Rapid Electronics Prototyping with Arduino

Arduino Projects: Interfacing with Common Hardware

By

Benjamin Tyler, PE
PDH4Engineers

September 18, 2022

Contents

Introduction....4
Author Introduction....5
Note About Source Code In This Course....6
Suggested Course Materials....7
Part One – Interfacing with Common Hardware....9
Limit Switch....9
Explanation of Code....12
Thermistor....13
Explanation of code....15
Phototransistor....15
Explanation of code....18
Ultrasonic Sensor Module....18
Explanation of code....20
Servo 20
Stepper Motors....22
Explanation of code....28
RGB LEDs....29
Explanation of code....32
Real-Time Clock (RTC) Module and I2C EEPROM....32
Explanation of code....36
Part 2: Computer-Controlled I/O Device....38
Explanation of code....42
Explanation of code....45
Conclusion....46
Figure 1: A typical limit switch....9
Figure 2: Wiring diagram for limit switch....11
Figure 3: Thermistor wiring diagram....14

Figure 4: Phototransistor wiring diagram....17
Figure 5: Typical ultrasonic module....18
Figure 6: Ultrasonic module wiring....19
Figure 7: Servo wiring....21
Figure 8: Structure of an H-bridge circuit (in red) driving a DC motor. Image: Cyril BUTTAY [CC BY-SA 3.0], from Wikimedia Commons....23
Figure 9: Unipolar stepper with center taps tied together....24
Figure 10: 28BYJ-48 coils, from datasheet....25
Figure 11: Stepper motor wiring....26
Figure 12: Common cathode RGB LED....30
Figure 13: NeoPixel wiring....31
Figure 14: DS3231 module....33
Figure 15: DS3231 module wiring....34
Figure 16: Arduino Device Controlled By Computer....38

Introduction

This course provides a practical introduction to controlling common hardware with Arduino. Arduino is an ideal platform for rapid development of programmable electronics. Arduino has simplified the software and hardware aspects of electronics development. The Arduino ecosystem has lots of freely available and useful libraries that greatly simplify coding your projects. Additionally, there is a plethora of open, documented development boards that you can use as a guide for the hardware aspect of your projects. You leverage all the freely available resources as an engineer to rapidly develop a working solution.

In this course, we will create several practical projects with Arduino. The first part of this course focuses on interfacing with hardware like stepper motors, servos, DC motors, and RGB LEDs. The goal is to give you a working knowledge of using Arduino with different hardware but not a deep understanding of the theory behind the hardware. The second part of the course implements an Arduino controller that communicates via serial with a host computer. The Arduino controller reads sensors, controls actuators, and communicates with the host. It essentially operates as dumb I/O for the host. The computer runs a Python program that controls the flow of the system logic and processes data.

If you are not familiar with Arduino or have not done much programming before, I recommend you take the preceding Introduction to Arduino class first. The previous class covers topics like installing Arduino, installing libraries, digital and analog I/O, and I2C communication.

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.

Note About Source Code In This Course

Source code for this course is under the “MIT No Attribution” license. Do not use this code for medical, aviation, or other safety-critical applications. You may use this code freely, even for commercial applications. License included below:

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***

The following are Amazon affiliate links to suitable kits and parts:

Elego Uno Super Start Kit

Elegoo Uno Complete Starter Kit

Elegoo Complete Uno Starter Kit

Lafvin Super Starter Kit

Adafruit NeoPixel Strip

Disclaimer: PDH4Engineers is a participant in the Amazon Services LLC Associates Program, an affiliate advertising program designed to provide a means for sites to earn advertising fees by advertising and linking to Amazon.com.

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

- Arduino Uno, Leonardo, Zero, Mega, Micro, Nano, etc
- solderless breadboard
- 10k Ohm resistor
- 220 Ohm resistor (can be up to 1k, does not need to be exactly 220)
- tactile switch
- LED
- ultrasonic sensor
- DS3231 module

- servo
- stepper motor and driver module/ULN2003A H-bridge or similar
- NeoPixel. This probably won’t be included in any of the popular beginner kits.

Part One – Interfacing with Common Hardware

One of Arduino's strengths is that it is makes interfacing with hardware very easy. Usually, all the low level details have been taken care of for you. In this part we will do several mini projects that interface with common devices including: limit switches, thermistors, phototransistors, RGB LEDs, servos, stepper motors, ultrasonic range finders, rotary encoders, and real-time clocks.

We will use common libraries where possible.

Limit Switch

Parts needed for this mini project:

- Arduino Uno board (or other Arduino board)
- solderless breadboard
- tactile push button switch
- jumper wires

Limit switches are common and often necessary components in motion control applications. They are placed in a moving part's path to limit its range of motion. Limit switches are used for safety, preventing damage to equipment, and establishing home coordinates for control. Limit switches come in different form factors that can be triggered by linear or rotational motion.

Figure 1: A typical limit switch

In this mini project, we will use a miniature tactile switch to simulate a limit switch. Many common limit switches are momentary switches, but they usually have spade type terminals. We can avoid the fuss of soldering or using connectors by using a tactile switch you probably already have if you bought an Arduino kit.

In the previous Introduction to Arduino course, we interfaced with a tactile switch without a library. Most of the available Arduino libraries for buttons and switches rely on polling. If your sketch's loop() runs quickly each loop, polling is a viable solution. If there is a chance that a loop might take long enough that it would delay reading the limit switch, using an interrupt on pin change is a good idea. The ezButton library is a popular library for buttons and different kinds of switches.

While manually coding buttons is easy enough with Arduino, the ezButton library makes things… simpler. It takes care of debouncing and setting pullups. In addition, it is non-blocking and allows the use of multiple buttons. Install it using the Library Manager in Arduino IDE by going to Tools→Manage Libraries.

Note that the button in our case will use an internal pullup, so the pin connected to the button will read HIGH when not pressed and LOW when pressed.

Some functions in this library include:

ezButton(int pin) – this creates a new instance of the ezButton class for the pin number passed to it.

setDebounceTime(int time)- this sets the debounce delay for a pin to prevent it from chattering. If this function is not called, the debounce delay is automatically set under the hood with a default value.

loop()- must be called each time the main loop in order to update the button's state.

getState()- returns whatever the current state is, 1 for logic HIGH or 0 for logic LOW.

isPressed()- returns true if the button has been press since the last loop.

isReleased()- returns true if released since the last loop.

ezButton can also keep track of the times a number has been pressed.

Please build the circuit below:

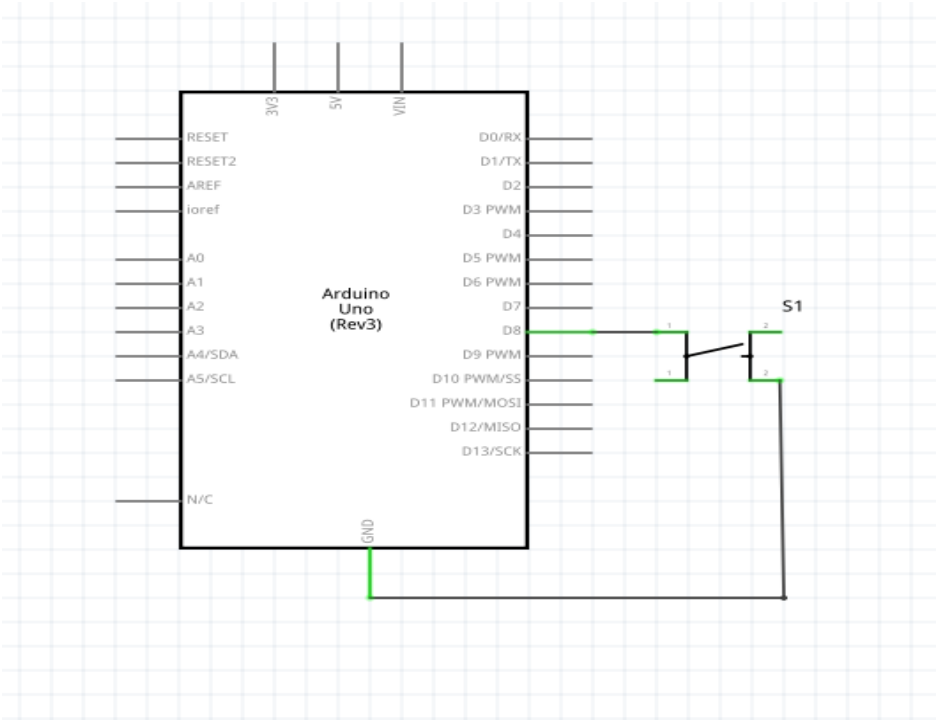


Figure 2: Wiring diagram for limit switch

Next, copy/paste or type the following code into the Arduino IDE. Make sure you have installed the ezButton library. Our code is a contrived example where we have a hypothetical CNC machine with a milling tool that moves along a rail. One end of the rail has a limit switch. Pressing the button simulates the milling tool hitting the limit switch. Don't read too deep into hypothetical aspect of the code, as it's only purpose is to demonstrate ezButton operation. After uploading code to your board, open the Serial Monitor.

```
/* Demonstrate ezButton library for use with limit switches*/
#include <ezButton.h>
// 1
ezButton limitSwitch(8);
bool isToolMoving = true;
bool isToolHome = false;

// 2
```

```
void setup() {
  limitSwitch.setDebounceTime(40);
  Serial.begin(9600);
}

// 3
void loop() {
  limitSwitch.loop();

  if (limitSwitch.isPressed()) {
    Serial.println("Limit switch pressed");
    isToolHome = true;
    isToolMoving = false;
  }
// 4
  if (limitSwitch.isReleased()) {
    Serial.println(“Limit switch released”);
    isToolHome = false;
    isToolMoving = true;
  }

  if (isToolMoving) {
    Serial.println(“Tool is moving”);
  }

  if (isToolHome) {
    Serial.println(“Tool is in HOME position”);
  }
}/* end program */
```

Explanation of Code

1. An ezButton object is created for the limit switch. Pin 8 is set up as an input and the internal pullup is set.

2. The limit switch debounce time is set. This might need to be adjust up or down to prevent bouncing yet not take too long.

3. The loop() method for the limitSwitch object needs to be called each iteration of the main loop in order to update the switch’s internal state. If a switch press is detected, a message is printed.

4. If a switch release is detected, a message is printed.

Thermistor

Parts needed for this mini project:

- Arduino Uno board (or other Arduino board)
- solderless breadboard
- jumper wires
- NTC thermistor
- 10k resistor

Thermistors are resistors that change resistance with temperature, more so than ordinary resistors. Thermistors can be used as temperature sensors or as way to limit current. Thermistors are similar to RTDs (resistance temperature detectors), except that thermistors are made from ceramics or polymers as opposed to metals in RTDs. There are two types of thermistors: NTC and PTC. NTC thermistors decrease in resistance as temperatures rise, whereas PTC thermistors increase in resistance as temperatures rise. This makes PTC thermistors suitable as current limiters. NTC thermistors have become popular as temperature sensors due to accuracy being as good as ± 0.1 °C from 0 °C to 70 °C.

Thermistor resistance does not scale linearly with temperature, so the approximation is calculated using the Steinhart-Hart equation. The Thermistor library by panStamp takes care of this calculation for you. Install the Thermistor library in the Library Manager.

For this mini lab, we build a circuit with a voltage divider consisting of a NTC thermistor and a 10k resistor. The thermistor is 10k Ohms at nominal 25 °C, and to get better accuracy, use a 1% 10k resistor. For demonstration purposes, use whatever 10k resistor you have on hand.

Wire your components as shown below:

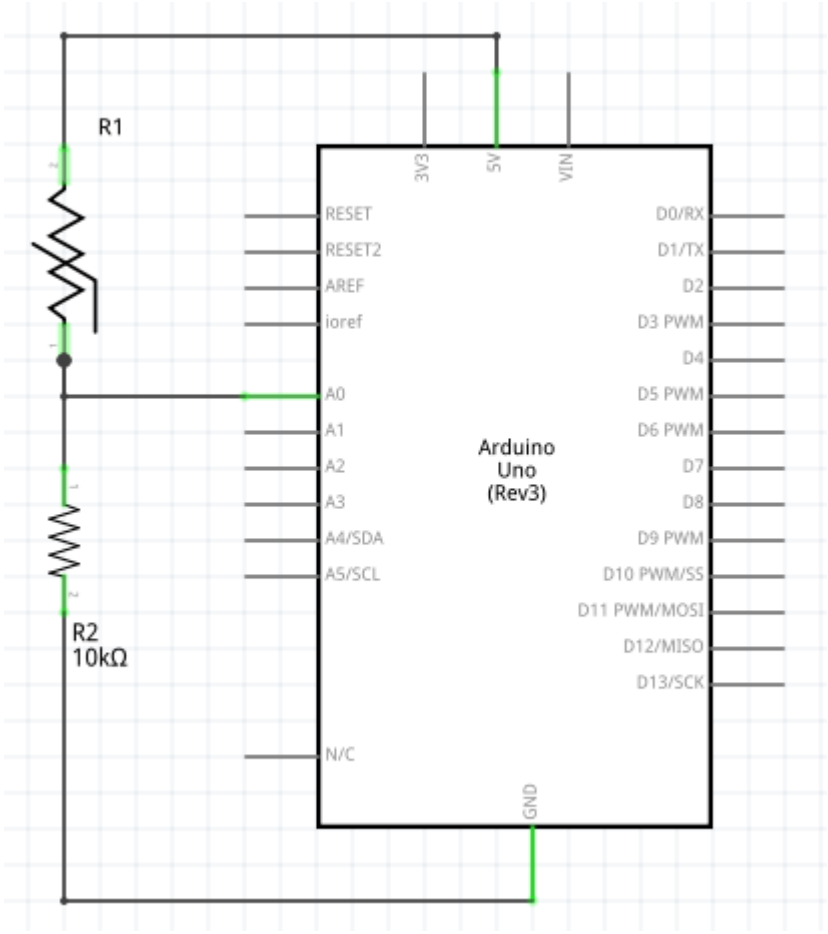


Figure 3: Thermistor wiring diagram

The Thermistor library is simple to use. You only need to create a Thermistor object and read from it.

THERMISTOR(uint8_t AdcPin,
uint16_t nominalResistance,
uint16_t betaCoefficient,
uint16_t serialResistor) – this constructs a THERMISTOR object. Pass in the the nominal resistance at 25 °C, the beta coefficient, and the resistance of the series resistor.

read()- returns the temperature in increments of 0.1 °C.

Type or copy/paste the code below into Arduino IDE:

```
/* Thermistor test */
#include “thermistor.h”

THERMISTOR thermistor(A0, 10000, 3950, 10000);

void setup() {
  Serial.begin(9600);
  Serial.println(“Temp in deg C:”);
}

void loop() {
  Serial.println(thermistor.read());
  delay(1000);
}
```

Explanation of code

This sketch creates the thermistor object and reads from it every second. Instead of reading the temperature and storing it to a variable, read() is called inside println(). Storing the value from read() to a variable and printing the variable is fine, too.

Phototransistor

Parts needed for this mini project:

- Arduino Uno board (or other Arduino board)
- solderless breadboard
- tactile push button switch
- jumper wires
- phototransistor

In this mini project we interface with a phototransistor. The phototransistor is a semiconductor device that passes more current proportional to intensity of light exposure. Before the phototransistor was popular, photocells, or light dependent resistors, were commonly used for light detection applications. Photocells are made with cadmium sulfide, so they are not RoHS compliant.

Phototransistors are useful for things that need to detect light levels like nightlights, street lamps, and cameras. They can also act as a switch when a known light source shining on the phototransistor is impeded. They tend to be accurate and have a linear response.

The datasheet for the transistor lists specs for certain conditions, for example a supply voltage of 5V and a series resistor of 1k Ohms. In darkness, the transistor might be rated for 100nA, so the Arduino analog input would read a voltage of 100nA x 1k = 100uV. The Arduino Uno has a 10-bit analog to digital converter, so each step of the ADC is 5V/1023 (2^{10} – 1 because one step is at 0) = 4.89mV. So, in darkness, the ADC will read 0, whereas at the other end in total brightness, the device saturates. At saturation, there will be a voltage drop of about 0.3V across the transistor. Therefore, the top voltage reading will be about 4.7V.

Wire the circuit below:

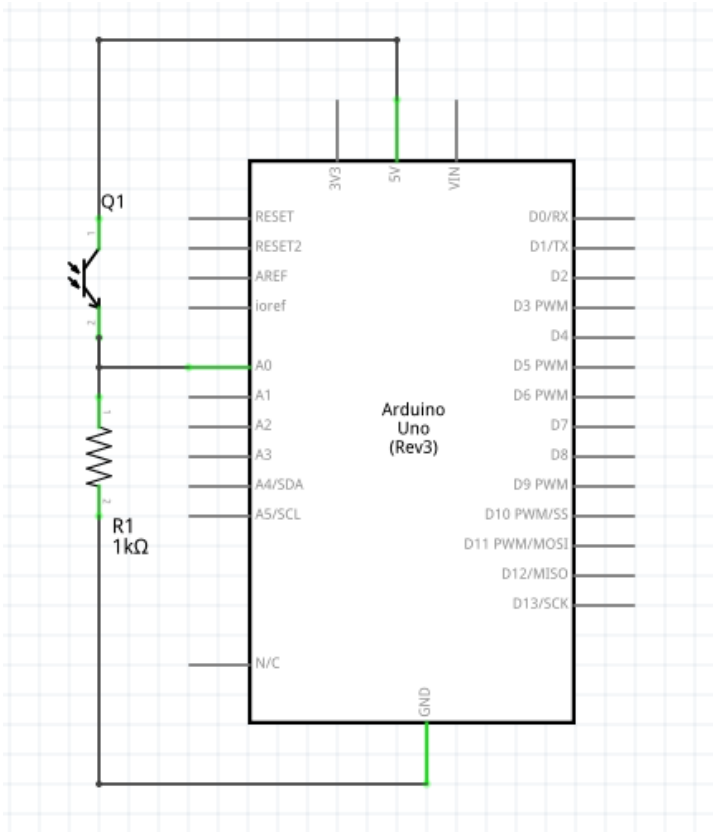


Figure 4: Phototransistor wiring diagram

The code for this is nearly identical to the last mini project. Type or copy/paste the code below and start the Serial Monitor:

```
/* phototransistor test */
int reading = 0;

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

void loop() {
  reading =  analogRead(A0);
  Serial.print("The brightness reading is: ");
  Seral.println(reading);
  delay(1000);
```

```
}
```

Explanation of code

In this sketch, we create a variable to hold the current reading, read from the analog input, and print the reading value. The raw analog to digital converter value is use here, but in a real use case it would probably need to be scaled to lux.

Ultrasonic Sensor Module

Parts needed for this mini project:

- Arduino Uno board (or other Arduino board)
- solderless breadboard
- jumper wires
- ultrasonic module

Ultrasonic sensors are useful for measuring distances up to about 5m for common low-cost sensor modules. Some typical applications include distance measurement, object avoidance for robotics, and level sensing.

Figure 5: Typical ultrasonic module

The ultrasonic modules work by sending a pulse out one transducer and receiving the reflected pulse in the other transducer. The HC-SR04 is a typical module used with Arduino. It has max range of 4m and min range of 2cm. It operates at 40kHz, and the ping it emits consists of an 8-cycle burst.

To use the module, the Arduino first generates a 10 microsecond pulse on the Trig pin, and then when an echo is detected, the module generates a pulse on the Echo pin that indicates the distance. The Arduino measures the length of the pulse from the Echo pin to calculate distance to the object reflecting the ultrasonic wave. The calculations for distance follow:

pulse round trip time in µs = Echo pin pulse duration

speed of sound = 340 m/s = 0.034 cm/µs

distance in cm = (Echo pulse duration)/2 * 0.034

The Arduino code for the above operations is straightforward enough, but there is a nice library called Ultrasonic by Erick Simões. Install the library in the Library Manager. Build the circuit below:

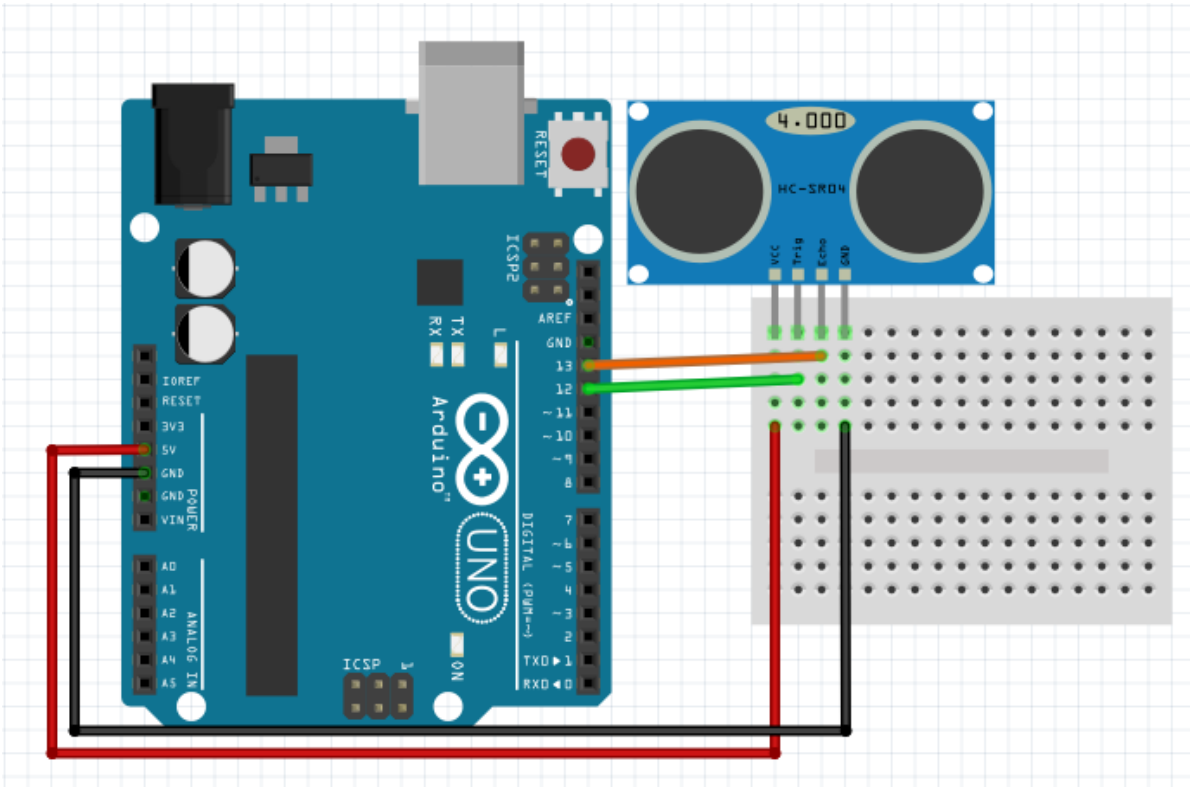


Figure 6: Ultrasonic module wiring

In our circuit, we have pin 12 hooked up to Trig and pin 13 hooked up to Echo. Pin 12 is an output, and pin 13 is an input. The Ultrasonic library takes care of all the hardware setup. To use the Library, create an Ultrasonic object and run the read() method. We can also specify a timeout period, as reading in a pulse is a blocking operation, and there might not be an echo if there is not an object in range and field of view of the sensor.

Type or copy/paste the code below into Arduino IDE:

```
/* ultrasonic test */
#include <Ultrasonic.h>

Ultrasonic sensor(12, 13, 30000UL);

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

void loop() {
  Serial.print("Distance in cm: ");
  Serial.print(sensor.read());
  Serial.print(", inches: ");
  Serial.println(sensor.read(INC));
  delay(2000);
}
```

Explanation of code

In the above code the timeout is set to 30,000µs which works out to a distance of about 5m. The UL after the timeout value stands for unsigned long, which tells the compiler to use the right kind of variable that can hold larger numbers. The second time read() is called, we pass a constant INC (declared in the library) to it in order to get a reading in inches.

Servo

Parts needed for this mini project:

- Arduino Uno board (or other Arduino board)

- solderless breadboard
- jumper wires
- servo

A servo is a geared motor than can rotate up to 180 degrees. Servos can be controlled by analog or digital signals. Analog servos tend to be less expensive, have low frequency audio noise, and have lower power consumption. However, then tend to be slower, have less torque, lower resolution (fewer steps), and have a larger deadband. The deadband is the amount of time a control pulse needs to change in order for the servo to move. A larger deadband means less precise movement. Digital servos tend to be faster, more precise, have more torque, and have a smaller deadband. The enhanced features come with higher prices and higher power consumption. Since they use higher frequency PWM, they produce higher pitch audio noise.

A typical servo that comes with Arduino kits is the SG90. It has a stall torque of 1.6kg/cm and has a deadband of 5µs. It can be controlled by using the PWM output from an Arduino board.

Wire up the circuit below. Double check the power, ground, and pulse pins, as they can vary.

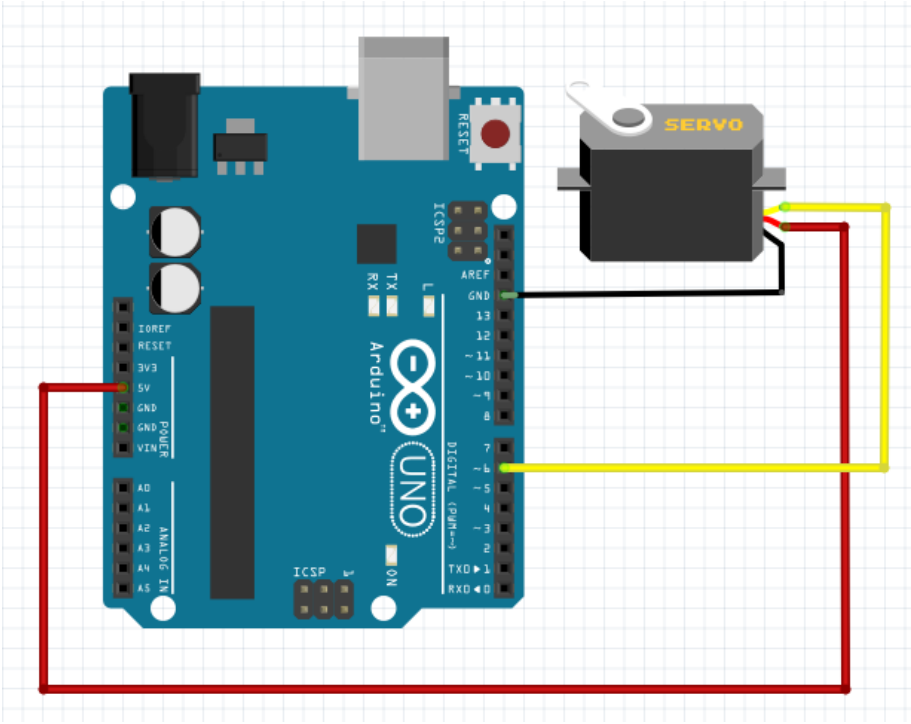


Figure 7: Servo wiring

Now let's create a program that turns the servo all the way back and forth. The Servo library comes included with Arduino IDE, so it does not need to be installed. Type or copy/paste the code below:

```
/* servo test */
#include <Servo.h>

int servoPin = 6;

Servo servo;

void setup() {
  servo.attach(servoPin);
}

void loop() {
  for (int i = 0; i <= 180; i++) {
    servo.write(i);
    delay(20);
  }
  servo.write(0);
  delay(3000);
}
```

Explanation of code

In this code we set the servo to sweep from 0 to 180 degrees and then finally reset to 0.

Stepper Motors

Parts needed for this mini project:

- Arduino Uno board (or other Arduino board)
- solderless breadboard with power and ground rails
- 6x F-M and 1x M-M jumper wires
- unipolar stepper motor such as 28BYJ-48
- ULN2003 H-bridge chip or break out board

- power supply module with 9V power adapter OR an alternative power supply that can source over 500mA

Stepper motors are brushless DC motors that divide each rotation into a fixed number of steps. The distance traveled in one step is repeatable and only has a small non-cumulative error. Steppers are rugged, reliable, and precise. They have high torque at low speeds and at startup. A controller sends pulses in specific patterns to the motor coils, thereby turning the motor shaft. The sequence of pulses determines the rotation direction, and the frequency of the pulses sets the speed. Some steppers can be driven in half steps to give twice the angular resolution.

To drive the stepper, the Arduino controller sends pulses to an H-bridge that energizes the motor coils. The motor needs more current than a microcontroller output pin can provide, so the H-bridge functions like a voltage-controlled current switch. Inside the H-bridge are high current transistors arranged such that they can drive a stepper motor in either direction.

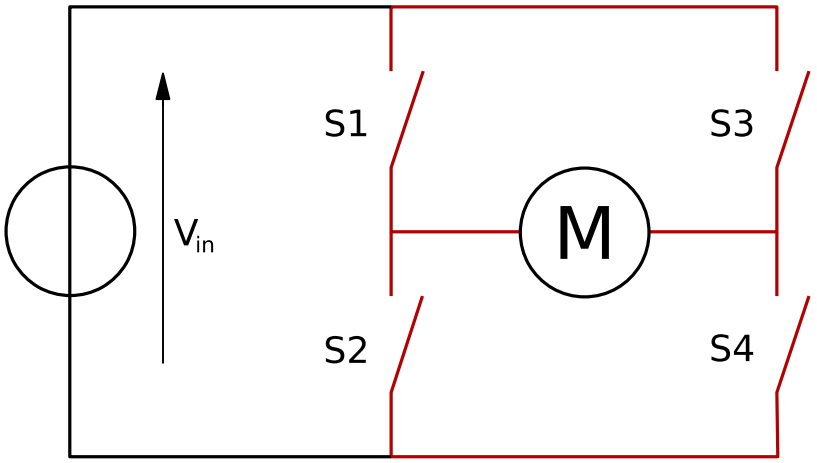


Figure 8: Structure of an H-bridge circuit (in red) driving a DC motor. Image: Cyril BUTTAY [CC BY-SA 3.0], from Wikimedia Commons

One important feature of steppers is they can be controlled in an open loop. Provided the stepper is not missing steps, the stepper's position can be tracked by counting steps. Sometimes it is preferable to have a home position for the controller, in which case a mechanical limit switch, optical sensor, or a Hall effect sensor can be used to detect when a stepper reaches the home position. A closed loop control that uses a rotary encoder for feedback tracks position more accurately.

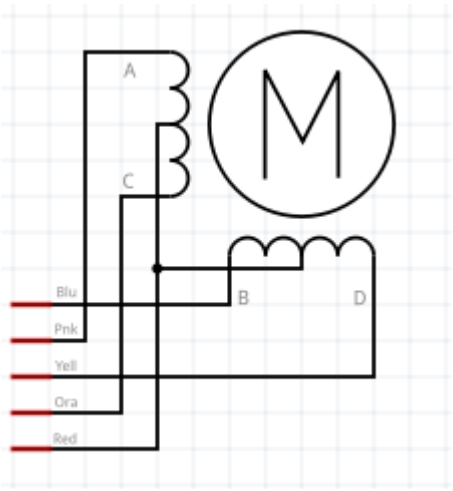


Figure 9: Unipolar stepper with center taps tied together

A typical unipolar stepper has six wires, with two wires connecting to the coils at center taps. The tapped coils can act as separate coils, giving a total of four coils. Bipolar steppers usually have four wires, lacking the center-tapped wires. Generally speaking, unipolar steppers are easier to control, but bipolar steppers can generate more torque.

One common stepper motor included with many kits is the 28BYJ-48. It is a unipolar stepper with the following specs:

- Rated voltage: 5V
- Number of phases: 4
- Stride angle: 5.625°/64
- DC resistance: 50 ±7%(25℃) Ω
- In traction torque: >34.3mN.m
- Friction torque: 600-1200 gf.cm
- Pull in torque: 300 gf.cm

The 28BYJ-48 has gears internally to increase torque and reduce speed. The gear reduction is specified to be 64:1, but some intrepid hobbyists have found the exact ratio is about 63.68395:1. Each manufacturer might be slightly different or have different specified ratios like 16:1 or 32:1.

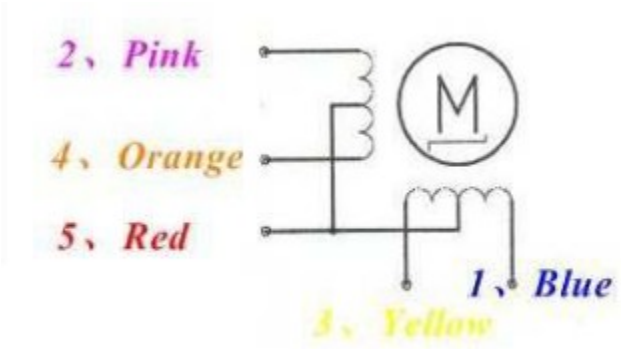


Figure 10: 28BYJ-48 coils, from datasheet

In this mini lab we are controlling a unipolar stepper with a ULN2003A H-bridge. The ULN2003A has seven channels, each with a Darlington transistor that can conduct 500mA. Each channel has clamping diodes to protect it from the back EMF from inductive loads. In order to control a unipolar stepper with four channels, each of the center-tapped wires is tied to the power supply. To keep things simple, the stepper will operate in open loop.

Please note that since the motor draws a lot of current, the voltage regulator chip feeding the motor will get hot. If the regulator and H-bridge do not have heatsinks, be careful not to run the motor too long.

Build the circuit below:

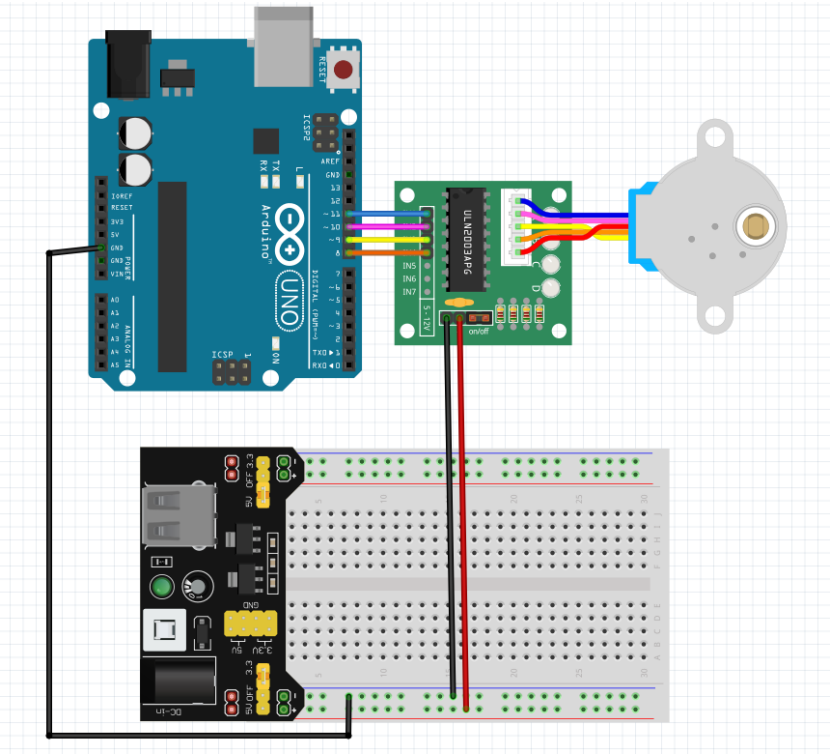


Figure 11: Stepper motor wiring

If you do not have a power module like in the diagram above, you can wire a 9V battery to the power and ground pins of the ULN2003A driver board or wire a barrel connector that a wall wart power adapter plugs into.

The sketch for this mini lab uses the Stepper library. It comes with Arduino IDE, so there is no need to install anything. Some functions we use from the library include:

Stepper(totalStepsForOneRotation, pin1, pin2, pin3, pin4) – this constructs a Stepper object that holds that state of the stepper motor. pin1 and pin2 connect to the one coil, and pin3 and pin4 connect to the other coil. Double check the motor's wiring diagram to make sure each pair of pins corresponds to a single coil, not different coils.

.setSpeed(int speed) – this sets the stepper speed in rpm (revolutions per minute)

.step(int numberOfSteps) – this tells the stepper to step the specified number of times. A positive number of steps rotates the stepper shaft in one direction, and a negative number rotates in the opposite direction.

In this sketch, we send a number of steps over serial, and the Arduino alternates stepping the motor clockwise and counterclockwise. Type or copy/paste the code below:

```
// 1
/* stepper test */
#include <Stepper.h>
#define BUFFLEN 10

const int steps = 64;
const int gearReduction = 32;
const int totalSteps = steps * gearReduction; // needed by Stepper
char inBuff[BUFFLEN];  // holds characters read from serial
int bufferIndex = 0;   // index for input buffer
int inSteps = 0;       // the number of steps, converted from buffer
char inChar;          // holds each individual char read from serial
int clockwise = 0;     // rotation direction

Stepper stepper(steps, 8, 10, 9, 11);

// 2
void setup() {
  Serial.begin(9600);
  stepper.setSpeed(200);
  memset(inBuff, 0, sizeof(inBuff));
}

// 3
void loop() {
    if (Serial.available() > 0) {
      inChar = Serial.read();
      Serial.print(inChar);
      if (isDigit(inChar)) {
        inBuff[bufferIndex] = inChar;
```

```
        bufferIndex += 1;
      } else {
        inSteps = atoi(inBuff);
        memset(inBuff, 0, sizeof(inBuff));
        bufferIndex = 0;
      }
    }

// 4
   if (inSteps) {
    if (clockwise) {
      Serial.print("clockwise steps: ");
      Serial.println(inSteps);
      stepper.step(-inSteps);
      dir = 0;
    } else {
      Serial.print("counter clockwise steps: ");
      Serial.println(inSteps);
      stepper.step(inSteps);
      dir = 1;
    }

    inSteps = 0;
    delay(1000);
   }
}
```

Explanation of code

1. We set up all the variables. Since we are reading from a serial port, we need a buffer to hold what is read. We have an index and the value the buffer converts to. The Stepper object takes the total number of steps for one rotation of the stepper shaft and the pins corresponding to the stepper's coils.

2. In setup, we set the stepper's speed to 200 rpm, and set all the bytes in the input buffer to 0 with memset. This has the benefit of automatically null terminating the sequence of chars read from serial, as well as making sure the memory does not contain random garbage that could cause an error later.

3. Inside loop(), if a byte is in the serial port buffer, it is read. To be clear, we are dealing with two different buffers- one is part of the serial port, and one we have set up in software to hold all the all the bytes in a message. If the received character is a number, it is written to the input buffer and the buffer index is incremented. Eventually a newline or other non-numeric is detected, and the digits in the buffer are converted to an integer. Then the buffer and buffer index are reset.

4. If the value for number of steps is nonzero, the servo then moves that number of steps. The number of steps is reset, and there is a one second delay.

One thing to note is that there is no error handling performed. Do not cut and paste this code into something critical.

RGB LEDs

Parts needed for this mini project:

- Arduino Uno board (or other Arduino board)
- solderless breadboard
- jumper wires

Adding colored lights to a project can help convey certain ideas like stop, warning, or ready, as well as make the project more aesthetically pleasing to look at. RGB LEDs offer an easy, attractive way to light up a project, and they can be reconfigured on the fly.

Common four pin RGB LEDs come in two types, common anode and common cathode. These are controlled by using three PWM outputs to drive the LED's red, green, and blue pins. This is straightforward, so we'll look at a more interesting kind of RGB LED.

Figure 12: Common cathode RGB LED

The NeoPixel is an individually addressable and serial-controlled RGB LED module. There are three parts known as the NeoPixel, including the WS2812, WS2811, and SK6812. NeoPixels can be daisy-chained and typically come packaged as a strip of NeoPixels. Only one I/O pin is needed as opposed to three pins for a four pin RGB LED. One downside is that the refresh rate is lower than other four pin LEDs, due to the serial protocol of the devices. There is a small propagation delay for each node that becomes apparent the more nodes there are in a chain.

The way the protocol works is that each NeoPixel needs 24 bits of data, 8 bits for each color. The 8 bit byte value can have a value between 0 and 255. When a pulsetrain arrives at a NeoPixel, it strips off the first three bytes it receives for itself and then passes through the rest of the pulses. So the number of bytes needed to control a strip of NeoPixels is N * 3. The bytes are sent with the most significant digit first. The first byte is green, the second is red, and the third is blue.

Since the timings for the NeoPixel protocol are tight, libraries for the NeoPixel have code written in assembly for various microcontroller architectures.

One thing to note is that NeoPixels are stated to support 5V logic, but in practice 3.3V logic works fine. This course uses the Arduino Uno which has 5V logic, but most other Arduino boards are 3.3V. If the voltage creates a problem, then you will need to level-shift the voltage using a chip like the 74AHCT125.

For this mini lab, install the Adafruit_NeoPixel library. Wire the circuit below. For demonstration purposes, the circuit uses a strip of NeoPixels from Adafruit, but it can be a single Neopixel or several Neopixels chained together in another shape. You might need to solder a pin header to the NeoPixel board in order to plug it into the breadboard, or solder jumper wires directly to the NeoPixel board. Make sure you wire to the data input (DIN) pin, not the data output (DOUT) pin

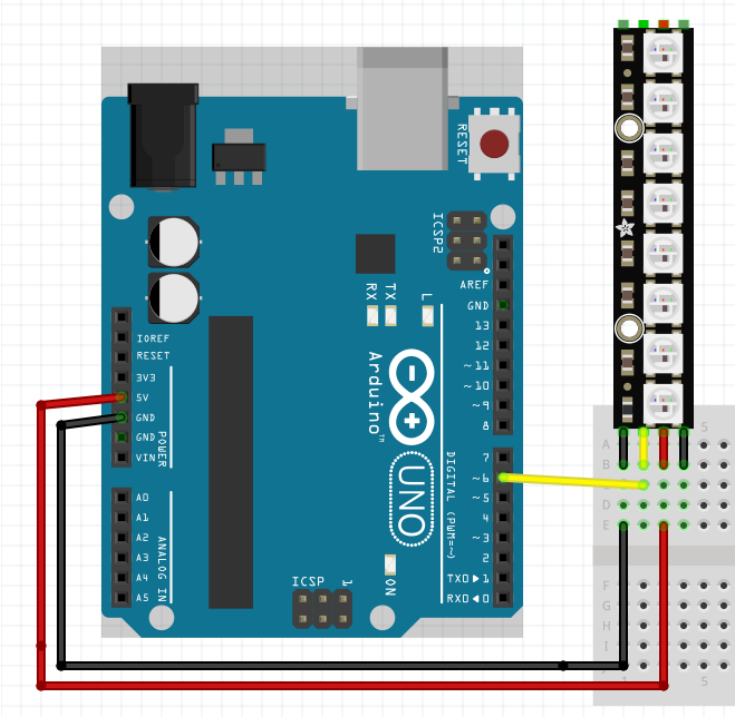


Figure 13: NeoPixel wiring

Now we will create a sketch that sets each NeoPixel to a random color. Type or copy/paste the following code into Arduino IDE:

```
// 1
/* NeoPixel test */
#include <Adafruit_NeoPixel.h>
#define     DATA_PIN          6
#define     PIXEL_COUNT       8  // number of daisy-chained NeoPixels
#define     PAUSE             1000 // time in ms to pause between refreshes

Adafruit_NeoPixel pixels(PIXEL_COUNT, DATA_PIN, NEO_GRB + NEO_KHZ800);
```

```
void setup() {
  pixels.begin();
}
// 2
void loop() {
  pixels.clear();
  for(int i = 0; i < PIXEL_COUNT; i++) {
    pixels.setPixelColor(i, pixels.Color(random(255), random(255),
random(255)));
    pixels.show();
    delay(PAUSE);
  }
}
```

Explanation of code

1. Here we establish variables for which pin we are using to communicate with the NeoPixels, how many NeoPixels we have, and how much time to pause between refreshing the NeoPixel values. Then we create an object for the NeoPixel strip, and the last two parameters are common communication settings. See the library documentation on Github for other settings.

2. The pixels are cleared. Then each NeoPixel in the strip of NeoPixels has random RGB values written. The pixels are then shown and the loop delays.

Real-Time Clock (RTC) Module and I2C EEPROM

Parts needed for this mini project:

- Arduino Uno board (or other Arduino board)
- solderless breadboard
- jumper wires
- DS3231 module

Real-time clocks keep accurate time and can use a battery backup for when the Arduino is off or in low power mode. The RTC can track time to the nearest second and report back the current year, month, week, day, hour, and second

Some commonly used RTCs are the DS3231 and DS1307, and many available modules use one of these RTC chips. In our case, we'll use the newer DS3231. Its features include I2C communication, an integrated crystal, two programmable time-of-day alarms, and a 32.768 kHz square wave output pin.

EEPROM is electrically erasable programmable read only memory. EEPROM is suitable for persisting data while a device is turned off and for storing data that is written often, ie an application like datalogging. EEPROM tends to have more write cycles than flash memory. The AT24C32 is rated for at least 1,000,000 write cycles, whereas flash memory on microcontrollers is typically rated for around 1,000 write cycles. The AT24C32 EEPROM on the DS3231 module use in this example comes with 32 kB of nonvolatile memory.

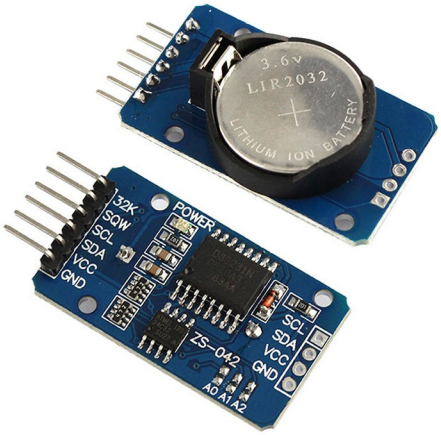


Figure 14: DS3231 module

Install RTClib by Adafruit and AT24Cxx by Manjunath, and wire the circuit below. The Arduino Uno picture does not clearly label its SCL and SDA pins, but in the top right pin header, the first pin is used for SCL and the second for SDA.

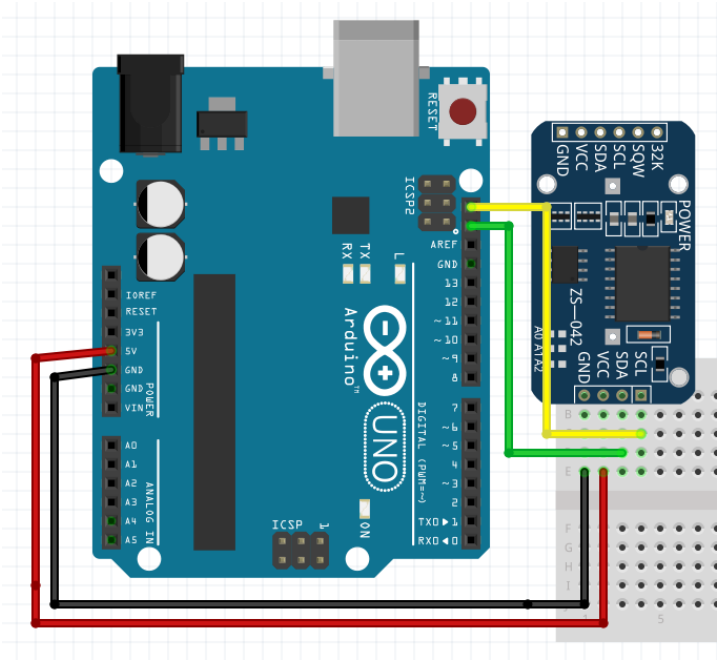


Figure 15: DS3231 module wiring

Some of the classes and functions we will use from the RTCLib include:

RTC_DS3231() – constructor for a DS3231 object.

.begin (TwoWire *wireInstance=&Wire) – starts I2C communication and returns true if successful.

.adjust (const DateTime &dt) – sets the date and time for the DS3231.

.now() - returns a DateTime object with the current date/time.

DateTime (uint16_t year, uint8_t month, uint8_t day, uint8_t hour=0, uint8_t min=0, uint8_t sec=0) – constructor for a DateTime object.

.unixtime() - returns the number of seconds since midnight, January 1, 1970.

There are other useful features like alarms and power failure detection that are referenced in the docs:
https://adafruit.github.io/RTClib/html/index.htm

Some classes and functions we will use from the AT24Cxx library include:

AT24Cxx(uint8_t i2c_address, uin32_t eeprom_size) – creates a serial EEPROM object at the specified I2C address. The size is in kilobytes.

.read(uint16_t address) – reads one byte from the specified address.

.write(uint16_t address, uint8_t value) – writes one byte to the specified address.

.update(uint16_t address, uint8_t value) – writes one byte to the specified address if it is different from the value already stored there. The purpose is to reduce the number of write cycles to the EEPROM chip.

One thing to keep in mind with this library for reading EEPROM chips: it reads and writes one byte at a time, so for any data type larger than one byte you will need to compute how many bytes to read/write. Some other libraries might make this easier. Something else to be aware of is this library does not do any error checking. If you accidentally read or write to an address that does not exist, the library will not tell you.

Now let's create a simple demonstration for the DS3231. Type or copy/paste the following code into Arduino IDE:

```
#include <RTClib.h>
#include <Wire.h>
#define AT24C32_ADDR 0x57
// 1
RTC_DS3231 rtc;
AT24Cxx eeprom(AT24C32_ADDR, 32);
int rom_addr = 31;

void setup() {
// 2
  Serial.begin(9600);
  if (!rtc.begin()) {
    Serial.println("Error starting DS3231, check wiring");
    while (true) delay(10);
  }
  rtc.adjust(DateTime(F(__Date__), F(__Time)));
// 3
  int rand = random(255);
  Serial.print("Random number written to EEPROM: " + String(rand));
  eeprom.update(, rand) //.write() would also work
```

```
}
// 4
void loop() {
  DateTime time = rtc.now();
  Serial.print("Seconds since midnight 1/1/1970: ");
  Serial.println(time.unixtime());
  Serial.print("Date: ");
  Serial.print(time.month(), DEC);
  Serial.print('/');
  Serial.print(time.day(), DEC);
  Serial.print('/');
  Serial.print(time.year(), DEC);
  Serial.print(", Time: ");
  Serial.print(time.hour(), DEC);
  Serial.print(":");
  Serial.println(time.minute(), DEC);
  Serial.print("Value from EEPROM: ");
  Serial.println(eeprom.read(rom_address), DEC);
  delay(5000);
}
```

Explanation of code

1. We create objects for the real-time clock and EEPROM. We set the EEPROM's serial address and size in kilobytes, and then the address in the EEPROM we read and write from. In my test unit, there was already data in the first 25 bytes of memory, so I chose to use an memory location a few bytes after the existing values. I've kept everything in decimal format for simplicity, but it is common to use hexadecimal with memory locations.

2. The real-time clock is started. If there is a problem, the sketch stalls. Then the DS3231 is adjusted to current time. The F(__Date__) and F(__Time) macros get the current time while compiling the code. If a battery is installed, the RTC will keep the time when the Arduino board is powered off.

3. A random number is generated and then written to EEPROM.

4. In each loop the Arduino reads the DS3231 and then reads the random number written to EEPROM.

This concludes the first part of the course. You should now be familiar with several useful pieces of hardware. Next we will create the beginnings of a practical application that can incorporate what we have worked on so far.

Part 2: Computer-Controlled I/O Device

In this project, we will create an Arduino device that can be controlled by another computer, essentially turning it into a “dumb” I/O device. Often there is a need to collect data or control things using DAQ or SCADA devices (data acquisition, supervisory control and data acquisition) in laboratory, industrial or other environments. Using Arduino for DAQ or SCADA is very convenient, inexpensive, and quick to implement. Keep in mind that that Arduino is suitable for quick and dirty applications, but it would be prudent to use a professional-grade hardware solution if controlling something that could cost a lot of money or impact safety.

Our system will consist of two main components: the Arduino board with firmware that reads and executes serial commands, and a computer (PC, Mac, Linux running on a toaster, etc) that runs a Python script. The Python program running on the computer will handle all the data processing, control, and data storage.

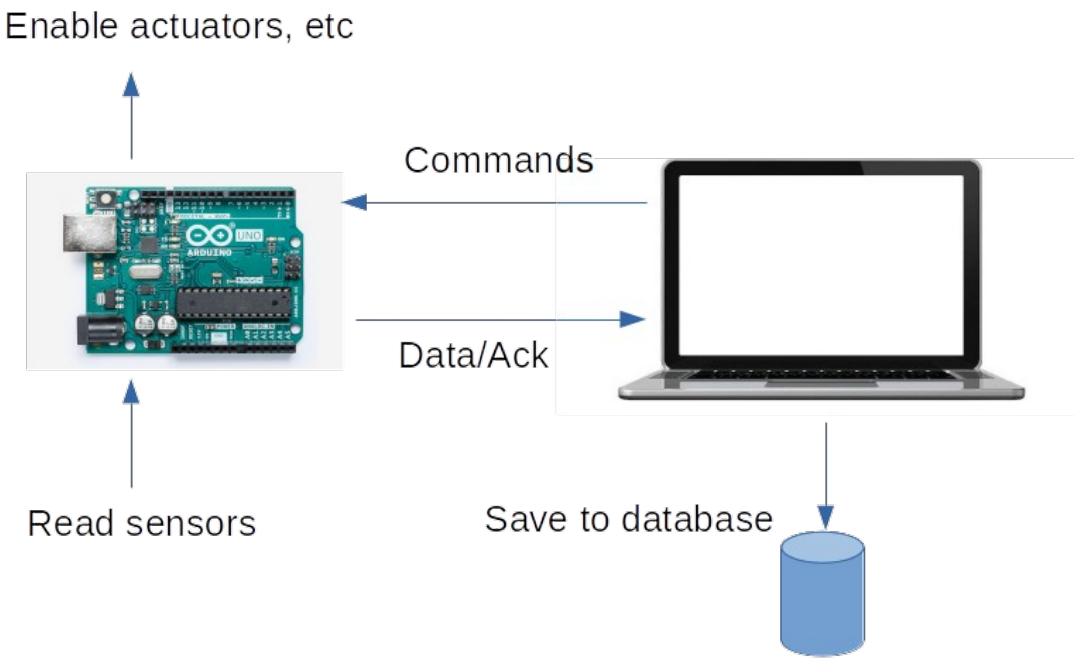


Figure 16: Arduino Device Controlled By Computer

On the computer side, the Python program will run in a loop. Every 10 seconds, the program will tell the Arduino to set the LED brightness to a random value. Every 20 seconds, the computer will ask the Arduino to read its ultrasonic sensor. For the sake of simplicity, we will write to a CSV file instead of a database.

The Arduino will output to an LED and read from an ultrasonic sensor. The LED is a stand-in for other things you might want to control, like a heating element, fan, pump, or valve. The ultrasonic sensor is something we used earlier in this course, so it stands in for an arbitrary sensor you might want to read. The Arduino will run in a loop that waits for a command on the serial line. Once a command is sensed, that command is executed, and the result, if any, is communicated back to the computer. A command frame consists of three parts:

1. First, the 8-bit char. The command is an alphabetic, one letter command.

2. Optionally, a value for the Arduino to process. It can be variable length.

3. Finally, a non-alphanumeric character is detected, indicating the end of the command.

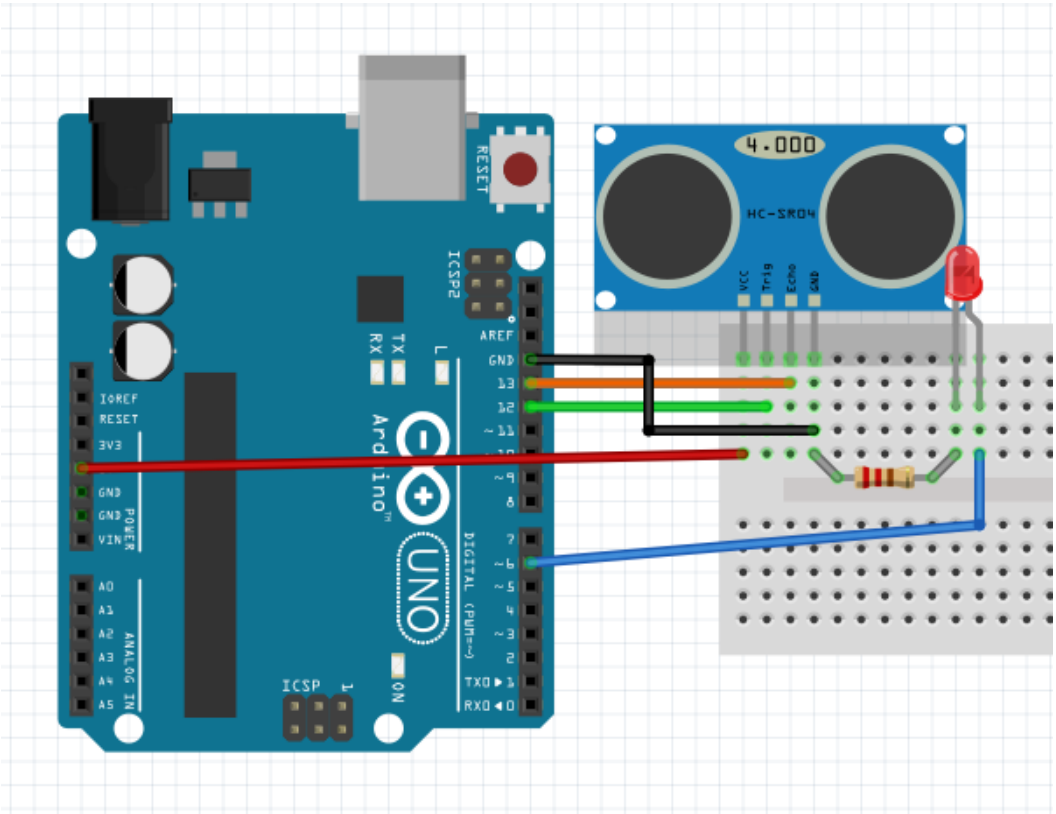
For demonstration purposes, our system will perform simple tasks. On the computer side, we will write data to a CSV file instead of a database.

For this project you need:

- an Arduino Uno board (or any Arduino board)
- solderless breadboard
- jumper wires
- an LED
- 220 Ohm resistor (can be close in value)
- ultrasonic module

Note
View the Arduino language reference here:
https://www.arduino.cc/reference/en/

Build the circuit below:



Type or copy/paste this in the Arduino IDE:

```
/* Arduino as external I/O */
#include <Ultrasonic.h>
#define BUFFLEN 10
// 1
const int led = 6;
char inBuff[BUFFLEN];
int bufferIndex = 0;
char inChar;
char cmd = 0;   // first byte from command message
int flag = 0;   // indicates if done receiving command message

Ultrasonic sensor(12, 13, 30000UL);

void setup() {
  Serial.begin(9600);
  memset(inBuff, 0, sizeof(inBuff));
```

```
}
// 2
void loop() {
    if (Serial.available() > 0) {
      inChar = Serial.read();
      if (isAlphaNumeric(inChar)) {
        if (bufferIndex == 0) {
          cmd = inChar;
        }
        inBuff[bufferIndex] = inChar;
        bufferIndex += 1;
      } else {
        flag = 1;
        bufferIndex = 0;
      }
    }
// 3
   if (flag) {
     switch(cmd){
       case 'l':
         changeLED();
         break;
       case 'u':
         readUltrasonic();
         break;
       default:
         break;
    }

    memset(inBuff, 0, sizeof(inBuff));
    cmd = 0;
    flag = 0;
   }
}
// 4
void changeLED() {
  int value = atoi((inBuff + 1));
  Serial.println(value);
  analogWrite(led, value);
}
```

```
void readUltrasonic() {
  Serial.println(sensor.read(INC));
}
```

Explanation of code

1. This sketch starts off similarly to the `stepper` mini lab with setting up variables for serial communication. This time is a bit different, because each message consists of a command character and an optional value. The optional value could be a number or string. In this lab we only use the optional value as an integer.

2. Each loop, if there is a byte in the serial input buffer, it is read and stored to a buffer. The first byte read is assumed to be a letter that corresponds to a command. We are using the letters ‘l’ and ‘u’ to control the LED and read the ultrasonic sensor.

3. If a finished message flag is detected, process the command. If the command is not valid, nothing happens. Then the input buffer, command char, and flag are reset.

4. Functions handle setting the PWM driving the LED and reading the ultrasonic sensor.

The above code can be tested using the Serial Monitor. Send a ‘u’ to get the ultrasonic measurement or ‘l’ and a number from 0 to 255 to set the LED brightness.

Now let’s work on the second part of this project, a Python script that runs on a computer. If you do not already have Python 3 installed, go to ***www.python.org***, and install the latest version of Python 3. Next you need to install Pyserial. Using the terminal in Mac OS or Linux and the command prompt in Windows, type and run:

python3 -m pip install pyserial

See ***https://pyserial.readthedocs.io/en/latest/pyserial.html#installation*** if there are any installation issues.

Once you have installed Pyserial, open IDLE, the Python shell, and type:

import serial

If there is no error, Pyserial has install properly. Next we need to find the serial port your Arduino board is using. You can either check in Arduino IDE or by running a command line too. In Arduino IDE, go to Tools→Port, and note what the port is. Alternatively in your system's terminal or command shell, run:

```
python3 -m serial.tools.list_ports
```

```
Your serial port will look something like "/dev/ttyACM0" in Mac
OS or Linux and "COM1" in Windows. Make sure the Arduino Serial
Monitor is not still connected before running the code below.
```

Next, in IDLE, click File and New File. Type or copy/paste the following code. Take care to keep the indentations consistent, since Python cares about spacing.

```
// 1
import serial
import time
import csv
import random
import threading

exit = threading.Event()
```

```
ser = serial.Serial('/dev/ttyACM0', 9600) # substitute your serial port
// 2
def setLED():
    rand = random.randint(0, 255)
    print('setting LED to: ' + str(rand))
    msg = bytes('l' + str(rand) + '\n', 'utf-8')
    ser.write(msg)
// 3
def getUltrasonic():
    ser.write(b'u\n')
    distance = str(ser.readline(), 'utf-8').rstrip('\r\n')
    print('ultrasonic sensor value in inches: ' + distance)

    with open('demo.csv', 'a', newline='') as f:
        writer = csv.writer(f)
        writer.writerow([time.ctime(), distance])

def quit(signo, _frame):
    exit.set()

def main():
    print('Press ctl-c to stop')
// 4
    while not exit.is_set():
        setLED()
        exit.wait(5)
        getUltrasonic()
        exit.wait(5)


    print("Wrapping up")
    ser.close()
// 5
if __name__ == '__main__':
    import signal
    for sig in ('TERM', 'HUP', 'INT'):
        signal.signal(getattr(signal, 'SIG'+sig), quit);

    main()
```

Explanation of code

1. First we include all the necessary libraries. We use the threading library to handle time delays and exit from a while loop. Explaining threads is beyond the scope of this course, but threads make the code easier to use. Then we create a serial port object with the name of the port and the connection speed. Substitute your serial port into the serial object constructor.

2. This function controls the LED. It generates a random number, and then a string is formed by combining the letter ‘l’, the random number, and a newline, ‘\n’. The string is then converted to an array of bytes, as the serial library only transmits and receives the byte type.

3. This function queries the ultrasonic sensor. We send a command ‘u\n’, and read bytes that we convert to a string. The readline function appends a carriage return and newline, so we strip those off the string with rstrip. Next we open a file in append mode and create a csv writer from the file handle. The writer then can append a row containing the date and sensor value.

4. The quit function and exit.is_set have to do with the threading library and how the program handles interrupts. We use a keyboard interrupt to get out of a for loop. The for loop calls setLED and getUltrasonic, each separated by a 5 second delay. Press control-c to exit this loop. Using threading for the delays instead of timer delays allows us to exit from the while loop quicker.

5. The final part of this program is a Python pattern. If the script, or module, is run by itself, then the code in the if statement runs. If the module is imported by another module, the code in the if statement does not run. This lets a module run as a standalone program or as part of another program.

Conclusion

In this course we have explored interfacing several common hardware components with Arduino. In the second part of the course, we made a practical project that could incorporate any of the hardware we covered.

Thank you for taking this course. I hope you got some value from it. Feel free to email any feedback you have.