), and code.

Arduino is also the name of the company in Italy that develops the Arduino boards and the IDE. The founders of Arduino wanted to make building things with microcontrollers easy enough to use to the point where non-technical people like artists could quickly get code running on a board.

From the Arduino.cc website:

> Arduino is an open-source electronics platform based on easy-to-use hardware and software. **Arduino boards** are able to read inputs - light on a sensor, a finger on a button, or a Twitter message - and turn it into an output - activating a motor, turning on an LED, publishing something online. You can tell your board what to do by sending a set of instructions to the microcontroller on the board.
>
> Arduino boards come with everything they need to run code, and allow other boards that add functionality. In addition to the microcontroller, each Arduino board typically has a USB to serial converter, power regulator, resistors and capacitors, and pin headers or solder pads that provide access to I/O pins.
>
> The microcontroller contains a microprocessor, random access memory (RAM), program memory(NOR flash memory, OTP ROM, or ferroelectric RAM), and programmable input and output peripherals. Common peripherals are digital I/O pins, serial ports, analog to digital converters (A/D), pulse width modulation outputs (PWM), comparators, and others.
>
> The microcontroller is pre-programmed with Arduino bootloader firmware. The bootloader allows the board to be programmed via serial port, so a dedicated hardware programmer is unnecessary.

Arduino has become very popular due to a few important factors.

1 The boards are inexpensive and do not require specialized hardware. The barrier to entry used to be higher for embedded development, because a chip programmer or an in-circuit debugger had to be purchased in addition to the development board. One minor drawback is that Arduino boards typically do not have dedicated hardware for debugging, like JTAG, so debugging is done by printing to the serial console.

2 Writing code for Arduino is easy. Arduino code hides all the details particular to each microcontroller's architecture, so the same code will run on different microcontrollers with little or no modifications. This greatly simplifies coding for users. Code is written in the Arduino IDE, and the IDE runs on Windows, Mac, Linux, and the web. This makes coding for Arduino available to everybody with a computer.

3 There is a large ecosystem of expansion boards, called "shields." Shields plug in to Arduino boards in order to add capabilities like ethernet, stepper motor drivers, temperature and humidity sensors, wifi, battery charging, OLED screens, and more.

4 The hardware designs are open source, so anyone can manufacture their own Arduino-compatible boards. The circuit schematics and board layout files are available for you to download and use.

5 There is a ton of published open source projects, with code and schematics included. You can repurpose existing designs for your own needs.

Arduino Uno and Other Arduino Boards

For this class we use the Arduino Uno, although any Arduino board is suitable. There are many variants of Arduino boards available. They come with different microcontrollers and configurations for specialized purposes. Some are specialized for IoT (internet of things), wearables (electronics embedded in clothing), motor control, or a multitude of other purposes. All the official Arduino board designs are fully open sourced.

The Uno has the following features:

- **Processor:** ATMega328P microcontroller made by Microchip
- **Flash memory:** 32KB
- **RAM:** 2KB of SRAM
- **Digital I/O:** 14 digital I/O pins, six of which provide PWM output
- **Analog inputs:** 10 bit analog inputs, 0 to 5V
- **Clock speed:** 16 MHz crystal oscillator (some clones are clocked at 12 MHz)
- **Operating Voltage:** 5V
- **EEPROM:** 12KB
- **Input Voltage:** 7 to 12V in the DC power jack

While the specs of the ATMega32P might not seem very impressive, there is a lot you can accomplish with only 2KB of RAM. Keep in mind that it is powered by 5V, and 5V is less common these days with microcontrollers. Make sure that any sensors or devices that you connect to the Uno are 5V-tolerant, otherwise you will need to use level shifting between devices. Other Arduino boards run at 3.3V or other voltages.

More modern Arduino boards often have an Arm core, a powerful 32-bit microprocessor, or even other powerful 32-bit or 64-bit processors. Arduino programs, or sketches, are mostly portable between architectures. The newer microprocessors have more memory, higher clocks, efficient low power modes, and sometimes have specialized instructions for signal processing or AI. These features open a whole new world of applications for you to develop.

To get an idea of all the features of the ATMega or whatever microcontroller you choose, take a look at the datasheet. Datasheets tend to make for dense reading, but they are indispensable. Microchip tends to have decent documentation for their chips. The datasheet for the ATMega328P is here: ***https://www.microchip.com/wwwproducts/en/ATmega328P***

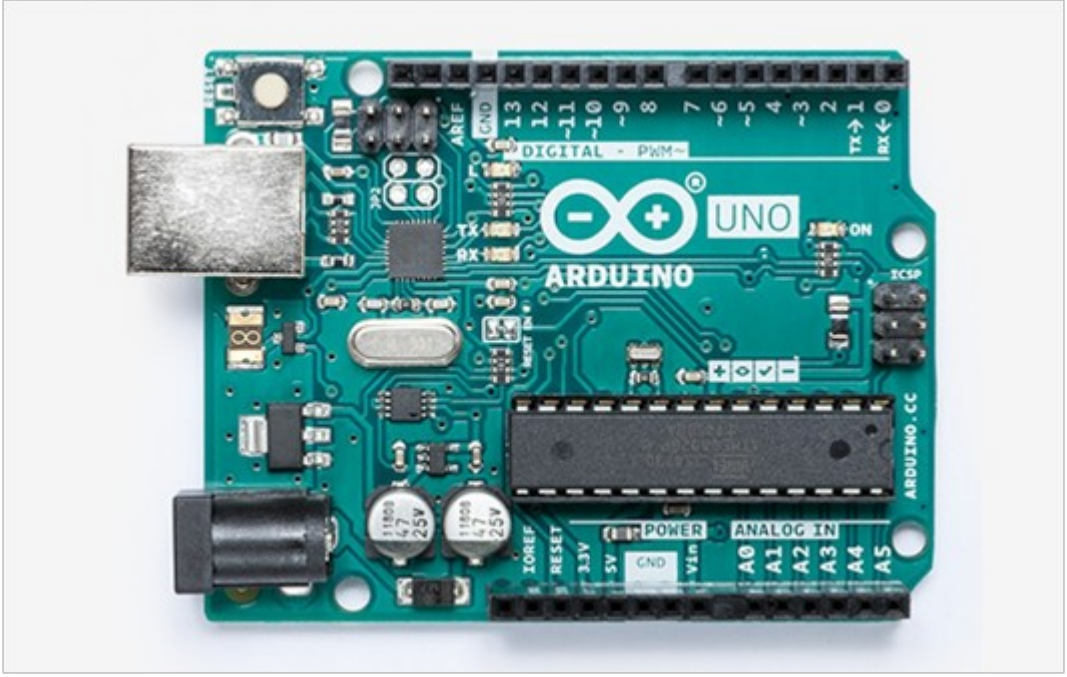

Figure 1 Uno Board

Many of the I/O pins are multiplexed with different functionalities like digital, analog, and serial. The pins are configured in code to act as inputs or outputs and digital or analog. The digital I/O pins are tristate logic. That means that the pins can output high or low logic voltages, and in input mode they are high impedance.

The analog inputs measure continuous voltage signals. The A/D converter for on the ATMega328P is 10 bits, so the A/D produces an integer between 0 and 1023 that linearly corresponds with an analog signal between 0V and 5V. Pins labeled with tilde (~) can be used as analog output. These pins actually output PWM (pulse width modulation). A low pass RC filter can convert PWM to a true analog signal if necessary.

The I2C serial interface uses two of the analog pins, A4 and A5. Two digital pins, D0 and D1, also serve as the RX and TX pins on the UART serial interface.

See the Uno schematic here:
https://www.arduino.cc/en/uploads/Main/arduino-uno-schematic.pdf

Note

There are many Arduino boards to suit different uses. They are all programmed in the same way, but have different sizes, costs, and peripherals. See here for a list of official Arduino boards: https://docs.arduino.cc/

Shields

Shields are add on boards that plug into Arduinos and provide more functionality like GPS, Bluetooth, ethernet, stepper motor drivers, and more. Shields are designed to plug into the header sockets on the Arduino board. Many shields pass through the signals, so it is possible to stack shields on the Arduino. Make sure the shields and particular Arduino board are compatible- some might have different pin assignments. Here is a sample of shields from Sparkfun Electronics:

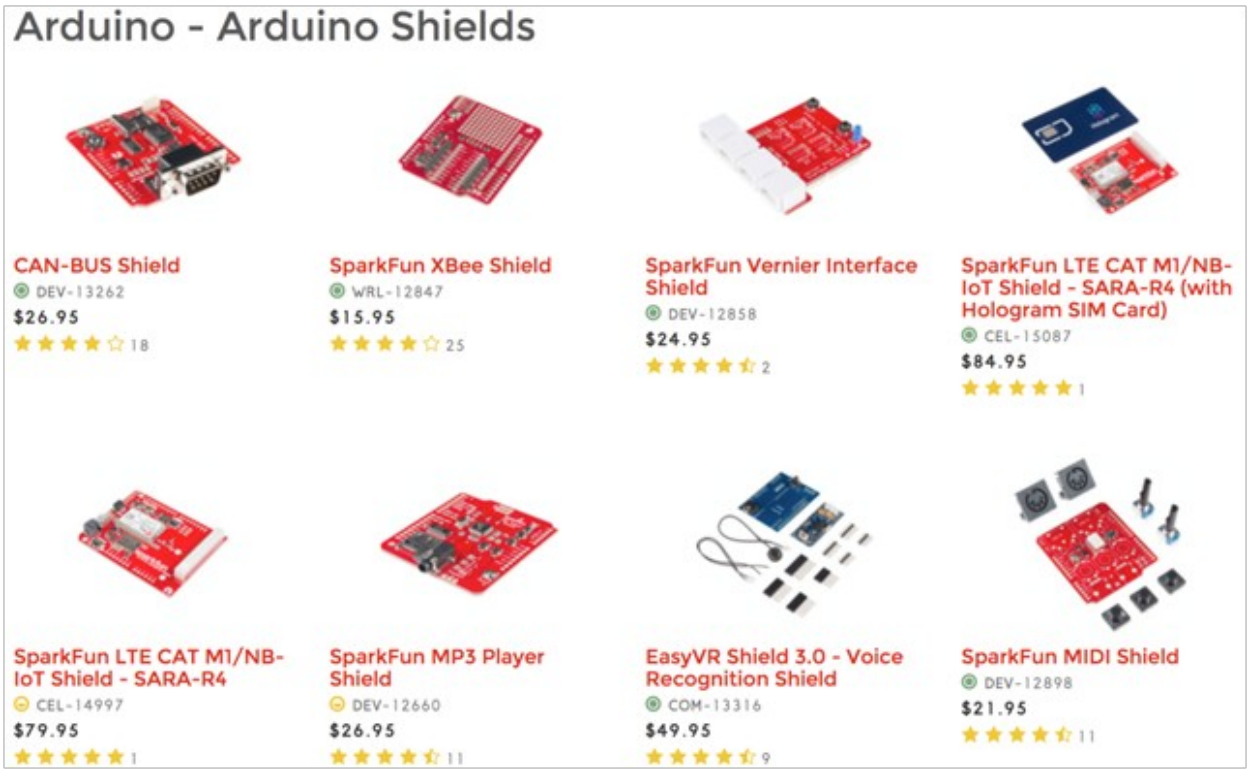

Figure 2 Shields Add-On Boards

Note

For an extensive list of shields along with compatibility details, see shieldlist.org .

Arduino Language

Arduino does not have its own language, although people commonly refer to the "Arduino language." The language used is in fact C++ and is compiled with a C++ compiler, though the Arduino core library provides many user friendly functions that simplify the coding experience. You are free to use object oriented programming (OOP) style coding as in C++, procedural style found in C, or a mix of the two styles.

If you need a quick refresher for what OOP or procedural mean, see:
https://en.wikipedia.org/wiki/Procedural_programming

https://en.wikipedia.org/wiki/Object-oriented_programming

Arduino provides high level, easy to use functions for reading and writing I/O, math, time delays, serial communication, bit manipulation, and character manipulation. We will use many of these functions in the following projects.

The core library provides an abstraction layer that hides the details of the underlying hardware. That means you do not have to port your code to use different Arduino boards or microprocessors- it has all been done for you. Arduino IDE automatically uses the correct core library when a particular board is selected.

See here for an overview of the functions included in the core library:
https://www.arduino.cc/reference/en/

An Arduino source file or application is also known as a “sketch.” The sketch has two main components: the setup() function and the loop(). All the configuration is done in setup(), as well as any code that only needs to run once. The rest of the code goes into loop().

Libraries

Libraries add functionality to Arduino. In addition to the main Arduino core library, Arduino IDE comes bundled other libraries. To add a library to a sketch, used the include command followed by the name of the library, like this:

```
#include <Wire.h>
```

In the above example, the sketch can use the functions defined in the Wire library, which is for I2C serial communication. The includes go at the top of the sketch.

Some other standard libraries include:

- EEPROM - reading and writing to "permanent" storage. Be aware that not all microcontrollers have real EEPROM, so this library will write to flash. Flash memory has fewer write cycles than EEPROM, so be aware what kind of memory you are writing to.
- Ethernet - for connecting to the internet using the Arduino Ethernet Shield, Arduino Ethernet Shield 2 and Arduino Leonardo ETH
- Firmata - for communicating with applications on the computer using a standard serial protocol
- GSM - for connecting to a GSM/GRPS network with the GSM shield
- LiquidCrystal - for controlling liquid crystal displays (LCDs)
- SD - for reading and writing SD cards
- Servo - for controlling servo motors
- SPI - for communicating with devices using the Serial Peripheral Interface (SPI) Bus
- SoftwareSerial - for serial communication on any digital pins
- Stepper - for controlling stepper motors
- TFT - for drawing text , images, and shapes on the Arduino TFT screen
- WiFi - for connecting to the internet using the Arduino WiFi shield

You can add more libraries to Arduino IDE by downloading existing libraries or writing your own. In Arduino IDE, select Sketch > Include Library. To use the Library Manager, select Tools > Manage Libraries. Alternatively, you can manually copy the library to the arduino/libraries directory.

For more about libraries see:
https://www.arduino.cc/en/Reference/Libraries

Setting Up the Arduino IDE

The Arduino IDE has a text editor where you can write your sketches, and it lets you upload sketches to the Arduino board. The Arduino IDE editor has several helpful features:

- syntax highlighting
- many code examples
- reference and troubleshooting documentation
- an output pane for viewing the build process and memory usage of sketches
- a serial monitor that enables two way communication with the board
- a serial plotter for viewing board output in graphical form

Download the Arduino IDE for your system here:
https://www.arduino.cc/en/Main/Software

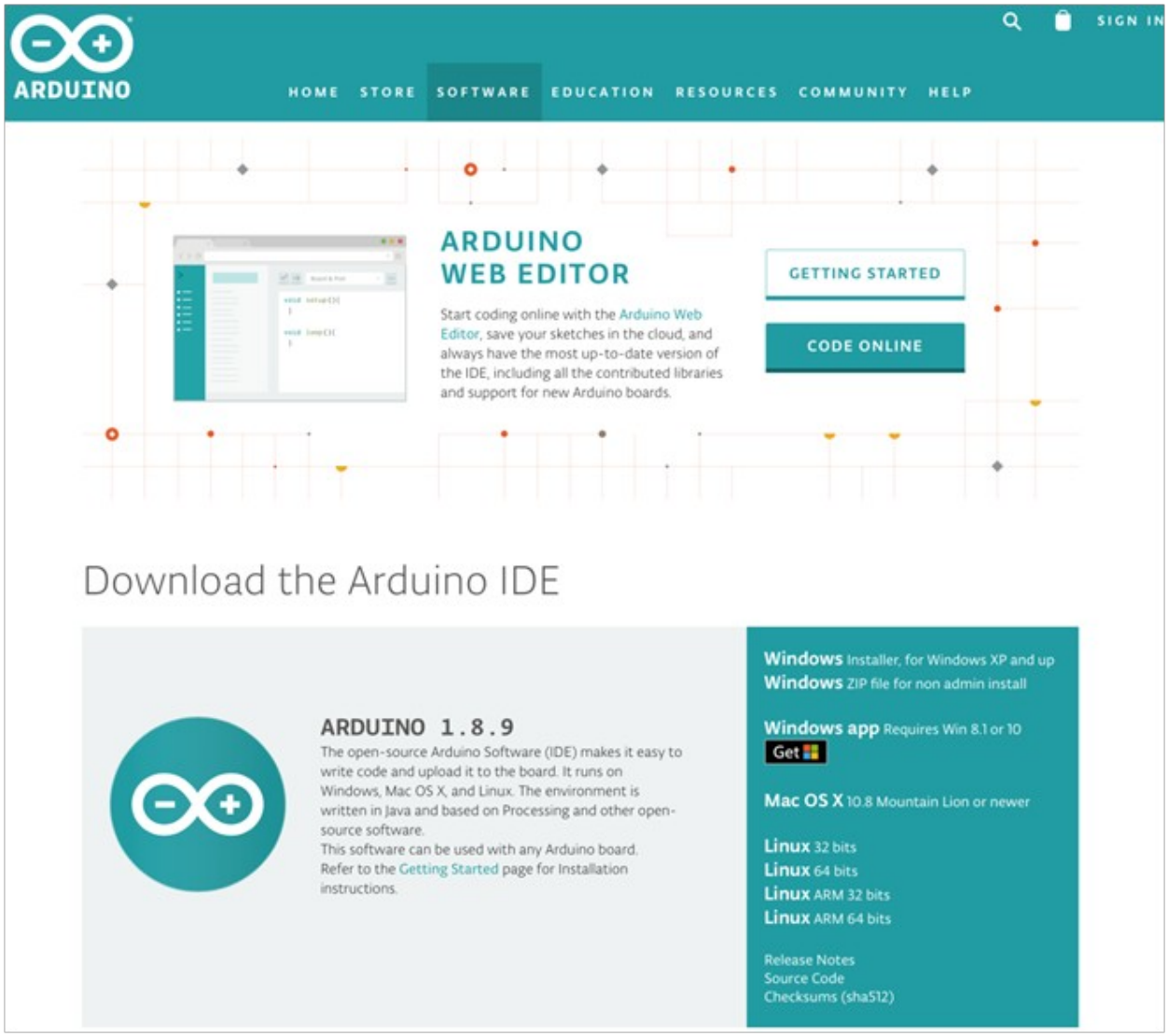

Figure 3 Arduino IDE

To avoid confusion, this course will use the installed version of the Arduino IDE, not the web editor.

After installing, connect the Arduino board to your computer via USB. In the IDE, go to Tools > Board, and select Arduino/Genuino Uno (or whatever board you are using). Next select the communications port the Arduino is using. Go to Tools > Ports, and select the right port. If you are on Linux, add your user to the dialout group. If you have a generic Arduino board, you might need to install a serial driver, but most likely it will just work.

To see some example projects that you can run for your board, go to File > Examples.

Two buttons in the tool bar at the top of each sketch are very useful. The check button compiles the sketch, and the right arrow button uploads the sketch to the board. Output from the build and upload processes show in the bottom pane. Any bugs or errors will appear there, too.

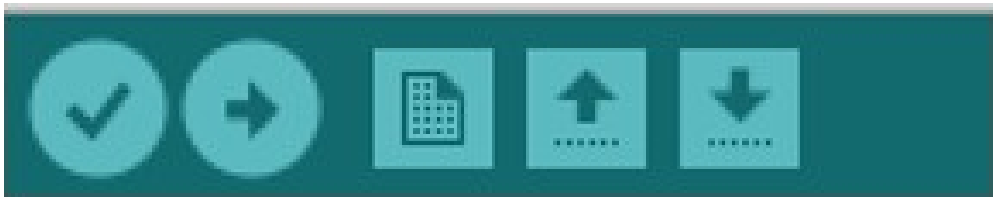

Figure 4 IDE Tool Bar

Debugging

Debugging with Arduino is done by printing to the serial monitor. To enable the serial monitor, click Tools > Serial Monitor.

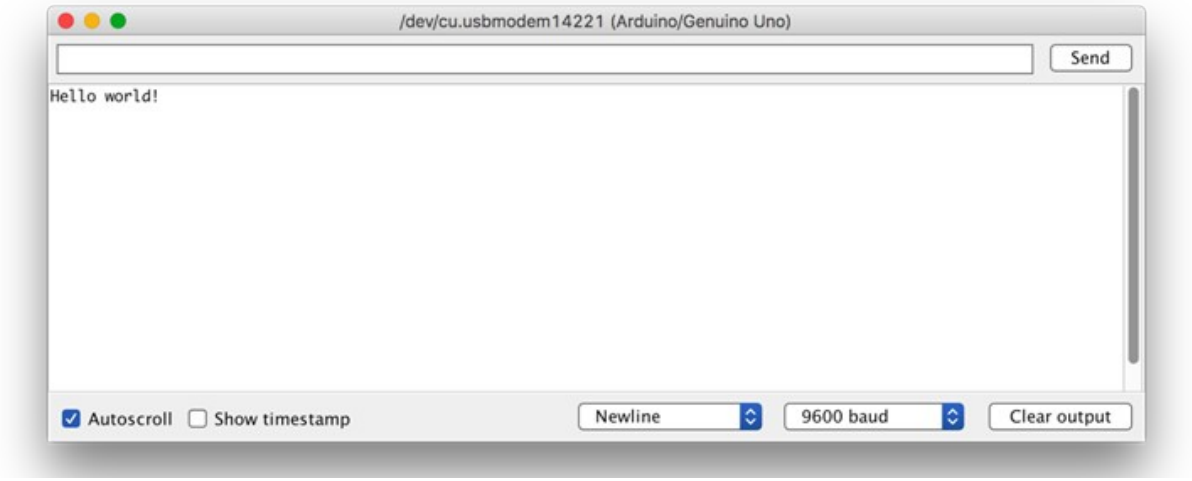

Figure 5 Debugger Screen

It is good practice to put some print statements throughout the code, so you can track where the program is currently executing. Be aware that printing to serial is slow, so too much printing can cause timing issues if timing needs to be tight.

Keep in mind that the serial monitor might tie up the same serial port you need to use to upload code to the chip. If you upload code and there is an error, make sure to close the Serial Monitor, and then uploading the code should work.

Sketch 1: Digital I/O and Delays

This sketch introduces two aspects of Arduino, digital I/O and timing. The digital pins can drive things like LEDs and transistors or read the state of switches. To use a digital I/O pin, the pin first needs to be initialized to be an input or output. Arduino boards have the pins numbered, so use the pin's number when initializing the pin. Arduino has some constants already defined for some boards. For example, the LED_BUILTIN is pin 13. The delay functions let you do create timed sequences. We will create a Morse code blinker with a digital output and delays.

Here are some Arduino functions related to serial communication, digital output, and delays:

- **Serial.begin(9600)** initializes the serial port and sets the Baud rate at 9600.
- **Serial.println(text)** prints text to the serial port. You can see the printed text in the Serial Monitor.
- **pinMode(pin, OUTPUT)** initializes the pin as an output. Other available values for mode are INPUT and INPUT_PULLUP. INPUT_PULLUP enables the ATMega's internal pullup, so an external pullup resistor is not necessary.
- **digitalWrite(pin, HIGH/LOW)** outputs HIGH or LOW to a pin.
- **delay(value)** pauses the sketch for the given value of milliseconds. Be aware that the delay() function does not let other code run at the same time.
- **delayMicroseconds(value)** does the same thing as delay() but delays the sketch for the given value of microseconds.

First let’s demonstrate how we can use the above commands to make the Uno’s built-in LED toggle every second. A pin, defined as a LED_BUILTIN is initialized as an output. Then every second the pins state changes, and loop() repeats the state changes.

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.println(“Hello World!”); // the usual first program
}

/* obligatory hardware version of
Hello World */

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

Also note that I demonstrated the two ways to leave comments in code, the single line comment // and the multiline comment bookended with /* */.

Let's examine a more interesting sketch that blinks Morse code for "hello word". Morse code is used for transmitting text messages with the characters encoded as dots and dashes (aka dits and dahs). The basic unit of time is the duration of the dot. The duration of a dash is equivalent to three dots, the delay between letters is three dots, and the delay between words is seven dots. There is a delay equivalent to a dot after each dot or dash within a letter.

This sketch uses the built-in LED, so no external circuit is necessary. Copy and Paste or type the following code into the text editor of Arduino IDE.

```
/*
Morse Code Blinker
*/
/* 1 */
#define DOT (200)       // delay for dot in milliseconds
#define DASH (3 * DOT)  // delay for dash
#define SPACE (DOT + 1) //delay after each dot or dash, see note 6
#define LSPACE (4 * SPACE) // delay for space between letters
#define WSPACE (7 * SPACE)// delay for space between words

/* 2 */
const int morse[] = {DOT, SPACE, DOT, SPACE, DOT, SPACE, DOT,
LSPACE, DOT, LSPACE, DOT, SPACE, DASH, SPACE, DOT, SPACE, DOT,
LSPACE, DOT, SPACE, DASH, SPACE, DOT, SPACE, DOT, LSPACE, DASH,
SPACE, DASH, SPACE, DASH, WSPACE, DOT, SPACE, DASH, SPACE, DASH,
LSPACE, DASH, SPACE, DASH, SPACE, DASH, LSPACE, DOT, SPACE, DASH,
SPACE, DOT, LSPACE, DOT, SPACE, DASH, SPACE, DOT, SPACE, DOT,
LSPACE, DASH, SPACE, DOT, SPACE, DOT, WSPACE}; // HELLO_WORLD_

const int length = sizeof(morse)/sizeof(morse[0]);

int i = 0;

/* 3 */
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);
  Serial.println("HELLO WORLD in Morse code!");
}

/* 4 */
void loop() {
  if (morse[i] == DOT || morse[i] == DASH) {
       digitalWrite(LED_BUILTIN, HIGH);
  } else {
       digitalWrite(LED_BUILTIN, LOW);
  }

  delay(morse[i]);
/* 5 */
  i++;
  if (i == length) {
    i = 0;
   Serial.println("flashing HELLO WORLD again");
  }
} /* end */
```

A sketch has two function blocks, setup() and loop(). Setup() runs once at the beginning. This is where code for configuring peripherals goes. Loop() runs over and over unless a stop condition is reached.

Let's examine what is happening in this sketch.

1 The Morse code for DOTs, DASHes, and spaces are defined with preprocessor macros. The #define macros are evaluated at compile time, not run time, and the compiler substitutes their values throughout the source code.

2 The Morse code for "hello world" is encoded as an array of integers in morse[]. Since you can stick any arbitrary message into morse[], we need to calculate its length. Determining the length of the array is done with a common C trick: take the number of bytes used by the array divided by the number of bytes used by a single array element. Think of it as sizeof(array of ints)/sizeof(int). Remember, the definition of what an int, or integer, depends on the architecture. In this case we are using an 8-bit microcontroller, so ints are 8 bits each. It should be noted that it would be better to specify what kind of int we are using, signed or unsigned, long or long long if necessary, but for the sake of demonstration we are playing a bit fast and easy by just using a plain int.

6 An integer variable i stores how many times loop() has run. Be aware that the variable will roll over if you loop through the code more times than the int can hold.

7 setup() initializes the LED_BUILTIN pin to be an output. Then it initializes the serial port and outputs a message. LED_BUILTIN is a constant defined by which board you've selected in the IDE.

8 loop() is responsible for blinking the LED. First it looks to see if the current element of morse[] is a space. If not, it turns on the LED for the duration of a dot or dash. If the current element is a space, the LED is turned off for the duration of a space. The delay(X) function stops the execution for X number of milliseconds. Keep in mind that delay(), at least on the Uno, takes the system clock and divides it down to a time of approximately 1ms, so it isn't very precise.

9 i is incremented. If i is equal to the length of the morse array, it is reset and a message prints.

10 DOT and SPACE are supposed to be the same length, but for the sake of simplicity 1 is added to SPACE to differentiate it from DOT.

A cleaner implementation of this code would probably use data structures like enums and structs to store the Morse encodings, but I tried to keep things simple.

Using a digital output and delays, we have blinked an LED to send a message. The functions in this sketch could be used to make other useful things like timer relays, lighting controls, infrared remote controls, or anything that uses a timed sequence of events.

For more information about Morse code:
https://en.wikipedia.org/wiki/Morse_code

Sketch 2: Button Debouncing

Buttons are electromechanical devices, and whenever a button is pressed there is some bouncing for a short time. Bouncing can cause a button press to register several times, so the button needs to be debounced. Other switches and relays typically have a rating for bounce time during which switch closures need to be ignored. For this sketch we will debounce in software, but oftentimes using some form of hardware debouncing is appropriate.

The Arduino functions related to digital input and timing are:

- **digitalRead(pin)** reads the logic level of the pin, HIGH or LOW.
- **millis()** gives the number of milliseconds since sketch began running. The value overflows after about 50 days for a board with a 16 MHz crystal..
- **micros()** gives the number the of microseconds since the sketch began running. The value overflows after about 70 minutes for a board with a 16 MHz crystal.

For this sketch you will need:

- momentary button
- 10k Ohm resistor
- breadboard
- male-male jumper wires (22 gauge solid core wire works, too), preferably in red, black, and blue colors

Wire the components as shown:

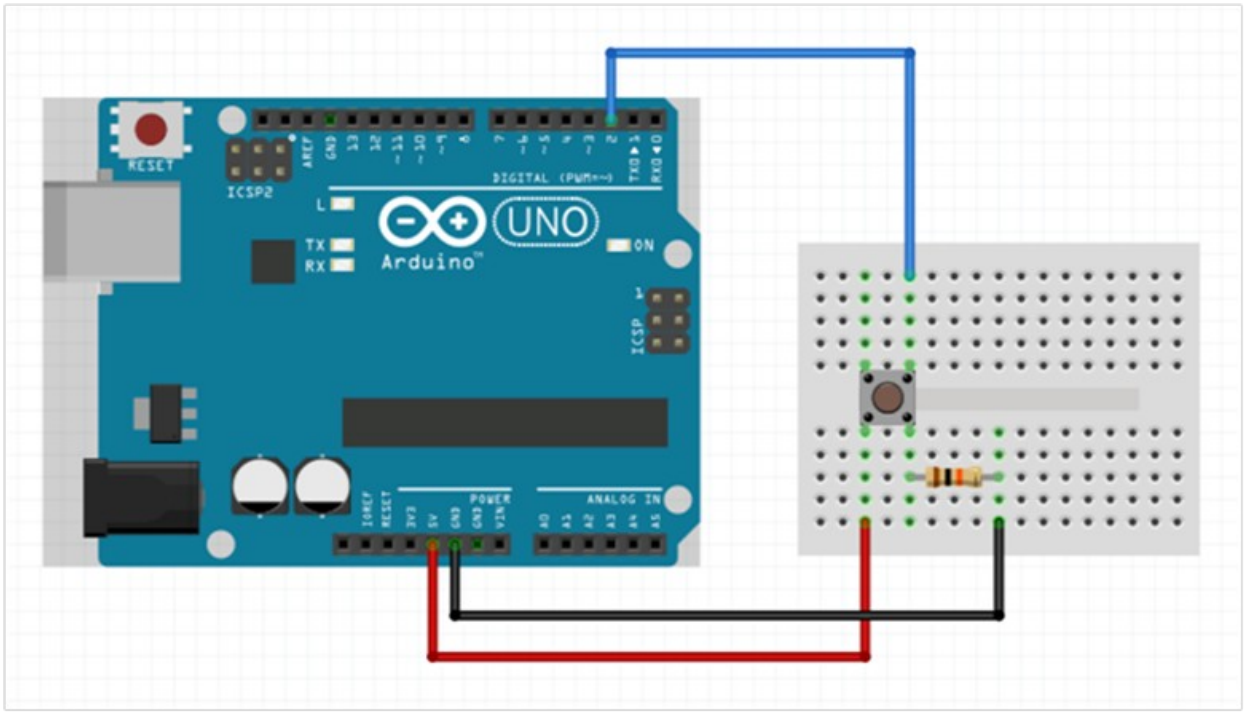

Figure 6 Button Debouncing Circuit

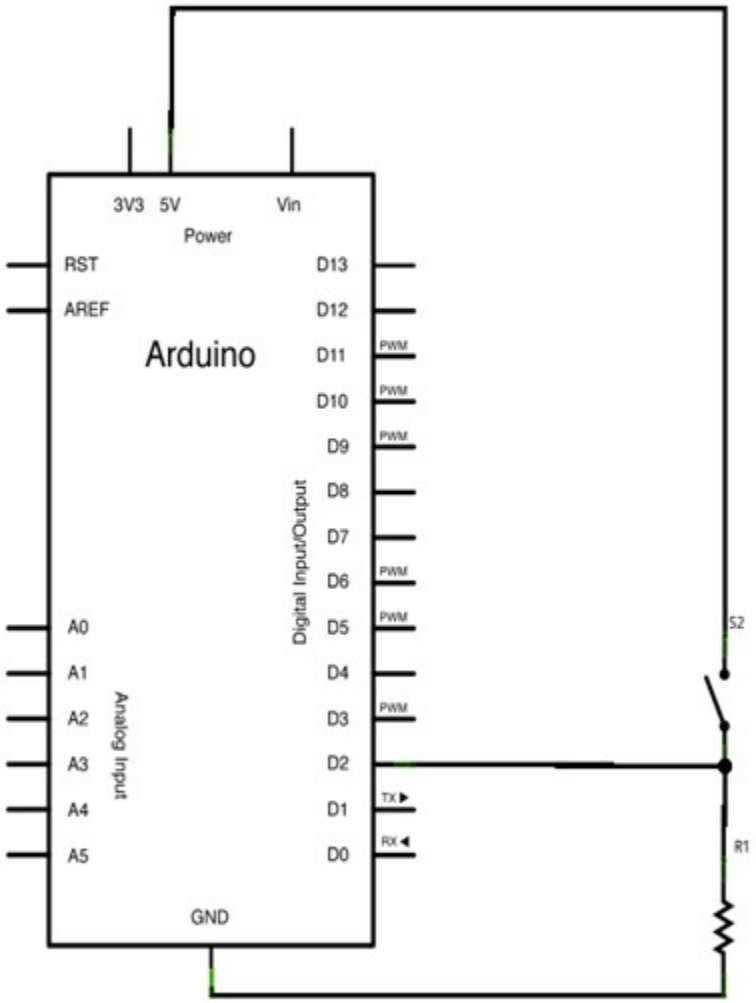

Figure 7 Button Debouncing Schematic
https://www.arduino.cc/en/uploads/Tutorial/button_sch.png

Open the Debounce example sketch. It is at File > Examples > 02. Digital > Debounce. The sketch is included here with annotations.

```
/*
  Debounce
Each time the input pin goes from LOW to HIGH (e.g. because of a
push-button   press), the output pin is toggled from LOW to HIGH or
HIGH to LOW. There's a minimum delay between toggles to debounce the
circuit (i.e. to ignore noise).
  The circuit:
 LED attached from pin 13 to ground
 pushbutton attached from pin 2 to +5V
 10k Ohm resistor attached from pin 2 to ground
Note
```

```
On most Arduino boards, there is already an LED on the board
connected to pin 13, so you don't need any extra components for this
example.
 created 21 Nov 2006   by David A. Mellis,  modified 30 Aug 2011by
Limor Fried
modified 28 Dec 2012 by Mike Walters
modified 30 Aug 2016  by Arturo Guadalupi

 This example code is in the public domain.
http://www.arduino.cc/en/Tutorial/Debounce
*/
/* 1 */
// constants won't change. They're used here to set pin numbers:
const int buttonPin = 2;    // the number of the pushbutton pin
const int ledPin = 13;      // the number of the LED pin

// Variables will change:
int ledState = HIGH;         // the current state of the output pin
int buttonState;             // the current reading from the input
pin
int lastButtonState = LOW;   // the previous reading from the input
pin

// the following variables are unsigned longs because the time,
measured in
// milliseconds, will quickly become a bigger number than can be
stored in an int.
unsigned long lastDebounceTime = 0;  // the last time the output pin
was toggled
unsigned long debounceDelay = 50;    // the debounce time; increase
if the output flickers

/* 2 */
void setup() {
  pinMode(buttonPin, INPUT);
  pinMode(ledPin, OUTPUT);

  // set initial LED state
  digitalWrite(ledPin, ledState);
}

/* 3 */
void loop() {
  // read the state of the switch into a local variable:
  int reading = digitalRead(buttonPin);

  // check to see if you just pressed the button
  // (i.e. the input went from LOW to HIGH), and you've waited long
enough
  // since the last press to ignore any noise:

  // If the switch changed, due to noise or pressing:
  if (reading != lastButtonState) {
    // reset the debouncing timer
```

```
      lastDebounceTime = millis();
    }
  /* 4 */
    if ((millis() - lastDebounceTime) > debounceDelay) {
      // whatever the reading is at, it's been there for longer than
  the debounce
      // delay, so take it as the actual current state:

      // if the button state has changed:
      if (reading != buttonState) {
        buttonState = reading;

        // only toggle the LED if the new button state is HIGH
        if (buttonState == HIGH) {
          ledState = !ledState;
        }
      }
    }
  /* 5 */
    // set the LED:
    digitalWrite(ledPin, ledState);

    // save the reading. Next time through the loop, it'll be the
  lastButtonState:
    lastButtonState = reading;
  } /* end */
```

Let's examine what is happening in this sketch.

1 Constants and variables are initialized here. The important things to notice here is that there are variables to hold the button state, store the time of the last state change, and set the debounce delay. The debounce delay should be just long enough to avoid erroneous button presses. The initial button state is LOW, not pressed. The LED is on.

2 Setup() initializes the input and output pins, as well as driving the output high.

3 Each loop reads the state of the input pin. This is known as polling. If the new reading of the pin's state does not match the previous state, the debouncing timer is reset.

4 While waiting for the debounce delay, the button state is ignored. Once the system has waited for the bounce delay, button state is then compared with the current reading. If a button state change is detected, the button state is set to the reading. If the button state is HIGH, meaning the button has been depressed, then the LED is toggled.

5 Sets the output pin and stores the previous state of the button.

The method shown in this sketch is better than simply using a delay() after a state change is detected, because a delay() blocks other work from being done. Try reducing the debounceDelay variable to a value less than 50. At some point you will observe the LED flicker. If you were to look at the voltage across the switch with an oscilloscope, you could observe how long the bouncing lasts.

Now that you know how to debounce buttons, you can incorporate buttons and other electromechanical switches into your designs.

For a comprehensive guide to debouncing:
http://www.ganssle.com/debouncing.htm

Sketch 3: Analog I/O

In this sketch we will read and write analog values. Microcontrollers in Arduino boards come with an analog to digital converter (A/D) to read analog voltages between 0V and the power supply voltage, usually either 3.3V or 5V. The Uno comes with a 10 bit A/D (analog to digital converter) and six available analog inputs. The Uno's A/D requires about 100 microseconds to take a sample, resulting in max sampling rate of 10 kHz. The microcontrollers in Arduino boards typical have analog comparators as well. The comparators return a true or false depending if the input voltage is above or below a reference voltage. We won't discuss comparators further here, but they are available in case you have a need for them.

Analog output is achieved with PWM (pulse width modulation). While PWM is not true analog, it is a close enough approximation for many applications. The Uno has an 8 bit PWM, so a writing a value of 0 to the PWM produces a square wave with a 0% duty cycle (off), and a value of 255 produces a square wave with 100% duty cycle (on). The frequency of each PWM pin is determined by an internal counter and a prescaler that divides the base frequency. Pins 5 and 6 use timer0, which runs at base frequency of 62500 Hz. The base frequency is scaled down to about 1000 Hz for pins 5 and 6. Pins 9 and 10 use timer1, and pins 2 and 11 use timer2. timer1 and timer2 have a base frequency of 31250 Hz, which is divided down to about 500 Hz. These slow PWM frequencies are suitable for things like blinking an LED, but not for other things like switching mode power supplies. See the note at the end of this lesson about changing the PWM frequencies. Higher end microcontrollers than the ATMega have more PWM options.

Let's examine the functions related to analog. For most applications, using only the first two functions with the default settings is sufficient.

- **analogRead(pin)** - reads the analog voltage at the pin, which in the case of the Uno is a 10 bit integer between 0 and 1023.
- **analogWrite(pin, value)** - outputs a PWM wave on pin. The output value is between 0 to 255, corresponding to 0% to 100% duty cycle.
- **analogReference()** - sets the top of the input range. Defaults to 5V for the Uno. See

  https://www.arduino.cc/reference/en/language/functions/analog-io/analogreference/

- **analogReadResolution(value)** - for Arduinos with more than 10 bits A/D resolution. If the Arduino has a 16 bit A/D, set value to 16, and analogRead() will return a value between 0 and 4095. The input value can be up to 32.
- **analogWriteResolution(value)** - for Arduinos with more than 8 bits PWM resolution.

For this sketch, we need the following parts:

- 100k Ohm potentiometer
- LED
- 220 Ohm resistor

Wire the circuit as shown here:

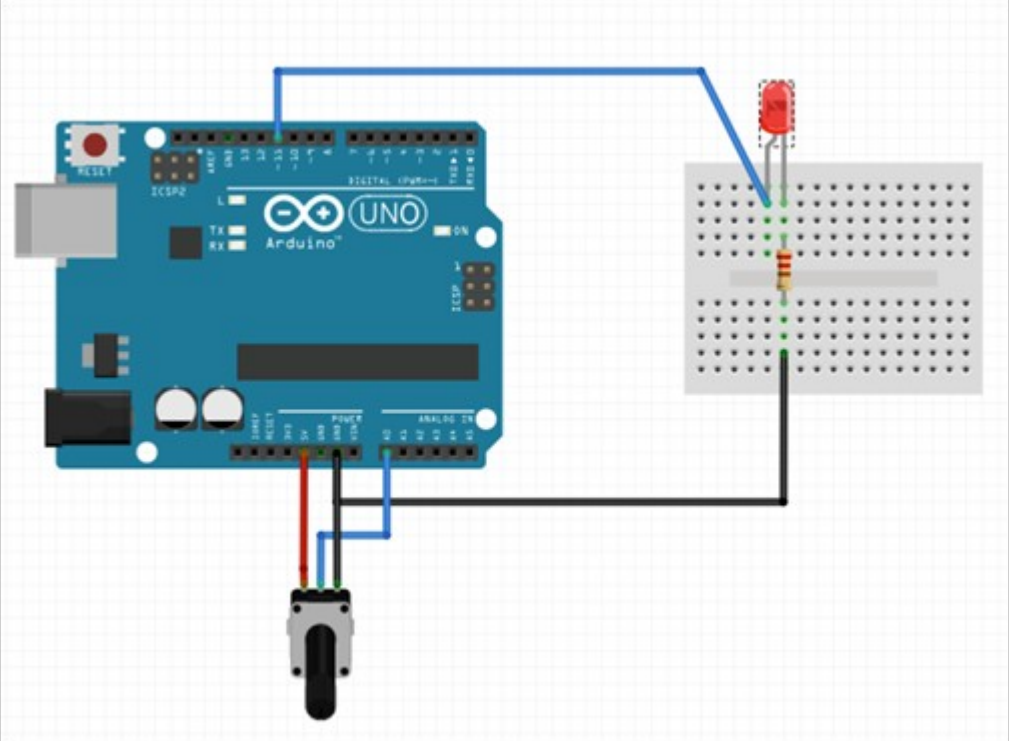

Figure 8 Analog Circuit

This is the circuit schematic:

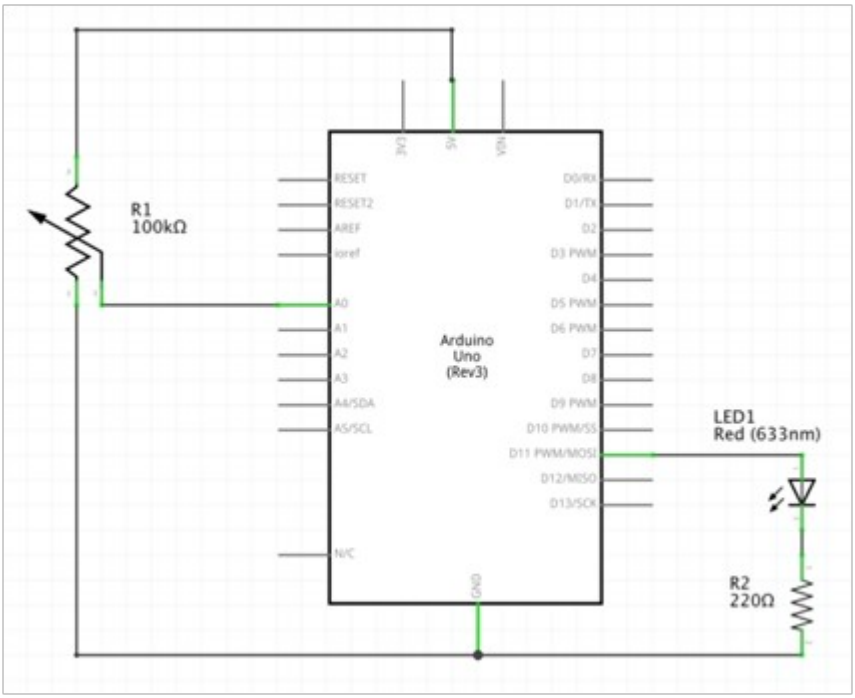

Figure 9 Analog Schematic

Type or copy and paste the following code into Arduino IDE.

```
/*
 * Dimmer using Analog Input and PWM
 */

/* 1 */
const int led = 11;
const int potentiometer = A0;
int reading = 0;
int pwm = 0;

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

/* 2 */
void loop() {
  reading = analogRead(potentiometer);
  Serial.println(reading);
/* 3 */
  pwm = map(reading, 0, 1023, 0, 255);
  analogWrite(led, pwm);
  delay(20);
}/* end */
```

1 Pins for LED and potentiometer are set. Pin mode for LED pin is set.

11 Each loop, the A/D is read and then printed to the serial console.

12 The value is then scaled with the map() function to between 0 and 255. The value is then fed to PWM output. The LED brightness is determined by the PWM duty cycle. Finally, the loop delays for 20ms.

In case more than six PWM outputs are need, bit-banging PWM is an option. Any digital pin can bit-bang PWM. Let's look at an example.

```
/*
 * Bit-banging Pulse Width Modulation Example
 */
const int pin = 7;
const int period = 1000; // period is 1000us, so frequency is 1kHz
const int onTime = 200; // arbitrary value for demonstration

void setup() {
  pinMode(pin, OUTPUT);
}

void loop() {
  digitalWrite(pin, HIGH);
  delayMicroseconds(onTime);
  digitalWrite(pin, LOW);
  delayMicroseconds(period - onTime);
} /* end */
```

In this example, pin 7 is toggled at 1KHz with about 20% duty cycle (200 microseconds on, 800 off for a total of 1000). This should only be used as a conceptual example as the **delayMicrosecond()** calls will block other work from being done.

In review, reading and writing analog values is easy with Arduino, because all the low level configuration has already been done. By default, the Uno offers slow PWM frequencies. There is more about PWM and how to change the frequencies at the following links.

https://playground.arduino.cc/Code/PwmFrequency/

https://www.arduino.cc/en/Tutorial/SecretsOfArduinoPWM

Sketch 4: Functions, Random Numbers, and Strings

In this lesson we review how to use functions and generate random numbers. In addition to the standard setup() and loop() functions, you can write other functions in sketches. Code is much easier to read and is better organized if all the code is not put in setup() and loop(). The sketch in this lesson uses function calls and random numbers to implement a simple guessing game.

The functions need to follow the C/C++ format:

```
returnType  functionName(type arg1, type arg2, ...) {
  //code body;
  return value; //if the function's return type is void, nothing
is returned
}
```

Functions can appear before or after setup() and loop(). Functions can be called before they are defined, which is not the case in C and C++. The return type can be anything that is also available in C++: int, float, bool, string, and others. If the function does not return a value, void is used. The function name should make its purpose clear

See here for more about data types available with Arduino:
https://www.arduino.cc/reference/en/#variables

Here is an example of using a function:

```
/*
Function Example
*/
void setup() {
  Serial.begin(9600);
}

void loop() {
  int duration = exampleFunction1();
  delay(duration);
  exampleFunction2();
}

int exampleFunction1() {
  return 1000;
}

void exampleFunction2() {
  Serial.println("This is a test");
} /* end */
```

Oftentimes we need to use random numbers. Arduino provides random() and randomSeed(). randomSeed() initializes the pseudo random number generator. In order to get a different sequence of generated numbers each time the sketch runs, seed the generator with a random input like analogRead() on a floating pin, like this:

```
randomSeed(analogRead(0));
```

Then to get a random number call random(). To get an integer value between 0 and MAX - 1, call

```
int randomNumber = random(MAX);
```

Note

C and C++ are zero indexed, so counting starts with zero. The range goes from zero inclusive to MAX exclusive, which explains why we subtract one from MAX. Off-by-one bugs often occur when zero indexing is not considered.

To get an integer between a range of MIN and MAX - 1, call

```
int randomNumber = random(MIN, MAX);
```

Let's examine how to use strings briefly. First we examine how to analyze individual characters, then how to use the C++ String type to create and manipulate strings. In this lesson's sketch, we need the Aduino board to process user input. To do this, we can use the following Arduino functions. Remember, char is an 8-bit variable type used to hold an ASCII character.

- **isAlpha(char)** – returns true if char is a letter.
- **isDigit(char)** – returns true if char is a digit 0 – 9.
- **isSpace(char)** – returns true if char is a space.
- **isPunct(char)** – returns true if char is a punctuation mark.

It's important to distinguish C++ string objects from char and C-style arrays of chars. With Arduino it is generally easier to use C++ string objects. For simplicity, we treat an indivual characters as a char type and convert the char to a C++ String object whenever we need to manipulate it. Now let's create a String object using some of its common constructors.

```
String string1 =  "This is a string"; // creates a String constant
String string2 = String('S'); // creates a String from a char
String string3 = String(123); // creates string from integer
```

Using C++ String objects makes string manipulation easier. To concatenate two strings, use the + operator like this:

```
String string4 = string1 + ", fantastic"; // string4 is "This is a
string, fantastic"
```

We can use the comparison operators ==, !=, >, <, >=, and <= to compare two strings, which is useful for sorting and alphabetizing strings.

```
if(string1 == string2) Serial.println(“The strings are equal!”);
```

The toInt() function lets you convert the String representation of an integer to an integer.

```
int number = string3.toInt();
```

Now let's demonstrate using functions with random() and strings. We will make a guessing game where the user chooses a number between 1 and 5. Open the Serial Monitor to view sketch output and to send keystrokes.

```
/*
Guessing Game
*/
/* 1 */
void setup() {
  Serial.begin(9600);
  randomSeed(analogRead(0));
}

void loop() {
  Serial.println("Choose a number between 1 and 5");
/* 2 */
  int number = getRandom();
  while( !Serial.available() ) ;
  char rawGuess = Serial.read();

  int guess = analyzeInput(rawGuess); //see note 6

/* 3 */
  if ( (guess >= 1) && (guess <= 5) ) {
    Serial.println("You chose: " + String(guess));
  } else {
    Serial.println("Character not between 1 and 5");
    return;
  }

/* 4 */
  Serial.println("Arduino chose: " + String(number));
  if (number == guess) {
    Serial.println("You got it!");
  } else {
    Serial.println("Bad luck! Try again.");
  }
}

/* 5 */
int getRandom() {
  return random(1, 6);
} /* end */
```

```
/* 6 */
int analyzeInput(char input) {
  if(isAlpha(input)){
    Serial.println("Input is a letter");
    return -1;
  }
  if(isSpace(input)){
    Serial.println("Input is a space");
    return -1;
  }
  if(isDigit(input)){
    return String(input).toInt();
  }
  return -1;
}
```

Let's review what is happening in this sketch.

1 The serial port is initialized, and the random number generator is seeded with a reading from the A/D converter.

2 The **getRandom()** function is called and returns an integer. Then the sketch waits to receive a character from the serial port that the user typed. The value is converted from a char to an int by subtracting the char '0'. If you look at a table of ASCII values, ‘0’ has a value of 48 or 0x30. So, subtracting ‘0’ from itself is 0, ‘1’-‘0’ is 1, etc.

3 Check if the input is between 1 and 5. If not, skip to the next loop by calling return. The guess value is converted to a string with **String()** and then concatenated with the preceding text.

4 The guess valued is compared with the machine generated random number. If they match, a congratulatory message is printed to the Serial Monitor.

5 **getRandom()** calls **random()** and returns an integer. The user’s input is read from Serial Monitor.

6 The user input is tested to see if it is valid. If it is valid, it is converted to an integer and returned.

This sketch covered several key concepts: using functions other than setup() and loop(), how to use random(), how to skip to the next iteration of loop(), and reading serial input.

Sketch 5: Interrupts

In the button debouncing sketch above, we used polling to detect when the button state changes. Reading an input with a loop wastes processor time. Interrupts provide a more responsive way to detect a button press or sensor output. Interrupts tell the microcontroller to stop what it is doing and to execute code in the interrupt service routine (ISR).

The Uno has two available interrupts on pins 2 and 3. Other Arduino boards have more interrupts available.

To use an interrupt first attach the interrupt to a pin:

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, TRIGGER);
```

attachInterrupt() does not take a pin number directly; it needs the number of an interrupt. **digitalPinToInterrupt()** gets the number of an interrupt associated with a pin. The ISR is the name of the function acting as an interrupt handler. Do not put a lot of code in the ISR. It needs to be fast and not block the rest of the program.

The ISR can trigger on four possible events:

- **LOW** triggers when the pin is low.
- **RISING** triggers when the pin transitions from low to high.
- **FALLING** triggers when the pin transitions from high to low.
- **CHANGE** triggers any time the pin's value changes.

Use the same circuit as in the Debounce sketch above. Now let's use an interrupt. Type or copy and paste the following code into Arduino IDE.

```
/*
Interrupt
*/
/* 1 */
const int interruptPin = 2;
int ledState = LOW;          // the current state of the output
pin

/* 2 */
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(interruptPin, INPUT);
  attachInterrupt(digitalPinToInterrupt(interruptPin), handler,
HIGH);
}

/* 3 */
loop() {
  digitalWrite(LED_BUILTIN, ledState);
}

/* 4 */
void handler() {
  ledState = !ledState;
} /* end */
```

1 First the input pin and the initial LED state are set.

2 Pins are set as output and input. An interrupt is attached to pin 2.

3 The ledState, HIGH or LOW, is written to the pin.

4 Every time the button is pushed, the handler() function runs. The LED output is toggled.

In case you ever need to run critical code that cannot be interrupted, use **nointerrupts()**. This disables interrupts. To re-enable interrupts, use **interrupts()**.

For both of the button related sketches, we used an external resistor to pull down the input pin to ground. The ATMega microcontroller has built-in pullup resistors of 20K Ohm or more. In the case where using the microcontroller's internal pullups is desired, use **pinMode(pin, INPUT_PULLUP)** to enable the internal pullups. Be aware that external circuits on pins with internal pullups can divide the voltage down under 5V, making the input read LOW.

In review, interrupts provide a way to respond to events with the lowest delay possible without wasting processor time. Interrupt services routines should be as short as possible so the sketch can return to executing other code.

Sketch 6: I2C and External Libraries

In this lesson we will cover the I2C serial bus and how to use external libraries. With some help from libraries, we will make a thermometer with a digital display.

First let's start with the I2C bus. It is a lower speed serial bus that is used to connect to peripheral chips over short distances. Its speed ranges from 100 kbit/s to 5 Mbits/s. It can be implemented in hardware or software. I2C requires two lines, SCL and SDA. SCL carries the clock signal, and SDA handles the data. It uses a master-slave layout, and a master can communicate with several slave devices on the same bus.

In depth explanation for how I2C works (you can skip this if you want): Each slave device has a 7-bit or 10-bit address. The master initiates transmitting by sending a START signal followed by the slave address. After the address the master sends a bit to indicate a write (0) or read (1). Then the master sends or receives data based on the write/read bit. The slave carries out the complimentary action and then sends an ACK. The master can continue a transaction of writes or reads by continuing to send bytes or receive them and sending an ACK back to the slave. The master halts the message with a STOP signal.

The Uno has a hardware I2C port. Pin A4 also serves as SDA, and Pin A5 handles SCL. To demonstrate I2C in action, let's put together the hardware for this lesson.

For this lesson, we need the following parts:

- 3 or 4 pin DHT11 temperature and humidity sensor
- 10k resistor if DHT11 has 4 pins
- 1602 LCD with I2C interface (Hitachi HD44780 LCD with PCF8574A I2C module)
- 4 male-female jumper wires, breadboard, 5 male-male jumper wires OR 7 male-female jumper wires (wire components directly to Uno and power the DHT11 with 3.3V instead of 5V from Uno)

If using the 4 pin DHT11, wire the components as in this diagram:

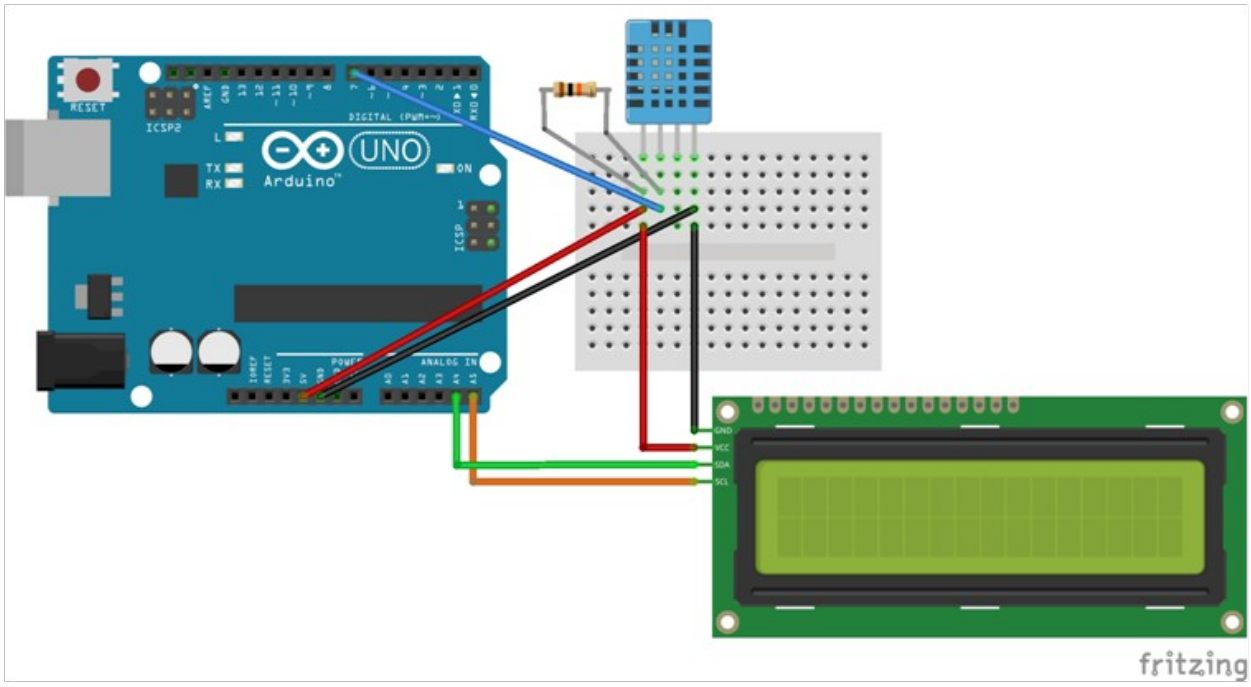

Figure 10 4-Pin, I2C Serial Bus Components

For the 3 pin DHT11:

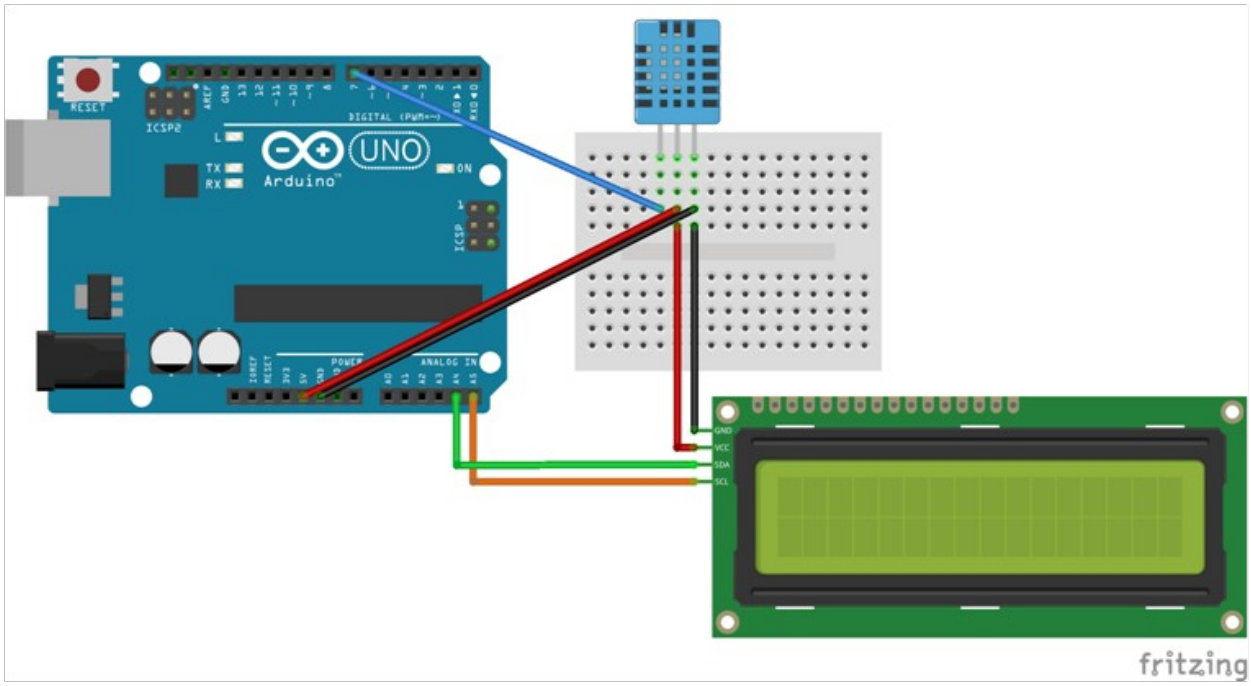

Figure 11 3-Pin, I2C Serial Bus Components

The schematic with 4 pin DHT11:

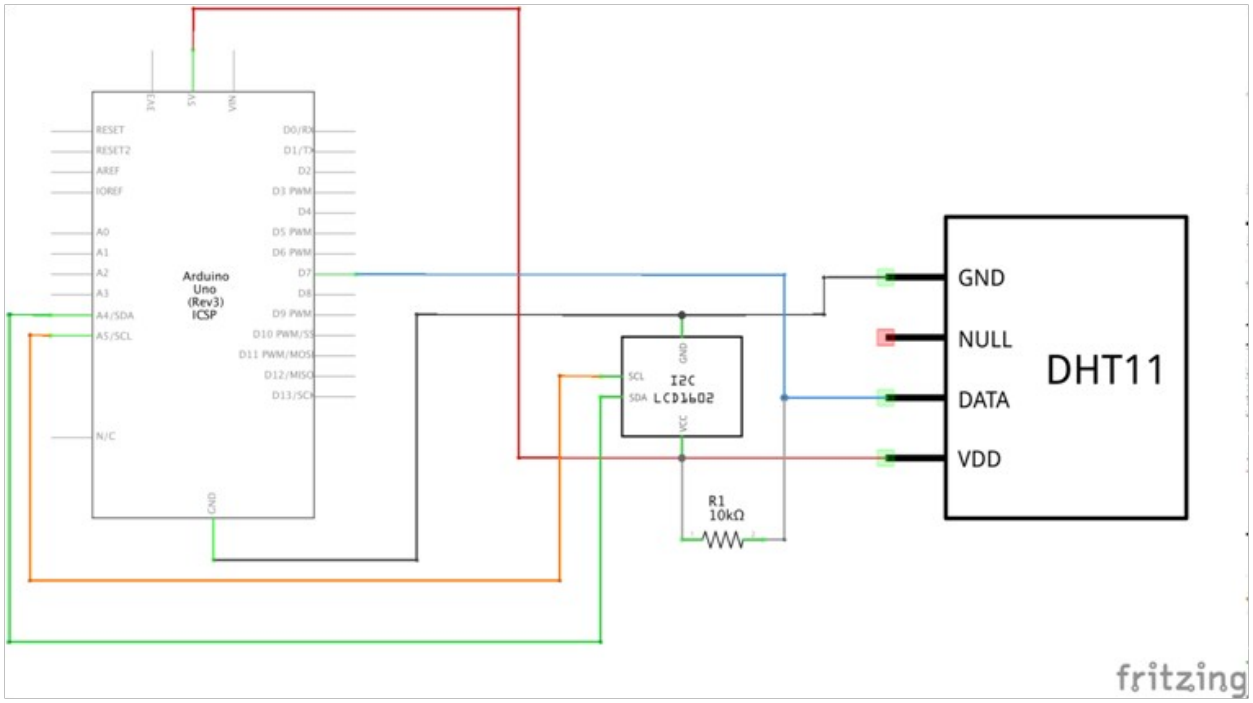

Figure 12 4-Pin, I2C Serial Bus Schematic

Schematic with 3 pin DHT11:

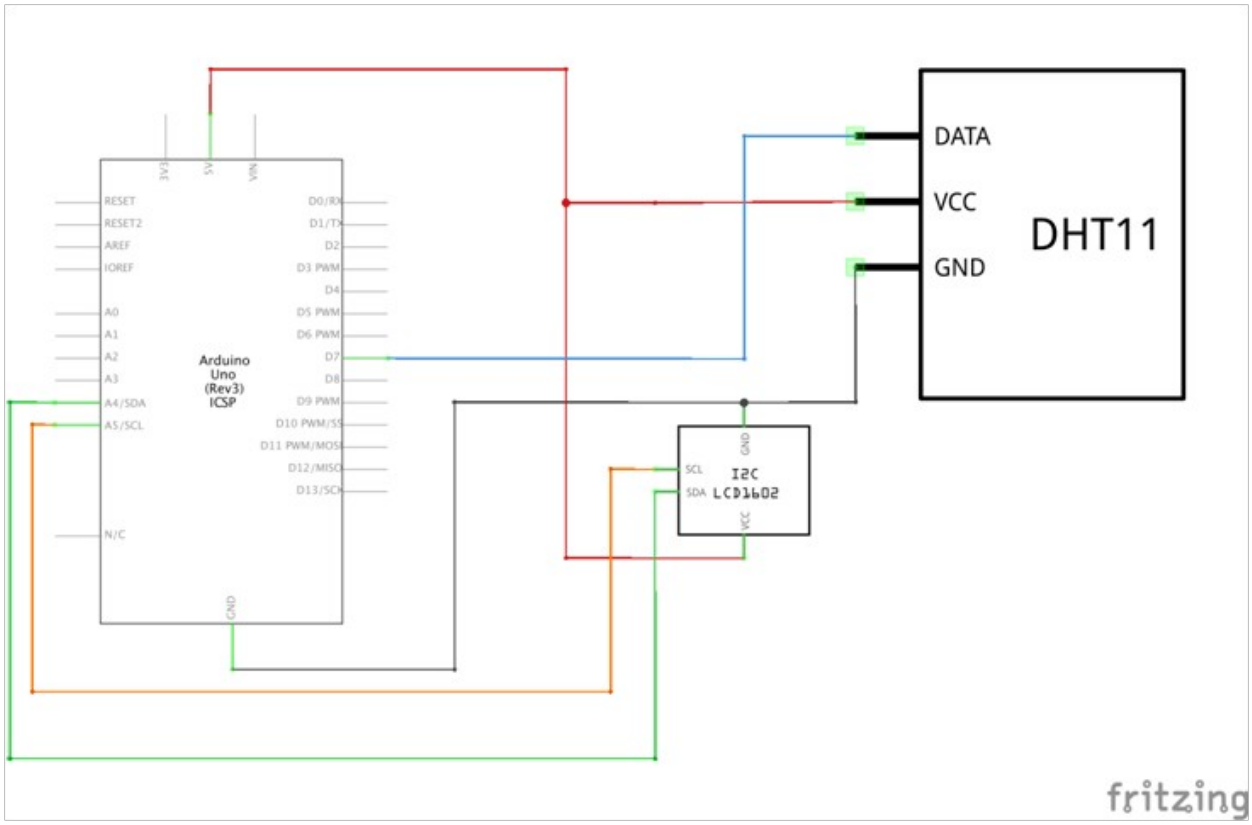

Figure 13 3-Pin, I2C Serial Bus Schematic

Let's examine a few of the functions available with Wire.

- **Wire.begin(address)** - Initiate the Wire library. If the address is given, join the I2C bus as a slave at that address. If no address, join the I2C bus as master.
- **Wire.beginTransmission(address)** - Start transmitting to the I2C slave device at address.
- **Wire.write(value OR string OR data, length)** - If device is a slave, write data to bus in response to master. If device is a master, queue data for transmission.
- **Wire.endTransmission()** - Ends transmission to a slave device and transmits data queued by **Wire.write()** - This function returns one byte indicating the status of the transmission.
- **Wire.requestFrom(device, length)** – used by the master to request the number of bytes “length” from the address of the device.
- **Wire.read()** – receives a byte and then the byte is returned by the function. The byte can be interpreted as a char or int.
- **Wire.available()** – returns the number of bytes waiting to be read.

Let's examine a sketch using Wire, an I2C scanner. Run the following code with the Serial Monitor open.

```
/*
 I2C Scanner
*/
/* 1 */
#include <Wire.h>

/* 2 */
void setup() {
  int deviceCount = 0;
  Serial.begin(9600);
  Wire.begin();
  while (!Serial);
  Serial.println("Beginning scan");
/* 3 */
  for(int address = 1; address < 127; address++ ) {
    Wire.beginTransmission(address);
    int result = Wire.endTransmission();
/* 4 */
    if (result == 0) {
      Serial.print("I2C device at 0x");
      Serial.print(address, HEX);
      deviceCount++;
    }
  }
/* 5 */
  if (deviceCount == 0) {
    Serial.println("No I2C devices found\n");
  }
}
/* 6 */
void loop() {}  //empty loop
/* end */
```

1 The Wire library is included. The Wire library is already installed with Arduino IDE.

2 All the code is going into setup(), so it only runs once. The variable deviceCount stores the number of devices found. Serial communications is initialized, and the sketch waits for it to be ready.

3 A for-loop cycles through all the possible addresses between 1 and 127. Wire.beginTransmission() and Wire.endTransmission() are called to get a response from each address.

4 If transmission to an address is a success, the result is printed to the Serial Monitor, and deviceCount is incremented.

5 If no devices are found, a messages is printed to the Serial Monitor.

6 No code is inside loop(). loop() will simply run forever doing nothing after setup() has run the code once.

The sketch will not find any devices yet if the Uno is not connected to anything. With the circuit we built above, the sketch should find one device, the LCD.

Arduino IDE comes with many libraries built-in, but sometimes it is necessary to install more libraries. Oftentimes libraries are used for device drivers, and in our case we need drivers for the DHT11 temperature/humidity sensor and for the 1602 LCD. The 1602 LCD driver is built on the Wire library. Wire comes with Arduino IDE and provides an easy to use interface for I2C. While we will not use Wire in the later sketch, the LCD driver we will use is based on Wire.

Now we need to import two libraries. In Arduino IDE, go to Sketch->Include Library->Manage Libraries. In the Library Manager, type "SimpleDHT" and install it. The list of libraries should filter down to a few choices. Next, install "LiquidCrystal I2C" by Frank de Brabander.

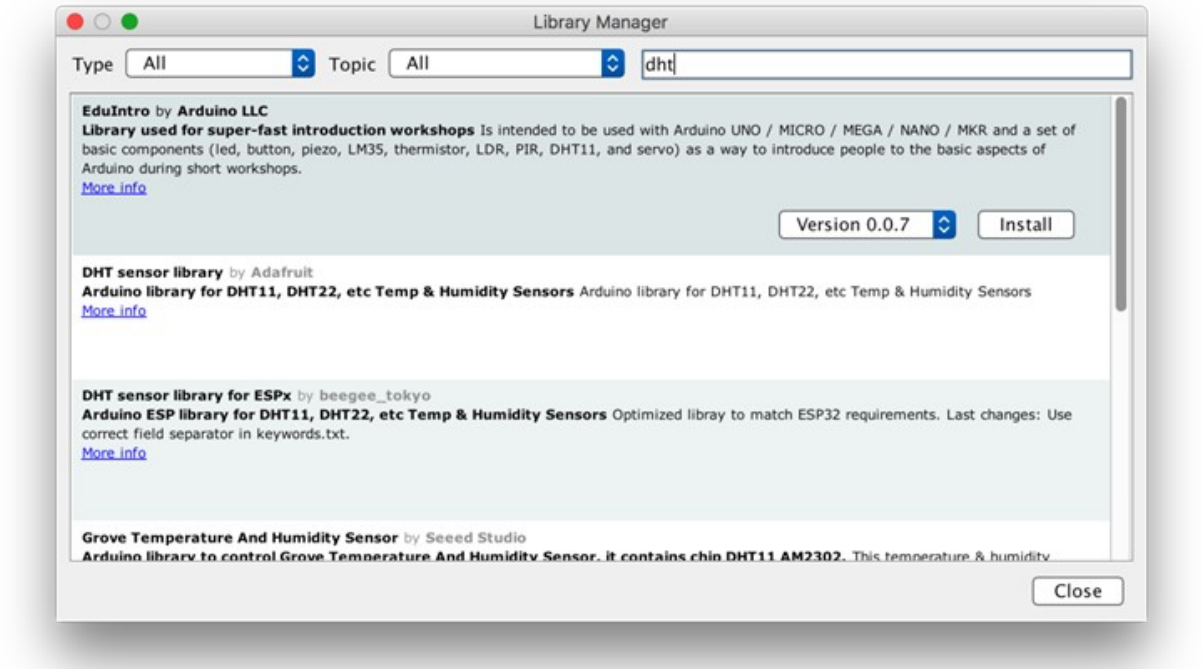

Figure 14 Library Window

The SimpleDHT library makes using the DHT11 sensor very straight forward. To use the library, include it with, and then create a SimpleDHT11 object. Assign a pin for communicating with the sensor. In our case, it is pin 7. Create two variables for temperature and humidity. Then use the read() function, with the address of the two variables passed as parameters. A minimal working program looks like this:

```
/*
 * DHT11 Demo
 */
#include <SimpleDHT.h>
/* 1 */
byte temperature = 0;
byte humidity = 0;
const int pin = 7;
SimpleDHT11 dht11;

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

void loop() {
/* 2 */
  if (dht11.read(pin, &temperature, &humidity, NULL)) {
    Serial.print("Error reading DHT11.");
    return;
  }
/* 3 */
  Serial.print(String(temperature) + " *C, ");
  Serial.print(String(humidity) + " %");
  delay(5000);
} /* end */
```

1 Variables for temperature and humidity need to be 8 bits, so the **byte** type is used.

2 If there is an error with read(), an error message is printed to Serial Monitor, and then the return skips execution to the next run loop.

3 The variables are converted to strings, concatenated with related text, and then printed to Serial Monitor.

Now let's briefly examine the Liquid Crystal I2C library. First, LiquidCrystal_I2C object needs to be initialized with the LCD's address, the number of characters per line, and the number of lines. Here are some functions that can be used:

- **init()** - Initializes the LCD driver.
- **backlight()** - Turns on the backlight LED.
- **print(text)** - Prints text to the LCD.

- **home()** - Returns the cursor to position 0,0 of the LCD.
- **clear()** - Clears all contents and settings of the LCD.
- **setCursor(x, y)** - Moves the cursor to the x^{th} position in the line and to the y^{th} line.

Now let's look at a simple sketch that uses the LCD.

```
/*
 * 1602 LCD Demo
 */
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

/* 1 */
LiquidCrystal_I2C lcd(0x27,16,2);

void setup() {
 Serial.begin(9600);
 lcd.init();
 lcd.backlight();
 lcd.print("I love Arduino!");
}

void loop() {}
/* end */
```

1 The LiquidCrystal_I2C object is created and initialized with the address to 0x27, 16 chars, and 2 lines. 0x27 is the address for LCDs with the PCF8574 chip. LCDs with the PCF8574A chip have the address is 0x3F. You should have seen one of these values with the I2C scanner sketch above. Use 0x3F if you have the PCF8574A chip.

Now that we have used the DHT11 sensor and the 1602 LCD, let's combine them to make a thermometer with a display.

```
/*
 * Digital Thermometer with LCD
 */
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <SimpleDHT.h>

byte temperature = 0;
byte humidity = 0;
const int pin = 7;
SimpleDHT11 dht11;
LiquidCrystal_I2C lcd(0x27,16,2);

void setup() {
 Serial.begin(9600);
 lcd.init();
 lcd.backlight();
}
void loop() {
  if (dht11.read(pin, &temperature, &humidity, NULL)) {
    Serial.print("Error reading DHT11.");
    return;
  }
  string text = String(temperature) + " *C, \n" + String(humidity)
+ " %");
  lcd.print(text);
} /* end */
```

Now you have experience importing libraries and using them. There is a vast ecosystem of libraries available for you to use in your projects.

For more information, see these links.

- I2C bus: https://en.wikipedia.org/wiki/I²C
- DH11 Datasheet: ***https://cdn-learn.adafruit.com/downloads/pdf/dht.pdf***

Using an Arduino In Production

While Arduino is better suited for prototyping, using it in production is viable. Leveraging the design effort that has gone into the Arduino ecosystem can save a lot of time and money, but you should be aware of some of the caveats.

If your company has a problem with using GPL code in production or a similar issue with other licenses, make sure to audit the libraries your project includes to make sure they align with business needs. If a license requires code to be published, that might affect commercial viability of a product.

Another consideration for libraries included in a product is code quality. Libraries have a wide range of code quality, and so you must determine if the code you are using is robust enough for production. Some issues, like poor memory management, might not pop up until the devices has run for an extended time. Ultimately it is up to you to determine whether or not the code is suitable for production. The core libraries on the more established microcontrollers have been battle tested.

Debugging things like memory leaks in sketches by printing to serial can be arduous, and Arduino IDE only offers this limited debugging method. Atmel Studio for offers JTAG debugging and can be used with AVR and SAM microcontrollers. Debugging with JTAG allows you to step through the code and view memory contents, very useful for debugging. Another alternative development environment is PlatformIO, which runs on top of Microsoft VSCode.

Arduino might become too limiting for your application, or Arduino might not be well supported for the microcontroller in your project. In that case, it might be preferable to use the native SDK for that chip. The setup() and loop() structure of a sketch might not suit an application well. If you are fighting the Arduino architecture or you need to deal with low level details abstracted by Arduino, maybe using the native SDK is preferable. If there is a possibility of more complexity being added to the project later, that is something to be considered. For example, if the code needs to do several different tasks, perhaps a RTOS (real-time operating system) with native SDK is better suited for the application.

It is likely you will want to design a custom circuit board for your application. The designs for all the Arduino boards are public, so they are a good starting point for your designs. Reusing designs will save time and effort. The same components from the Arduino boards and shields could be use in custom circuit board designs to save costs.

In conclusion, using Arduino in production is feasible, even advantageous, if your code fits within Arduino's structure and level of hardware abstraction. It's up to you to determine if the included library code is good enough and free of legal problems.

Conclusion

In this course we have explored what the Arduino platform is and how to use it. We have covered digital I/O, analog I/O, serial communication, interrupts, and using imported libraries. You now have the foundation to make some gadgets by harnessing all the resources Arduino provides. I hope you have enjoyed the course and gotten something out of it.