An Arduino Reference for Experimental Psychologists

Notes from a Workshop

Michael Perone
Department of Psychology
West Virginia University

Edition

First edition, created April 30, 2017

Author's Contact Information

Michael Perone
Department of Psychology
West Virginia University
53 Campus Drive
Morgantown, WV 26506-6040
Michael.Perone@mail.wvu.edu

Acknowledgements

Descriptions of, and advice about, Arduino coding techniques, and development of the sketches herein, were informed and guided by material at www.arduino.cc and the *Arduino Programming Notebook*, August 2007 edition, by Brian W. Edwards.

The circuit drawings were prepared with Fritzing software available at www.fritzing.org.

The website https://arduino-info.wikispaces.com was a valuable resource, particularly in preparing the circuits and sketches involving the stepper motor and the infrared remote control.

A few illustrations were adapted from open-source materials designed by Linz Craig, Nick Poole, Prashanta Aryal, Theo Simpson, Tai Johnson, and Eli Santistevan of Sparkfun Electronics.

The physical computing schematic in Part 1 was contributed to the public domain by Nevit Dilmen (Own work) [CC0], via Wikimedia Commons (https://commons.wikimedia.org/wiki/File%3APhysical_computing.svg).

The "Anatomy of a C function" in Part 4 is from https://www.arduino.cc/en/Reference/FunctionDeclaration.

The resistor decoder in the Quick Reference is by Bret Victor, http://worrydream.com/ResistorDecoder/.

License

Contents

Introduction 5

Part 1: Physical Computing & Basic Prototyping Materials

The Arduino Uno Microcontroller 6
Physical Computing 8
Building Prototypes with an Arduino and a Breadboard 10

Part 2: Sensors & Actuators

Buttons 11
Potentiometers and Their Kin 11
Advanced Sensors 12
Light Emitting Diodes (LEDs) 13
Piezo Speakers 13
Liquid Crystal Display (LCD) Panels 13
Motors 14
Personal Computers 15

Part 3: Circuit Elements

Resistors 16
NPN Transistors 16
Optcouplers 17
Relays 17

Part 4: Programming Elements

Arduino Integrated Development Environment (IDE) 19
Arduino Libraries 19
Basic Structure of an Arduino Sketch 20
Variables and Constants 21
Array Variables 22
Arithmetic Operators 22
Comparison Operators 23
Boolean Operators 23
Functions 24

Part 5: Common Programming Tasks

Communication through the Serial Port 26
Timing 28
Digital Input 30
Digital Output 34
Analog Input 35
Analog Output 36
Branching 38
Looping 40
Doing Math 40
Making Sounds 43

Part 6: Sample Sketches & Circuits

Blink *(with circuit drawing*) 45
Reading Serial Strings 46
Reading Serial Strings as Parameters *(with circuit drawing)* 48
Count Button Presses *(with circuit drawing)* 50
Count Button Presses Debounced 52
Adjustable Tone *(with circuit drawing)* 54
LCD Hello World *(with circuit drawing)* 56
LCD Recycling Hello World 60
Thermistor 62
Joystick Simple 63
Joystick Refined 64
Joystick RGB LED 66
Joystick Ultrasonic RGB LED 68
Servo Sweep 72
Photocell 73
Photocell Response Count 74
Two Buttons 76
Stepper Sweep 78
Stepper by Steps 79
Stepper by Degrees 80
Temperature Humidity Monitor 82
Remote Signal Reception 83
Remote Signal Decoding Elegoo 84
Transistor to Relay *(with circuit drawing)* 86
Optocoupler Test (*with circuit drawing)* 88
Analog IO with PWM *(with circuit drawing)* 90

Part 7: Exercises 92

Resources 94

Appendices

Overview 96
Elegoo Uno Project Super Starter Kit 97
Escobar & Perez-Herrera (2015) 98
Stepper 360 Dial 107

Quick Reference 109

Introduction

This document pulls together, in what I hope is a handy format, some topics explored in a "microcontroller workshop" in the WVU Department of Psychology in the spring semester of 2017. This is not intended to be a comprehensive treatment of anything in particular, but rather a compendium of information that I think is interesting or useful. My conception of what is "interesting" or "useful" is that of an experimental psychologist who sees microcontroller technology as a means of supporting basic research. Your individual needs may lead you to a different opinion. Still, I hope there is enough overlap between us to make this material helpful to you.

The workshop was largely a show-and-tell: Each participant bought a kit with an Arduino Uno microcontroller development board, a breadboard and jumper wires, and a variety of devices for detecting events in the world or making events happen in the world – this being the essence of physical computing. Each week we assembled a few circuits with some of the devices and paired each circuit with an Arduino "sketch" (program) to make it go. The idea was to illustrate how you can handle inputs and control outputs with relatively straightforward code and simple circuits.

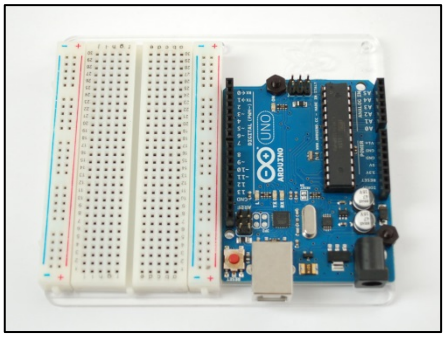
We used the Elegoo Uno Project Super Starter Kit, and we fooled around with these devices:

- *For input:* buttons, potentiometer, joystick, photoresistor (a.k.a photocell), thermistor, DHT11 temperature and humidity module, infrared receiver and handheld remote control

- *For output:* light emitting diodes (LEDs), liquid crystal display (LCD) panel, servo motor, stepper motor, active buzzer, passive buzzer

- *Circuit elements:* resistors, NPN transistor, relay, 4N25 optocoupler (the last was my addition; it was not included in the kit), and – of course – the Arduino Uno, breadboard, and jumper wires.

Some weeks into the semester, we recognized that our circuits were suffering from loose connections, partly because the Arduino and breadboard were separate parts that could move and put strain on the jumper wires, and partly because the Elegoo breadboard was, well, maybe not of the highest quality. We made improvements by using an Adafruit mounting plate to hold the Arduino firmly aligned with a half-size breadboard.

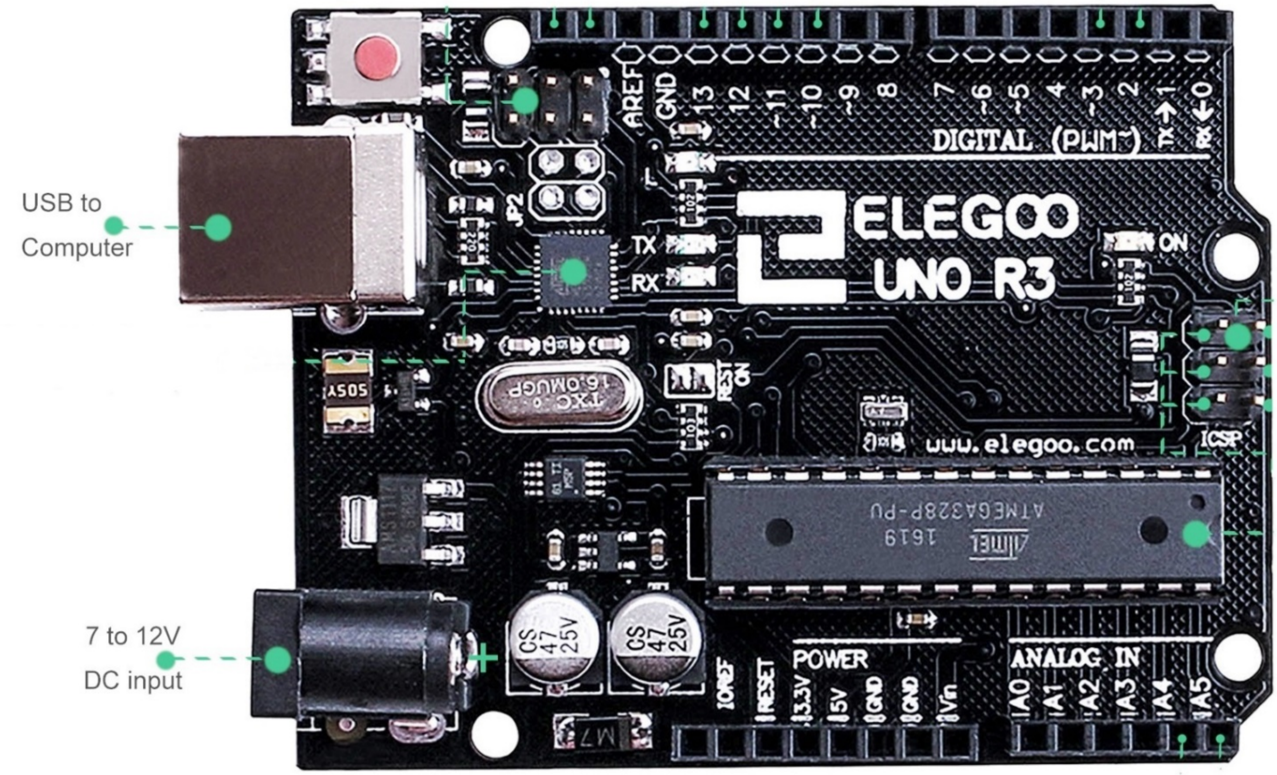


This *Reference* is organized into seven sections as described in the table of contents. Parts 1 through 3 provide simple (one might say "simplistic" – but give me some slack, I'm a psychologist, not an electrical engineer) descriptions of the hardware used in the workshop. Part 4 describes essential elements of Arduino programming. This material is not comprehensive by any stretch of the imagination, but it does cover what you need to know to get started on some serviceable coding projects. Part 5 describes some ways to accomplish common tasks; these ways are not the only ways – they may not even be the best ways – but they are fairly easy to understand and they work. Part 6 reproduces some of the sketches and circuits we used in the workshop in case it may be useful to have them at hand. Part 7 offers some exercises to give you practice in coding.

You can find the official reference for Arduino programming at www.arduino.cc/en/Reference/HomePage. And you can find an excellent set of tutorials at www.arduino.cc/en/Tutorial/HomePage. When you have a specific problem to solve, a bit of Googling can be a time-saver: The members of the large Arduino programming community have a wide range of interests and experience, and they are generous in sharing their knowledge.

Part 1: Physical Computing & Basic Prototyping Materials

The Arduino Uno Microcontroller

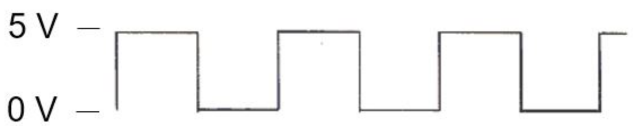


Here, shown larger than life, is the Elegoo company's version of the venerable Arduino Uno Revision 3 (hereafter, "the Arduino"). This is an essential part our physical computing tool kit for building prototypes of circuits that can sense and control environmental events.

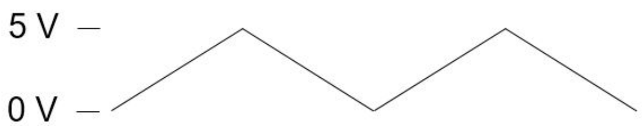
The Arduino can be powered through a USB cable attached to a personal computer (PC). It also has a power jack so it can be powered with a 9V battery or a plug-in power supply that provides between 7V and 12V. Regardless of the power source, the Arduino will regulate the voltage downward: It operates at 5V.

Along the top of the Arduino (as it is oriented in this photo) are 14 digital pins numbered 0 through 13. These can be configured as *inputs* to sense discrete (digital) events such as button presses, or as *outputs* to make things happen by turning devices on and off. Pins 0 and 1 are used when the Arduino communicates with the PC through the USB cable; to avoid interference, we won't use them. Note that six of the pins are marked with dashes: 3, 5, 6, 9, 10, and 11. These pins are capable of *pulse width modulation* (PWM) which is a method for simulating analog output using digital pins. Digital devices have only two states on and off or, in electronic terms, HIGH (5 volts for our Arduino) or LOW (0 volts). The state of an analog device is continuously variable. Pulse width modulation simulates an analog output signal by alternating between HIGH and LOW at frequencies established by the programmer. More information about PWM is in Part 5 of this *Reference*.

At the bottom right of the Arduino are six analog input pins numbered A0 through A5. Whereas a digital input can detect only HIGH and LOW states (5V and 0V), an analog pin can detect a wide range of states anywhere between 5V and 0V. More information about analog input is in Part 5. See also the *Quick Reference* section, which includes a table summarizing the functions of all 20 of the input/output pins.



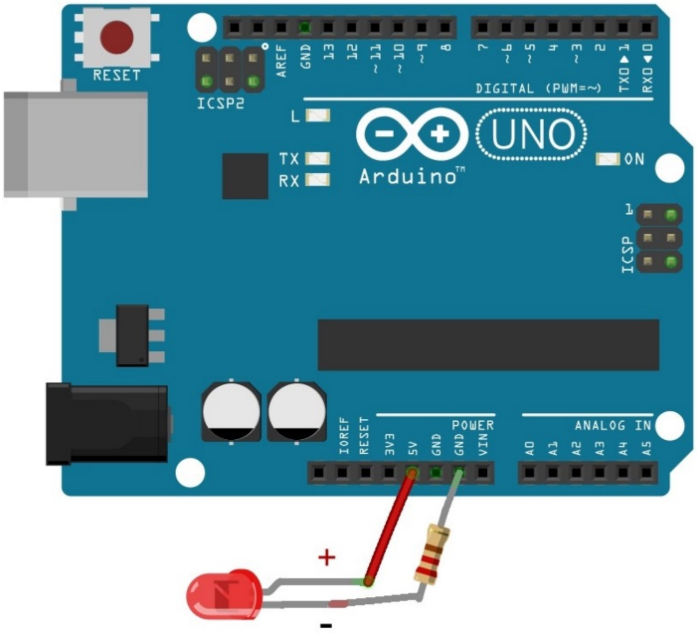
Digital Signal

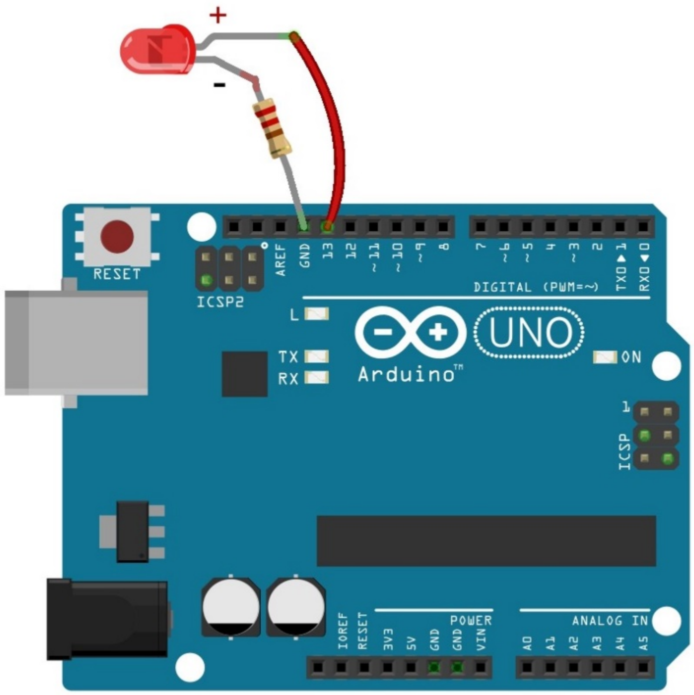


Analog Signal

Also along the bottom of the Arduino are several power-related pins. Our interest is in the pins labeled 5V and GND (ground, 0V). The 5V pin is akin to the positive terminal of a battery, and the GND pin is akin to the negative terminal. There's another GND pin on the top of the board, next to Digital Pin 13. Electric current flows when an appropriate device completes a circuit between 5V and GND. The circuit illustrated at right turns on a light emitting diode (LED). The positive lead of the LED is connected to 5V and the negative lead is attached to GND via a resistor. (Even though the Arduino's power is modest, it would burn out the LED quickly without that resistor.)

In this circuit, the Arduino is not doing anything interesting; it is just being used to power the LED. We are treating the Arduino as if it were a battery. Of course we want the Arduino to do more, such as turning things on and off under the control of a program.



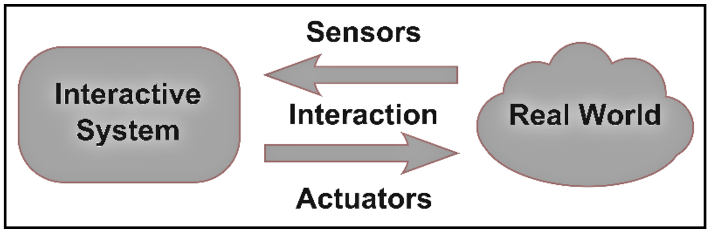


The Arduino plays a more interesting role in the second circuit, to the left. As before, the negative lead of the LED is connected to GND through a resistor. The positive lead, however, is connected to Digital Pin 13. With this arrangement, we can program the Arduino to use the pin to output current to the LED, turning the LED on by setting the pin to HIGH and turning it off by setting the pin to LOW.

Programs that run on the Arduino are called "sketches." Sketches are written on the PC in a variant of the C++ programming language and uploaded to the Arduino through the USB cable. The sketch will run continuously whenever the Arduino is powered. To stop the sketch, you can disconnect the Arduino's power source (e.g., by unplugging the USB cable) or press the reset button mounted on the upper left corner of the board. The reset button only stops the sketch momentarily: After about a second, it starts over.

Information and advice about writing Arduino code can be found in Parts 4 and 5, and over two dozen of the sketches prepared for the workshop can be found in Part 6 (complete with embarrassing typographical errors in the comments and awkward code here and there). Sketches can be written using the Arduino Integrated Development Environment described in Part 4. It is available for free at www.arduino.cc.

Physical Computing

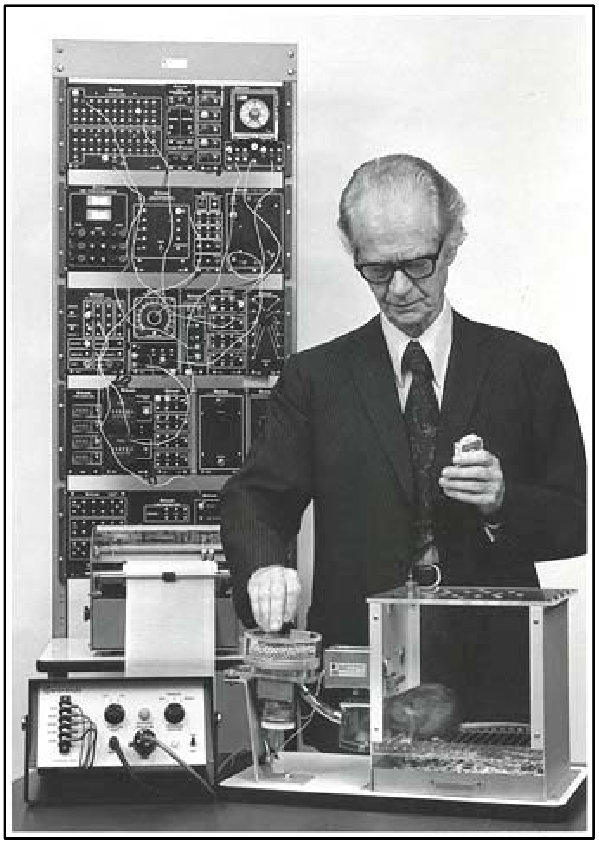


The term *physical computing* is applied to arrangements of computer hardware and software that interact with the environment by detecting events and making things happen. The diagram shows the essential parts. In our case, the "interactive system" is the Arduino microcontroller and the sketch that is running on it. The "real world" is that part of the environment that can affect the Arduino or be affected by it. *Sensors* are devices that provide a way for the environment to affect the Arduino through its input pins. *Actuators* are devices that provide a way for the Arduino to affect the environment though its output pins or serial communications port. Electronic circuits connect the sensors and actuators to the Arduino's input and output pins, and the Arduino's software (the sketch) determines how the Arduino will interpret information from the sensors and control the actuators.

Physical computing is ubiquitous in modern life. Consider your microwave oven. Its embedded microcontroller interacts with the world by receiving your commands through a keypad. By pressing a few keys on the outside of the oven, you can tell it how long to cook and at what power level. The microcontroller interprets your key presses and activates devices that, for example, generate radio waves that heat your food and start the motor that turns the food in circles. Your oven probably has some internal sensors, too, that transmit information to the microcontroller about the state of the food you are cooking. For example, a sensor might measure the temperature of the food and the microcontroller's software may respond by turning off the radio waves and carousel motor when the target temperature is reached.

As a physical computing system, the microwave oven interacts with a pretty small part of the world. Other systems are more expansive. Home security systems employ a wide range of sensors to detect motion, the sound of breaking glass, the concentration of carbon monoxide, smoke, water in places where it shouldn't be, and whether specific doors and windows are open. The systems can respond to the information from these sensors by texting the homeowner, sounding an alarm, or calling the police or fire department.

Physical computing is common in basic experimental psychology. Researchers in the experimental analysis of behavior could be said to have gotten an early start. A precursor of today's systems came into wide use in the second half of the twentieth century thanks to B. F. Skinner and Ralph Gerbrands of Harvard University. They developed a system of controlling events in behavioral test chambers and recording the responses of animals using circuits that combined electromechanical switches, steppers, relays, counters, motors, and timers. The system had sensors – lever switches to detect behavior – and actuators – lamps, speakers, and food dispensers for delivering stimuli. There were no microcontrollers to control this stuff; instead, the systems used rather sophisticated electromechanical circuits such as the one behind B. F. Skinner in the photo at right.



(To learn more about this period in the history of experimental psychology, visit the Behavioral Apparatus Virtual Museum curated by Kennon A. Lattal at aubreydaniels.com/institute/museum.)

Electromechanical control circuits were eventually replaced by computer technology. First came minicomputers which, by today's standards, weren't all that "mini." When fully equipped with the components needed to store programs and data – components such as a paper tape puncher/reader (surprisingly common in the 1970s because they provided cheap storage) or disk drives (expensive) – a minicomputer might stand six feet tall and anywhere from about two to five feet wide.

The first mass-produced minicomputer was the PDP-8 from the Digital Equipment Corporation. ("PDP" was short for "Programmable Data Processor.") It served as the hardware platform for a programming language developed specifically to control behavioral experiments and collect data. The language, "SKED" (and later, "Super SKED" and "SKED-11") was designed by Arthur Snapper at Western Michigan University, implemented on the PDP-8 (and later the PDP-11), and widely used for about 20 years starting in the early 1970's.

In the late 1980's, Thomas Tatham, a behavior analyst trained at Temple University, developed a variant of SKED for the desktop microcomputers that had become popular. The language, originally called Med State Notation, was first sold by Med Associates, Inc. in 1987 along with a system of hardware modules and cables to link the computer to behavioral test chambers (the company also sold the chambers). The language, renamed MED-PC, has been upgraded several times. Today, Med's software and hardware, in conjunction with a PC, is probably the most widely used physical computing system in experimental psychology.

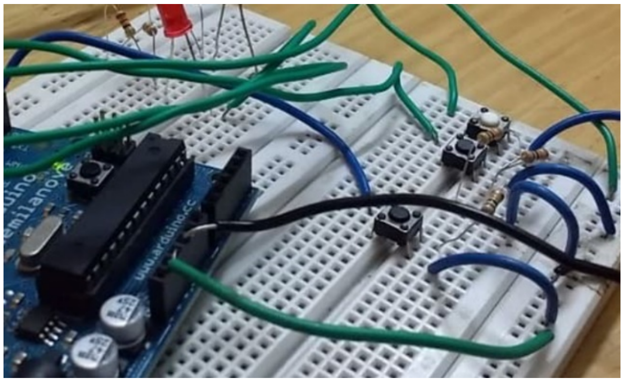
The Med system has a significant limitation: it costs a lot of money. A system based on the Arduino can be made without much money, but it requires substantial technical knowledge plus the time and inclination to tinker and build. The purpose of the microcontroller workshop is to introduce young behavior analysts to physical computing with the Arduino, in the hope that some of them might be inspired to tinker and build.

Rogelio Escobar, a professor of psychology at the National Autonomous University of Mexico, has achieved a high degree of technical sophistication in the development of electronic equipment for experimental control and behavioral recording. He has done magnificent work in creating physical computing systems for the study of operant behavior, and he generously shares his work – in both hardware and software – on his web site, http://analisisdelaconducta.net/. His work is not restricted to physical computing; he also has designed the experimental environments in which behavior take place: the test chamber. His website includes files you can use to build rat chambers using a 3D printer!

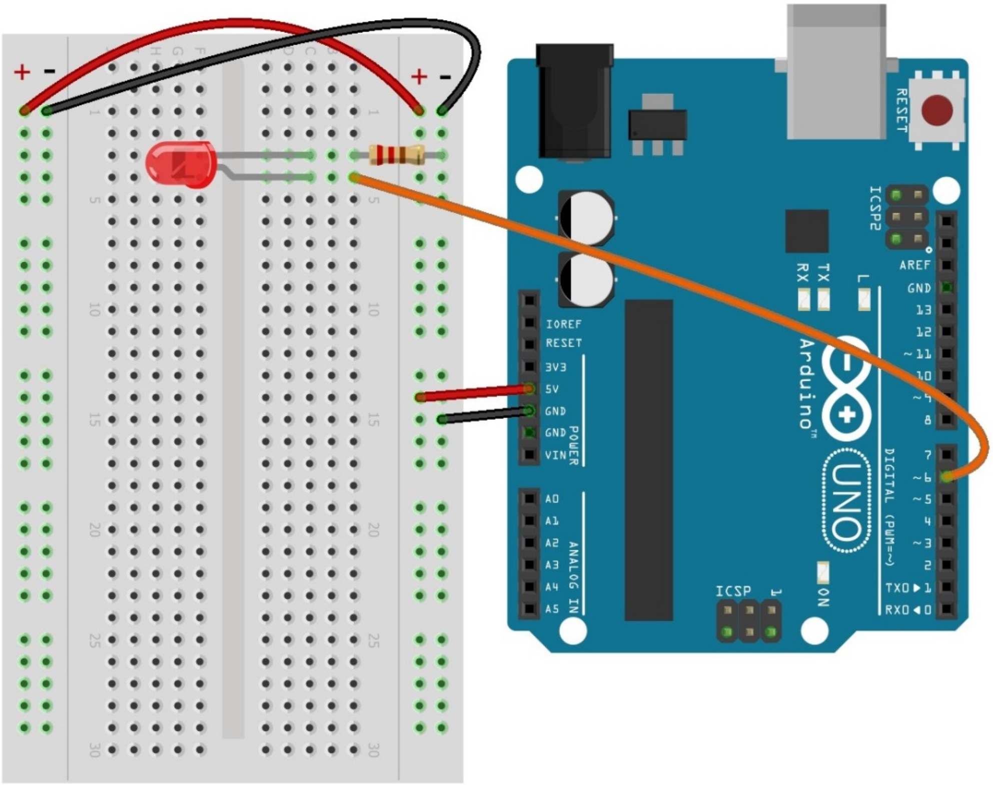
An article by Dr. Escobar and his student Carlos A. Perez-Herrera, published in the *Journal of the Experimental Analysis Behavior*, is included as an appendix to this *Reference*. The article describes a physical computing system that uses an Arduino to interface behavioral test chambers with a PC running a Visual Basic program.

Building Prototypes with an Arduino and a Breadboard

Our Arduino is intended for building prototypes of physical computing systems. Header strips mounted on the edges of the board allow you to connect wires simply by pushing one end into the header. Circuits with sensors and actuators are built on a “breadboard” that has rows of tiny sockets that likewise allow you to insert buttons, LEDs, resistors, etc., by pushing their leads into the sockets and to connect them to one another by pushing wires into the sockets. Building circuits this way is relatively quick and easy – no soldering – but the resulting product is fragile. If you need a durable version of your circuit, you will need to solder the microcontroller and various circuit elements together – a task beyond the scope of our little workshop.



A typical breadboard, shown below alongside an Arduino, has four sections. The sections on each edge have two vertical columns of sockets. The sockets in each column are connected to one another. This means that a wire inserted into any socket within a column is electrically connected to the wires inserted into other sockets within the same column. These sections are called “power rails” because they normally are used as a convenient way to get GND and 5V to the parts of a circuit. As in the illustration, a wire is run from a GND pin on the Arduino to one of the sockets in the column labeled “—.” Now all the sockets in that column are connected to GND. Another wire is run from a 5V pin on the Arduino to one of the sockets in the “+” column, so that the sockets in that column are connected to positive.



The two middle sections are organized into horizontal rows. Within each section, the five sockets in each row are connected. This allows you connect various components without solder. Consider the red LED in the illustration. One of its leads is connected to a resistor which itself is connected to GND. The other lead is connected to a wire that is connected to Pin 6 of the Arduino.

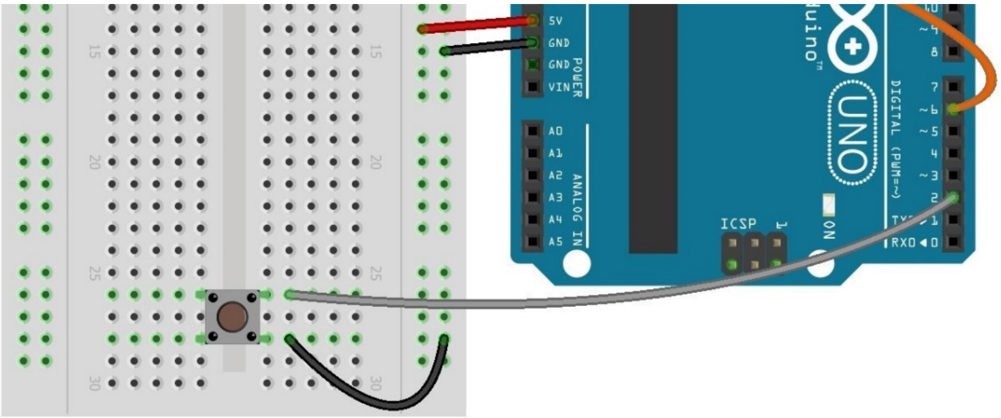
As you build circuits on your breadboard, remember: The sockets on the power rails are connected *vertically for the full length of the board*. The sockets in the middle sections are connected *horizontally in sets of five*.

Part 2: Sensors & Actuators

Buttons

As already noted, digital inputs have two states, LOW and HIGH, corresponding to 0V (i.e., GND) and 5V. If you are building a prototype circuit, the simplest way to figure digital input into the design is with an electrical switch, commonly in the form of a push button. In our workshop, we use a common button like the one shown here. It has two sets of leads, each pair constituting the end points of a switch. Pressing the button connects the two leads. If the switch is in a properly designed circuit, pressing the button causes electrical current to flow across the leads.

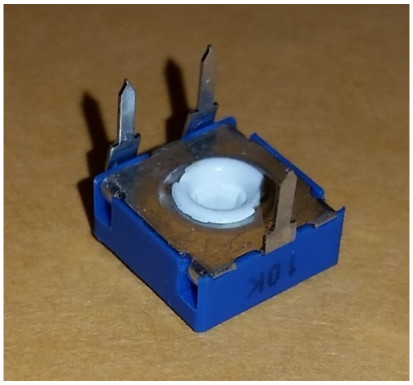
Although there are two switches in this device, there is only one button and pressing it operates the left switch and the right switch simultaneously. The button is designed to be mounted across the gap running down the middle of your breadboard. This keeps the left switch and the right switch from interfering with one another.

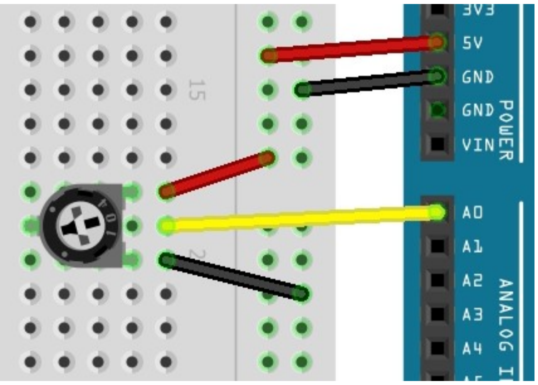


The illustration at right shows the kind of circuit to use with buttons in our workshop. One lead of the button's right-side switch is connected to GND through the power rail section of the breadboard. The other lead is connected to Pin 2 of the Arduino. Pressing the button closes the switch between these two leads, sending GND to the Arduino. More information about how the Arduino's pins handle input from buttons can be found in Part 5 ("Common Programming Tasks") in the section labeled "Digital Input."

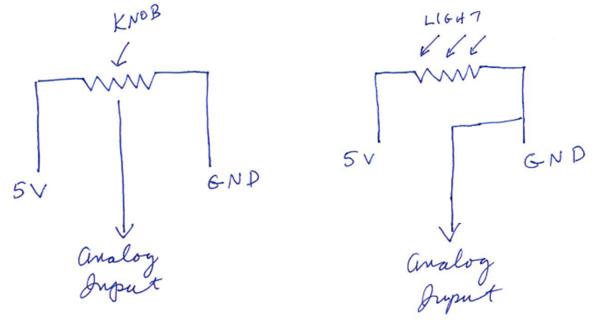
Potentiometers and Their Kin

A *potentiometer* – "pot" for short – is a variable resistor. At bottom left is a photo the underside of a common pot that fits into a breadboard. It has three leads. The "side" leads are the ones in the back of the photo and the "middle" lead is the one in front. On the top of the pot is a screw slot or a knob that can be turned. This varies the resistance between the side leads and the middle lead. In the breadboard illustration, one side lead is connected to 5V and the other to GND. The middle lead is connected to one of the Arduino's analog inputs. As the pot is turned, the voltage on the middle pin is varied from 0V to 5V.





The pot is a great model for a variety of analog sensors. In our workshop we work with *photoresistors* and *thermistors.* These work on the same principle as the pot; the key difference is that resistance is varied not by turning a knob but rather by changes in light or temperature. When our photoresistor is wired as shown here, there is a direct relation between the intensity of the light falling on it and the voltage sent to the Arduino. With our thermistor wired in the same way, there is a direct relation between the temperature near its surface and the voltage sent to the Arduino.



Potentiometer (left) and photoresistor

We also played with a joystick. This device is just a pair of potentiometers. The resistance of one is changed by moving the stick vertically; the resistance of the other is changed by moving the stick horizontally.

Advanced Sensors

We played with some more sophisticated sensors:

- *DHT11* to measure temperature and humidity
- *HC-SR04 Ultrasonic Ranging Monitor* to measure distance
- *Infrared remote control* and *IR detector/demodulator* to receive inputs from a handheld remote control

These devices are easy to connect to the Arduino; see the circuit notes in these sketches in Part 6:

- DHT11: Temperature Humidity Monitor
- HC-SR04: Joystick Ultrasonic RGB LED
- Infrared remote: Remote Signal Reception & Remote Signal Decoding Elegoo

The devices area also surprisingly easy to use within a sketch because each is supported with special libraries that give you access to functions to program the Ardunio to interact with the devices and convert the input into meaningful forms (e.g., measures of temperature and humidity information) with little or no calculation. (I'll say a bit more about Arduino libraries in Part 4.) Here are the libraries we used for these devices:

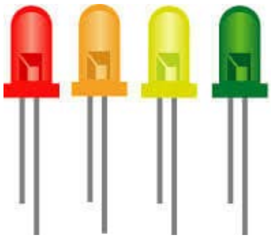
- DHT11: SimpleDHT
- HC-SR04: NewPing
- Infrared remote: IRremote

For more information about these devices, see:

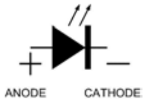
- DHT11: https://learn.adafruit.com/dht/
- HC-SR04: https://www.cytron.com.my/p-sn-hc-sr04
- Infrared remote: https://arduino-info.wikispaces.com/IR-RemoteControl

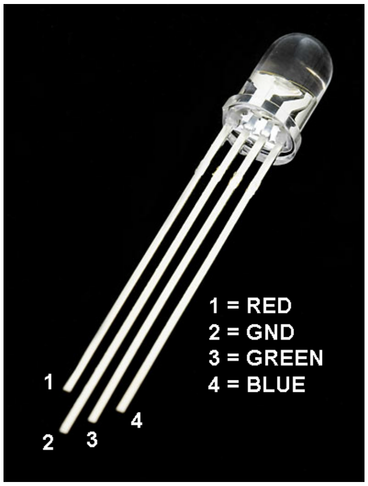
Light Emitting Diodes (LEDs)

LEDs are highly efficient lamps and put out a bright light with relatively little electrical current. They have two leads. The longer lead is positive (or “anode) side of the circuit and the shorter lead is the negative (or cathode). The output of an Arduino digital pin will overpower a standard LED, so a resistor must be added in series to reduce the current. The resistor will dim the LED as well. If you want a bright light, use a 220-ohm resistor. If you want something dimmer, try stronger resisters until you find something that suits your needs.



In my circuits – in our workshop as in the illustrations in this Reference – the LED’s negative lead is connected to GND through a 220-ohm resistor, and normally an Arduino output pin is attached to the positive lead. Setting the pin to HIGH lights the LED. You may encounter circuits that put the resistor on the other side – the negative lead is connected directly to GND and the positive lead is connected to the Arduino through the resistor. Both arrangements accomplish the same results.

In schematic circuit diagrams, an LED is shown by this symbol: 



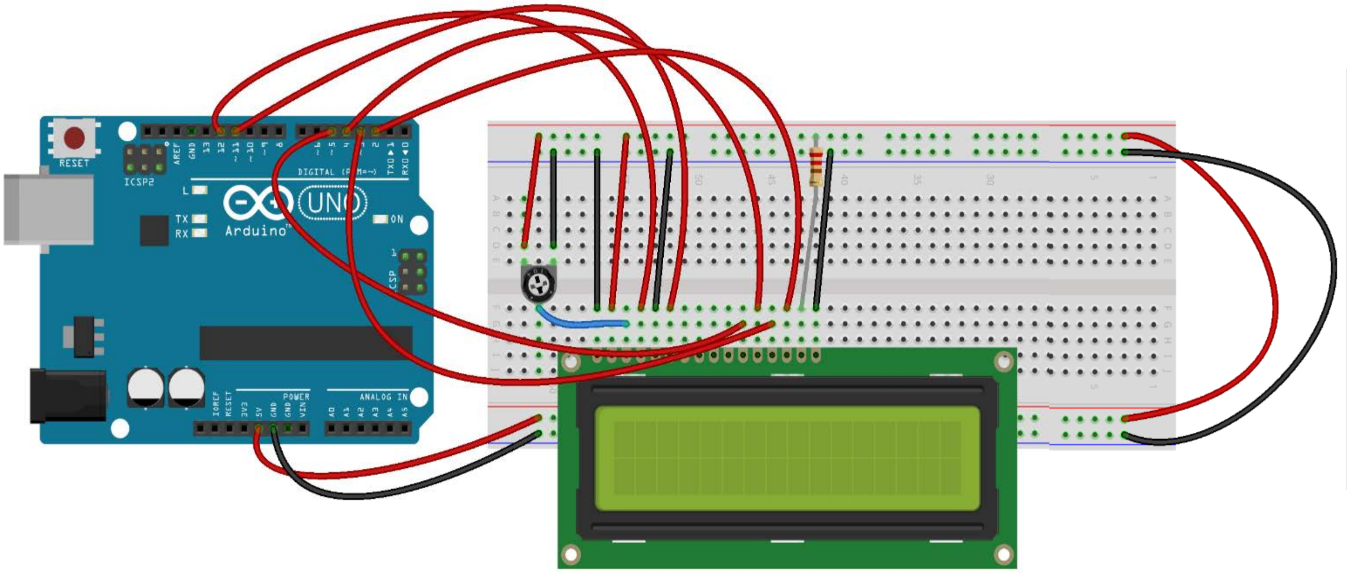
Our workshop also uses a special LED: the “RGB LED. “ This is essentially a combination of three LEDs in a single package with a common GND lead. It’s fun to use because by varying the signal to the Red, Green and Blue leads, you can create light of any color. The signal can be varied by adjusting the voltage or by using the analogWrite() function discussed in Part 5.

Piezo Speakers

We can produce sound using the *piezo speaker* in our Elegoo kit. This simple device can be operated directly by an Arduino digital output, but it produces sound at low volume. The speaker has two leads. The one marked as positive (see the + sign with a circle around it in the photo) is connected to the Arduino output pin; the other is connected GND. The Arduino’s tone() function sends the signals to the speaker to produce the sound. To make louder sounds with the Arduino, the signal can be sent to an amplifier and then on to a proper 4- or 8-ohm speaker. Low-cost amplifiers can be found at various sources including www.adafruit.com.

Liquid Crystal Display (LCD) Panels

Our Elegoo kit included a 2 x 16 LCD Display. This device can display 2 lines of 16 characters each. LCDs are easy to program, but they require a lot of wiring. If your goal is to provide your sketch with a means of communication – for example, to show the values of variables as the sketch runs – then it will be much easier to use the serial monitor that is included in the Arduino IDE (see Part 5 for information about how to use the serial monitor). If your goal is to create a self-contained device that displays information – for example, temperature, humidity, counts of behavior – then an LCD may be a good choice.



LCDs have a parallel interface: The Arduino has to manipulate several pins on the LCD at once to control the display. The interface consists of these pins:

- A *register select (RS) pin* that controls where in the LCD's memory data are written. You can select either the data register, which holds what goes on the screen, or an instruction register, which is where the LCD's controller looks for instructions on what to do next.
- A *Read/Write (R/W)* pin that selects reading mode or writing mode
- An *Enable pin* that enables writing to the registers
- *8 data pins (D0 -D7).* The states of these pins (HIGH or LOW) are the bits that you're writing to a register when you write, or the values you're reading when` you read.
- There's also a display contrast pin *(Vo),* power supply pins (*+5V* and *Gnd*) and LED Backlight (*A* and *K* on the LCD in our kit) pins that power the LCD, control the display contrast, and turn on and off the LED backlight, respectively.

The process of controlling the display involves putting the data that form the image of what you want to display into the data registers, then putting instructions in the instruction register. The LiquidCrystal library simplifies this for you so you don't need to know the low-level instructions. The library allows you to control LCDs that are compatible with the Hitachi HD44780 driver, and our LCD fits this description.

Hitachi-compatible LCDs can be controlled in two modes: 4-bit or 8-bit. The 4-bit mode requires fewer I/O pins from the Arduino. For displaying text on the screen, you can do most everything in 4-bit mode. The sample sketches in Part 6 show how to control a 2x16 LCD in 4-bit mode. Also included in the sketches is pin-by-pin instructions on how to wire the LCD’s interface to the Arduino. See “LCD Hello World” and “LCD Recycling Hello World.”

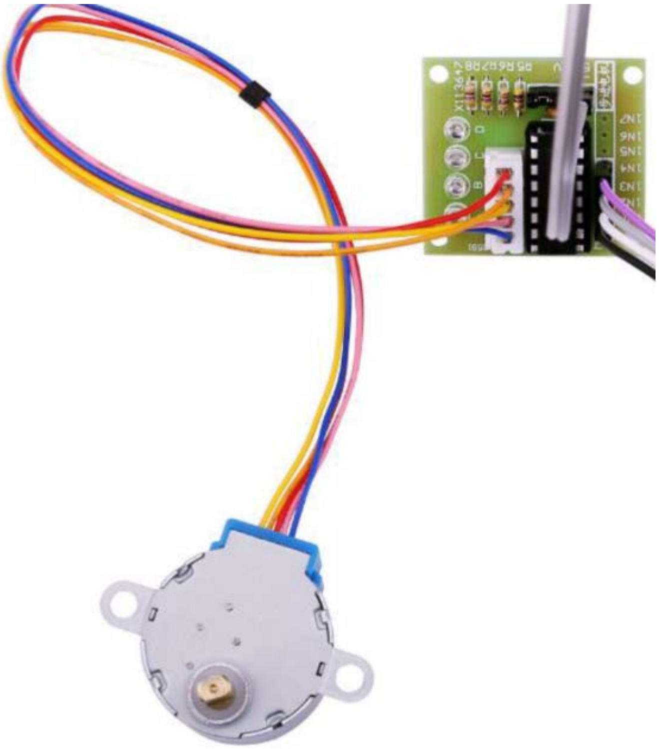
Motors

Our kit includes three motors: a conventional di*rect-current (DC) motor*, a *servo motor*, and a *stepper motor.* The leads on the DC motor were so flimsy – mine kept falling off and had to be re-soldered several times – that I decided not to use it in our workshop.

A conventional DC motor is designed to rotate a shaft continuously, with the speed of rotation proportional to the applied voltage. A servo motor is designed to rotate the shaft through a 180-degree arc, moving in either direction. A stepper motor can rotate the shaft through the entire 360 degrees. The key feature of servos and steppers is that you can exert fine control over the position of the shafts. They are used in systems with mechanical parts that must be moved precisely, such as copy machines, scanners, 3D computers, and robots.

Programming servos and steppers with the Arduino is straightforward because well-developed libraries provide easy-to-use functions. The libraries in the workshop are named, appropriately, “Servo” and “Stepper.”

The motors in our workshop are popular models, and you can find a lot of information about them online. Our servo is the Tower Pro Micro Servo SG90 and the stepper is the 28BYJ-48. A tutorial on the servo is at https://www.intorobotics.com/tutorial-how-to-control-the-tower-pro-sg90-servo-with-arduino-uno/. A guide to the stepper is at https://arduino-info.wikispaces.com/SmallSteppers.



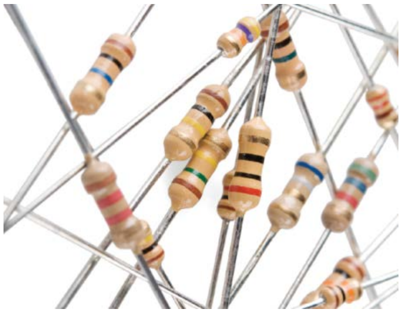
Information about connecting the servo and stepper to the Arduino can be found in these sketches in Part 6: “Servo Sweep,” ”Stepper Sweep,” “Stepper by Steps,” and “Stepper by Degrees.” The stepper came with a printed circuit board that interfaces it with the Arduino (see photo at left), so the connections are easy.

Personal Computers

It may seem odd to include the PC in this discussion, but in fact it can serve both as a sensor (input device) and as an actuator (output device). In our workshop the PC is used for both functions by communicating with the PC through the serial monitor built into the Arduino IDE. See Part 5 of this Reference for information on how to incorporate serial communication into an Arduino sketch and see Part 6 for many sketches that incorporate serial communication for *output* (e.g., “Count Button Presses”). Some sketches use the PC as an *input* device; see, in particular, “Reading Serial Strings” and “Reading Serial Strings as Parameters.” Finally, among the appendices is an article by Escobar and Perez-Herrera (2015) that describes how to use the Arduino to interface behavioral test chambers with a PC running a Visual Basic program.

Part 3:
Circuit Elements

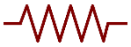
Resistors



Resistors are electronic components that limit the flow of electrons through a circuit. The amount of electrical resistance is measured in “ohms,” commonly symbolized with the Greek letter omega:

Ω

In schematic diagrams of circuits, a resistor is show by this symbol:



The resistance is designated by bands on the component. The first two bands represent the most significant digits of the resistor’s value. The third band is a multiplier. The last band is either gold or silver and indicates the tolerance (error) in the resistor: gold = 5%, silver = 10%. This bit of information also allows you to orient the resistor: The gold or silver band is the last. A handy chart to decode the bands of a resistor is included in the Quick Reference section.

Resistance is related to electric current and voltage in an orderly fashion. Think of electric current as the flow of electrons; it is measured in amperes (abbreviated “A”). Voltage is electric pressure, measured in volts. Ohm’s Law puts them all together:

R = V/I *or* **I = V/R** *or* **V = I x R**

where R = resistance, V = voltage, and I = amps . If you don’t like “I” as the expression for electrical current, blame the French: It stands for “intensité de courant,” the French term for current flow. (In case you’re wondering: Georg Ohm, for whom the law is named, was German.)

Ohm’s Law is helpful when you need to figure out how much resistance to put into a circuit to limit the flow of current. In our workshop, we generally power LEDs with 5V from the Arduino, but we add a 220-ohm resistor. So we limit the current to .023 amp. Plugging 5 and 220 into I=V/R gives us I = 5/220 = .023 amp or 23 milliamps (23 mA).

A nice tutorial on resistors is at https://learn.sparkfun.com/tutorials/resistors#power-rating

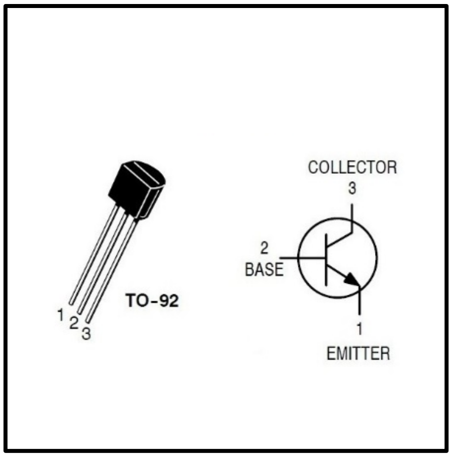
NPN Transistors

The NPN transistor is a common bi-polar junction transistor that can be used as an electronic switch. A very low current can be used to operate the transistor, and the transistor can switch higher currents. It might help to relate to this a switch you encounter every day: the light switch on the wall of your office. It takes almost no energy to operate the switch (you can flick it on or off with a finger) but the switch itself can handle a lot of energy – enough to turn on the overhead fluorescent lights and light up the entire room.

Our Arduino's power supply is limited: It can supply up to about 40 mA from each pin. That is not enough to drive some (many) actuators. The transistor comes to the rescue: The Arduino has enough juice to operate it. By turning the transistor on and off, the Arduino can control devices that require higher currents, just as you can control currents high enough to light a room by flicking a wall swtich.

Our Elegoo kit includes a widely used general-purpose transistor, the PN2222A. To operate it, positive voltage is to the "Base" pin. This creates a connection between the "Collector" and "Emitter" pins. We can use an Arduino digital output pin to operate the transistor – a very low current will be drawn – and let the higher current flow across the Collector and Emitter to operate the actuator that draws a lot of current.

In our workshop we use the transistor to control a relay that requires more juice than our Arduino output pins can provide. The sketch and circuit are in Part 6; see "Transistor to Relay."

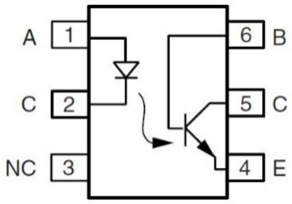


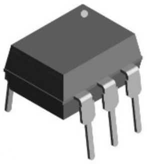
Our transistor. To identify the pins, orient the transistor so the flat side is facing you. Then, from left to right, are the 1. Emitter, 2. Base, and 3. Collector.

For a good tutorial bi-polar junction transistors, see https://learn.sparkfun.com/tutorials/transistors.

Optocouplers

An *optocoupler* – also known as an *optoisolator* – is a transistor that is operated by light rather than by the application of an electric current. This allows you to have two power sources communicate safely – when, for example, relatively high-voltage devices in the lab must be sensed by the low-voltage Ardunio. It can be used to protect the Arduino from high voltages.





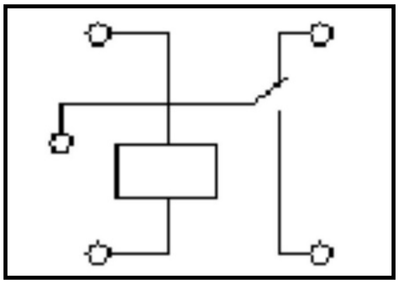
Here is the circuit diagram of the 4N25 optocoupler, a widely used model, alongside a drawing of the chip. This is the one in our workshop (it is not included in the Elegoo kit, however). To orient the chip, look for a little circle in one corner; this marks Pin 1. (It won't be as easy to see as the one in the drawing.) From there, the pins are numbered sequentially in counter-clockwise order. Passing current across Pins 1 and 2 lights an internal infrared LED. This causes the internal phototransistor to close the circuit between Pin 4 (the transistor's Emitter) and Pin 5 (the Collector) . The infrared LED can handle relatively high voltages (with an appropriately sized resistor in series with it), and the idea is to run the high-voltage output of lab devices across Pins 1 and 2. If we run the Arduino Uno's 5V current across Pins 4 and 5, we can send 5V back to the Arduino to detect the input.

More information is available in Part 6; see "Optocoupler Test."

Relays

Whereas a transistor is an electronic switch (with no moving parts), a *relay* is an electromechanical switch. It consists of an electromagnetic coil and a switch. Running current through the coil creates a magnetic field that pulls the switch into place. When the current is removed, the magnetic field collapses and the switch returns to its "normal" or resting position.

A small amount of electricity can energize the coil and move the switch. The switch itself, however, can handle high currents at high voltages. Our Elegoo kit includes a Songle SRD-05VDC-SLC-C Relay, which is widely used in microcontroller applications. A 5V direct-current power supply, such as the one on board our Arduino Uno, is all that is needed to operate it, but it can switch 10 A at 125 V of alternating current – enough to turn household appliances on and off.



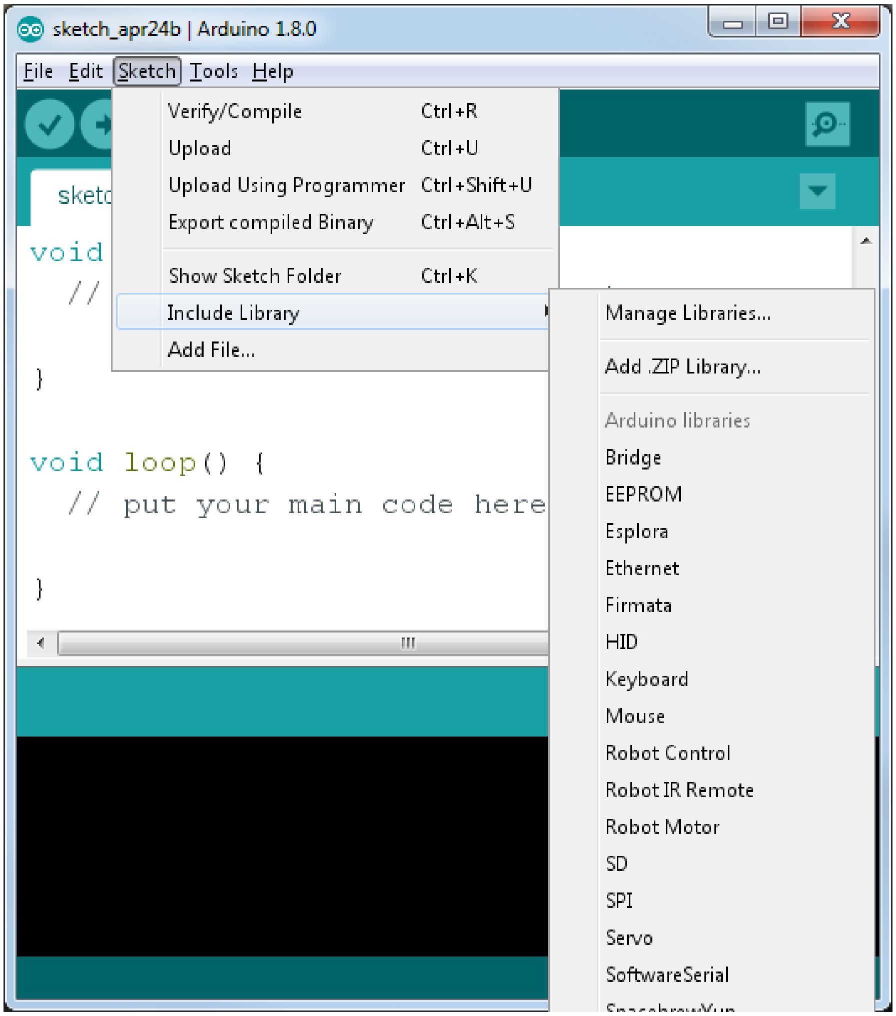
Circuit diagram as viewed from the top of the Songle relay. Pinout, counter-clockwise from top left: 1. Operate coil, 2. Common of switch, 3. Operate coil, 4. Normally open side of switch, 5. Normally closed side of switch.

Although the Arduino's power supply has enough juice to operate the relay, the output pins themselves are limited to about 40 mA, and that's not enough. But we can overcome this limitation by using the output pin to operate a transistor, and let the transistor switch the higher current required by the relay. More information is in Part 6; see "Transistor to Relay."

When the relay is in its normal state, there is an electrical connection between the "Common" pin and the "Normally Closed" pin. When the relay is operated – that is, when its coil is energized – the switch moves, disconnecting the Common pin from the Normally Closed pin and creating, instead, a connection between the Common pin and the "Normally Open" pin. You can hear this happening: Moving the switch makes an audible click. To energize the coil and operate the relay, one side of the coil must be connected to positive voltage and the other side to GND.

Part 4:
Programming Elements

The Arduino Integrated Development Environment (IDE)



```
void
  //

}


void loop() {
  // put your main code here

}
```

Probably the most common tool for writing Arduino sketches and uploading them to your Arduino is the integrated development environment that is available for free at www.arduino.cc. Here you can download versions of the IDE for the Windows, Mac OS X, and Linux operating systems. You can also find straightforward instructions for installing the IDE and getting started writing sketches.

Arduino Libraries

The Arduino programming language accessible within the IDE is a variant of C/C++. Its capabilities can be extended with libraries. A library adds functions beyond those available within the base language.

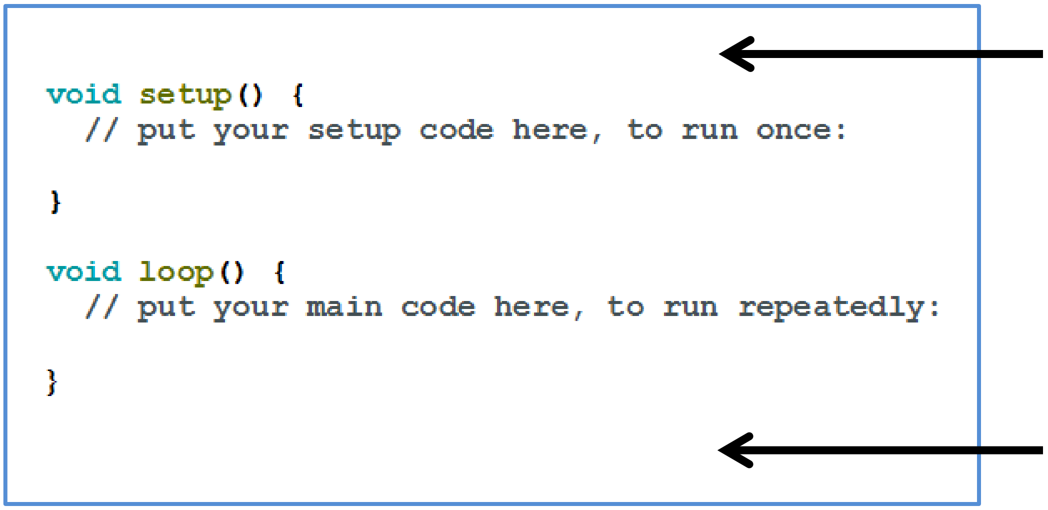
Many libraries have been designed to help you write code for specific sensors or actuators. Such libraries do most of the heavy lifting in terms of highly technical code, allowing you to concentrate on the purpose of your sketch and write code that is more straightforward.

The IDE comes with many popular libraries, and adding them to your sketch is simple. You just select the library from the pull-down menu accessed by clicking *Sketch* and then *Include Library* (see above). Or you can manually add the code that references the library. For example, suppose you want to use the "Servo" library. You could click *Servo* in the library menu, or you could add this code at the top of your program: #include <Servo.h>

After the reference to the library has been added to your sketch, you can use any of the library's functions in your sketch. Of course you have to know about the functions to be able to use them, and unfortunately the quality of the documentation for libraries is uneven. The best documentation is for the libraries listed at www.arduino.cc (e.g., check out the documentation for the Servo library at https://www.arduino.cc/en/Reference/Servo).

You can install additional libraries – and add them to the menu – by following these directions: https://www.arduino.cc/en/Guide/Libraries.

Basic Structure of an Arduino Sketch



```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Add library references and declare global variables up here.

If you add your own functions, put them down here.

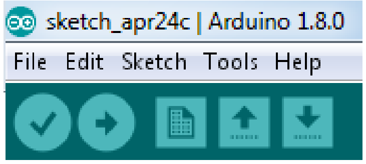
In the Arduino IDE, a blank sketch page looks like this. The sketch must include the setup() and loop() functions. Code in the setup() function will run just once, when your sketch starts. The sketch starts when either the Arduino is powered or the reset button is pressed. Code in the loop() function will run from top to bottom and then repeat as long as the Arduino has power or the reset button is left alone.

The space above the setup() function is where you should put references to libraries. This is also the place to declare variables and constants that are available to (can be “seen by”) the entire sketch. The space below the loop() function is a good place to put the code for any functions that you may provide. I’ll say more about variables, constants, and user-created functions later in Part 4.

Curly braces – { } – define the beginning and end of functions and some other blocks of code such as “if” blocks (more on “if” in Part 5, “Branching”). A opening brace must be followed eventually by a closing brace.

A semicolon marks the end of a statement. Forgetting to end a line of code with a semicolon will lead to errors when you compile the program. Because the error messages in the Arduino IDE are primitive by today’s standards, you may have trouble figuring out what’s wrong. When you get a confusing compiler error, start by looking for places where you forgot to type a semicolon.

Comments are preceded, on a line by line basis, by two forward slashes: //. A multi-line comment can be preceded by /* and terminated by */. There are many examples in Part 6.



“Compiling” means translating your sketch, which is written in a language for humans (believe it or not), into machine language that can be executed by the Arduino’s central processor, the Amtel ATmega328/P. You can compile your sketch by clicking the first control at the upper left of the IDE window – the circle with the check box. If there are problems, error messages will appear at the bottom of the window. If your code passes muster, the message will be “Done compiling”.

You also will make good use of the second control – the circle with the right-pointing error. Clicking it compiles the sketch and, if it passes muster, uploads it to the Arduino through the USB cable.

A final point: Be careful about upper and lower case. The compiler won’t correct your typos.

Variables and Constants

Variables are locations in a computer's memory that can store data and have a name (hopefully one that is meaningful to the programmer). The content of a variable is sometimes called its *value*. Values can change as a sketch is executed. For example, you might use a variable to keep track of the number of times a rat presses a lever; your sketch would increment the value with each press.

Variables are classified by the kind of data they are capable of storing. These are the classifications in our workshop:

- int: This class of variable can store an integer (a whole number) between -32,768 and 32,767.
- long: An integer that can vary from -2,147,483,648 to 2,147,483,647.
- float: This is for floating-point numbers, that is, numbers with a decimal point.
- String: This holds text, that is, a string of characters (note the upper-case 'S").
- boolean: A variable of this type can hold either of two values: true or false.

Integers are processed faster that floating-point variables. Unless a floating-point number is essential, use integers.

You *declare* (create) a variable by stating the data type followed by the name you want to use:

- `int respCount;`
- `long sessionTime;`
- `float respRate;`
- `String ratID;`

You can store a value in a variable when you declare it. For example:

- `int interTrialInterval = 5000;`
- `boolean reinforcerIsReady = false;`

Constants also are locations in a computer's memory that have a name and can store data. They differ from variables in that their values are fixed. Once a constant is declared, its value cannot be changed. Constants are used to make code more readable. For example, if an intertrial interval is going to be 5,000 milliseconds throughout an experimental session, it is prudent to declare a constant with a meaningful name. Your code will make more sense if it refers to something meaningful like "interTrialInterval" rather than "5000". You declare a constant by putting *const* before the data type in the declaration statement and assigning the fixed value:

- `const int interTrialInterval = 5000;`
- `const String ratID = “Chewa”;`

Variables and constants have a property called *scope*. When the scope is *global*, the variable or constant can be "seen" by any function throughout the sketch. To be global, a variable or constant must be declared at the top of the sketch, before the setup() function.

When a variable (or constant) is declared within a function, it is *local* to that function. A local variable can be seen only by the function in which it was declared.

The Arduino IDE includes some pre-established global constants, including: HIGH, LOW, INPUT, INPUT_PULLUP, OUTPUT, LED_BUILTIN, true, false. You can read about them at www.arduino.cc/en/Reference.

Array Variables

An *array* is a collection of variables with a single name, differentiated by an index number. The most common way to declare an array is this:

```
int responseLatency[100];
```

This statement would create an array of 100 integers, with index values from 0 to 99. The first element in the array would be responseLatency[0] and the last would be responseLatency[99]. Note that the index is enclosed within square brackets, not parentheses. That's to differentiate array variables from functions.

You can create an array of any data type: int, long, float, String, Boolean, and so forth.

Arrays are often used within *for* loops (discussed in Part 5 in the section on "Looping") where the loop counter is used as the index for each element of the array. With just a few lines of code, tens, hundreds, or thousands of elements can be accessed. Arrays allow for highly efficient coding.

Arithmetic Operators

Arithmetic is, for the most part, straightforward. Here are the operators :

= Assignment operator, e.g. x = 3 assigns x the value of 3. Not to be confused with == which is a comparison operator.

+ Addition

- Subraction

* Multiplication

/ Division

% Modulo. Returns the remainder when one integer is divided by another. If x = 17 and y = 5 then x % y returns 2.

The key thing is to recognize that a single equals sign (=) tells the compiler to assign a value, with the flow of information going from right to left. Some examples:

- `ratID = “Chewa”; // puts the string “Chewa” into ratID`
- `ITI = 3000; // puts the number 3000 into ITI`
- `oldTime = newTime; // puts the value of newTime into oldTime`
- `sum = trialOne + trialTwo; // adds the value of 2 variables and puts them into sum`

Comparison Operators

These operators are used to compare two values, normally within some kind of decision-making structure (see “Branching” in Part 5). The key thing here is the comparison to see if two values are equal. The comparison operator consists of two consecutive equals signs (==), which is to distinguish the comparison of values from the assignment of values as described above. For example, this code is valid:

```
if (digitalRead (respPin) == LOW){ // input detected
   respCount = respCount + 1; // so count it
 }
```

This code, with just one character missing (the second ‘=’) is invalid:

```
if (digitalRead (respPin) = LOW){ // input detected
   respCount = respCount + 1; // so count it
 }
```

The problem is two-fold. First, this is an easy mistake to make. Second, the compiler will not catch it; both blocks of code will pass muster. Yet the code in the second example will not yield the desired results. And you might read, re-read, and re-re-read your code without finding your mistake. You have been warned.

== Equal to. Not to be confused with = which is the arithmetic assignment operator.

!= Not equal to

< Less than

> Greater than

<= Less than or equal to

>= Greater than or equal to

Boolean Operators

In Boolean logic (named after 19th century mathematician George Boole), expressions are either true or false, and comparisons involving these expressions also are true or false. An “expression” could be a constant, a variable, or a comparison. The Boolean operators are “and”, “or”, and “not”, symbolized as follows:

&& Logical “and”

|| Logical “or” (these characters are typed by pressing your keyboard’s back-slash key with the Shift key held down)

! Logical “not” (negation)

Consider this bit of code:

```
if ((digitalRead (respPin) == LOW)&& reinforcerIsReady{
   respCount = respCount + 1; // count response
   digitalWrite (reinforcerPin, HIGH);// turn on reinforcer
 }
```

The parenthetical material in the if statement has a comparison [digitalRead (respPin) == LOW] and a Boolean variable (take my word for it), reinforcerIsReady. If the comparison is true and the variable is true, then the if statement is true and the next two lines of code will be executed.

The following table shows the results of the Boolean operations with different combinations of X and Y values. In the table X and Y could be Boolean variables, Boolean constants, or expressions yielding Boolean results (i.e., the result of the expression would be true or false).

- **X && Y** is true only when both X and Y are true.
- **X || Y** is true when either X or Y (or both) is true.
- **! X** is the negation of X; the expression is true when X if false, and false when X is true.

X	Y	X && Y	X || Y	! X
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Functions

A *function* is a block of code that performs some well-defined duty (it carries out a function) and can be called to action by other code within the sketch. Much of the Arduino programming system consists of predefined functions. Here are a few examples, all of which are described in more detail in Part 6 and in the Quick Reference.

- **sqrt(*x*)** – This function accepts one parameter: a number or a variable containing a number. It calculates the square root of that number and returns the result.

- **noTone(*p*)** – This function accepts one parameter: the number of an output pin that is being used to play a tone. It stops the tone. It returns nothing.

- **millis()** – This function accepts no parameters. It returns the time since the Arduino started, in milliseconds. You might wonder why the parentheses are needed when the function accepts no parameters. The parentheses designate that “millis” is the name of a function.

Although each of these functions performs a specific action, they differ from one another in certain ways. Two of them need information (*parameters*) from the programmer in order to perform the action [sqrt(), noTone()]. One needs no parameter but returns a value [millis()]. One needs a parameter but returns no value [noTone()].

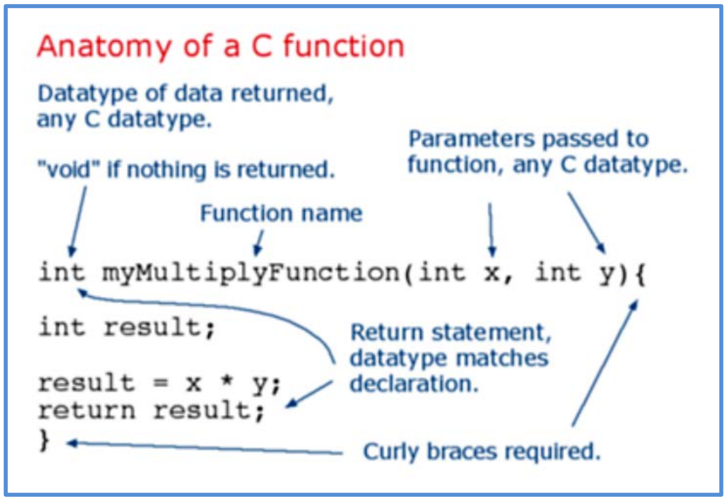
You are not limited to the built-in functions. Arduino libraries add specialized functions to serve a variety of purposes, often in connection with a specific sensor or actuator. And you can write your own functions. If your sketch needs to carry out a set of instructions more than once, you can put those instructions into a function and then call that function from anywhere in your sketch.

To declare a function that returns a value, you must (a) indicate the type of value to be returned (e.g., int, long, float, Boolean, etc.), (b) provide a name for the function, and (c) list the parameters, if any, that the function will receive. The function’s code must be enclosed within curly braces and it must end with a return statement that designates the information to be sent back to the section of the sketch that called the function.

To declare a function that does *not* return a value, you must (a) begin with the word *void*, (b) provide a name for the function, and (c) list the parameters, if any, that the function will receive. Again, the function's code must be enclosed within curly braces, but there is no return statement.

In the example at right, the myMultiplyFunction() is designed to receive two ints, multiply them, and return the result as an int.

The part of the sketch that calls the function could look like this:


Anatomy of a C function
Datatype of data returned, any C datatype.
"void" if nothing is returned.
Function name
Parameters passed to function, any C datatype.
Return statement, datatype matches declaration.
Curly braces required.

```
int myMultiplyFunction(int x, int y){
int result;

result = x * y;
return result;
}
```

```
totalResponses = myMultiplyFunction(responsesPerTrial, trials);
```

Here I am assuming that totalResponses, responsesPerTrial, and trials are previously declared ints.

Suppose your sketch is controlling an operant conditioning session in which a pigeon sometimes earns 2-s access to food and sometimes earns 6-s access. You could accomplish this with a function that accepts two parameters: the output pin that is wired to the food hopper and the amount of time the hopper should be raised to give the pigeon access to the food. The function might be:

```
void raiseHopper(int hopperPin, int milliseconds) {
  // This function raises the hopper, waits, then lowers the hopper
  digitalWrite(hopperPin, HIGH);
  delay(milliseconds);
  digitalWrite(hopperPin, LOW)
}
```

To give the pigeon 2-s access to food, the sketch in the loop() might say:

```
 if (respCount >= fixedRatio) {
    raiseHopper(3, 2000);
 }
```

Here I am assuming that the hopper is controlled by Pin 3.

The order of the parameters is important: The values in the calling statement must line up with the variables in the function's declaration. In this example, raiseHopper() understands 3 as the output pin for the hopper and 2000 as the duration because the function's declaration lists the pin first and the duration second.

Your code can pass parameters to a function in the form of literal values (e.g., 3, 2000) or in the form of variables or constants that store appropriate values. Suppose int "feeder" stores the number of the pin wired to the hopper and int "access" stores the number of milliseconds the hopper should be raised. The calling code could be

```
raiseHopper(feeder, access);
```

For more about writing functions, see https://www.arduino.cc/en/Reference/FunctionDeclaration.

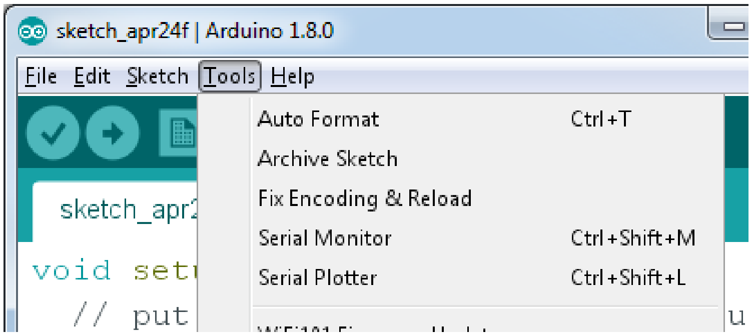
Part 5: Common Programming Tasks

Communication through the Serial Port

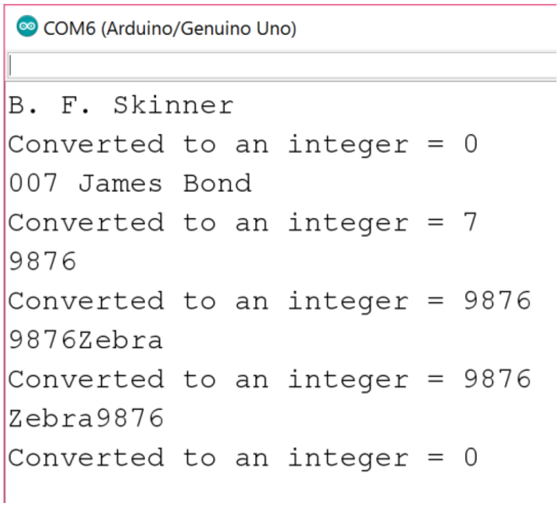
The USB cable that you use to transfer sketches to the Arduino IDE on your personal computer (PC) also allows the PC to communicate with the sketch as it runs on the Arduino. The communications protocol uses Digital Pins 0 and 1, and that is why you rarely encounter sketches in which these pins are used for general input-output purposes.

The easiest way for the PC and Arduino to communicate is through the serial monitor that is built into the Arduino IDE. You can activate the serial monitor via the “Tools” tab in the IDE’s menu bar. A pull-down menu in the lower right corner of the monitor’s form allows you to set the baud rate, which is the speed of communication in bits per second. The monitor’s baud rate must match the baud rate in your sketch (more on this below). The default rate of 9600 baud is commonly used.

The field at the top of the serial monitor form is used to send information to the Arduino: Just type what you want and click the “Send” button. The large field, the one that occupies most of the form, displays information that the Arduino sketch sends to the PC.



You can write your own programs to establish communication between a PC and an Arduino. One might imagine a program on the PC that allows an experimenter to enter the parameters of an experiment and send them to the Arduino. The Arduino sketch, in turn, would receive the parameters and use them in controlling an experimental session. The article by Escobar and Perez-Herrera (2015), included as an appendix, provides an illustration.



```
B. F. Skinner
Converted to an integer = 0
007 James Bond
Converted to an integer = 7
9876
Converted to an integer = 9876
9876Zebra
Converted to an integer = 9876
Zebra9876
Converted to an integer = 0
```

When you send information to the Arduino through the serial monitor, the information is received as a string of characters. You may, however, wish to convert the information into a number. This can be accomplished by saving the received information into a String variable and then using the *toInt()* function which returns a long number. In Part 6, the sketch “Reading Serial Strings as Parameters” shows how to use the toInt() function. Sample output from the sketch is shown at left.

The Arduino supports a lot of serial communication functions (see www.arduino.cc/en/Reference/Serial). Reviewed here are the ones that are used in the sample sketches in Part 6 of this *Reference*.

Serial.begin(*baud*);

This function sets up the communications link between the Arduino and the PC. The "baud" parameter is replaced with a number representing the desired baud rate (e.g., 9600). It should appear in the setup() function of your sketch. Example:

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

Serial.setTimeout(*milliseconds*);

This sets the amount of time, in milliseconds, that the sketch will spend reading the serial port before moving on. This function should appear in the setup() function of your sketch. The default is 1000, that is, 1 second. That's a long time to wait. If you anticipate short strings, the time limit can be brief. I suggest that you play with different limits to see what suits your purpose. Example:

```
Serial.setTimeout(10);
```

Serial.available()

This function returns the number of characters that are available to be read from the serial port. The characters are in the serial receive buffer and waiting to be read. If the result is 0, then there is nothing in the buffer, that is, nothing to read. This function is commonly used within an "if" structure. If the function returns a value greater than zero, we know there is something to read and we can take appropriate action. For example,

```
if (Serial.available() > 0) { // characters are in the buffer, so read them
```

Serial.readString()

This function tries to read a string of characters from the serial port. If will persevere until it reaches the time limit imposed by the "Serial.setTimeout()" function. Here is an example in which the string is read and saved in a string variable, then converted to a number for further processing.

```
if (Serial.available() > 0) {
    // yes, there are characters in the buffer, read them into string variable
    String inputString = Serial.readString();
    // if possible, convert the string to a number and store it
    // if the string does not beging with a numeral, a zero is returned
    long number = inputString.toInt();
```

Serial.print(*string*); and Serial.println(*string*);

These two functions send a string of characters from the Arduino through the serial port to the PC. They differ in one respect: "print" simply sends the string whereas "println" sends the string and follows with a newline character that causes the next string to be printed on a new line. The "string" parameter can be a string variable or a literal such as "Number of Responses." Example:

```
Serial.print("Lever Presses: ");
Serial.println(responseCount); // responseCount is an integer variable
```

If responseCount held a value of 27, the following would appear in the serial monitor:

```
Lever Presses: 27
```

Most of the sketches in Part 6 incorporate serial communication. These include “Reading Serial Strings,” “Reading Serial Strings as Parameters,” “Count Button Presses,” “Count Button Presses Debounced,” “Joystick Simple,” “Joystick Refined,” and many more.

Timing

There are four timing functions available to Arduino programmers, two for creating delays and two for marking the time.

delay(*milliseconds*);

This pauses the sketch for the specified number of milliseconds (one-thousandth of a second). For example:

```
delay(1000); // pause for a second
```

It is important to understand that “pausing” the sketch means that the main loop is halted during this time: no inputs are read, no outputs written, no conditions tested, no calculations made. In some circumstances, this may be exactly what you want. In other circumstances, you may need timing to go on concurrently with other tasks. This can be accomplished using the millis() function described below – and some additional coding.

delayMicroseconds(*microseconds*);

This function pauses the sketch for the specified number of microseconds (one-millionth of a second). This should be used only when very, very short pauses are required, otherwise use delay(). For example:

```
delayMicroseconds(100); // pause for a tenth of a millisecond
```

The maximum allowable pause with this function is 16,383 microseconds, or in other words, 0.016383 second.

millis()

This function returns the time in milliseconds since the Arduino began running the current sketch – that is, since the Arduino was powered up or reset. This number will overflow (go back to zero), after approximately 50 days. As long as your sketch does not run for that long, you should be in good shape. Because the result of millis() can be such a large number, use a long variable, rather than an int, to store the result.

```
long currentMilliseconds = millis(); // mark the time
```

Calculating the time between events is straightforward: The formula is Current Time – Time of Previous Event. For example, to calculate the time between successive button presses, you would follow these steps: (a) Record the time of the first press. (b) At the second press, get the current time and subtract the time of the first press. This yields the time between the two presses. (c) Record the time of the most recent press so that you can repeat the calculation when another press occurs. Here is some code that would accomplish the task, assuming it is in the main loop of the sketch. After each press, the code prints (to the serial port) the time that has elapsed since the previous press. (For simplicity, I’ve omitted the code to declare the variables, set up serial communication, and detect the presses.)

```
interResponseTime = millis() - previousResponseTime; // time between responses
Serial.println(interResponseTime); // sent result to serial port
previousResponseTime = millis(); // record time of this response
```

The sketches in this *Reference* use millis() to correct for contact bounce in noisy switches. For an example, see the sketch entitled "Count Button Presses Debounced."

micros()

This function returns the time in microseconds since the Arduino began running the current sketch – that is, since the Arduino was powered up or reset. There are 1,000 microseconds in a millisecond or, in other words, 1,000,000 microseconds in a second. Because micros() can return such a large number, use a long variable, rather than an int, to store the result. For example:

```
long currentMicroseconds = micros();
```

On our Arduino Uno, the timing resolution is 4 microseconds. That's not bad: Timing is accurate to 0.000004 second!

The micros() function has an important limitation: This number it returns will overflow (go back to zero), after approximately *70 minutes*. Your sketch must start and end within that period, otherwise the results from micros() will be unreliable.

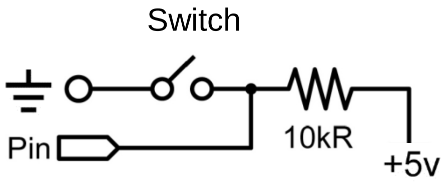
Digital Input

The Arduino’s pins can be configured as inputs or outputs. Input pins receive signals from the environment. Digital signals are discrete – the signal is “on” or “off” – as compared to analog signals that can vary continuously across a range of values. A digital input pin, then, is configured to detect “on” or “off” states of the device that is connected to it. The most common example in a prototype circuit is a button. In the psychology laboratory, the connected devices would be designed to detect behavior, including rat levers, pigeon keys, or photocell circuits that detect a mouse’s nose poke.

In digital circuits, “on” translates to HIGH which means, in the case of our Arduino Uno, there is 5V on the pin. “Off” translates to LOW which means there is 0V (i.e., GND) on the pin.

The Arduino’s input pins are “high impedence:” Left alone, they are susceptible to electrical noise in the environment and will flip randomly between HIGH and LOW. This obviously is unacceptable if the pins are to be used to detect the state of a button, lever, key, or photocell. One solution is to tie the input pin to 5V so that when at rest, the pin is HIGH. We then wire up the input switch (button, key, etc.) so that it turning the switch on (e.g., pressing the button) puts GND on the pin, pulling it LOW. Our sketch looks for the pin to go LOW and when it does, we count that as a response.

Here is a diagram of the circuit:

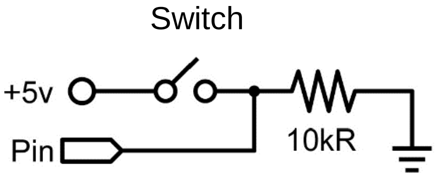


“Pullup” Input Circuit

⏚ = GND

With the switch open, as shown here, the pin is connected to 5V through a 10K-ohm resistor. This holds the pin in its HIGH state (it will be “pulled up”). One side of the switch is connected to GND (i.e., 0V) and when it is closed, the pin will be connected to both GND and 5V. Because the path to GND has no resistor in it – it is literally the “path of least resistance” – that connection takes precedence, and the pin will be brought to a LOW state.

If it seems odd to you to set up a system in which a button press is detected when the input goes LOW rather than HIGH, you can flip the 5V and GND and create a “pulldown” circuit:



“Pulldown” Input Circuit

In this circuit, the pin is held LOW (“pulled down”) until the switch is closed, at which point 5V flows to the pin, bringing it HIGH. With this circuit, a button press is detected when the input goes HIGH.

Either strategy will work, but the pullup strategy is the easier because the circuitry is built-in. If you choose the pulldown strategy, you must provide the resistor and build the circuit. If you choose the pullup strategy, you simply activate the built-in circuit within your Arduino sketch. This *Reference* assumes the pullup strategy. *EXCEPTION: For technical reasons, Pin 13 has trouble with the INPUT_PULLUP mode, so don’t use it with Pin 13.*

pinMode(*pin*, INPUT_PULLUP);

This function, which should be part of the setup() function of your sketch, configures the designated pin as an input and activates the internal pullup circuit. The "pin" parameter can be the actual pin number or a variable or constant that holds the pin number; the advantage of the latter is that the variable can convey the function of the pin. Consider these two blocks of code:

```
pinMode(8, INPUT_PULLUP);
pinMode(9, INPUT_PULLUP);

const int leftLever = 8;
const int rightLever = 9;
pinMode(leftLever, INPUT_PULLUP);
pinMode(rightLever, INPUT_PULLUP);
```

The first block of code is simple. The second requires an extra step: creating two integer constants to hold the pin numbers. Except in the simplest sketches, however, the extra effort is worthwhile. Using meaningful names instead of numbers to refer to the pins will make your code easier to write, read, and debug.

digitalRead(*pin*)

This function returns the current state of the pin, LOW or HIGH. The result can be stored in a variable or acted upon immediately. A couple examples:

```
newButtonState = digitalRead(leftLever);  // read pin, save result

if (digitalRead(startButton) == LOW){  // button is pressed, so do something…
```

Counting Discrete Input Events

Your sketch can execute its loop() function rapidly. A single input event (button press, key peck, etc.) may be detected hundreds of times. For example, a rat may press a lever and hold it down for a half-second before releasing it and pressing it again. Your sketch may "see" that first press several hundred times, but you only want to count it as a single response. To do this, you need to keep track of *changes* in the state of the input.

To track changes in a button, for example, we need to create two variables. One is used to save the input state the last time a change was detected; the other is used to hold the current input state. Each time we read the input pin, we compare its new (current) state with the old state. If the new state does not match the old state, that means the state has changed, in which case we need to (a) record the fact that a change was detected, by updating the "old state" variable, and (b) evaluate the new state. If the new state is LOW (assuming a pullup circuit is being used), then the button has been pressed and we count it. If the new state is HIGH, it means that the button has been released and we can move along without incrementing our counter.

The sketch on the next page will do the trick. Each new button press increments a counter. The button must be released and pressed anew for another increment. This is bare-bones code to show the essential logic involved in detecting and counting discrete input events. The sketch doesn't do anything with the information except to send the value of the counter to the serial port so you can view it on the serial monitor.

```
// COUNTING DIGITAL INPUTS
// define some integers
const int respButton = 8; // button will be connected to Digital Pin 8
int respCount; // for counting responses
int newButtonState; // for keeping track of changes in button’s state
int oldButtonState; // for keeping track of changes in button’s state


void setup() {
  pinMode(respButton, INPUT_PULLUP);  // input with internal pullup
  Serial.begin(9600);  // Open a serial port
}

void loop() {
  // read the state of the response button
  newButtonState = digitalRead(respButton);
  // has the state changed since our last read?
  if (newButtonState != oldButtonState) {
    // yes, the button state has changed so make a note of it
    oldButtonState = newButtonState;
    // is the new state of the button LOW, i.e., is button pressed?
    if (newButtonState == LOW){
      respCount = respCount + 1; // yes, button is pressed, count it
      Serial.println(respCount); // send the count to the serial port
    }
  }
}
```

Debouncing

The electrical contacts inside a switch can vibrate briefly when the switch changes state – for example, when a pushbutton is pressed. The vibrations, called “contact bounce,” generate spurious open/close transitions that your sketch may read as multiple presses. This problem may be corrected in hardware with specialized circuitry. Or it may be corrected in software with a simple strategy: Whenever a change in the input’s state is detected, further changes are ignored for a little while – a very little while, perhaps 5 milliseconds or less (depending on how “noisy” your switch may be). The idea is that the changes in the input that take place within milliseconds of one another are caused by contact bounce and should be ignored.

To ignore contact bounce – to add “debouncing” code to our sketch – we need a function that tells time: We will use the millis() function that returns the number of milliseconds that has elapsed since the Arduino began running the current sketch.

The next sketch adds debouncing code to the sketch we just reviewed. The logic is s straightforward. When we detect a change in the state of the button (LOW or pressed, HIGH or released), we note the time. Then, when we detect further changes in the button state, we ignore them if they have occurred too soon after the last recorded state change. "Too soon" is operationalized in the debounceDelay constant, which is set here to 3 milliseconds. If you happen to be using very noisy switches, you may need to set a longer delay.

To facilitate comparison of the two sketches, the code added for the purpose of debouncing is highlighted in yellow.

```
// COUNTING DEBOUNCED DIGITAL INPUTS
// define some integers
const int debounceDelay = 3; // 3-millisecond delay for debouncing
long lastChangeMoment; // to record the moment of last input change
long elapsedChangeTime; // to record the passage of time since last change
const int respButton = 8; // button will be connected to Digital Pin 8
int respCount; // for counting responses
int newButtonState; // for keeping track of changes in button’s state
int oldButtonState; // for keeping track of changes in button’s state


void setup() {
  pinMode(respButton, INPUT_PULLUP);  // input with internal pullup
  Serial.begin(9600);  // Open a serial port
}

void loop() {
  // read the state of the response button
  newButtonState = digitalRead(respButton);
  // calculate how much time has passed since last button change
  elapsedChangeTime = millis() - lastChangeMoment;
  // has the state changed since our last read AND has the debounce delay passed?
  if ((newButtonState != oldButtonState) && (elapsedChangeTime >= debounceDelay)) {
    // yes, the button state has changed and enough time has passed,
    // so we will pay attention to the change
    oldButtonState = newButtonState; // record the change in state
    lastChangeMoment = millis(); // record the time of this change
    // is the new state of the button LOW, i.e., is button pressed?
    if (newButtonState == LOW){
      respCount = respCount + 1; // yes, button is pressed, count it
      Serial.println(respCount); // send the count to the serial port
    }
  }
}
```

Digital Output

A digital output pin is configured to turn the device connected to it on or off. Most devices are designed so that setting the output pin HIGH turns it on and setting the pin LOW turns it off. I have encountered devices that work the opposite way, but in this *Reference* the assumption is that HIGH = on and LOW = off.

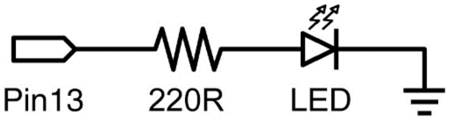
pinMode(*pin*, OUTPUT);

This is all it takes to configure a pin for output. The “pin” parameter can be the number of the pin or a constant or variable with a meaningful name that holds the number of the pin (e.g., “pelletDispenser”).

digitalWrite(*pin*, HIGH); and digitalWrite(*pin*, LOW);

The digitalWrite function is used to change the state of the output pin; setting it HIGH allows 5V to flow and setting it LOW grounds it.

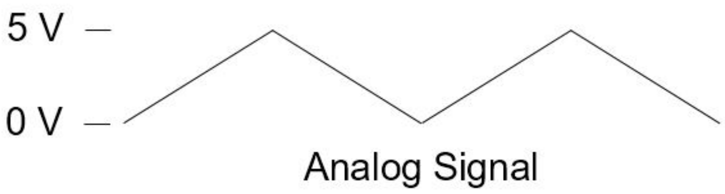
The “Blink” sketch is the classic demonstration of digital output. The sketch is designed to control an LED. The positive lead of the LED is connected to an output pin (Pin 13 in this example) and the negative side is connected to GND. Note also the 220-ohm resistor in series with the LED.



```
void setup() {
  pinMode(13, OUTPUT);      // initialize digital pin 13 as an output.
}

void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

Analog Input

Our Arduino has six input pins numbered A0 through A5 that are capable of analog input. Whereas a digital input can detect only LOW and HIGH states (0V and 50V), an analog pin can detect a wide range of states anywhere between 0V and 5V. When used for analog input, these pins connect to an analog-to-digital (A/D) converter. The A/D converter translates the voltage received by the analog pin to an integer between 0 and 1,023. If the voltage on the pin is 0 (i.e., GND), the A/D converter returns 0. If the voltage is 5, the A/D converter returns 1,023. Intermediate voltages lead to intermediate digital conversions. The point is this: Although an analog signal can have infinite variation, our Arduino will digitize the variation into a finite set of 1,024 values (i.e., 0 through 1,023).



You may be interested to know that Pins A0 through A5 do not have to be used for analog input. They can be configured for digital input or output using the code described in the previous two sections. If you use one of these pins for digital input or digital output, and then you want to switch the pin to analog input, you should (a) explicitly set the pin mode to simple input (i.e., with no pullup resistor circuit), and then pause for a few milliseconds before attempting to read the analog pin. For example:

```
pinMode(pin, INPUT);  // simple input mode
delay(10); // pause briefly to allow electrical noise to quiet down
```

This is the function for analog input:

`analogRead(`*`pin`*`)`

The “pin” parameter can be the number of the pin a constant or variable with a meaningful name that holds the number of the pin (e.g., “temperatureSensor”). You may be wondering what kind of variable or constant should be used to store A0, A1, etc. You can use an int because A0 (and the other analog pin designations) is a built-in constant that holds the actual pin number. Here are the numbers:

A0=14, A1=15, A2=16, A3=17, A4=18, A5=19.

In other words, you can set the variable or constant to A0 or to 14. These are equivalent statements:

```
const int temperatureSensor = A0;
const int temperatureSensor = 14;
```

As noted above, the analogRead() function will return a value between 0 and 1,023. You probably will want to convert the value to one that is more meaningful – for example, degrees Farenheit if your analog sensor is responding to temperature. The map() function may come in handy:

`map(`*`val, fromMin, fromMax, toMin, toMax`*`)` where

val = the reading from the analog pin,
fromMin = the lowest possible reading
fromMax = the highest possible reading
toMin = the lowest possible value in the converted unit of measurement
toMax = the highest possible value in the converted unit of measurement

The map() function uses integer math. This would appear to pose a problem if, for example, you wanted to convert the reading to voltage. To illustrate, suppose you wrote this code to get the voltage at Pin A0:

```
int pinReading = analogRead(A0);
int volts = map(pinReading, 0, 1023, 0, 5);
```

The value read from the pin can be from 1 to 1,023. We want to convert that to something between 0V and 5V. This seems like an easy way to do it. And yes, it will work, but not very well because in this code "volts" can take on just six possible values: 0, 1, 2, 3, 4, 5. We won't be able to measure fractional changes in voltage, even though our instrument is capable of it. Our A/D converter is capable of discriminating variations in voltage of 5.0/1,204 = 0.005V.

To get around the integer limitations of the map() function, we can expand the conversion range and then use floating-point math to break the results into decimal fractions. For example:

```
int newPot = analogRead(A0); // read pin connected to 10K potentiometer
// then convert the reading to voltage in the next 2 lines
float potVolts = map(newPot, 0, 1023, 0, 500); // convert to voltage x 100
potVolts = potVolts / 100.0; // convert to voltage expressed to nearest .01 V
Serial.println(potVolts); // show the voltage
```

By the way, it turns out that the potentiometer is a pretty good model for analog input in general. No matter what kind of sensor you connect to an analog pin, it will input variations in voltage. It doesn't matter if the sensor is a potentiometer, a thermistor, a photoresistor, etc. Of course, if you've attached a thermistor, you probably don't really care about the input voltage; instead, you will want to convert the input signal to temperature in degrees Fahrenheit or degrees Celsius. Or if you've attached a photoresistor, you may want to convert the signal to lux, a measure of illumination. The coding to accomplish these things is not hard, but you will have to do some research to figure out the conversion formula. Instructions may come with the sensor. In many cases, sensors come with specialized Arduino libraries that do the conversions for you – and then all you have to do is figure out which library functions to use.

Analog Output

As you may have surmised, an analog output pin is capable of sending out a wide range of voltages between 0V and 5V. On this topic, I have good news and bad news. First, the bad: Our Arduino Uno does not have any analog output pins! Some microcontroller boards in the Arduino extended family do have them, but the Uno is not one of them. The good news is that our Arduino does provide a partial work-around: It has six digital pins that can mimic analog output through a technique called Pulse Width Modulation (PWM). Here is the function:

```
analogWrite(pin,dutyCycle);
```

Here "pin" is the PWM-capable digital pin, configured as a output. On our Arduino, "pin" must be 3, 5, 6, 9, 10, 11 (or, of course, a variable or constant storing one of those numbers). The other parameter, "dutyCycle," is a value between 0 and 255. It expresses the part of the pin's normal cycle during which the pin will be HIGH. For example, a dutyCycle value of 63 is 25% of the way between 0 and 255. This value will cause the pin to be HIGH for 25% for the cycle (the "duty" part) and LOW for the other 75%. A value of 127 would have the pin be HIGH for 50% of each cycle and LOW for the other 50%. (As you might guess, at the limits, a dutyCycle of 0 will keep the pin LOW throughout the cycle, and a dutyCycle of 255 will keep the pin HIGH.)

If you attach motor to, say, Pin 3, adjusting the value of the dutyCycle will change the speed of the motor in relatively fine gradations – 256 gradations (0-255) to be exact. If you attach an LED, adjusting the dutyCyle will change the apparent brightness off the LED. The higher the value of the dutyCycle, the longer the motor or LED are powered.

Branching

Code in an Arduino sketch is executed in linear fashion, statement by statement, unless otherwise directed. Departures from a purely linear flow can be arranged in various ways. Here are three of them.

if

The simple if structure tests a condition and, if the result is true, some code is executed. It works like this:

```
if (expression){
   … code that will be executed if the expression is true…
 }
```

Here the expression in parentheses can be anything that is Boolean true or Boolean false. This is commonly some kind of comparison [e.g., **`(responseCount >= fixedRatio)].`** If the comparison is true – if the value of responseCount is greater than or equal to the value of fixedRatio, then the code inside the curly braces will be executed and the program will continue with the statement after the second curly brace. If the comparison is false, the program will skip the code inside the curly braces.

if/else

This is a straightforward extension of the simple if structure. It works like this:

```
if (expression) {
  … code that will be executed if the expression is true…
 }
  else {
  … code that will be executed if the expression is false…
 }
```

Take careful note of the two pairs of braces. The first pair encompasses the code that will be executed if the expression is true. The second pair of braces encompass the code that will be executed if the expression is false.

switch/case

This structure allows you to specify the code to be executed not just in one (if) or two (if/else) cases, but in any number of cases. It works like this:

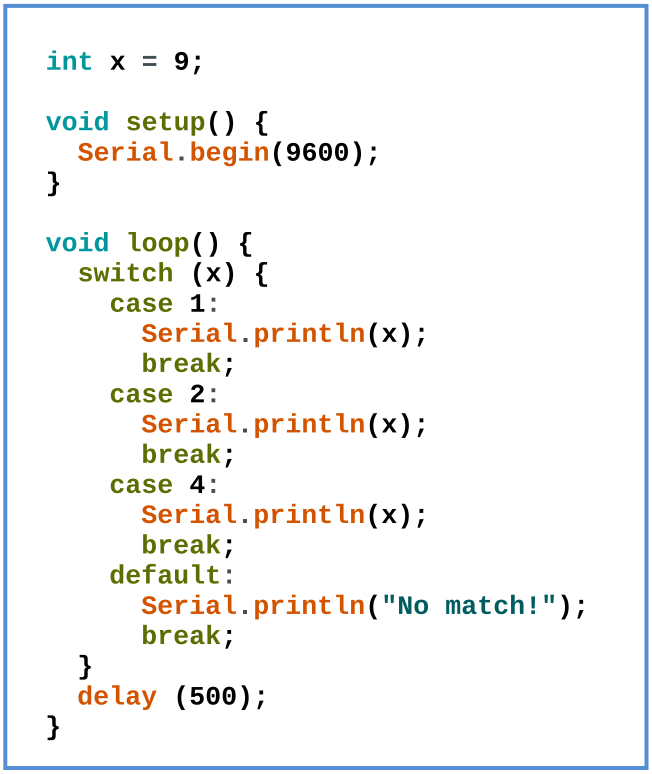
```
 switch (variable) {
    case 1:
      … code that will be executed if variable == 1…
      break;
    case 2:
      … code that will be executed if variable == 2…
      break;
    case 3:
      … code that will be executed if variable == 3…
      break;
    default:
      … code that will be executed if no match is found…
      break;
 }
```

An int or string variable is listed in the first line. This is followed a series of cases, each indicating a possible value (A, B, C) that could match the value of the variable in the first line. The values must be literal integers or literal strings. When the first match is encountered, the code that follows it is executed and then the program flow skips to the end and picks up execution with the first line after the curly brace. The *break* statement at the end of each case is required to mark the end of the code to be executed when a match is found; when the break statement is reached, the program flow breaks out of the switch/case structure.

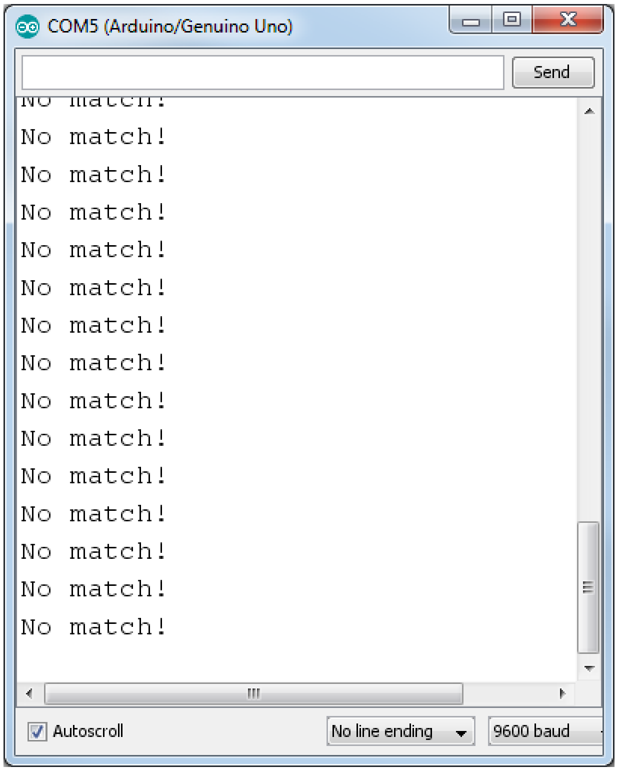
In the example above, I've listed three possible cases, but there is no practical limit to the number of cases. In Part 6, there is a sketch with a switch/case that has over 20 cases; see "Remote Signal Decoding Elegoo."

You also have the option of including a catch-all block of code to be executed if none of the cases matches the variable. This *default* block of code is listed last.

Here's a little sketch to play with:



```
int x = 9;

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

void loop() {
  switch (x) {
    case 1:
      Serial.println(x);
      break;
    case 2:
      Serial.println(x);
      break;
    case 4:
      Serial.println(x);
      break;
    default:
      Serial.println("No match!");
      break;
  }
  delay (500);
}
```



```
No match!
No match!
No match!
No match!
No match!
No match!
No match!
No match!
No match!
No match!
No match!
No match!
No match!
No match!
No match!
```

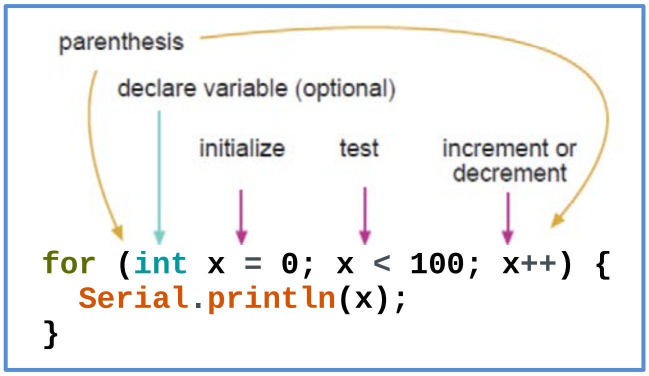
The int x is set to 9. As a result, the default code sends then phrase "No match!" to the serial monitor. What will happen if you set x to 1? Or 2? Or 3? Or 4? Or 44?

Looping

Of the various looping statements, I will consider just one.

for

The classic *for* loop, as implemented in C, has this general format:



```
for (int x = 0; x < 100; x++) {
  Serial.println(x);
}
```

The loop's header has three parts:

- *Initialization:* Declare an integer type variable (if it hasn't been declared already) and set a starting value.
- *Test:* A comparison involving the variable. The loop will execute until this comparison is false.
- *Increment or Decrement:* An expression that increases or decreases the value of the variable after each loop.

In the example shown above, the int x starts at 0. The code within the loop – that is, the code within the curly braces that mark the beginning and end of the loop – will be executed once. Then x will be incremented or decremented and the resulting value will be tested. In this example, if it is true that x < 100, the code will be executed again, and x will be incremented and tested again. Eventually, the test comparison will be false (i.e., x will not be less than 100), at which point the loop is over. In the example, the values 0 through 99 would be sent to the serial monitor.

You might be wondering about this expression: x++. This is coding shorthand for x = x + 1. If you prefer, you can say x = x + 1. You can increment in other steps sizes: If we changed the increment to x = x + 2, the loop would print 0, 2, 4, 6, … 98.

Doing Math

The code to perform arithmetic is straightforward; you pretty much use the arithmetic operators described in Part 4 as you did in grade school. The only difference is that the result is on the left of the equation because the assignment operator sends values to the left. For example, to add variables a and b and store the result in c:

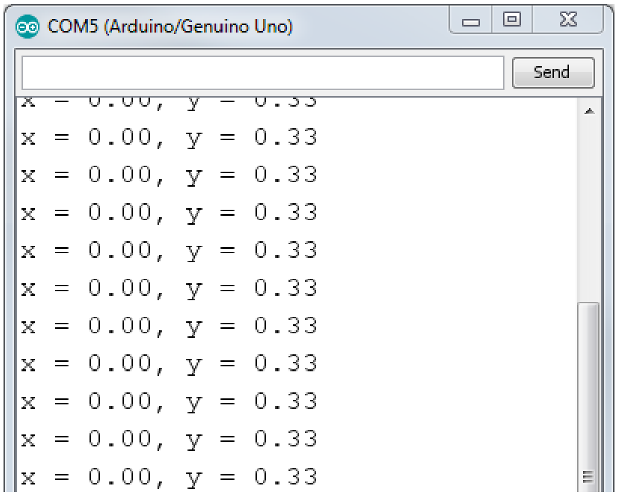
```
c = a + b;
```

If you have to code an expression involving, say, addition and division, you need to worry about the order of operations. I can't remember the rules about this, so I just use parenthesis to make the desired order explicit.

`c = a + b / c;` is ambiguous to me, so I would say `c = (a + b) / c;`

If you want to code arithmetic with floating point variables, you need to designate any literal values as floating point numbers even if they happen to be whole numbers. Consider this sketch:

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

void loop() {
 float x = 1/3;
 float y = 1./3.;
 Serial.print("x = ");
 Serial.print(x);
 Serial.print(", y = ");
 Serial.println(y);
 delay(1000);
}
```



```
x = 0.00, y = 0.33
x = 0.00, y = 0.33
x = 0.00, y = 0.33
x = 0.00, y = 0.33
x = 0.00, y = 0.33
x = 0.00, y = 0.33
x = 0.00, y = 0.33
x = 0.00, y = 0.33
x = 0.00, y = 0.33
x = 0.00, y = 0.33
x = 0.00, y = 0.33
```

We divide 1 by 3 and store the result in x. We expect 0.33, but we get 0.00. The reason is that the compiler is treating 1 and 3 and integers, and 0 is the correct result for integer division in this case. We have to tell the compiler to treat the literal values of 1 and 3 and floating point numbers. We do that by adding the decimal points. If we designate 1 as ‘1.” and 3 as ‘3.’ then our sketch will do floating point division and yield the expected result, which we have stored in the float variable y in this example.

In addition to the arithmetic operators, the Arduino language provides a variety of mathematical functions. Among the most commonly used are these:

abs(*x*)

Returns the absolute value of x.

constrain(*x*,*a*,*b*)

Constrains x to the range from a to b inclusive. For example, if x = 106 then contrain(x, 0, 100) would return 100.

max(*x*,*y*)

Returns the higher of two numbers. If x = 5 and y = 1 then max(x, y) returns 5.

min(*x*,*y*)

Returns the lower of two numbers. If x = 5 and y = 1 then min(x, y) returns 1.

pow(*b*,*e*)

Raises b to the eth power. If b = 10 and e = 3 then pow(b, e) will return 1000. Note that e can be a fraction: If b = 10 and e = .5 then pow(b, e) = 3.16.

sqrt(*x*)

Returns the square root of x.

Sometimes you may need to convert a variable from one type to another. Here are some common conversion functions. In each case, the parameter x can be of any valid numerical data type.

```
int(x)    long(x)    float(x)
```

convers x to an int, a long, or a float, respectively. (Thank you, Captain Obvious!)

If your sketch needs to receive numerical information from a user or a PC program through the serial port, the information actually will be received as a string of characters. The user might intend ’12.5’ to mean ‘twelve-and-a-half’ but your sketch won’t see it that way unless you convert from a string to a numerical value. If the string of characters is received into a string variable named ‘string’ then you can say

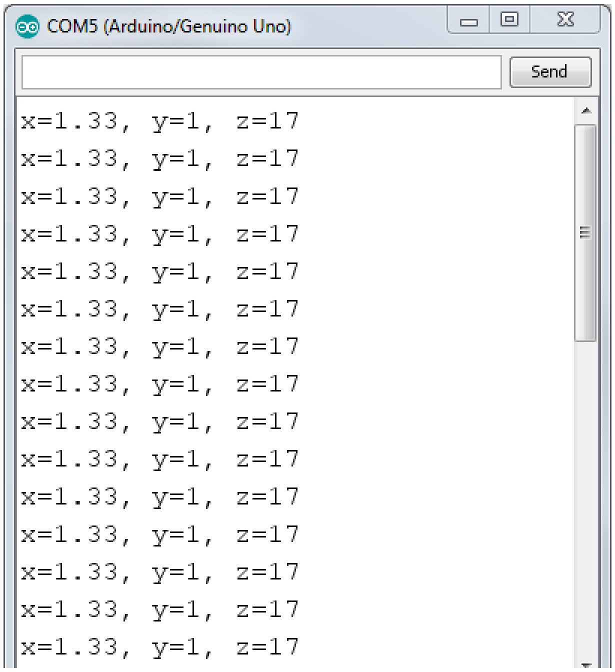
```
string.toFloat()
```

to convert it to a float or

```
string.toInt()
```

to conver it to an int. Here is a sample sketch and the results. Note what happens when the string “1.33” is converted to float variable x versus what happens when it is converted to int variable y.

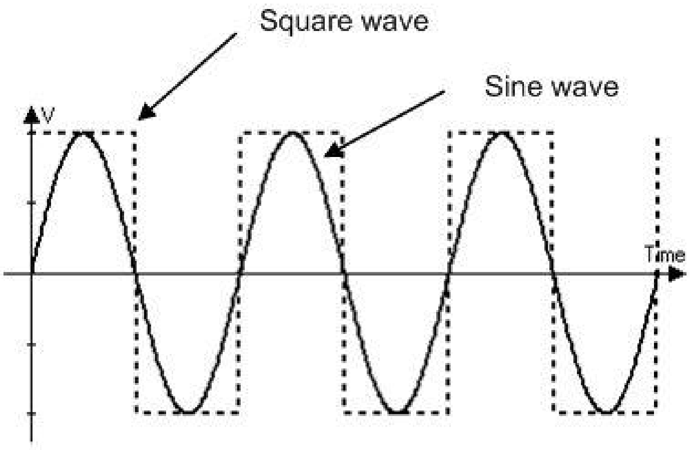
```
String one = "1.33";
String two = "17";

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

void loop() {
  float x = one.toFloat();
  int y = one.toFloat();
  int z = two.toInt();
  Serial.print("x=");
  Serial.print(x);
  Serial.print(", y=");
  Serial.print(y);
  Serial.print(", z=");
  Serial.println(z);
  delay(1000);
}
```



```
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
x=1.33, y=1, z=17
```

Making Sounds

A pure tone is a sine wave. Creating a sine wave requires a true analog output pin and, as we have noted, our Arduino Uno lacks one. It can, however, generate high frequency square waves by turning a digital pin HIGH and LOW rapidly. Attach the pin to a speaker (and perhaps a suitable amplifier) and you can get a tone. It won't sound exactly like a pure (sine-wave) tone, but it will be close enough for many purposes.



We have two functions for tones, one to start a tone of a specific frequency and the other to stop the tone.

tone(*pin*,*frequency*,*duration*);

This function generates a square wave on the designated pin (50% on, 50% off), at the designated frequency (in Hz), for the specified duration in milliseconds. The last parameter is optional; if omitted, the tone will play continuously until stopped by the noTone() function described below. The minimum frequency is 31 Hz and the maximum is 65,535 Hz. *On the Arduino Uno, this function will interfere with Pulse Width Modulation [i.e., with the analogWrite() function] on Pins 3 and 11.*

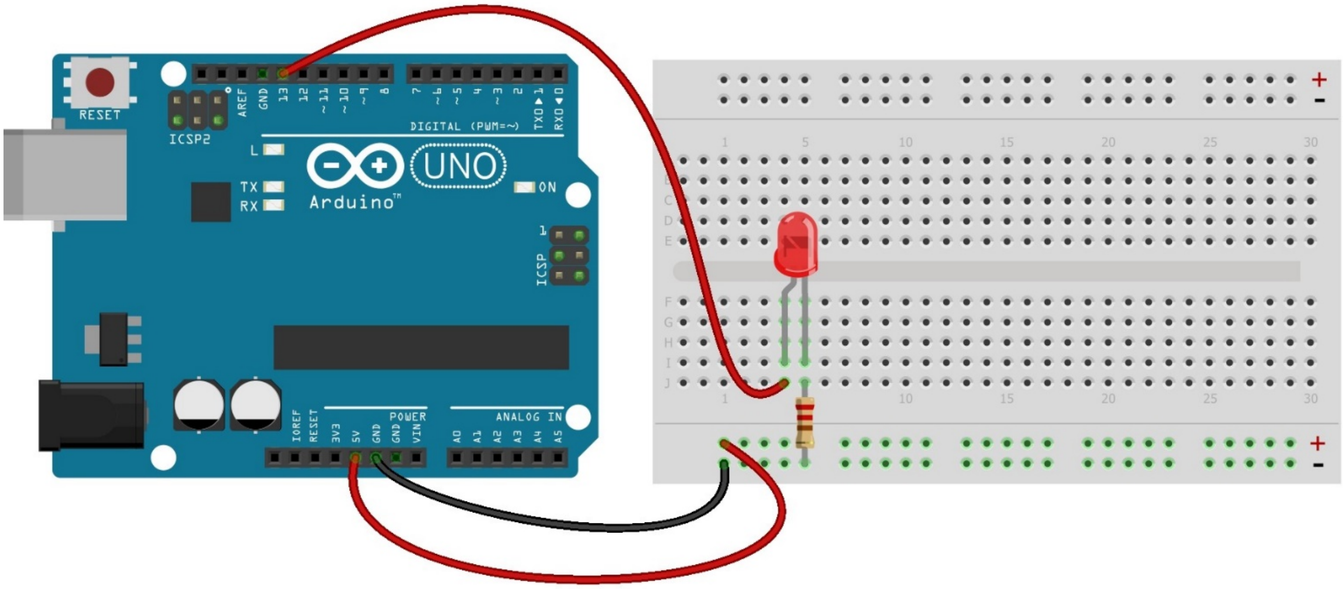
noTone(*pin*);

This just turns off the square wave generated by a previous tone() function on the designated pin.

Part 6:
Sample Sketches & Circuits

Blink

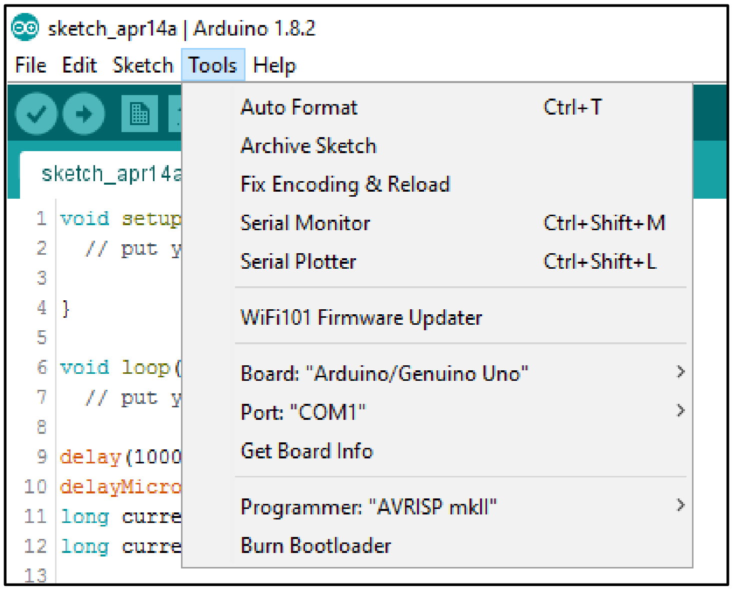
```
/
*
  Blink

  This sketch alternates, every second, between setting an output pin (Digital Pin
  13) HIGH (on) and LOW (off).  It is called “Blink” because it is assumed that an
  LED is attached to Pin 13. When the pin is HIGH, the LED will be lit; when the
  pin is LOW, the LED will be dark.

*/

void setup() {
  pinMode(13, OUTPUT);      // initialize digital pin 13 as an output.
}

void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```



"Blink" Circuit. Negative side of the LED is connected to GND through a 220-ohm resistor. Positive side of the LED is connected to Pin 13, which is pinMode (OUTPUT). When the sketch sets Pin 13 to HIGH, +5V is connected to the positive LED, completing the circuit and lighting the LED.

Reading Serial Strings

This sketch does not require any hardware beyond an Arduino connected to a PC. The sketch demonstrates the use of the Arduino IDE's serial monitor and a way for the Arduino sketch to communicate with the PC via the serial monitor. You can activate the serial monitor via the "Tools" tab in the IDE's menu bar.



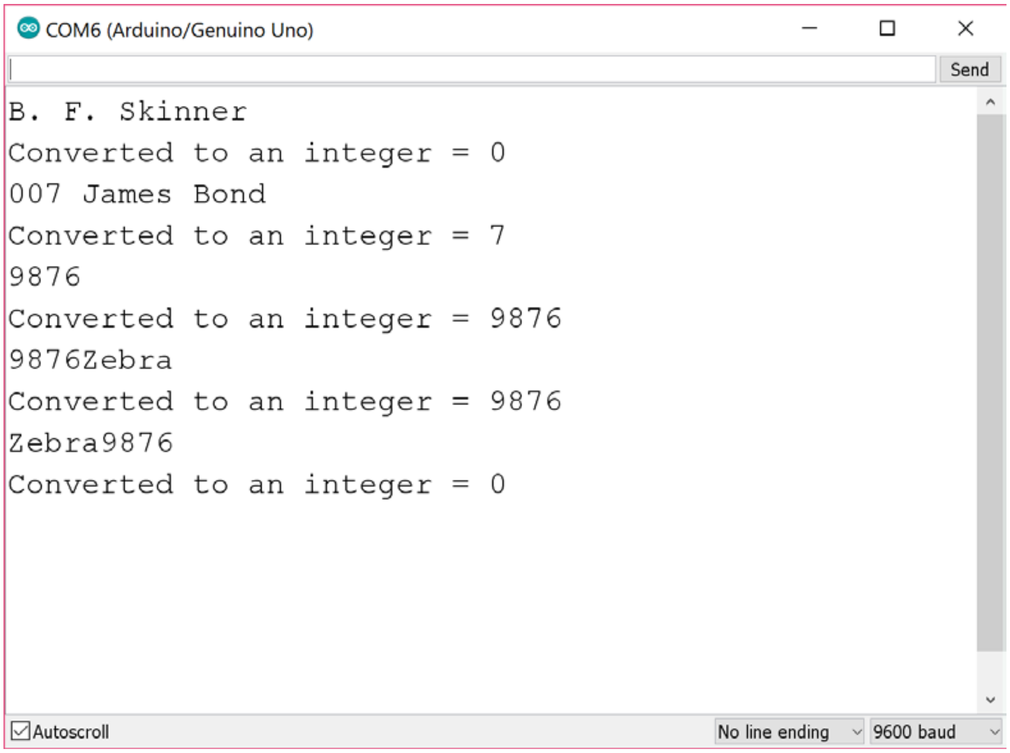
```
void setup
  // put y

}

void loop(
  // put y

delay(1000
delayMicro
long curre
long curre
```

```
/*
Reading_Serial_Strings

  This sketch demonstrates one way to:
  (a) receive a string that a user sends from the serial port;
  (b) if the string begins with a numeral, convert it to an integer;
  (c) send the string to the serial port;
  (d) send the integer to the serial port.

  Use the Serial Monitor to send strings to the sketch,
  as well as to see the results that the sketch sends back.

  Note use of these functions:

  Serial.begin - to establish serial communications
  Serial.setTimeout - to the set the amount of time the sketch will spend reading
     the serial port before moving on
  Serial.available - returns the number of characters in the serial input buffer
  toInt - converts a string that begins with a numeral to an integer integer
  Serial.readString - reads a string from the serial port until the
     timeout limit elapses (as set by Serial.setTimeout)
  Serial.print - sends characters to the serial port
  Serial.println - sends chacters to the serial port, followed by a line feed (so
     the next thing sent to the port is shown on a new line)

  Try sending these strings and see what happens:

  B. F. Skinner
  Will White
  007 James Bond
  9876
  9876Zebra
  Zebra9876

  M. Perone
  WVU Psychology Department
  February 1, 2017
*/
```

Listing continues…

Reading Serial Strings *cont'd*

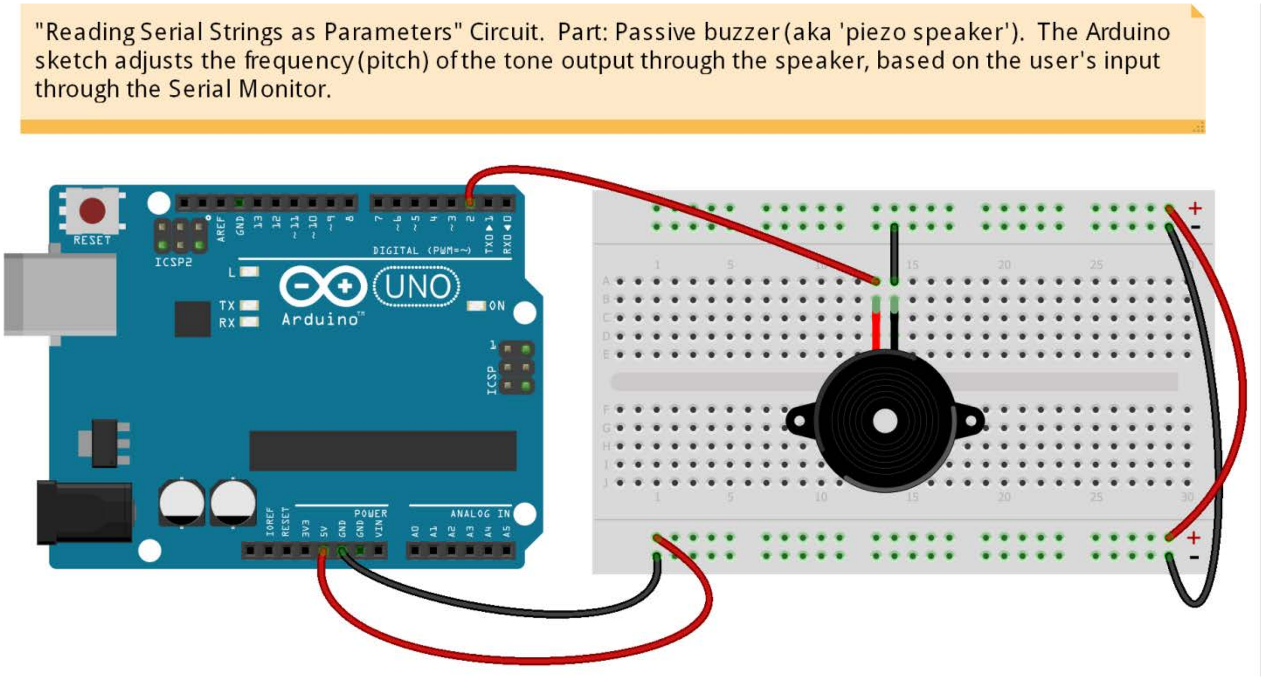
```
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  // set the time limit for reading a string to 10 ms
  // if you anticipate only short strings, a shorter limit
  // can be used; if you anticipate longer strings, the time
  // will need to be longer. play with the limit and the size
  // of the strings you send and see what happens.
  Serial.setTimeout(10);
}

void loop() {
  // are there any characters in the serial input buffer?
  if (Serial.available() > 0) {
    // yes, there are characters in the buffer, so...
    // ...read them into string variable
    String inputString = Serial.readString();
    // ...and send the contents of the string variable back out
    Serial.println(inputString);
    // if possible, convert the string to a number and store it
    // if the string does not beging with a numeral, a zero is returned
    long number = inputString.toInt();
    // send the result to the serial port
    Serial.print("Converted to an integer = ");
    Serial.println(number);
  }
}
```



```
B. F. Skinner
Converted to an integer = 0
007 James Bond
Converted to an integer = 7
9876
Converted to an integer = 9876
9876Zebra
Converted to an integer = 9876
Zebra9876
Converted to an integer = 0
```

Sample output

Reading Serial Strings as Parameters



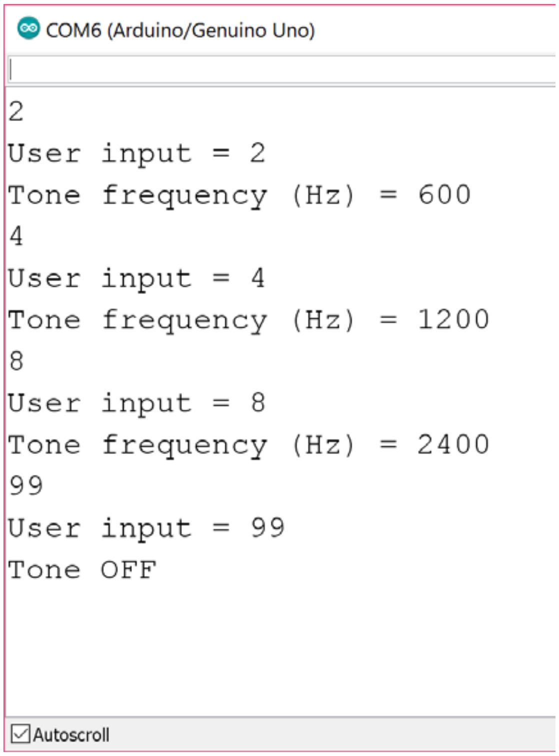
"Reading Serial Strings as Parameters" Circuit. Part: Passive buzzer (aka 'piezo speaker'). The Arduino sketch adjusts the frequency (pitch) of the tone output through the speaker, based on the user's input through the Serial Monitor.

```
/*
  Reading_Serial_Strings_as_Parameters

  This sketch demonstrates one way to:
  (a) receive a string that a user sends from the serial port;
  (b) if the string begins with a numeral, convert it to an integer;
  (c) use the integer to establish the frequency of a tone.

  This sketch extends "Reading_Serial_Strings." You can use the Serial Monitor
  to send strings to the sketch, and see the results that the sketch returns..

  Note use of these functions:
  Serial.begin - to establish serial communications
  Serial.setTimeout - to the set the amount of time the sketch will spend reading
     the serial port before moving on
  Serial.available - returns the number of characters in the serial input buffer
  toInt - converts a string that begins with a numeral to an integer
  Serial.readString - reads a string from the serial port until the timeout limit
     elapses (as set by Serial.setTimeout)
  Serial.print - sends characters to the serial port
  Serial.println - sends chacters to the serial port, followed by a line feed (so
     the next thing sent to the port is shown on a new line)
  tone - sends a tone of a certain frequency through a designated output pin
  noTone - turns off the tone at a designated pin

  Circuit Notes:
  Piezo speaker (passive buzzer):
  Connect negative pin to GND
  Connect positive pin to Digital Pin 2

  M. Perone
  WVU Psychology Department
  February 1, 2017
*/
```

Listing continue…

Reading Serial Strings as Parameters *cont'd*

```
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  // set the time limit for reading a string to 3 ms
  // if you anticipate only short strings, a short limit
  // can be used; if you anticipate longer strings, the time
  // will need to be longer. play with the limit and the size
  // of the strings you send and see what happens.
  Serial.setTimeout(3);
}

void loop() {
  // are there any characters in the serial input buffer?
  if (Serial.available() > 0) {
    // yes, there are characters in the buffer, so...
    // ...read them into string variable
    String inputString = Serial.readString();
    // ...and send the contents of the string variable back out
    Serial.println(inputString);
    // if possible, convert the string to a number and store it
    // if the string does not beging with a numeral, a zero is returned
    long number = inputString.toInt();
    // is the number between 1 and 10?
    if (number > 0 && number < 11) {
      // yes, convert to a frequency
      int frequency = number * 300;
      // play the tone @ specified frequency
      tone(2, frequency);
      // send some info to serial port
      Serial.print("User input = ");
      Serial.println(number);
      Serial.print("Tone frequency (Hz) = ");
      Serial.println(frequency);
    }
    else
    {
      // turn tone off
      noTone(2);
      // and send info
      Serial.print("User input = ");
      Serial.println(number);
      Serial.println("Tone OFF");
    }
  }
}
```



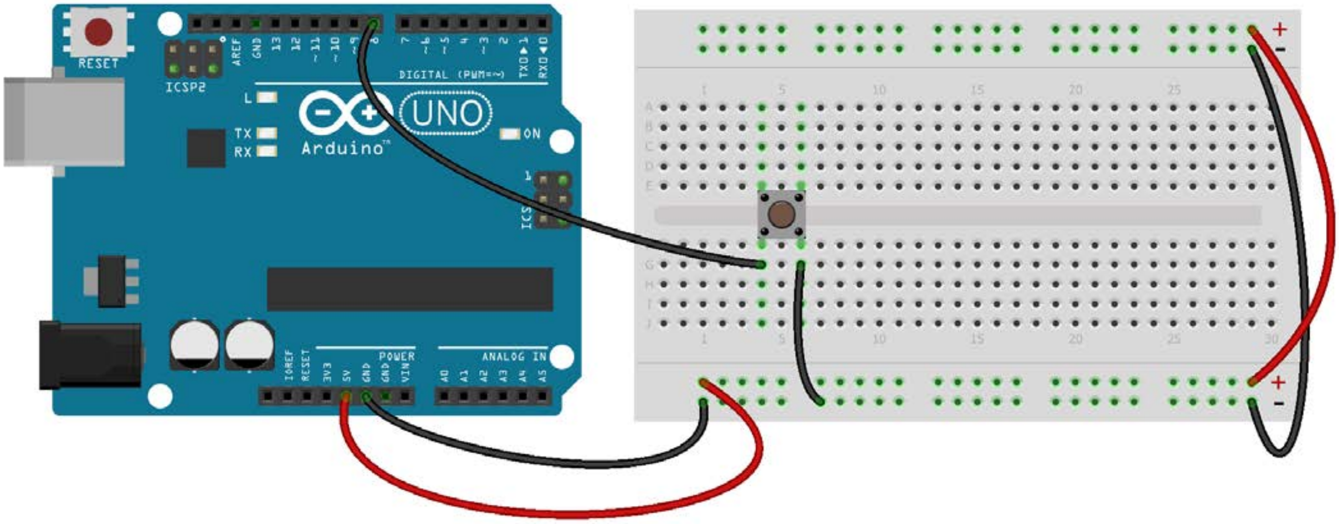
```
2
User input = 2
Tone frequency (Hz) = 600
4
User input = 4
Tone frequency (Hz) = 1200
8
User input = 8
Tone frequency (Hz) = 2400
99
User input = 99
Tone OFF
```

Sample output.

Count Button Presses

```
/*

Count_Button_Presses

This sketch illustrates how to count button presses or any the digital input).
We want to count each press just once: If the user holds the button down,
the sketch will see this thousands of times, but we only want to increment our
counter once.  The user must release the button and press it anew for the
response counter to be incremented.

Circuit Notes:

Button:
Connect one pin to GND
Connect the other pin to Digital Pin 8

Comments:

This sketch makes no effort to correct for bounce in the button's electical
contacts, and therefore extra presses may be counted.

M. Perone
WVU Psychology Dept
February 1, 2017

*/
```

"Count Button Presses" Circuit. Parts: Button. When pressed, the button provides GND to Pin 8 which is in pinMode (INPUT_PULLUP). The pin goes LOW when the button is pressed. The Arduino sketch counts the number of presses.



Count Button Presses *cont'd*

```
// define some integer variables
int respCount;
int newButtonState;
int oldButtonState;

void setup() {
  // Open a serial port
  Serial.begin(9600);
  // set Digital Pin 8 to input with internal pullup
  pinMode(8, INPUT_PULLUP);
}

void loop() {
  // read the state of the pin
  newButtonState = digitalRead(8);
  // has the state changed since our last read?
  if (newButtonState != oldButtonState) {
    // yes, the button state has changed so
    // make a note of it
    oldButtonState = newButtonState;
    // is the new state of the button
    // LOW, i.e., button is pressed?
    if (newButtonState == LOW){
      // yes, button is pressed, count it
      respCount = respCount + 1;
      // and send current count to serial port
      Serial.println(respCount);
    }
  }
}
```

Count Button Presses Debounced

This sketch uses the same circuit as "Count Button Presses"

```
/*

Count_Button_Presses_Debounced

This sketch illustrates how to count button presses or any the digital input).
We want to count each press just once: If the user holds the button down,
the sketch will see this thousands of times, but we only want to increment our
counter once.  The user must release the button and press it anew for the
response counter to be incremented.

This is an extension of "Count_Button_Presses." In this sketch,a correction is made
for possible bounce in the button's electical contacts.  The logic is s
straightforward. When we detect a change in the state of the button (LOW or
pressed, HIGH or released), we note the time.  Then, when we detect further changes
in the button state, we ignore them if they have occurred too soon after the last
recorded state change.  "Too soon" is operationalized in the debounceDelay
constant.

Circuit Notes:

Button:
Connect one pin to GND
Connect the other pin to Digital Pin 8

M. Perone
WVU Psychology Dept
February 1, 2017

*/
```

Listing continues…

Count Button Presses Debounced *cont'd*

```
// define some integer variables
int respCount;
int newButtonState;
int oldButtonState;
long lastButtonChangeTime;
const int debounceDelay = 3; // 3-ms debouce delay

void setup() {
  // Open a serial port
  Serial.begin(9600);
  // set Digital Pin 8 to input with internal pullup
  pinMode(8, INPUT_PULLUP);
}

void loop() {
  // read the state of the pin
  newButtonState = digitalRead(8);
  // is the state changed since our last read AND at least the debounce delay has
passed?
  if ((newButtonState != oldButtonState) && ((millis() - lastButtonChangeTime) >=
debounceDelay)){
    // Yes, the button state has changed and enough time has passed to pay
attention to the change
    // make a note of the change in state
    oldButtonState = newButtonState;
    // and make a note of the time of this change
    lastButtonChangeTime = millis();
    // is the new state of the button
    // LOW, i.e., button is pressed?
    if (newButtonState == LOW){
      // yes, button is pressed, count it
      respCount = respCount + 1;
      // and send current count to serial port
      Serial.println(respCount);
    }
  }
}
```

Adjustable Tone

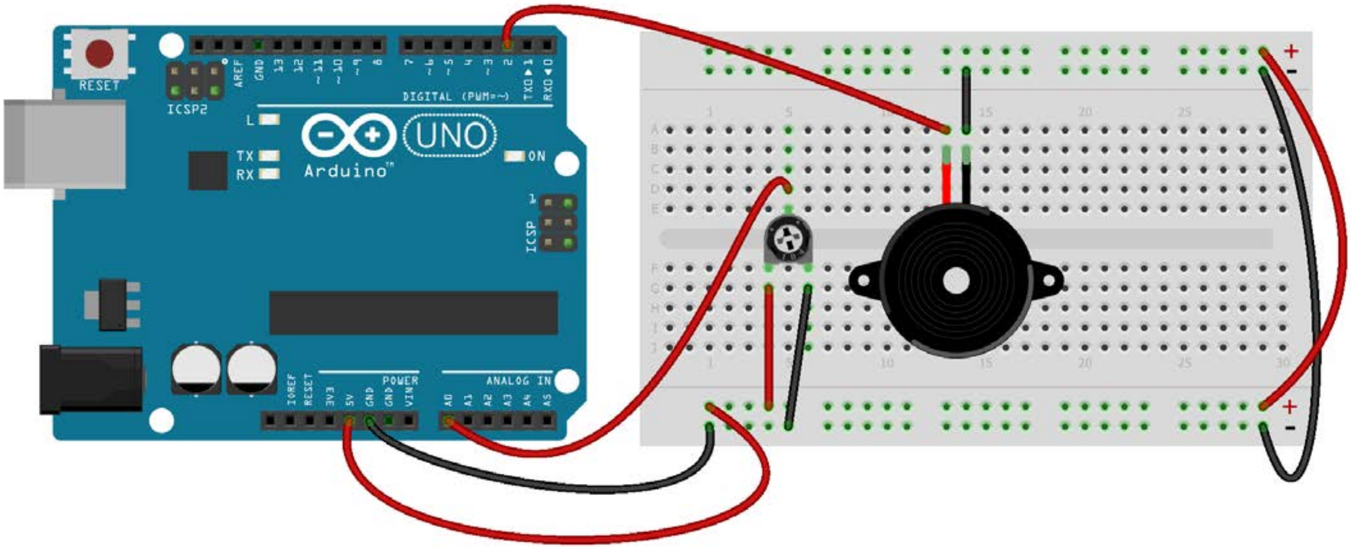
```
/*

  Adjustable_Tone

  This is a simple demonstration of analog input. A potentiometer provides
  input voltage to analog pin A0. The voltage is used to determine the
  frequency of a tone that is output to pin 2.

  Note use of these functions: analogRead, map, abs, tone.

  Circuit Notes:

  10K potentiometer:
  Connect 1 outer pin to GND and 1 outer pint to +5V;
  Connect middle pin to analog pin A0

  Piezo speaker (passive buzzer):
  Connect negative pin to GNS
  Connect positive pin to digital pin 2

  Comments:

  A certain amount of noise is inherent in analog input. A common
  way to compensate for this is to take multiple readings from the
  input pin and average them - a procedure called "smoothing."  In
  this sketch, I have taken a simpler approach.  After reading the
  voltage on the analog pin and converting it to a number between
  500 and 1500 (which will be used as my lowest and highest tone
  frequencies),I check to see if the number has changed by at least
  10 since the last reading.  If it has changed that much, I adjust
  the frequency of the tone; otherwise, I leave the tone as-is.  This
  eliminates (most) tiny fluctuations in tone that would arise from
  variablity in the voltage at the analog input.

  M. Perone
  WVU Psychology Dept
  January 26, 2017

*/
```

Listing continues…

Adjustable Tone *cont'd*

```
int voltage;           // voltage read from potentiometer connected to pin A0
int newToneFrequency; // frequency (Hz) of tone to be output
int oldToneFrequency; // last frequency (Hz) that was output
int change;            // to save change from old frequency to new frequency

void setup() {
  pinMode(A0, INPUT); // set analog pin A0 to input
}

void loop() {
  // read the voltage from analog input pin
  voltage = analogRead(A0);
  // convert voltage to a number between 500 and 1500
  // this will be new tone frequency
  newToneFrequency = map(voltage, 0, 1023, 500, 1500);
  // calculate difference between old and new frequencies
  change = newToneFrequency - oldToneFrequency;
  // convert change to absolute value (get rid of negative numbers)
  change = abs(change);
  // if new tone is at least 10 Hz different from old tone...
  if (change > 9) {
    // ... go ahead and adjust the frequency that is output to the speaker...
    tone(2, newToneFrequency);
    // ... and save the value in oldToneFrequency
    oldToneFrequency = newToneFrequency;
  }
}
```

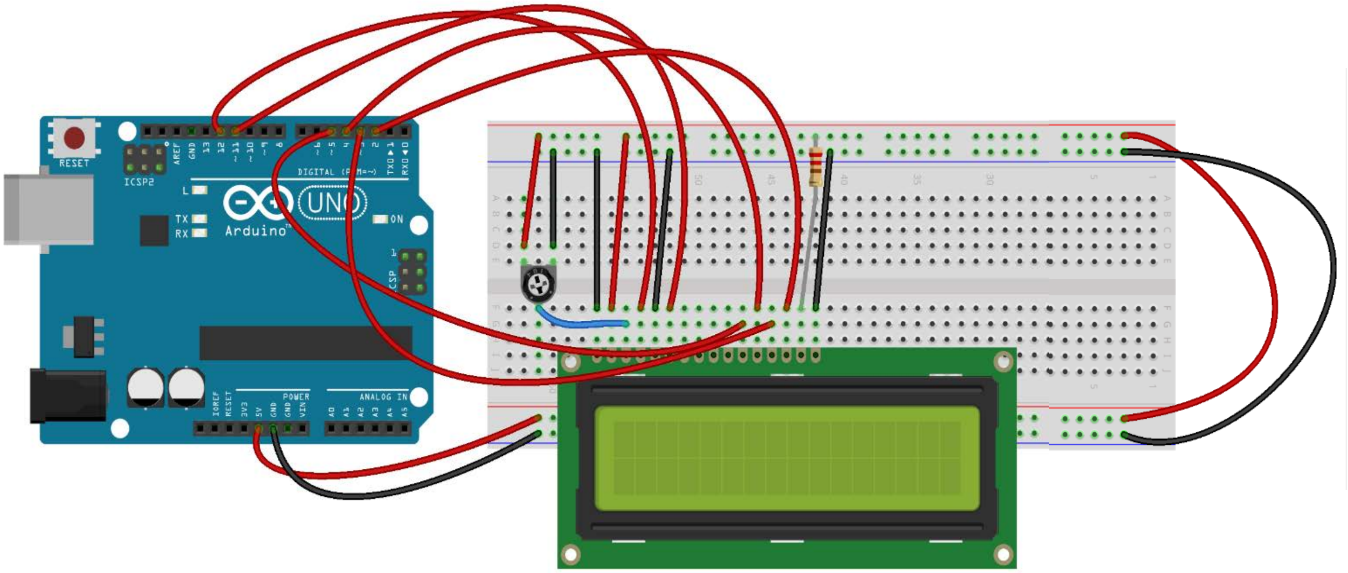
"Adjustable Tone" Circuit. Parts: 10K poteniometer, passive buzzer (aka 'piezo speaker'). The potentiometer is used for an analog input. As this input voltage changes, the Arduino sketch adjusts the frequency (pitch) of the tone output through the speaker.

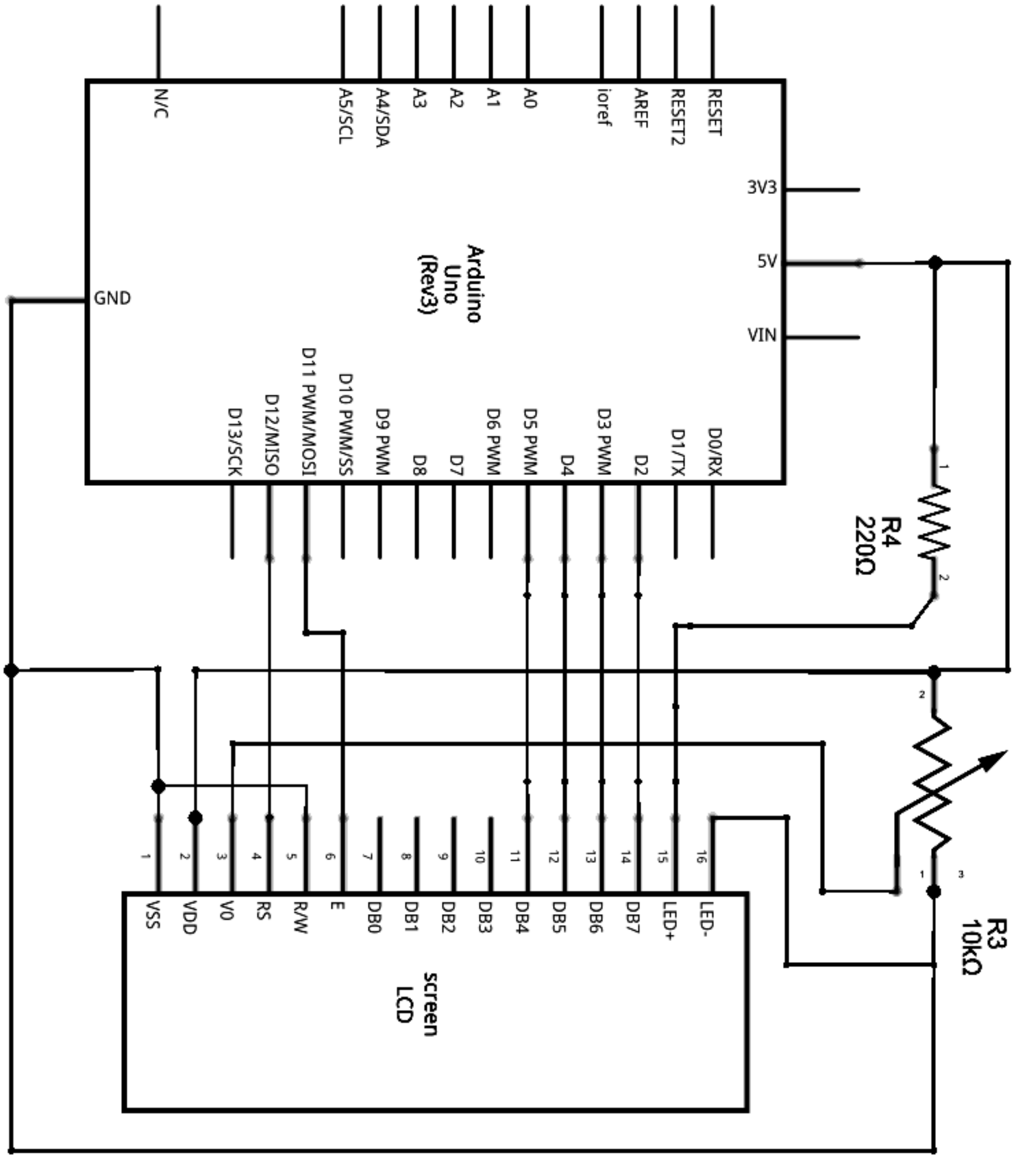


LCD Hello World

```
/*
  LCD Hello World

  Demonstrates the use a 16x2 LCD display. The LiquidCrystal library works with all
  LCD displays that are compatible with the Hitachi HD44780 driver. There are many
  of them out there, and you  can usually tell them by the 16-pin interface.
  This sketch prints "Hello World!" to the LCD in English, Spanish, Klingon, and
  Italian. It also shows the time that has elapsed since the Arduino was reset.

  Circuit Notes:

  The circuit is not complicated, but there are a lot of wires.  The LCD has 16
  pins; we will make connections with 12 of them.

   1. VSS: GND on breadboard
   2. VDD: +5V on breadboard
   3. VO: Wiper (output) of 10K potentiometer. Connect one input pin of the pot to
      GND (on breadboard) and the other to +5V (on breadboard). Turning the pot
      will adjust the contrast of the display.
   4. RS: Digital Output 12. This is the "Register Select" pin of the LCD.
   5. RW: GND (on breadboard).  This pin sets the mode to Read or Write.
   6. E: Digital Output 11. This is the "Enable" pin of the LCD.
   7. D0: not used
   8. D1: not used
   9. D2: not used
  10. D3: not used
  11. D4: Digital Output 5
  12. D5: Digital Output 4
  13. D6: Digital Output 3
  14: D7: Digital Output 2
  15: A: +5V (on breadboard) through a 220-ohm resistor
  16: K: GND on breadboard).  The A and K pins power the backlight.

  Comments:

  For the various versions of the "Hello World" message, I send 16 characters to
  the LCD. The last several are just spaces. This is to ensure that a new
  version completely overwrites the old version of the message.

  Note the use of "switch...case".  From the Arduino Online Reference: Like "if"
  statements,"switch...case" controls the flow of programs by allowing programmers
  to specify different code that should be executed in various conditions. In
  particular, a switch statement compares the value of a variable to the values
  specified in "case" statements. When a case statement is found whose value
  matches that of the variable, the code in that case statement is run. The "break"
  keyword exits the switch statement, and is typically used at the end of each
  case. Without a break statement, the switch statement will continue executing the
  following expressions "falling-through") until a break, or the end of the switch
  statement is reached.

  Library originally added 18 Apr 2008 by David A. Mellis
  Library modified 5 Jul 2009 by Limor Fried (http://www.ladyada.net)
  Example added 9 Jul 2009 by Tom Igoe; Modified 22 Nov 2010 by Tom Igoe
  Modified 4 & 11 Feb 2017 by M. Perone, WVU Psychology Dept

  Previous code that formed the basis of this sketch is in the public domain:
  http://www.arduino.cc/en/Tutorial/LiquidCrystal
*/
```

LCD Hello World *cont'd*





LCD Hello World *cont'd*

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

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {
  // set up the LCD's number of characters per row, and rows:
  lcd.begin(16, 2);
  // Print a message to the LCD.
  lcd.print("Hello World!    ");
}

void loop() {
  // set the cursor to column 0, line 1
  // (note: line 1 is the second row, since counting begins with 0):
  lcd.setCursor(0, 1);
  // construct a string to show seconds since reset:
  String message = "Elapsed: ";
  message = message + (millis() / 1000) + " s";
  lcd.print(message);
  // If 5 s has elapsed, "Hello World!" is translated to
  // Spanish.  At 10 s, the translation is to Klingon(!), etc.
  long elapsedTime = millis() / 1000; // elapsed time in s
  switch (elapsedTime) {
    case 5:
      // position the cursor to Position 0, Line 0
      lcd.setCursor(0, 0);
      // print 16 characters
      lcd.print("Hola Mundo!     ");
      // that's the end of the code for this case
      break;
    case 10:
      lcd.setCursor(0, 0);
      lcd.print("Qo' Vlvan!      ");
      break;
    case 15:
      lcd.setCursor(0, 0);
      lcd.print("Salve Mondo!    ");
      break;
  }
}
```

LCD Recycling Hello World

This sketch uses the same circuit as "LCD Hello World"

```
/*
  LCD Hello World

  Demonstrates the use a 16x2 LCD display. The LiquidCrystal library works with all
  LCD displays that are compatible with the Hitachi HD44780 driver. There are many
  of them out there, and you can usually tell them by the 16-pin interface.
  This sketch prints "Hello World!" to the LCD in English, Spanish, Klingon, and
  Italian, and then recycles back to English. It also shows the time that has
  elapsed since the Arduino was reset.

  Circuit Notes:

  The circuit is not complicated, but there are a lot of wires.  The LCD has 16
  pins; we will make connections with 12 of them.

   1. VSS: GND on breadboard
   2. VDD: +5V on breadboard
   3. VO: Wiper (output) of 10K potentiometer. Connect one input pin of the pot to
      GND (on breadboard) and the other to +5V (on breadboard). Turning the pot
      will adjust the contrast of the display.
   4. RS: Digital Output 12. This is the "Register Select" pin of the LCD.
   5. RW: GND (on breadboard).  This pin sets the mode to Read or Write.
   6. E: Digital Output 11. This is the "Enable" pin of the LCD.
   7. D0: not used
   8. D1: not used
   9. D2: not used
  10. D3: not used
  11. D4: Digital Output 5
  12. D5: Digital Output 4
  13. D6: Digital Output 3
  14: D7: Digital Output 2
  15: A: +5V (on breadboard) through a 220-ohm resistor
  16: K: GND on breadboard).  The A and K pins power the backlight.

  Comments:

  For the various versions of the "Hello World" message, I send 16 characters to
  the LCD. The last several are just spaces. This is to ensure that a new
  version completely overwrites the old version of the message.

  Note the use of "switch...case".  From the Arduino Online Reference: Like "if"
  statements,"switch...case" controls the flow of programs by allowing programmers
  to specify different code that should be executed in various conditions. In
  particular, a switch statement compares the value of a variable to the values
  specified in "case" statements. When a case statement is found whose value
  matches that of the variable, the code in that case statement is run. The "break"
  keyword exits the switch statement, and is typically used at the end of each
  case. Without a break statement, the switch statement will continue executing the
  following expressions "falling-through") until a break, or the end of the switch
  statement is reached.

  Library originally added 18 Apr 2008 by David A. Mellis
  Library modified 5 Jul 2009 by Limor Fried (http://www.ladyada.net)
  Example added 9 Jul 2009 by Tom Igoe; Modified 22 Nov 2010 by Tom Igoe
  Modified 4 & 11 Feb 2017 by M. Perone, WVU Psychology Dept

  Previous code that formed the basis of this sketch is in the public domain:
  http://www.arduino.cc/en/Tutorial/LiquidCrystal
```

```
*/
```

LCD Recycling Hello World *cont'd*

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

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

// establish a variable to keep track of the message cycle
long cycleStartTime;

void setup() {
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  // Print a message to the LCD.
  lcd.print("Hello World!    ");
}

void loop() {
  // set the cursor to column 0, line 1
  // (note: line 1 is the second row, since counting begins with 0):
  lcd.setCursor(0, 1);
  // construct a string to show seconds since reset:
  String message = "Elapsed: ";
  message = message + (millis() / 1000) + " s";
  lcd.print(message);
  // If 5 s has elapsed, "Hello World!" is translated to
  // Spanish.  At 10 s, the translation is to Klingon(!), etc.
  // Get the current time and convert to seconds
  long elapsedTime = (millis() - cycleStartTime) / 1000;
  // At 5, 10, 15 s, print greeting to LCD
  switch (elapsedTime) {
    case 5:
      // position the cursor to Position 0, Line 0
      lcd.setCursor(0, 0);
      // print 16 characters
      lcd.print("Hola Mundo!     ");
      // that's the end of the code for this case
      break;
    case 10:
      lcd.setCursor(0, 0);
      lcd.print("Qo' Vlvan!      ");
      break;
    case 15:
      lcd.setCursor(0, 0);
      lcd.print("Salve Mondo!    ");
      break;
    case 20:
      // start over
      // show English greeting
      lcd.setCursor(0, 0);
      lcd.print("Hello World!");
      // record the start time of this cycle
      cycleStartTime = millis();
      break;
  }
}
```

Thermistor

```
/*
  Thermistor

  This sketch performs an analog read on a thermistor connected
  to Arduino Analog Pin 1.  It converts the raw reading to voltage.
  The raw reading can be from 0-1023 and the voltage from 0.0 to 5.0.
  The sketch sends the raw reading and the voltage to the serial
  port twice per second.

  A thermistor is a temperature-sensitive variable resistor. As the
  temperature of the thermistor is raised, the thermistor's resistance is
  lowered. This in turn leads to an increase in the voltage received from
  the thermistor. So there is a direct relation between the temperature
  and the voltage.

  A thermistor is wired in the same way as a potentiometer. One lead is connected
  To V.  The other lead is connected to an analog input (A1 in the sketch) AND to
  Ground via a 10K-ohm resistor.  The only difference is this: With a
  potentiometer, resistance is controlled by turning a knob, but with a thermistor,
  resistance is controlled by heat.

  Circuit notes:
  1. Thermistor Lead 1 to 5v (lead numbering is arbitrary)
  2. Thermistor = Lead 2 to Analog Input 1
  3. Thermistor Lead 2 (again!) to GND via 10K-ohm resistor

  Additiona factors are involved in converting the thermistor readings to
  temperature. A comprehensive tutorial (based on a nicer thermistor than the
  one in our kit) is here: https://learn.adafruit.com/thermistor/overview. We
  won't bother about this now because we have an easier way to read temperature
  using the DHT11, a device that directly returns temperature (and humidity).

  M. Perone, WVU Psyc Department, Mar 18, 2017, Mar 31, 2017
*/

int thermistorPin = 1; // refers to the Arduino pin so it has a nice name

void setup() {
  Serial.begin(9600); // set up serial communications
}

void loop() {
  int thermReading = analogRead(thermistorPin); // get input from thermistor
  float thermV = (thermReading * 5.0) / 1023.0; // convert to voltage
  Serial.print("Raw: "); // show the results...
  Serial.print(thermReading);
  Serial.print(",  Volts: ");
  Serial.println(thermV);
  delay(500); // wait a half-sec before next reading
}
```

Joystick Simple

```
/*

  Joystick_Simple

  This sketch reads the switch, x-axis, and y-axis of a joystick and sends the
  readings to the serial port. Readings are made and sent every 50 ms.

  It appears that the joystick board includes debouncing circuitry, so no need to
  handle that in software.

  Circuit Notes (I recommend that all connections be made via the breadboard):

  1. Joystick GND to GND
  2. Joystick +5V to 5V
  3. Joystick VRx to Pin A0
  4. Joystick VRy to Pin A1
  5. Joystick SW to Pin D2

  M. Perone, WVU Psyc Dept, Feb 18, 2017

*/

// Arduino pin numbers
const int switchPin = 2; // digital pin connected to switch output
const int xPin = 0; // analog pin connected to X output
const int yPin = 1; // analog pin connected to Y output

void setup() {
  pinMode(switchPin, INPUT_PULLUP);
  Serial.begin(9600);
}

void loop() {
  Serial.print("Sw:  ");
  Serial.print(digitalRead(switchPin));
  Serial.print("  X: ");
  Serial.print(analogRead(xPin));
  Serial.print("  Y: ");
  Serial.println(analogRead(yPin));
  delay(50);
}
```

Joystick Refined

```
/*

  Joystick_Refined

  This sketch reads the switch, x-axis, and y-axis of a joystick and sends the
  readings to the serial port - but only if the readings have changed.  Any change
  in the switch is reported; changes in x and y are reported only if the change
  exceeds some minimum value as designed in the minimumChange constant below.

  It appears that the joystick board includes debouncing circuitry, so no need to
  handle that in software.

  Note the use of the "displayStickStatus" function. For more information about
  This structured coding technique, see
  https://www.arduino.cc/en/Reference/FunctionDeclaration

  Circuit Notes (I recommend that all connections be made via the breadboard):

  1. Joystick GND to GND
  2. Joystick +5V to 5V
  3. Joystick VRx to Pin A0
  4. Joystick VRy to Pin A1
  5. Joystick SW to Pin D2

  M. Perone, WVU Psyc Dept, Feb 18, 2017

*/
```

Listing continues…

Joystick Refined *cont'd*

```
// Arduino pin numbers
const int switchPin = 2; // digital pin connected to switch output
const int xPin = 0; // analog pin connected to X output
const int yPin = 1; // analog pin connected to Y output

// To track changes in stick position and switch
int oldSwitch;
int newSwitch;
int oldX;
int newX;
int oldY;
int newY;
const int minimumChange = 3; // stick must change this much to trigger display
update

void setup() {
  pinMode(switchPin, INPUT_PULLUP);
  // next 3 lines: baseline reading of the stick
  oldSwitch = digitalRead(switchPin);
  oldX = analogRead(xPin);
  oldY = analogRead(yPin);
  Serial.begin(9600); // establish serial communication
}

void displayStickStatus() {
  // This function updates the display
  Serial.print("Sw: ");
  Serial.print(newSwitch);
  Serial.print("  X: ");
  Serial.print(newX);
  Serial.print("  Y: ");
  Serial.println(newY);
}

void loop() {
  int unsigned diff; // to track changes in X, Y coordinates
  // switch - no need to debounce this fancy switch
  newSwitch = digitalRead(switchPin);
  if (newSwitch != oldSwitch) {
    oldSwitch = newSwitch;
    displayStickStatus();
  }
  // X
  newX = analogRead(xPin);
  diff = newX - oldX;
  if (diff >= minimumChange) {
    oldX = newX;
    displayStickStatus();
  }
  // Y
  newY = analogRead(yPin);
  diff = newY - oldY;
  if (diff >= minimumChange) {
    oldY = newY;
    displayStickStatus();
  }
}
```

Joystick RGB LED

```
/*

  Joystick_RGB_LED

  This sketch reads the switch, x-axis, and y-axis of a joystick and sends the
  readings to the serial port - but only if the readings have changed.  Any change
  in the switch is reported; changes in x and y are reported only if the change
  exceeds some minimum value as designed in the minimumChange constant below.

  The X,Y values are used to adjust the red and blue leds within an RGB LED.

  It appears that the joystick board includes debouncing circuitry, so no need to
  handle that in software.

  Note the use of the "displayStickStatus" and adjustColor functions. For more
  information about structured coding techniques, see
  https://www.arduino.cc/en/Reference/FunctionDeclaration

  Circuit Notes (I recommend that all connections be made via the breadboard):

  Joystick GND to GND
  Joystick +5V to 5V
  Joystick VRx to Pin A0
  Joystick VRy to Pin A1
  Joystick SW to Pin D2

  Orient RGB LED so that the longest leg is the second from the left. Then make
  these connections from left to right:
  1. (Leftmost pin) to Digital Pin 3 via a 220-ohm resistor
  2. (Longest leg) to GND
  3. to Digital Pin 5 via a 220-ohm resistor
  4. to Digital Pin 6 via a 220-ohm resistor

  M. Perone, WVU Psyc Dept, Feb 18, 2017

*/


// Arduino pin numbers; note only 2 of 3 "color" outputs are used
const int switchPin = 2; // digital pin connected to switch output
const int xPin = 0; // analog pin connected to X output
const int yPin = 1; // analog pin connected to Y output
const int redPin = 3; // Digital Pin 3 supports PWM
const int greenPin = 5; // Digital Pin 5 supports PWM
const int bluePin = 6; // Digital Pin 6 supports PWM

// To track changes in stick position and switch
int oldSwitch;
int newSwitch;
int oldX;
int newX;
int oldY;
int newY;
const int minimumChange = 5; // stick must change this much to trigger display
update
```

Listing continues…

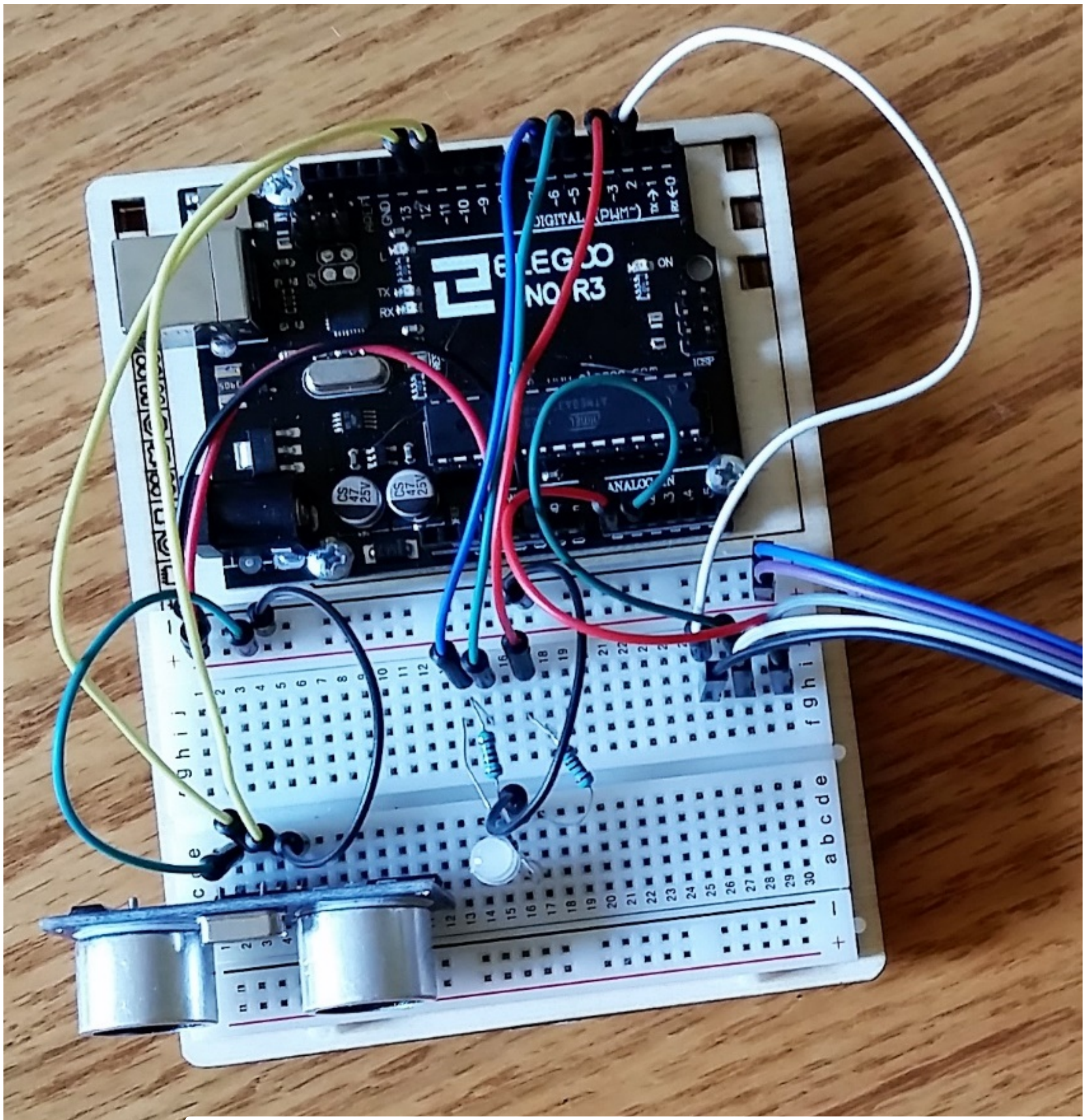
Joystick RGB LED *cont’d*

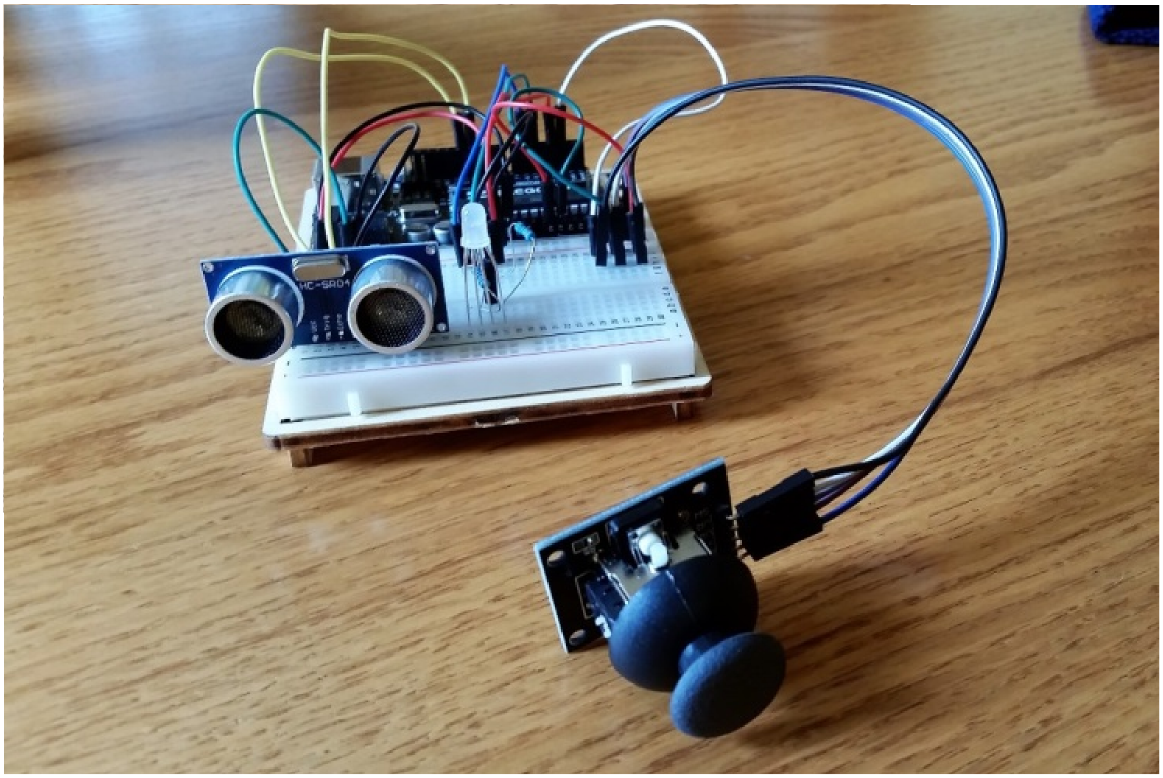
```
void setup() {
  pinMode(switchPin, INPUT_PULLUP);
  // next 3 lines: baseline reading of the stick
  oldSwitch = digitalRead(switchPin);
  oldX = analogRead(xPin);
  oldY = analogRead(yPin);
  // get RGB LED going...
  analogWrite(redPin,125);
  analogWrite(bluePin,125);
  analogWrite(greenPin,0); // not used
  Serial.begin(9600); // establish serial communication
}

void displayStickStatus() {
  // This function updates the display
  Serial.print("Sw: ");
  Serial.print(newSwitch);
  Serial.print("  X: ");
  Serial.print(newX);
  Serial.print("  Y: ");
  Serial.println(newY);
}

void adjustColor(){
    // Note use of "map" function, which works like this:
    //sensorValue = map(sensorValue, sensorMin, sensorMax, 0, 255);
    int redValue = map(newX, 0, 1023, 0, 255);
    int blueValue = map(newY,0,1023,0,255);
    analogWrite(redPin,redValue);
    analogWrite(bluePin,blueValue);
}

void loop() {
  int unsigned diff; // to track changes in X, Y coordinates
  // switch - no need to debounce this fancy switch
  newSwitch = digitalRead(switchPin);
  if (newSwitch != oldSwitch) {
    oldSwitch = newSwitch;
    displayStickStatus();
  }
  // X
  newX = analogRead(xPin);
  diff = newX - oldX;
  if (diff >= minimumChange) {
    oldX = newX;
    displayStickStatus();
    adjustColor();
  }
  // Y
  newY = analogRead(yPin);
  diff = newY - oldY;
  if (diff >= minimumChange) {
    oldY = newY;
    displayStickStatus();
    adjustColor();
  }
}
```

Joystick Ultrasonic RGB LED

```
/*

  Joystick_Ultrasonic_RGB_LED

  This sketch reads the switch, x-axis, and y-axis of a joystick and sends the
  readings to the serial port - but only if the readings have changed.  The X,Y
  values are used to adjust the red and blue leds within an RGB LED.

  The sketch pings an ultrasonic sensor every 50 ms. If an object is within 20 cm,
  the LED is turned off.

  Note the use of the "displayStickStatus", "adjustColor", and "LED" functions. For
  more information aboutstructured coding techniques, see
  https://www.arduino.cc/en/Reference/FunctionDeclaration

  Circuit Notes (I recommend that all connections be made via the breadboard):

  Joystick GND to GND
  Joystick +5V to 5V
  Joystick VRx to Pin A0
  Joystick VRy to Pin A1
  Joystick SW to Pin D2

  Orient RGB LED so that the longest leg is the second from the left. Then make
  these connections from left to right:
  1. (Leftmost pin) to Digital Pin 3 via a 220-ohm resistor
  2. (Longest leg) to GND
  3. to Digital Pin 5 via a 220-ohm resistor
  4. to Digital Pin 6 via a 220-ohm resistor

  Ultrasonic Sensor VCC to 5V
  Ultrasonic Sensor Trig to Digital Pin 12
  Ultrasonic Sensor Echo to Digital Pin 11
  Ultrasonic Sensor GND to GND

  M. Perone, WVU Psyc Dept, Feb 18, 2017

*/

// FOR ULTRASONIC SENSOR
#include <NewPing.h>      // Library for ultrasonic senSors
#define TRIGGER_PIN  12   // Arduino pin tied to trigger pin on the ultrasonic
sensor.
#define ECHO_PIN     11   // Arduino pin tied to echo pin on the ultrasonic sensor.
#define MAX_DISTANCE 300  // Maximum distance we want to ping for (in centimeters).
// Maximum sensor distance is rated at 400-500cm.
NewPing sonar(TRIGGER_PIN, ECHO_PIN, MAX_DISTANCE); // NewPing setup of pins and
maximum distance.
long lastPingTime;
int newSonicDistance;
const int minimumCentimeters = 20; // if something gets closer than this we turn
off LED
```

Listing continues…


ELEGOO
UNO R3
DIGITAL (PWM~)
ANALOG IN


HC-SR04

Joystick Ultrasonic RGB LED *cont'd*

```
// FOR JOYSTICK
const int switchPin = 2; // digital pin connected to switch output
const int xPin = 0; // analog pin connected to X output
const int yPin = 1; // analog pin connected to Y output
// To track changes in stick position and switch
int oldSwitch;
int newSwitch;
int oldX;
int newX;
int oldY;
int newY;
const int minimumChange = 5; // stick must change this much to trigger display
update

// FOR RGB LED (note: only the red and blue pins are actually used)
const int redPin = 3; // Digital Pin 3 supports PWM
const int greenPin = 5; // Digital Pin 5 supports PWM
const int bluePin = 6; // Digital Pin 6 supports PWM

void setup() {
  pinMode(switchPin, INPUT_PULLUP);
  // next 3 lines: baseline reading of the stick
  oldSwitch = digitalRead(switchPin);
  oldX = analogRead(xPin);
  oldY = analogRead(yPin);
  // get RGB LED going...
  analogWrite(redPin, 125);
  analogWrite(bluePin, 125);
  analogWrite(greenPin, 0); // not used
  Serial.begin(9600); // establish serial communication
}

// FUNCTION TO SEND DATA TO SERIAL PORT
void displayStickStatus() {
  Serial.print("Sw: ");
  Serial.print(newSwitch);
  Serial.print("  X: ");
  Serial.print(newX);
  Serial.print("  Y: ");
  Serial.print(newY);
  Serial.print(" Dist: ");
  Serial.println(newSonicDistance);
}

// FUNCTION TO CONTROL COLOR OF RGB LED
void adjustColor() {
  // Note use of "map" function, which works like this:
  //sensorValue = map(sensorValue, sensorMin, sensorMax, 0, 255);
  int redValue = map(newX, 0, 1023, 0, 255);
  int blueValue = map(newY, 0, 1023, 0, 255);
  analogWrite(redPin, redValue);
  analogWrite(bluePin, blueValue);
}
```

Listing continues…

Joystick Ultrasonic RGB LED *cont'd*

```
// FUNCTION TO TURN RGB LED ON AND OFF
// Note that it returns "true" if LED is on and "false" is LED is off.
bool LED(String status) {
  if (status == "Off") {
    analogWrite(redPin, 0);
    analogWrite(greenPin, 0);
    analogWrite(bluePin, 0);
    return false;
  }
  else
  {
    adjustColor();
    return true;
  }
}

void loop() {
  int unsigned diff; // to track changes in X, Y coordinates
  bool statusLED;
  // Ping the ultrasonic sensor every 50 ms
  if ((millis() - lastPingTime) > 49) {
    newSonicDistance = sonar.ping_cm(); // get distance in cm
    lastPingTime = millis();
    if (newSonicDistance < minimumCentimeters) {
      statusLED = LED("Off");
    }
    else
    {
      statusLED = LED("On");
    }
  }
  // switch - no need to debounce this fancy switch
  newSwitch = digitalRead(switchPin);
  if (newSwitch != oldSwitch) {
    oldSwitch = newSwitch;
    displayStickStatus();
  }
  // X
  newX = analogRead(xPin);
  diff = newX - oldX;
  if (diff >= minimumChange) {
    oldX = newX;
    displayStickStatus();
    if (statusLED == true) {
      adjustColor();
    }
  }
  // Y
  newY = analogRead(yPin);
  diff = newY - oldY;
  if (diff >= minimumChange) {
    oldY = newY;
    displayStickStatus();
    if (statusLED == true) {
      adjustColor();
    }
  }
}
```

Servo Sweep

```
/*

Servo_Sweep

This sketch moves a servo through its full range of motion, from
0 to 180 degrees, back and forth.  The number of degrees in each
change of position and the pause between each change of position
are stored in variables "degreeChange" and "pauseDuration".

Circuit Notes:

Servo brown wire to GND
Servo red wire to 5V
Servo orange wire to Digital Output 9

M. Perone, WVU Psyc Dept, Feb 24, 2017

 */

#include <Servo.h> //invoke servo librarry
Servo myservo;//create servo object to control a servo
int degreeChange = 5; // for incrementing or decrementing servo position
int pauseDuration = 200; // time between increments/decrements

void setup() {
  Serial.begin(9600);
  myservo.attach(9);//attach the servo on pin 9 to servo object
  myservo.write(0);//back to 0 degrees
  delay(2000);//wait a couple seconds
}

void loop() {
  // this for-loop increments the degrees from 0 to 180
  for (int degrees = 0; degrees < 180; degrees = degrees + degreeChange) {
    myservo.write(degrees); // move the servo to the designated postion
    Serial.print("Up "); // send position info to serial monitor
    Serial.println(degrees);
    delay (pauseDuration);
  }
  // this for-loop decrements from 180 to 0
  for (int degrees = 180; degrees > 0; degrees = degrees - degreeChange) {
    myservo.write(degrees); // move the servo to the designated position
    Serial.print("Down "); // send position info to serial montior
    Serial.println(degrees);
    delay (pauseDuration);
  }
}
```

Photocell

```
/*

  Photocell

  This sketch performs an analog read on a photocell - also known
  as a photoresistor - connected to Arduino Analog Pin 0.  It converts
  the raw reading to voltage. The raw reading can be from 0-1023 and
  the voltage from 0.0 to 5.0. The sketch sends the raw reading and the
  voltage to the serial port twice per second.

  A photoresistor is a light-sensitive variable resistor. As the intensity
  of the light falling on the photoresister is raised, the device's resistance is
  lowered. This in turn leads to an increase in the voltage received from
  the device. So there is a direct relation between the light intensity
  and the voltage.

  A photocell is wired in the same way as a potentiometer. One lead is connected to
  5V. The other lead is connected to an analog input (A0 in the sketch) AND to
  Ground via a 1K-ohm resistor.  The only difference is this: With a potentiometer
  Resistance is controlled by turning a knob, but with a photoresistor, resistance
  is controlled by the intensity of light.

  Circuit notes:

  1. Photocell Lead 1 to 5v (lead numbering is arbitrary)
  2. Photocell Lead 2 to Analog Input 0
  3. Photocell Lead 2 to GND via 1K resistor

  M. Perone, WVU Psyc Dept, Mar 19, 2017

*/

int lightPin = 0; // Arduino analog input to receive photocell

void setup() {
  Serial.begin(9600); // set up serial communication
}

void loop() {
  int reading  = analogRead(lightPin); // read photocell
  float photoV = (reading * 5.0) / 1023.0; // convert to voltage
  Serial.print("Raw: "); // show the results...
  Serial.print(reading);
  Serial.print(",  Volts: ");
  Serial.println(photoV);
  delay(500); // wait a bit
}
```

Photocell Response Count

```
/*

  Photocell_Response_Count

  This sketch increments a response counter every time a photobeam is broken.
  The circuit should have a white LED aimed at a photocell, without enough
  space in between them to allow a piece of cardboard to be swiped through the
  space. If the photocell reading drops sufficiently a reponse is counted. The
  threshold for counting a response is in the constant 'detectionThreshold'.

  The code also has some debouncing code.  This is because as an object passed
  Between the LED and the photocell, more than one reduced reading may be
  detected.  This problem is prevented by ignoring changes in the readings that
  occur "too soon" after the last recorded change. The constant 'debounceInterval'
  defines what is "too soon."

  A photocell is wired in the same way as a potentiometer. One lead is connected to
  5V. The other lead is connected to an analog input (A0 in the sketch) AND to
  Ground via a 1K-ohm resistor.

  Circuit notes:
  1. Photocell Lead 1 to 5v (lead numbering is arbitrary)
  2. Photocell Lead 2 to Analog Input 0
  3. Photocell Lead 2 to GND via 1K resistor
  4. LED Positive Lead (the longer one) to Digital Output 8
  5. LED Negative Lead (the shorter one) to GND via a 330-ohm resistor.

  M. Perone, WVU Psyc Dept, Mar 19, 2017

*/
```

Listing continues…

Photocell Response Count *cont'd*

```
int lightPin = 0; // analog input receiving photocell
int lightSource = 8; // digital output to control LED
int oldReading; // to keep tack of last photocell reading
int newReading; // to store current photocell reading
const int detectionThreshold = 150; // required reduction in reading to count
response
int responseCounter; // to count responses
int lastResponseTime; // for debounce
const int debounceInterval = 300; // for debounce

void setup() {
  pinMode(lightSource, OUTPUT); // set output pin mode
  digitalWrite(lightSource, HIGH); // turn on light source
  delay (100);// give it time to power up
  // next 2 lines: initialize photocell reading variables
  newReading = analogRead(lightPin); // read photocell
  oldReading = newReading;
  Serial.begin(9600); // set up serial communication
}

void loop() {
  newReading = analogRead(lightPin); // read photocell
  int difference = abs(oldReading - newReading);
  if ((difference > detectionThreshold) && ((millis() - lastResponseTime) >=
debounceInterval)) {
    // pay heed if reading has changed enough AND enough time has passed since last
change
    if (newReading < oldReading) { // and because reading has dropped, count a
response
      responseCounter = responseCounter + 1; // increment the counter
      // Next 6 lines: Send data to serial port
      Serial.print("Old: ");
      Serial.print(oldReading);
      Serial.print(", New: ");
      Serial.print(newReading);
      Serial.print(", Responses: ");
      Serial.println(responseCounter);
    }
    // save reading and time of this recorded change
    oldReading = newReading;
    lastResponseTime = millis();
  }
}
```

Two Buttons

The circuit for this sketch is a simple extension of the circuit for "Count Button Presses." You just add another button!

```
/*

  Two Buttons

  This sketch scans 2 digital input ports and detects debounced inputs. A
  correction is made for possible bounce in the button's electical contacts. All of
  the variables that are used to differential a button's status (low or high), time
  the debouncing correction, or count presses have been declared as 2-element
  arrays.  The first element is labeled 0, and the second is labeled 1.  Note also
  the use of the "for" control structure.

  Circuit Notes:

  Button 1: Connect one pin to GND on your breadboard. Connect the other pin to
  Digital Pin 8
  Button 2: Connect one pin to GND on your breadboard. Connect the other pin to
  Digital Pin 9


  M. Perone
  WVU Psychology Dept
  February 16, 2017

*/
```

Listing continues…

Two Buttons *cont'd*

```
// declare variable for counting button presses
int respCount[2];

// declare variables to calculate response rate in this session
long sessionStartTime;

// declare variables for keeping track of the button's state
int newButtonState[2];
int oldButtonState[2];

// declare variable and constant for debouncing code
int lastButtonChangeTime[2];
const int debounceDelay = 5; // 5-ms delay for debouncing

// set the response pins as D8, D9
int responsePin[2] = {8,9};

void setup() {
  // set the response pin to input with internal pullup
  pinMode(responsePin[0], INPUT_PULLUP);
  pinMode(responsePin[1], INPUT_PULLUP);
  // connect to serial port
  Serial.begin(9600);
  // record start time of this session
  sessionStartTime = millis();
}

void loop() {

  // *** LOOK FOR A BUTTON PRESS ***
  // read the state of the pin
  for (int x = 0; x < 2; x++) {

    newButtonState[x] = digitalRead(responsePin[x]);
    // is the state changed since our last read AND at least the debounce
    // delay has passed?
    if ((newButtonState[x] != oldButtonState[x]) &&
       ((millis() - lastButtonChangeTime[x]) >= debounceDelay)) {
      // Yes, the button state has changed and enough time has passed to pay
      // attention to the change
      // make a note of the change in state
      oldButtonState[x] = newButtonState[x];
      // and make a note of the time of this change
      lastButtonChangeTime[x] = millis();
      // is the new state of the button
      // LOW, i.e., button is pressed?
      if (newButtonState[x] == LOW) {
        // yes, button is pressed, count it
        respCount[x] = respCount[x] + 1;
        // and display it on the LCD
        Serial.print("Button A: ");
        Serial.print(respCount[0]);
        Serial.print("     Button B: ");
        Serial.println(respCount[1]);
      }
    }
  }
}
```

Stepper Sweep

```
/*

  Stepper_Sweep

  This sketch moves a 28BYJ-48 stepper motor through its full range of
  motion, clockwise and counter-clockwise, with a 2-s pause between
  each sweep. The 28BYJ-48 stepper motor requires 2048 steps for an
  entire revolution when using the standard Arduino stepper ibrary.
  A speed of 12 rpm works eliably in both directions in my testing.

  An informative guide to this stepper motor, with links to advanced
  stepper libraries, is here: https://arduino-info.wikispaces.com/SmallSteppers

  Circuit Notes:

  Our stepper motor comes with printed circuit board that interfaces it with the
  Arduino.  Plug the motor into the board (the connector will fit only one way),
  then make the following connections to the Arduino:

  Along one edge of the circuit board are 4 pins for power. A jumper
  covers the rightmost pair, leaving the first two available. Connect:
  "-" to GND
  "+" to 5V

  Along another edge are 4 input pins. Connect them to the Arudino as follows:
  IN1 to Digital 8
  IN2 to Digital 9
  IN3 to Digital 10
  IN4 to Digital 11

  M. Perone, WVU Psyc Dept, Mar 3, 2017

*/

#include <Stepper.h> // stepper library
const int stepsPerRevolution = 2048;  // steps per revolution, empirically derived
const long rpm = 12; // revolutions per minute, assigned by trial and error
// initialize the stepper library on pins 8 through 11; note that the syntax
// for this instruction is:
// Stepper nameOfStepperObject (stepsPerRevolution, PinToIn1, PinToIn3, PinToIn2,
PinToIn4)
Stepper myStepper(stepsPerRevolution, 8, 10, 9, 11);

void setup() {
  myStepper.setSpeed(rpm); // set stepper speed
}

void loop() {
    myStepper.step(2048); // clockwise
    delay(2000);
    myStepper.step(-2048); // counter-clockwise
    delay(2000);
}
```

Stepper by Steps

```
/*
  Stepper_by_Steps

  This sketch moves a stepper motor by the number of steps designated
  by the user via the serial communications interface. Our motor, the
  ubiquitous 28BYJ-48 requires 2048 steps for an entire revolution when
  using the standard Arduino stepper library, and a speed of 12 rpm works
  reliably in both directions in my testing.

  An informative guide to this stepper motor, with links to advanced
  stepper libraries, is here: https://arduino-info.wikispaces.com/SmallSteppers

  Circuit Notes:

  Our stepper motor comes with printed circuit board that interfaces it with the
  Arduino.  Plug the motor into the board (the connector will fit only one way),
  then make the following connections to the Arduino:

  Along one edge of the circuit board are 4 pins for power. A jumper
  covers the rightmost pair, leaving the first two available. Connect:
  "-" to GND
  "+" to 5V

  Along another edge are 4 input pins. Connect them to the Arudino as follows:
  IN1 to Digital 8
  IN2 to Digital 9
  IN3 to Digital 10
  IN4 to Digital 11

  M. Perone, WVU Psyc Dept, Feb 25, 2017, Revised Mar 3, 2017
*/

#include <Stepper.h> // stepper library
const int stepsPerRevolution = 2048;  // steps per revolution, empirically derived
const long rpm = 12; // revolutions per minute, assigned by trial and error
// initialize the stepper library on pins 8 through 11; note that the syntax
// for this instruction is:
// Stepper nameOfStepperObject (stepsPerRevolution, PinToIn1, PinToIn3, PinToIn2,
PinToIn4)
Stepper myStepper(stepsPerRevolution, 8, 10, 9, 11);

void setup() {
  Serial.begin(9600);// set up serial port
  Serial.setTimeout(20); // 10 ms to read string
  myStepper.setSpeed(rpm); // set stepper speed
}

void loop() {
  if (Serial.available() > 0) { // act only if instructed
    String instruction = Serial.readString(); // get the instruction
    int numberOfSteps = instruction.toInt(); // convert string to an integer
    Serial.print("Stepping: ");
    Serial.print(numberOfSteps);
    Serial.println(" times.");
    myStepper.step(numberOfSteps); // position the stepper
  }
}
```

Stepper by Degrees

The dial and pointer for this sketch are included as an appendix.

```
/*
  Stepper_by_Degrees

  This sketch positions a stepper motor at any degree position between
  0 and 360 (with 0 and 360 referring to the same position). The desired
  position is specified by the user via the serial communications interface.
  Our motor, the 28BYJ-48, requires 2048 steps for an entire revolution. This
  sketch translates steps to degrees. There is some   error in this because
  2048 is not wholly divisble by 360, but the results are not bad. With regard
  to motor speed, in my tests 12 rpm works reliably in both directions.

  To test the accuracy of the positioning, you will need a circular meter dial
  showing degrees, and a pointer that is affixed to the motor. These items are
  in the Appendices.  Cut out the meter face, afix to the motor,
  and then add the pointer.

  An informative guide to this stepper motor, with links to advanced
  stepper libraries, is here: https://arduino-info.wikispaces.com/SmallSteppers

  Circuit Notes:

  Our stepper motor comes with printed circuit board that interfaces it with the
  Arduino.  Plug the motor into the board (the connector will fit only one way),
  then make the following connections to the Arduino:

  Along one edge of the circuit board are 4 pins for power. A jumper
  covers the rightmost pair, leaving the first two available. Connect:
  "-" to GND
  "+" to 5V

  Along another edge are 4 input pins. Connect them to the Arudino as follows:
  IN1 to Digital 8
  IN2 to Digital 9
  IN3 to Digital 10
  IN4 to Digital 11

  M. Perone, WVU Psyc Dept, Feb 25, 2017, Revised Mar 3, 2017
*/
```

Listing continues…

Stepper by Degrees *cont'd*

```
#include <Stepper.h> // stepper library
const int stepsPerRevolution = 2048;  // steps per revolution, empirically derived
const long rpm = 12; // revolutions per minute, assigned by trial and error
// initialize the stepper library on pins 8 through 11; note that the syntax
// for this instruction is:
// Stepper nameOfStepperObject (stepsPerRevolution, PinToIn1, PinToIn3, PinToIn2,
PinToIn4)
Stepper myStepper(stepsPerRevolution, 8, 10, 9, 11);
int oldDegrees; // to track changes in degree settings
int newDegrees; // to track changes in degree settings

void setup() {
  Serial.begin(9600);// set up serial port
  Serial.setTimeout(20); // 20 ms to read string, more than enough
  myStepper.setSpeed(rpm); // set stepper speed
}

void loop() {
  if (Serial.available() > 0) { // act only if instructed
    String instruction = Serial.readString(); // get the instruction
    newDegrees = instruction.toInt(); // convert string to an integer
    if ((newDegrees != oldDegrees) && (newDegrees >= 0) && (newDegrees <= 360)) {
      // act only if the new position differs from the old, and the new position
      // is between 0 and 360 degrees
      // next line: calculate the steps needed to change the motor's position
      double change = ((newDegrees - oldDegrees) / 360.0) * stepsPerRevolution;
      int numberOfSteps = int(change); // convert to an integer
      myStepper.step(numberOfSteps); // position the stepper
      Serial.print ("New: "); // tell us about it...
      Serial.print(newDegrees);
      Serial.print(", Old: ");
      Serial.print(oldDegrees);
      Serial.print (", Change: ");
      Serial.print(change);
      Serial.print(", Steps: ");
      Serial.println(numberOfSteps);
      // update old degrees. In the special case in which the user has asked
      // to position the motor at 360 degrees, convert to 0 (because 0 and 360
      // refer to the same position, and it simplifies cacluations if we refer
      // to that position in a consistent way.
      if (newDegrees == 360) {
        oldDegrees = 0;
      } else {
        oldDegrees = newDegrees;
      }
    }
  }
}
```

Temperature Humidity Monitor

```
/*

Temperature_Humidity_Monitor

This sketch uses the Simple DHT library to take periodic readings of
temperature and humidity from a DHT11 sensor.  The sensor has been
mounted to a small circuit board with three pins as described below.

Circuit Notes:

These notes apply to the circuit board in the Elegoo Super Starter Kit.
Hold the circuit board with the senor facing you. The three pins from left
to right should be wired as follows:
Left pin (data pin) to Arduino Digital 2
Center pin to 5V
Right pin to GND

M. Perone, WVU Psyc Dept, Mar 20, 2017

*/

#include <SimpleDHT.h> // reference library
SimpleDHT11 myDHT11sensor; // library requires that we name our sensor
int pinForSensor = 2; // sensor data pin connected to Digital Input 2
byte tempC; // library requires byte to receive Celsius temp
byte humidity; // library requires byte to receive humidity
float tempF; // to save Celsius to Farenheit conversion
long sampleCount; // to count the number of readings

void setup() {
  Serial.begin(9600); // start serial communcations
}

void loop() {
  // next lines: read the sensor at pinDHT11 and receive temperature and
  // humidity. If results are NULL, the function is True and we should
  // print an error message and keep trying to read the sensor.
  if (myDHT11sensor.read(pinForSensor, &tempC, &humidity, NULL)) {
    Serial.print("Attempt to read DHT11 failed. Retrying.");
  }
  else {
    sampleCount = sampleCount + 1; // increment counter
    tempF = ((tempC * 1.8) + 32.0); // convert Celsius to Farenheit
    Serial.print(sampleCount);
    Serial.print(":  ");
    Serial.print(tempC);
    Serial.print(" C,    ");
    Serial.print(tempF);
    Serial.print(" F,    ");
    Serial.print((int)humidity); Serial.println(" % humidity");
    delay(2500); // read the sensor every 2.5 seconds
  }
}
```

Remote Signal Reception

```
/*

  Remote_Signal_Reception

  This sketch demonstrates receiving IR codes with the IRrecv function of the
  IRremote library. The sketch is designed to receive signals from the Elegoo
  Remote control from the company's "Super Starter Kit" using the IR
  detector/demodulator in the kit.  The signal pin of the board must be connected
  to the input "receivePin", which is assigned to Digital 11 in this particular
  sketch. The received code, in hexadecimal format, is sent to the serial monitor.

  I have tried several other remotes lying around my home, and they work with
  the IR detector/demodulator board and this sketch.

  The IRremote library was developed by Ken Shirriff, http://arcfn.com
  Office Website for the IRremote library: http://z3t0.github.io/Arduino-IRremote/
  Documentation here: https://github.com/z3t0/Arduino-IRremote/wiki

  Additional information here: https://arduino-info.wikispaces.com/IR-RemoteControl
  This website warns: "If you have a late version of Arduino with a library
  IRRobotRemote, it may conflict and you may have to remove that library.  Make
  sure to delete Arduino_Root/libraries/RobotIRremote, where Arduino_Root refers to
  the install directory of Arduino. The library RobotIRremote has similar
  definitions to IRremote and causes errors."

  Circuit Notes: On the IR board, there are 3 pins labeled G, R, Y.
  Connect as follows:

  G to GND
  R to 5V
  Y to Arduino Pin 11 (this is the signal)

  M. Perone, WVU Psyc Dept, Mar 4, 2017

*/

#include <IRremote.h> // library
int receivePin = 11; // receiver connected to Digital Pint 11
IRrecv irrecv(receivePin); // connect
decode_results signal; // create object for received signal

void setup() {
  Serial.begin(9600); // Set up serial communications
  irrecv.enableIRIn(); // Start the receiver
}

void loop() {
  if (irrecv.decode(&signal)) { // proceed only if we have a signal
    Serial.println(signal.value, HEX); // display it in hexadecimal format
    delay (250); // pause before reactivating receiver, to avoid extra signals
                 // from the noisy Elegoo remote we are using
    irrecv.resume(); // Get ready to receive the next value
  }
}
```

Remote Signal Decoding Elegoo

```
/*
  Remote_Signal_Decoding_Elegoo

  This sketch demonstrates receiving IR codes with the IRrecv function of the
  IRremote library. As each code is received, the sketch returns, via the serial
  port, the name of the button on the Elegoo remoted control that was pressed.
  The sketch is designed to receive signals from the Elegoo remote control from the
  company's "Super Starter Kit" using the IR detector/demodulator in the kit. The
  signal pin of the board must be connected to the input "receivePin", which is
  assigned to Digital 11 in this particular sketch. The received code, in
  hexadecimal format, is sent to the serial monitor.

  The IRremote library was developed by Ken Shirriff, http://arcfn.com
  Office Website for the IRremote library: http://z3t0.github.io/Arduino-IRremote/
  Documentation here: https://github.com/z3t0/Arduino-IRremote/wiki

  Additional information here: https://arduino-info.wikispaces.com/IR-RemoteControl
  This website warns: "If you have a late version of Arduino with a library
  IRRobotRemote, it may conflict and you may have to remove that library.  Make
  sure to delete Arduino_Root/libraries/RobotIRremote, where Arduino_Root refers to
  the install directory of Arduino. The library RobotIRremote has similar
  definitions to IRremote and causes errors."

  Circuit Notes: On the IR board, there are 3 pins labeled G, R, Y. Connect:
  G to GND; R to 5V; Y to Arduino Pin 11 (this is the signal)

  M. Perone, WVU Psyc Dept, Mar 15, 2017
*/

#include <IRremote.h> // library
int receivePin = 11; // receiver connected to Digital Pint 11
IRrecv irrecv(receivePin); // connect
decode_results signal; // create object for received signal

void setup() {
  Serial.begin(9600); // Set up serial communications
  irrecv.enableIRIn(); // Start the receiver
}

void loop() {
  if (irrecv.decode(&signal)) { // proceed only if we have a signal
    // compare the received value against the various possible valiue
    // (expressed in hexademical numerals)
    switch (signal.value) {
      case 0xFFA25D:
        Serial.println("Power");
        break;
      case 0xFF629D:
        Serial.println("Vol+");
        break;
      case 0xFFE21D:
        Serial.println("Func/Stop");
        break;
     case 0xFF22DD:
        Serial.println("Rewind");
        break;
```

Listing continues…

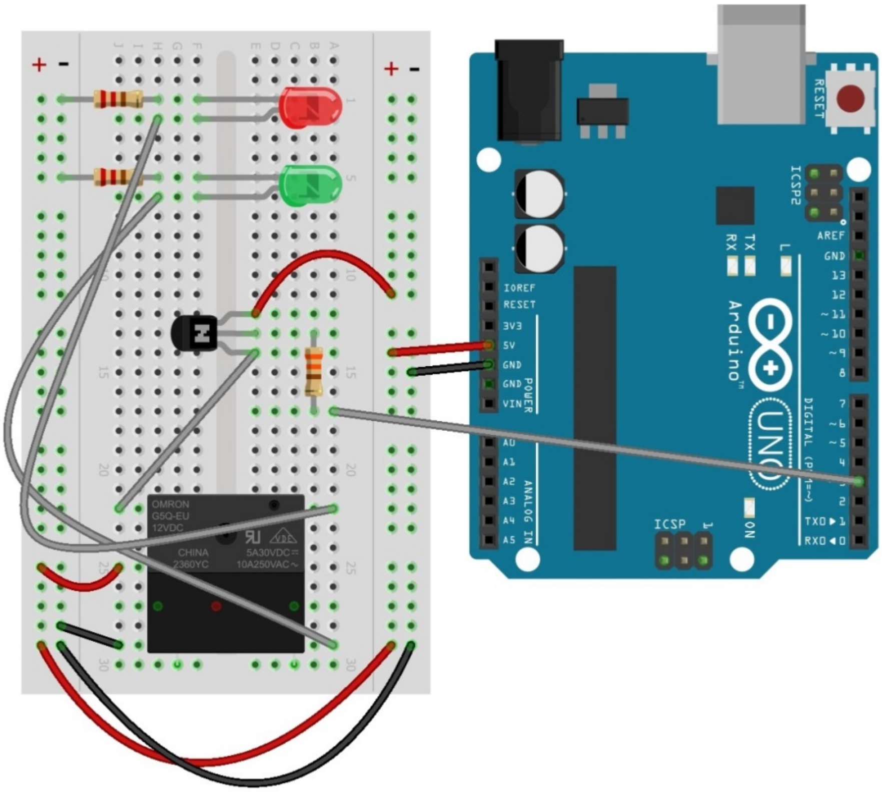
Remote Signal Decoding Elegoo *cont'd*

```
     case 0xFF02FD:
        Serial.println("Play/Stop");
        break;
      case 0xFFC23D:
        Serial.println("Fast Forward");
        break;
      case 0xFFE01F:
        Serial.println("Down");
        break;
      case 0xFFA857:
        Serial.println("Vol-");
        break;
      case 0xFF906F:
        Serial.println("Up");
        break;
      case 0xFF6897:
        Serial.println("0");
        break;
      case 0xFF9867:
        Serial.println("EQ");
        break;
      case 0xFFB04F:
        Serial.println("ST/Repeat");
        break;
      case 0xFF30CF:
        Serial.println("1");
        break;
      case 0xFF18E7:
        Serial.println("2");
        break;
      case 0xFF7A85:
        Serial.println("3");
        break;
      case 0xFF10EF:
        Serial.println("4");
        break;
      case 0xFF38C7:
        Serial.println("5");
        break;
      case 0xFF5AA5:
        Serial.println("6");
        break;
      case 0xFF42BD:
        Serial.println("7");
        break;
      case 0xFF4AB5:
        Serial.println("8");
        break;
      case 0xFF52AD:
        Serial.println("9");
        break;
    }
    delay (250); // pause before reactivating receiver, to avoid extra signals
                 // from the noisy Elegoo remote we are using
    irrecv.resume(); // get ready to receive the next value
  }
}
```

Transistor to Relay

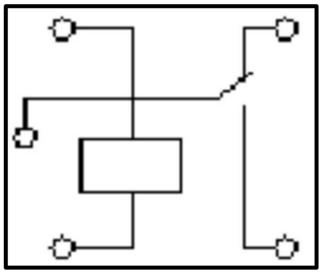
```
/*
  Transitor_to_Relay

  This trivially simple sketch is designed to accompany a relatively
  sophisticated circuit.  The sketch simply turns a digital output on
  and off. The circuit is designed to illustrate how to use a transistor
  to operate a relay, and to use the relay to control something.  In our
  circuit, we will used LEDs, but we could be controlling motors or other
  more interesting stuff.

  M. Perone, WVU Psyc Dept, Mar 26, 2017

*/

const int relayPin = 3; // let's use Digital Pin 3
const int onPause = 1000; // relay will be on this long
const int offPause = 3000; // relay will be off this long

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

void loop() {
  digitalWrite(relayPin, HIGH); // turn the relay on
  delay(onPause); // wait
  digitalWrite(relayPin, LOW); // turn the relay off
  delay(offPause); // wait
}
```

Transistor to Relay cont'd

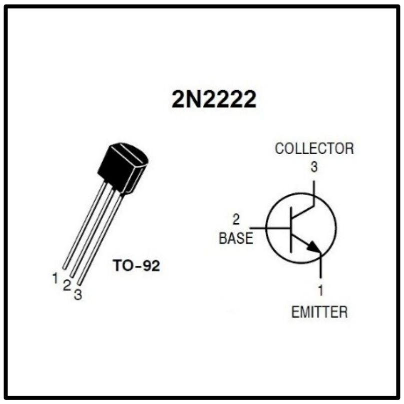


Parts

- Relay
- Transistor
- 1 Resistor @ 330 ohms
- 2 Resistors @ 220 ohms
- Red LED
- Green LED



Circuit diagram for the Songle SRD-05VDC-SL-C relay, as viewed from the top of the device. Pinout, counter-clockwise from top left: 1. Operate coil (apply 5V), 2. Common of switch, 3. Operate coil (apply GND), 4. Normally open side of switch, 5. Normally closed side of switch.



Our transistor. To identify the pins, orient the transistor so the flat side is facing you. Then, from left to right, are the 1. Emitter, 2. Base, and 3. Collector.

To operate the transistor with the Arduino, connect Transistor Lead 2 (the "base" of the transistor) to Arduino Digital 3 via a 330-ohm resistor.

To arrange the current to be switched by the transistor, (a) connect Transistor 3 (the "collector") to 5V, and (b) connect Transistor 1 ("emitter") to Relay 1 (the "operate" pin of the relay). When the Arduino puts 5V on transistor base, the collector and emitter will be connected and 5V will flow to the operate pin of the relay.

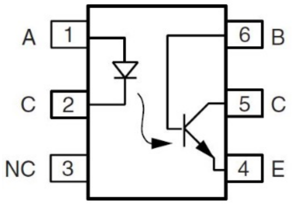
Connect Relay 3 (the other operate pin) to GND. When the transistor applies 5V to the Pin 1 of the relay, current will flow through the coil because the other side of the coil (Pin 3) is connected to GND. An electromagnetic field will move the relay's switch.

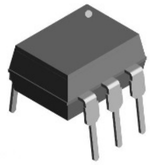
To arrange for current to switched by the relay: (a) connect Relay 2 (Common) to 5V, (b) connect Relay 4 (Normally Open) to the long lead (positive lead) of the green LED, and (c) connect Relay 5 (Normally Closed) to long lead of red LED.

Connect the short leads of both LEDs to GND via a 220-ohm resistor.

When the Arduino sketch turns on the transistor, the transistor operates the relay, and the relay turns on the green led (via the Normally Open pin of the relay). When the sketch turns off the transistor, the relay returns to it resting or "normal" state, and the red LED is lit (by current flowing through the Normally Closed pin).

Optocoupler Test

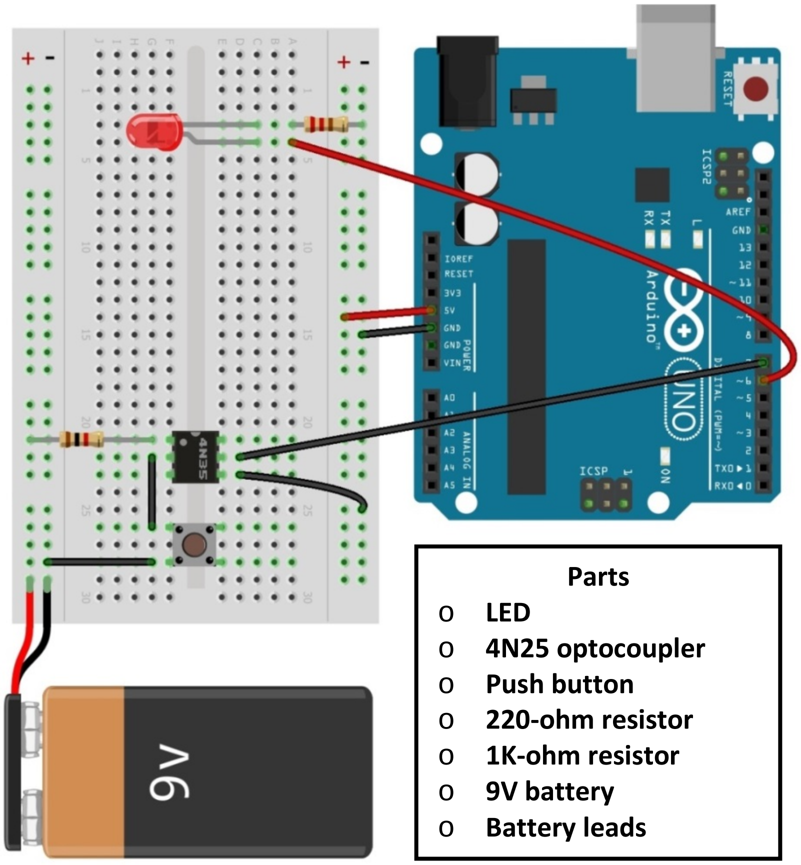




An optocoupler – also known as an "optoisolator" – is a transistor that is operated by light rather than by the application of an electric current. This allows you to have two power sources communicate safely – when, for example, relatively high-voltage devices in the lab must be sensed by the low-voltage Ardunio. It can be used to protect the Arduino from high voltages.

Here is the circuit diagram of the 4N25 optocoupler, a widely used model, alongside a drawing of the chip. To orient the chip, look for a little circle in one corner; this marks Pin 1. (It won't be as easy to see as the one in the drawing.) From there, the pins are numbered sequentially in counter-clockwise order. Passing current across Pins 1 and 2 lights an internal infrared LED. This causes the internal phototransistor to close the circuit between Pin 4 (the transistor's "emitter") and Pin 5 (the "collector.") The infrared LED can handle relatively high voltages (with an appropriately sized resistor in series with it), so will run the high-voltage output of lab devices across Pins 1 and 2. We will run the Arduino Uno's 5V current across Pins 4 and 5.

In the experimental psychology lab, operant conditioning chambers commonly are supplied with 24V to 28V. This is used to operate stimulus lamps and electromagnetic devices such as pellet dispensers. The 28V also is used to run through switches that are closed when, for example, a rat presses a lever. To count the presses, our Arduino needs to sense these switch closures. The problem is that the Arduino would be destroyed if we applied 24V to one of its input pins. The optocoupler solves the problem. On the lab side, Pin 1 is connected to the positive pole of the 24V lab power source via a resistor. Pin 2 is connected to one side of the lever's switch; the other side of the switch is connected to the GND pole of the 24V supply. When the rat presses the lever, 24V current flows across Pins 1 and 2 and the infrared LED is turned on. On the Arduino side, we need to apply GND from the Arduino's own power supply to an input pin in order for the Arduino to sense the input. Pin 4 of the optocoupler is connected to GND from the Arduino, and Pin 5 is connected to one of the Arduino's digital input pins. When the circuit across Pins 4 and 5 is closed, the Arduino's input pin is set LOW and the input is detected.



Parts

- o **LED**
- o **4N25 optocoupler**
- o **Push button**
- o **220-ohm resistor**
- o **1K-ohm resistor**
- o **9V battery**
- o **Battery leads**

When the rat presses the lever, 24V of lab power turns on the infrared LED, which turns on the phototransistor, which allows the Arduino's own 5V power to flow back to its input pin. The two sources of power never touch. They are "coupled" with light from the LED – they are opto-coupled.

To illustrate the use of the optocoupler in our workshop, we use a 9V battery in place of the 24V lab power supply, as illustrated here. Note that the left power rails of the breadboard are connected to the battery, and the right rails are connected to the Arduino's power.

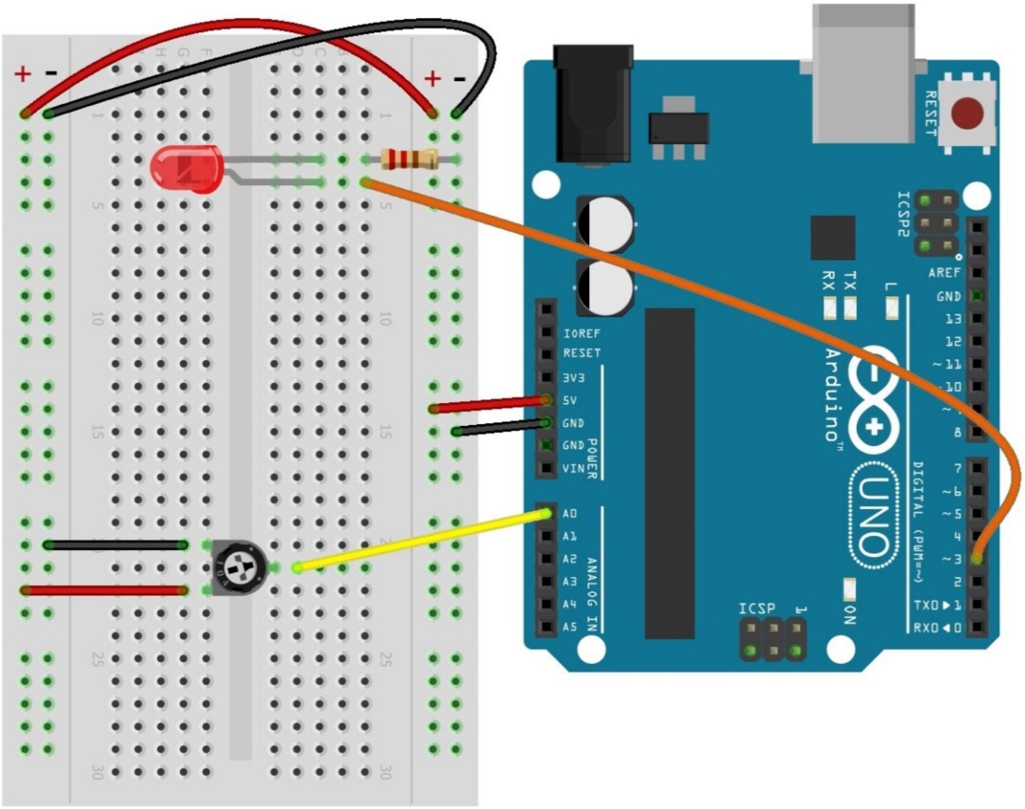
Optocoupler Test *cont'd*

```
/*

Optocoupler Test

This trivial sketch is designed to turn an LED on or off depending on
whether a button is pressed or released. It is intended to test a
circuit that is a bit more sophisticated: one in which an optocoupler
is used to link a button powered at 9V with the Arduino input pin. If
you wired the 9V output of the button directly to the Arduino's pin,
you would damage it.  The optocoupler safely isolates the high voltage
from the low-voltage that powers the Arduino.  (From he Arduino's
standpoint, 9V is high voltage.  This setup with a 9V battery is
designed to mimics common laboratory experiments in which 28V devices
must provide input to the Arduino.

Circuit Notes:

The 4N25 Optocoupler has six pins.  Pin 1 is marked with
a dot on the chip.  The numbering system is counter-
clockwise:  1   6
            2   5
            3   4

Our circuit uses four pins connected as follows:
   1 to +9V via a 1K-ohm resistor
   2 to one side of button
   4 to Arduino GND (via breadbroad, please)
   5 to Arduino Digital Pin 7

The other side of the button is connected to -9V.
Arduino Digital Pin 6 is connected to the positive lead of the LED
Negative lead of LED is connected to Arduino GND via 220k-ohm resistor

M. Perone, WVU Psyc Dept
April 12, 2017

*/

const int button = 7;
const int LED = 6;

void setup() {
  pinMode(button, INPUT_PULLUP); // input in pullup mode
  pinMode(LED, OUTPUT); // output
}

void loop() {
  if (digitalRead(button) == LOW) {
    digitalWrite(LED, HIGH); // button pressed, tunr on LED
  } else {
    digitalWrite(LED, LOW); // button released, turn off LED
  }
}
```

Analog IO with PWM

```
/*

Analog IO with PWM

This sketch reads a potentiometer, converts the result to a number between 0 and
255, and uses the new number to control the duty cycle of a pin capable of Pulse
Width Modulation that is connected to an LED.  By adjusting the pot, the user
changes the brightness of the LED. Note that small variations in the pot reading
are ignored. The sketch also converts the potentiometer reading to voltage.

Circuit Notes:

LED short lead to GND via 220-ohm resistor
LED long lead to Arduino Pin 3
10K Pot Side Pins: one to GND, one to 5V
10K Pot Middle Pin to Arduino Pin A0

M. Perone, WVU Psyc Dept, April 18, 2017

*/
```

Listing continues…



Analog IO with PWM *cont'd*

```
int oldPot; // to store old reading from pot
int newPot; // to store new reading from pot
int difference; // to store the difference in readings
float potVolts; // to store voltage with decimal fraction
const int analogPin = A0; // could have said 14 instead of A0

void setup() {
  Serial.begin(9600); // set up serial communications
}

void loop() {
  newPot = analogRead(analogPin); // read pin connected to 10K pot
  difference = newPot - oldPot; // calculate difference between old and new
  difference = abs(difference); // convert to absolute difference
  if (difference > 3) { // if the difference is 4 or more (range is 0-1023)...
    int dutyCycle = map(newPot, 0, 1023, 0, 255); // convert to duty cycle for PWM
    analogWrite(3, dutyCycle); // adjust brightness of LED via PWM
    // For fun, let's convert the newPot value to voltage
    potVolts = map(newPot, 0, 1023, 0, 500); // convert pot reading to volts x 100
    potVolts = potVolts / 100.0; // convert to voltage expressed to nearest .01 V
    // Now send numbers to serial port
    Serial.print("Analog Input: ");
    Serial.print(potVolts);
    Serial.print (" volts.  LED brightenss: ");
    Serial.println(dutyCycle);
    oldPot = newPot; // the new reading is now the old reading
  }
}
```

Part 7: Exercises

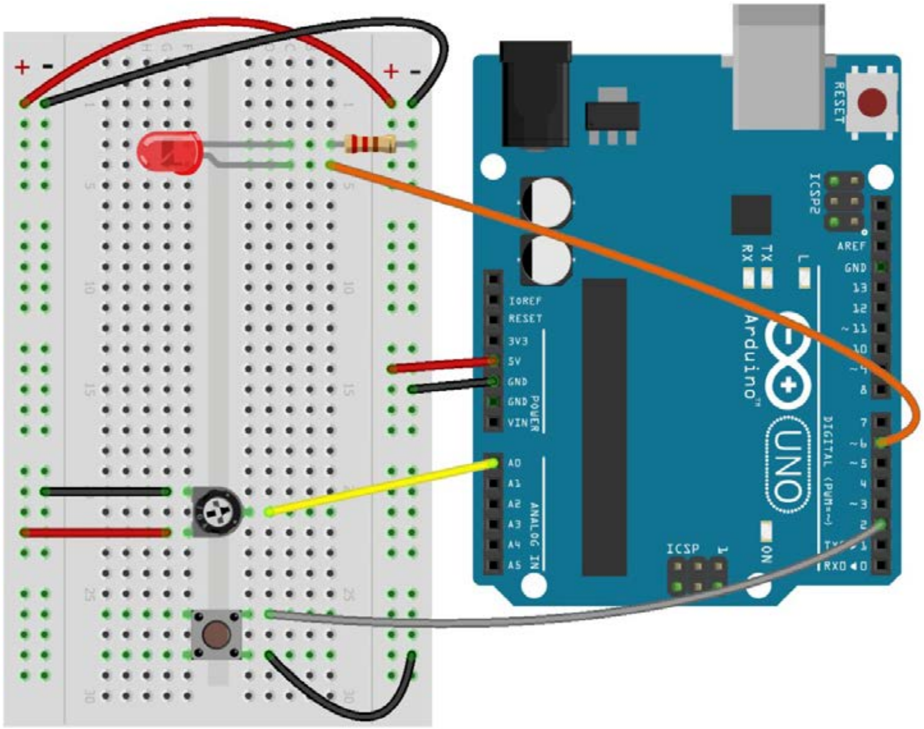
A later edition of this work may have a more systematic progression of exercise (easy to hard, simple to complex). These, however, are in no particular order.

1. Modify the Blink sketch by (a) changing the delay values and (b) changing the output pin (and associated wiring).

2. Build a circuit with a button and an LED. Write a sketch that: (a) Counts button presses. Each press, no matter how long in duration, increments the counter once. (b) Includes debouncing code. (c) After every 5 presses, turns on the LED for 2 seconds. During this time, button presses are ignored.

3. Modify the sketch in Exercise 2 to add data transmission through the serial port: (a) After each press, send the current value of the counter in this format: **`Responses: 1.`** (b) When the LED is turned on, send a message to the serial port indicating that the LED has been turn on, and include a count of the LED operations: **`LED 1`** … **`LED 2`** … etc.

4. Modify the sketch in Exercise 3 to give the user some control over the sketch: (a) When the user sends a number to the Arduino over the serial port, that number is used to decide how many button presses are required to light the button. If, for example, the user sends '25' then the LED will be turned on after every 25 responses. For guidance, look at the sketch entitled "Reading Serial Strings as Parameters" in Part 6.



5. Build the circuit shown here. A 10-k ohm potentiometer provides input to Pin A0, a button provides input to Pin 2, and Pin 6 outputs to an LED. Write a sketch that receives input from the potentiometer on Pin A0 and displays the raw result on the serial monitor. You will need to use the analogRead() function. You may wish to impose a delay of 500 milliseconds or so after every display – otherwise, you sketch will be printing like wild.

6. Using the circuit shown here, modify the sketch in Exercise 6 so that information is sent to the serial monitor only if the raw input value on A0 changes by at least 5. In this sketch, no delay should be needed.

7. Using the circuit shown here, write a sketch that receives input from the potentiometer on Pin A0, maps the input value to a number between 0 and 255, and uses that number to vary the brightness of the LED. You will need to use the Pulse Width Modulation capability of Pin 6 and this is accomplished with the analogWrite() function.

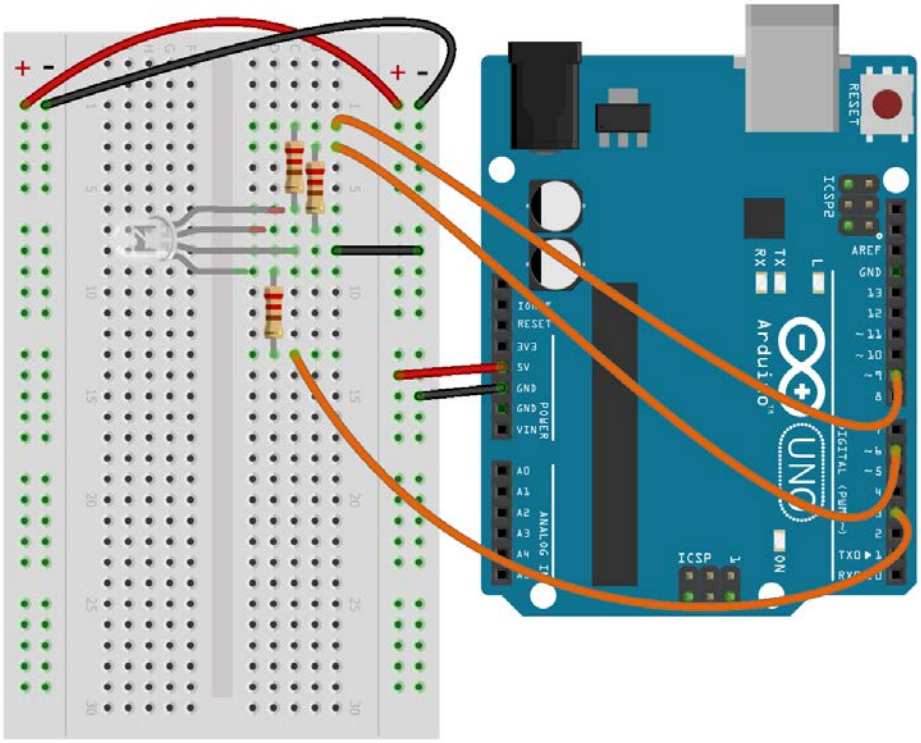
8. Using the circuit shown on the previous page, wite a sketch that receives input from the button on Pin 2. The sketch should count the button presses and, using analogWrite(), vary the brightness of the LED depending on the button count, as follows:

0 = 0
1 = 7
2 = 15
3 = 32
4 = 63
5 = 127
6 = 255

After 6 presses are counted and highest brightness is achieved, the next button press should reset the counter to 0 so the process can begin anew.

9. This exercise uses the Elegoo remote control, IR detector/demodulator, and an RGB LED: Build a suitable circuit and write a sketch so that pressing the 1, 2, and 3 buttons on the remote causes the red, green, and blue elements of the RGB LED to turn on and off. For example, pressing 1 should turn on red and pressing it again should turn off red (and over and over and over again). Pressing 2 should have the same effects on green, and pressing 3 should have the same effects on blue. Hint: You will need variables that keep track of the on/off state of the red, green, and blue elements of the LED. (I personally would use Boolean variables, but other approaches are equally valid.)

10. Part 6 includes a sketch called "Two Buttons." It uses arrays! Add two more buttons to the circuit described in the sketch, and modify the sketch so that it handles four buttons instead of two. This is easy if you understand how to use arrays.



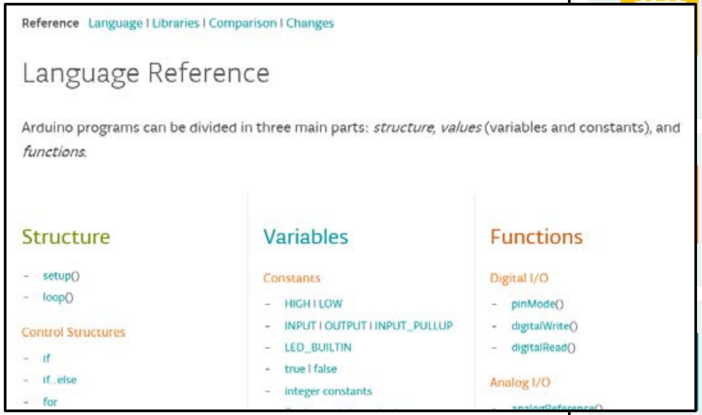
11. Build a circuit that controls the red, green, and blue elements of an REG LED with three PWM-capable pins. Be sure to put a 220-ohm resistor in series with each element of the LED. Using *for* loops, write a sketch that: (a) raises the brightness of the red element from 0 to 255 and then lowers it back to 0. You will need to delay for a while between each change in brightness, otherwise it will happen too fast and you won't see much of anything. You also will have to decide what step size to use as you increment and decrement the brightness. I suggest that you put the delay and step in variables at the top of the sketch so you can easily play with them. (b) For the green element, do the same as you did for the red. Use the same delay and step values. (c) For the blue, do the same as you did for the other colors. The final product should look like this: First, the red goes gradually from dark to full brightness and then back to dark. Next, the green does the same. Third, the blue does the same. And then we start over with the red.

Resources

www.arduino.cc

You can find the official reference for Arduino programming at www.arduino.cc/en/Reference/HomePage

And you can find an excellent set of tutorials at www.arduino.cc/en/Tutorial/HomePage



Reference Language | Libraries | Comparison | Changes

Language Reference

Arduino programs can be divided in three main parts: *structure*, *values* (variables and constants), and *functions*.

Structure

- setup()
- loop()

Control Structures

- if
- if...else
- for

Variables

Constants

- HIGH | LOW
- INPUT | OUTPUT | INPUT_PULLUP
- LED_BUILTIN
- true | false
- integer constants

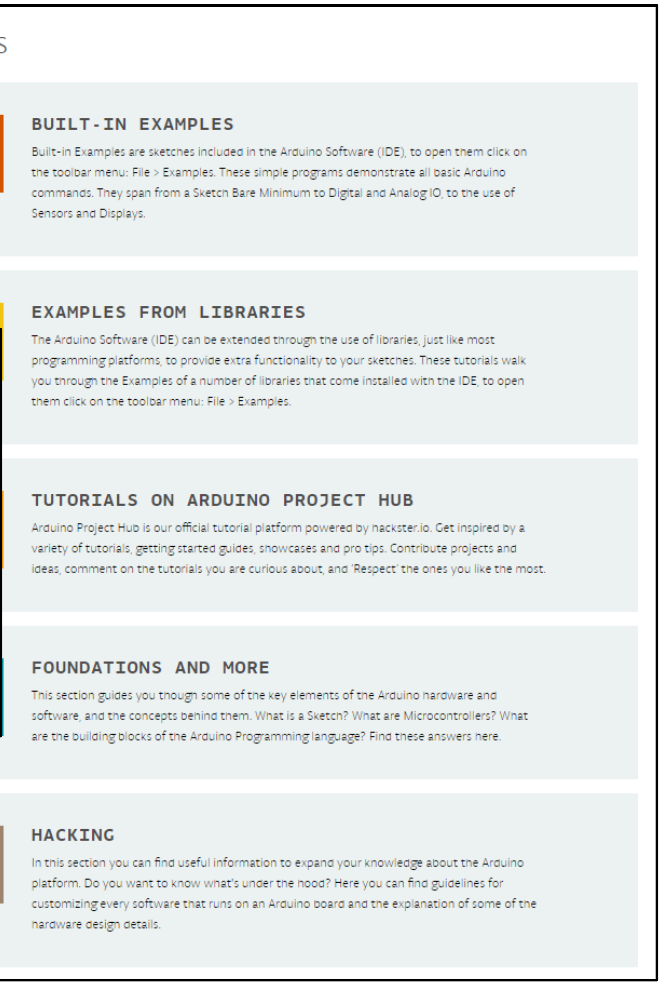
Functions

Digital I/O

- pinMode()
- digitalWrite()
- digitalRead()

Analog I/O

- analogReference()



Tutorials

BUILT-IN EXAMPLES

Built-in Examples are sketches included in the Arduino Software (IDE), to open them click on the toolbar menu: File > Examples. These simple programs demonstrate all basic Arduino commands. They span from a Sketch Bare Minimum to Digital and Analog IO, to the use of Sensors and Displays.

EXAMPLES FROM LIBRARIES

The Arduino Software (IDE) can be extended through the use of libraries, just like most programming platforms, to provide extra functionality to your sketches. These tutorials walk you through the Examples of a number of libraries that come installed with the IDE, to open them click on the toolbar menu: File > Examples.

TUTORIALS ON ARDUINO PROJECT HUB

Arduino Project Hub is our official tutorial platform powered by hackster.io. Get inspired by a variety of tutorials, getting started guides, showcases and pro tips. Contribute projects and ideas, comment on the tutorials you are curious about, and 'Respect' the ones you like the most.

FOUNDATIONS AND MORE

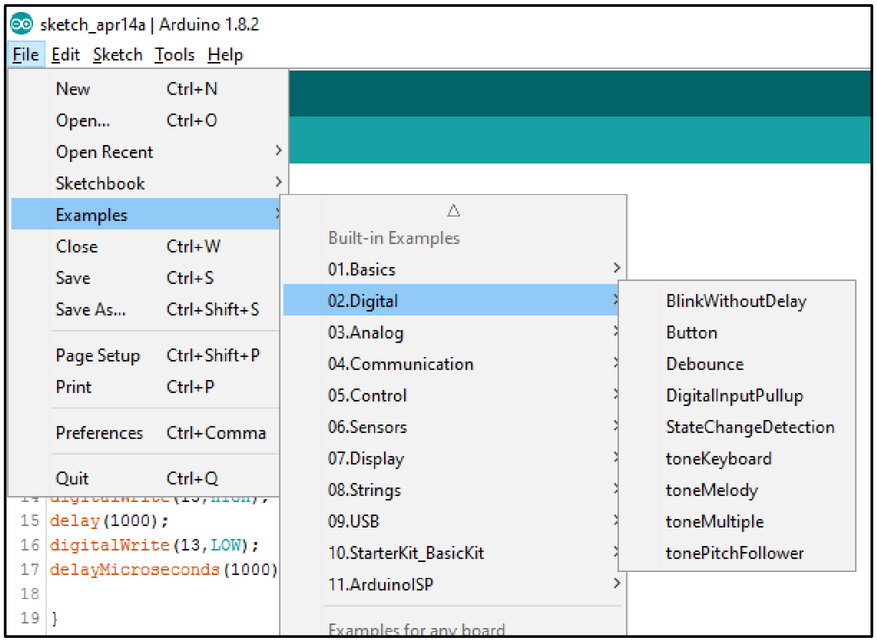
This section guides you though some of the key elements of the Arduino hardware and software, and the concepts behind them. What is a Sketch? What are Microcontrollers? What are the building blocks of the Arduino Programming language? Find these answers here.

HACKING

In this section you can find useful information to expand your knowledge about the Arduino platform. Do you want to know what's under the hood? Here you can find guidelines for customizing every software that runs on an Arduino board and the explanation of some of the hardware design details.

Examples in the Arduino IDE

Don't overlook the example sketches that come with your Arduino IDE; they illustrate ways to tackle a wide range of programming tasks. In the File menu, click Examples and you are on your way.

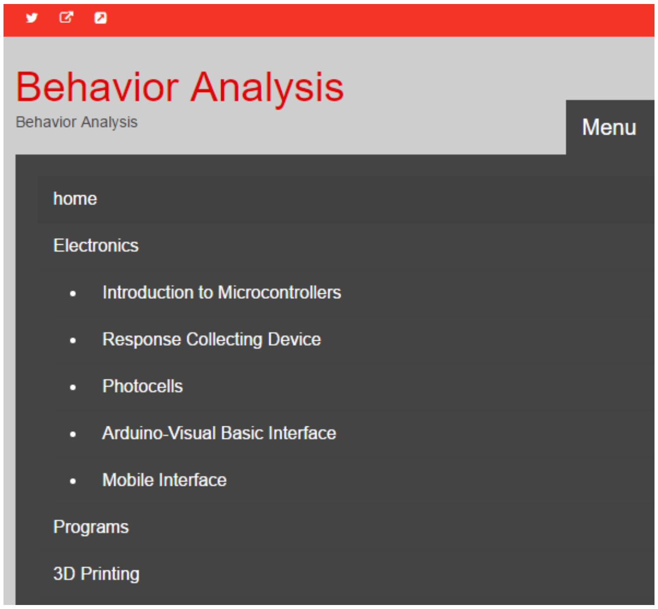


```
digitalWrite(13,HIGH);
delay(1000);
digitalWrite(13,LOW);
delayMicroseconds(1000)

}
```

Rogelio Escobar's Website

http://analisisdelaconducta.net/

Dr. Escobar is Professor of Psychology at National Autonomous University of Mexico. His research interests include basic and applied research in the experimental analysis of behavior, the history of precision instruments in behavior analysis and experimental psychology, and the development of electronic equipment for experimental control and behavioral recording. He has done impressive work in creating physical computing systems for the study of operant behavior, and he generously shares his results – in both hardware and software – on his web site. An article describing some of his work, published in the *Journal of the Experimental Analysis Behavior*, is included as an appendix to this *Reference*.



Other Online Resources

- Resistors: https://learn.sparkfun.com/tutorials/resistors#power-rating
- LCDs: http://www.arduino.cc/en/Tutorial/LiquidCrystal
- DHT11 temperature and humidity module: https://learn.adafruit.com/dht/
- HC-SR04 ultrasonic sensor: https://www.cytron.com.my/p-sn-hc-sr04
- Infrared remote control: https://arduino-info.wikispaces.com/IR-RemoteControl
- SG90 servo: https://www.intorobotics.com/tutorial-how-to-control-the-tower-pro-sg90-servo-with-arduino-uno/
- 28BYJ-48 stepper: https://arduino-info.wikispaces.com/SmallSteppers
- Thermistors: https://learn.adafruit.com/thermistor/overview
- Bi-polar junction transistors: https://learn.sparkfun.com/tutorials/transistors.
- All kinds of stuff: https://arduino-info.wikispaces.com
- About the role of apparatus in the history of behavioral psychology: the Behavioral Apparatus Virtual Museum curated by Kennon A. Lattal at aubreydaniels.com/institute/museum

Books

There are a lot of books out there. Here's a few that I think are helpful.

- *Getting Started in Electronics* by Forrest M. Mims
- *Electronics Cookbook: Practical Electronic Recipes with Arduino and Raspberry Pi* by Simon Monk
- *Programming Arduino: Getting Started with Sketches, Second Edition (2nd Edition)* by Simon Monk
- *Programming Arduino Next Steps: Going Further with Sketches* by Simon Monk

Appendices

Overview

Elegoo Uno Project Super Starter Kit

Page 97. A list of the components in the kit we used in the Spring 2017 workshoip.

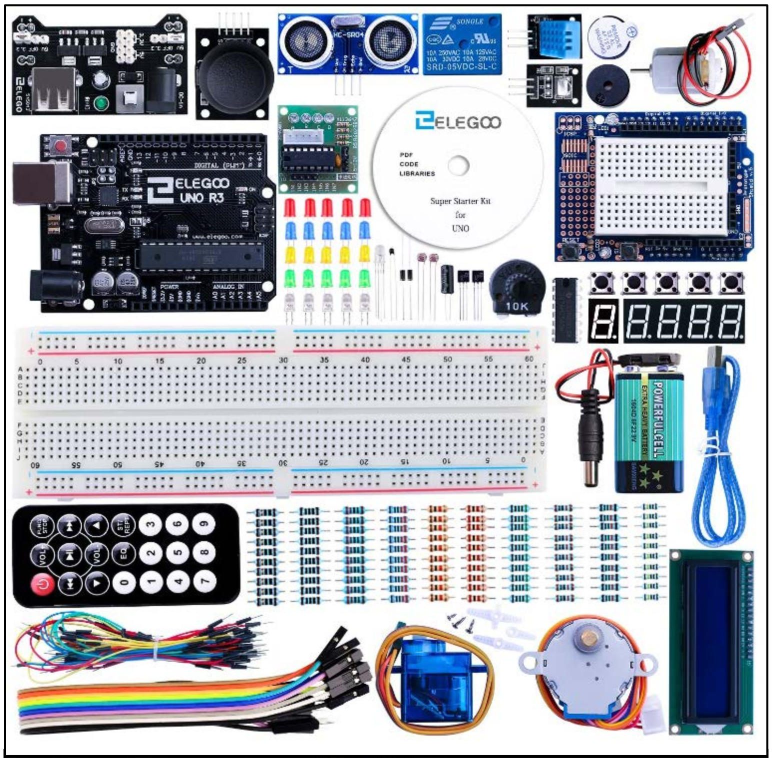
Escobar & Perez-Herrera (2015)

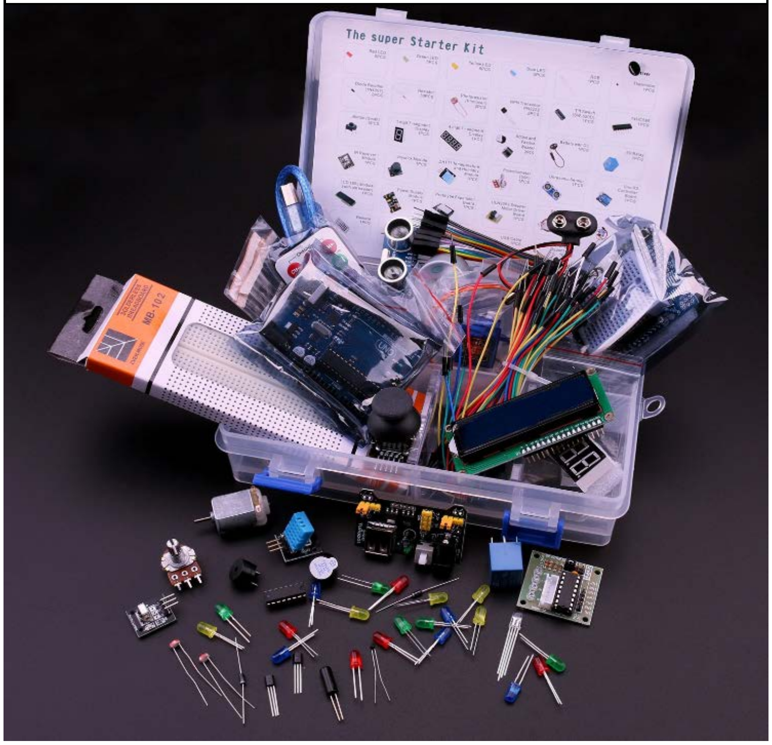
Page 98. An article describing a physical computing system for operant conditioning research developed in Rogelio Escobar's laboratory at the National Autonomous University of Mexico. The system uses an Arduino to interface behavioral test chambers with a PC running a Visual Basic program.

Stepper 360 Dial

Page 107. A dial and pointer that you can cut out and attach to a stepper motor. See the "Stepper by Degree" sketch that is reprinted in Part 6.

Elegoo Uno Project Super Starter Kit





Here are the components in the kit; the ones used in the Spring 2017 workshop are in bold.

1 Uno R3 Controller Board
1 LCD1602 Module (with pin header)
1 Breadboard Expansion Board
1 Power Supply Module
1 Joystick Module
1 IR Receiver
1 Servo Motor (SG90)
1 Stepper Motor (28BYJ-48)
1 ULN2003 Stepper Motor Driver Board
1 Ultrasonic Sensor (HC-SR04)
1 Temperature & Humidity Module (DHT11)
1 9V Battery with DC
1 65 Jumper Wire
1 USB Cable
1 Active Buzzer
1 Passive Buzzer
2 Potentiometer
1 5V Relay (Songle SRD-05VDC-SLC-C)
1 Breadboard
1 Remote
1 Tilt Switch
5 Button (small)
1 1 digit 7-segment Display
1 4 digit 7-segment Display
5 Yellow LED
5 Blue LED
5 Green LED
5 Red LED
1 RGB LED
2 Photoresistor
1 Thermistor
2 Diode Rectifier (1N4007)
2 NPN Transistor (PN2222)
1 IC 74HC595
30 Resistor
10 Female-to-male Dupont Wire

Available from Amazon.

Escobar & Perez-Herrera (2015)

JOURNAL OF THE EXPERIMENTAL ANALYSIS OF BEHAVIOR 2015, **103**, 427–435 NUMBER 2 (MARCH)

LOW-COST USB INTERFACE FOR OPERANT RESEARCH USING ARDUINO AND VISUAL BASIC

Rogelio Escobar and Carlos A. Pérez-Herrera

NATIONAL AUTONOMOUS UNIVERSITY OF MEXICO

This note describes the design of a low-cost interface using Arduino® microcontroller boards and Visual Basic programming for operant conditioning research. The board executes one program in Arduino® programming language that polls the state of the inputs and generates outputs in an operant chamber. This program communicates through a USB port with another program written in Visual Basic 2010 Express Edition® running on a laptop, desktop, netbook computer, or even a tablet equipped with Windows® operating system. The Visual Basic program controls schedules of reinforcement and records real-time data. A single Arduino® board can be used to control a total of 52 inputs/output lines, and multiple Arduino® boards can be used to control multiple operant chambers. An external power supply and a series of micro relays are required to control 28-V DC devices commonly used in operant chambers. Instructions for downloading and using the programs to generate simple and concurrent schedules of reinforcement are provided. Testing suggests that the interface is reliable, accurate, and could serve as an inexpensive alternative to commercial equipment.

Key words: interface, Arduino, Visual Basic, instrumentation, experimental control

One problem faced by those interested in conducting laboratory research in operant conditioning is the high cost of commercial equipment for controlling and recording experimental events. This problem is particularly acute in universities where resources are limited, and outside the United States where the costs are higher because of import taxes and shipping costs. Although inexpensive alternatives to commercial equipment have been offered, for example Palya and Walter's (1993) controller board, or a computer's parallel port (e.g., Escobar & Lattal, 2010; Gollub, 1991), these alternatives are now dated.

Advances in electronics have produced inexpensive technology that could be used in operant research. One example are microcontroller boards, which are compact input/output boards that can be programmed to activate devices such as lights or motors and to detect changes in the environment with a variety of sensors. Hoffman, Song, and Tuttle (2007) described how a microcontroller board can be used to control operant chambers. Their board, however, required extensive assembly and the schedules of reinforcement had to be programmed in C-like language.

In recent years, several easy-to-use preassembled microcontroller boards have become widely available (e.g., Arduino®, BASIC stamp®, Parallax Propeller®). The interface described in this note uses Arduino® Uno and Arduino® Mega 2560 microcontroller boards. The open-source Arduino® boards are inexpensive, and have been tested for precision, accuracy, and reliability (see D'Ausilio, 2012; Schubert, D'Ausilio, & Canto, 2013), properties that make them suitable for operant research. The most popular Arduino® board, the Arduino® Uno R3, is equipped with 14 digital input/output pins that can be used to control and record events in an operant chamber. Arduino® Mega 2560 R3 boards, in comparison, are more expensive but are equipped with 54 digital input/output pins[1]. The two boards can be used interchangeably in the system described in this paper.

This project was supported by the 2012 SABA International Development Grant and by grant TA300213–2 awarded by PAPIIT, DGAPA UNAM to the first author. We are indebted to Ernesto Huerta for his help in designing the interface, and to Armando Machado, William Palya, and one anonymous reviewer for their comments to previous versions of this paper. We thank Nadia Santillán and Katya Quiñones for their help during the tests. Those interested in using the interface described can contact the authors for technical support (rescobar@unam.mx or mnbaxs@hotmail.com).

Address correspondence to: Rogelio Escobar, Laboratorio de Condicionamiento Operante. 2o Piso Edificio C. Facultad de Psicología, Universidad Nacional Autónoma de México. Av. Universidad 3004. Col. Copilco-Universidad. C.-P. 04510.

doi: 10.1002/jeab.135

[1] Only 12 digital input/output pins are available in Arduino® Uno boards and 52 in the Arduino® Mega 256 boards because serial communication with the PC uses Digital Pins 0 and 1.

Arduino® is an open-source prototyping platform that consists of Arduino® boards and Arduino® integrated development environment (IDE), which is required for writing, debugging, and transferring programs to the microcontroller in the board. Arduino® boards communicate through a USB port with a personal computer (host PC) running the Arduino® IDE. This port is also used for powering up the board and for sending and receiving real-time data.

The microcontroller in the Arduino® boards executes the instructions of the program written in C-like Arduino® language. This program is required to read the state of inputs and to generate outputs. Once the program is transferred from the host PC to the board it is executed immediately and continuously until the board is turned off. To minimize the need of programming using this specific language, the authors created two free-distribution general-purpose programs designed for operant conditioning experiments (see section on *Arduino® Program*). These programs were designed to communicate with schedules of reinforcements written in the more familiar Visual Basic programming language; specifically Visual Basic 2010 Express Edition® (VB2010).

The structure of the interface is shown in Figure 1. The Arduino® board is programmed to translate contact closures in the operant chamber into values sent to a serial communication port through the USB port. These values are read and recorded by the VB2010 program that can also produce an output after the value is read. The VB2010 program sends a symbolic code to the serial communication port (also through the USB port) where the Arduino® program reads and interprets the symbolic code to activate lights and feeders in the operant chamber.

Programs for controlling operant chambers could be written exclusively in Arduino® language by users familiar with this programming language, thus eliminating the need for a VB2010 program (see e.g., Hoffman et al., 2007; for a similar strategy). The small amount of memory in microcontrollers and the basic user interface, however, limit the size and functionality of programs. VB2010, in contrast, enables researchers to program relatively complex experiments using the easy-to-learn and graphically oriented VB2010 IDE. Additionally, previous versions of Visual Basic® have been used in operant research (e.g., Cabello, Barnes-Holmes, O'Hora, & Stewart, 2002; Dixon & MacLin, 2003). Programs created previously can be modified with relative ease to communicate with an Arduino® board. VB2010 is free-distribution software and supports x86 and x64 Windows® architectures[2].

Hardware Assembly

Inputs

Digital pins detect the state of digital switches such as those used in levers or keys, which can be either on (closed contacts) or off (open contacts). Figure 2 shows a diagram of the connections between a micro switch and Digital Pin 8 on an Arduino® Uno board. A second input device can be connected to another digital pin (e.g., 9). The distribution of pins is similar in Arduino® Mega cards.

Outputs

The Arduino® board pins can power up directly 5-V DC devices drawing $<$ 40 mA of current. A regulated power supply must be added, however, to power up 28-V DC devices used in standard operant chambers such as bulbs and feeders. The upper section of Figure 3 shows an Arduino® Uno board connected to 28-V DC output devices using an inexpensive array of 5-V DC electromechanical relays and the integrated circuit ULN2803A. The lower section of Figure 3 shows another setup with solid-state relays, which are noiseless and faster than electromechanical relays. With both setups, for example, when the VB2010 program sends a symbolic code representing reinforcer delivery to the serial communication port, the Arduino® program detects the code and changes briefly the state of Digital Pin 12 from low to high (off to on). This activates the feeder in the operant chamber by changing the state of the relay. The electronic components can be arranged on a solderless breadboard for testing using male to male jumper

[2] Arduino® IDE runs on Windows®, Mac OS® X, and Linux® but the design of the present interface is limited to PCs running Windows® because of VB2010 requirements: Windows XP with service pack 3, Vista, 7, or 8 operating systems, 1.6 GHz or faster processor, 1024 MB RAM, and 3 GB of available disk space. VB2010 can be installed on a computer running Mac OS using a Virtual Machine but this configuration was not tested.

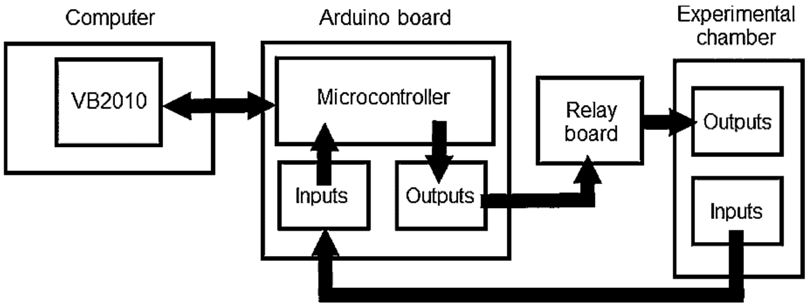


Fig. 1. Scheme of the interface structure. An Arduino® program detects the state of inputs in the operant chamber and sends a signal to the serial communication port where the VB2010 program in the host PC reads it. The VB2010 program also sends signals to the serial communication port where the Arduino® board reads it and activates the outputs in the operant chamber by changing the state of relays.

wires and later, if needed, transferred to a solderable breadboard or perfboard. Additionally, a screw shield (e.g., Itead® proto screw shield) can be added on top of Arduino® boards to secure the connections.

Software

Table 1 contains step-by-step instructions for setting up the interface. Steps 1 and 2 describe the installation of Arduino® IDE and VB2010 IDE in the host PC.

Arduino® Program

The source code of the Arduino® program can be downloaded from the authors' website (see Table 1, Step 3). Two versions are available. The code in the "Arduino_Program.ino" file is used when only one response is recorded, and the code in the "Arduino_Program_Concurrent.ino" file can be used when two responses are recorded (e.g., in concurrent schedules of reinforcement). Both programs establish serial communication with the PC, define the symbolic codes for activating outputs in the operant

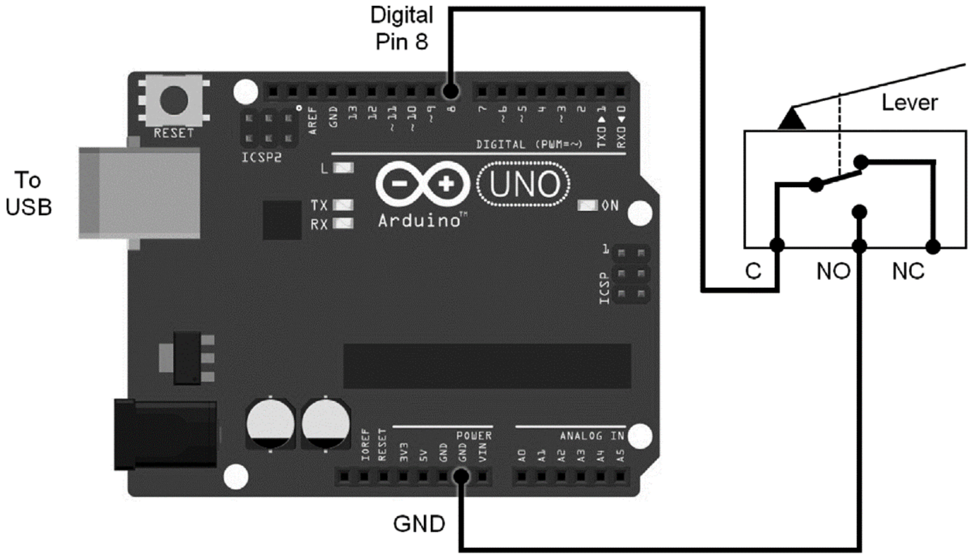


Fig. 2. Schematics of the connections between Arduino® Uno (schematic created using Fritzing® software [www.fritzing.com]) and a switch used in standard rat levers and pigeon keys. In some pigeon keys, the lever shown in the diagram is replaced with a small button.

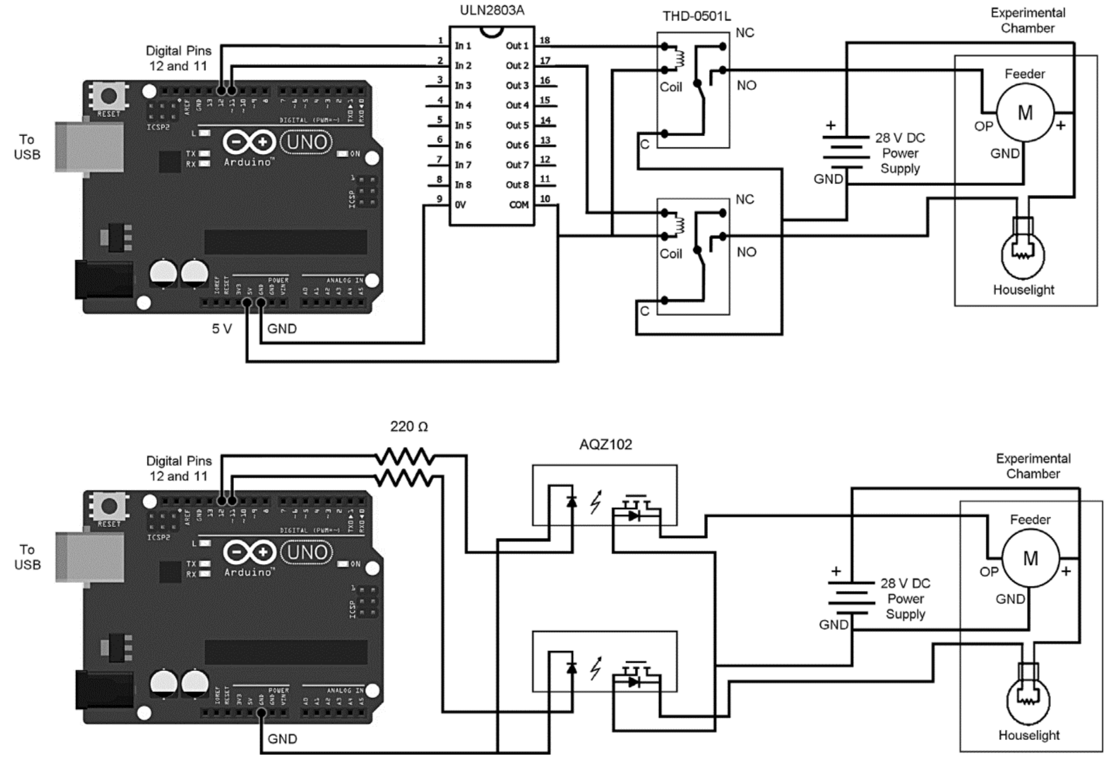


Fig. 3. Schematics of the connections between an Arduino® board and the outputs in an operant chamber (Arduino® schematic created using Fritzing® software). In the examples, only a houselight and a feeder are shown. Feeders are commonly connected with three wires, +28 V DC (red), operate (OP, white), and GND (black), as shown in the diagram. In feeders with two wires, the GND wire in the diagram is not used, and the GND wire in the feeder is connected as the OP wire. The upper diagram shows the connections using electromechanical 5 V DV relays (Sun Hold, Model THD-0501L). The lower diagram shows the connections with solid-state relays (Panasonic, Model AQZ102).

chambers, and poll continuously the state of digital inputs. Only one program can be uploaded to the microcontroller at a time and it is stored in the nonvolatile flash memory of the microcontroller. Therefore, unless a change in the number of inputs or outputs is required during an experiment, the Arduino® program is transferred to the Arduino® board only once (see Table 1, Steps 4 and 5).

Arduino® programs are divided into three sections. In the first section, variables are declared and given specific values that match the digital pins in which the devices in the operant chamber are connected (see Fig. 1). For example, in the "Arduino_Program.ino" file, the instruction "byte Houselight = 11" creates the variable "Houselight" with a value of 11. This value matches the digital pin where the houselight is connected.

The sections following the declaration of variables are "Setup()" and "Loop()". In the "Setup()" section, serial communication at 9600 baud rate is initialized with the instruction "Serial.begin(9600)", and each variable is set as input or output. For example, the instruction "pinMode(Houselight, OUTPUT)" sets Digital Pin 11 (the value assigned to the variable Houselight) to output mode. The instruction "pinMode(Response,INPUT_PULLUP)" , after Digital Pin 8 was set to input mode, enables an internal pull-up resistor that avoids random fluctuations of the input values when the switch contacts are opened. Because of this instruction, when no response is detected the value of the input is 1 and when a response is detected the value of the input is 0.

The instructions in the "Loop()" section of the code, known as the main loop, are executed and repeated after the board is powered on. Within the main loop, the program reads symbolic codes ("S", "R", or "E") sent to the serial communication port from the host PC,

Table 1

Checklist for running schedules of reinforcement using the Arduino®-VB2010 interface.

Step	Instructions
Arduino® and VB2010 IDEs	
1	Download Arduino® IDE from http://arduino.cc/en/Main/Software and install it in the host PC. Versions 1.0.2 to 1.0.6 were tested.
2	Download VB2010 IDE from http://www.visualstudio.com/downloads/download-visual-studio-vs#d-2010-express, and install it in the host PC. VB2010 includes SQL Server and Silverlight but these components are not required. Run once and close to allow VB2010 to create the "Projects" folder.
Arduino® program	
3	Download "Arduino_Program.zip" file from the authors' website: http://analisisdelaconducta.net/wp-content/uploads/2014/11/Arduino_Program.zip. Extract the file in any location in the host PC. This will create the Arduino_Program folder containing the "Arduino_Program.ino" file.
4	Connect the Arduino® board to the USB port and verify the assigned COM port using the "Device Manager" in the "Control Panel". The COM port differs for each board connected but remains constant for the same board.
5	Open "Arduino_Program.ino" file using Arduino® IDE and click the "upload" button to transfer the program to the Arduino® board. Before uploading, check board model by clicking "Tools" and then "Board". Verify that the COM port selected in "Serial Port" also in "Tools", matches the port displayed in the "Device Manager".
6	Serial monitor, Ctrl + Shift + m, inside Arduino® IDE can be used for testing inputs and outputs. If the devices in the chamber are connected correctly, a list of values = 1 change to 0 when a response is detected. Entering "S" or "R" and pressing the "Send" button activates the outputs.
VB2010 program	
7	Create folder "Data" in C:\.
8	Download "Visual_Basic_program.zip" from the authors' website: http://analisisdelaconducta.net/wp-content/uploads/2014/11/Visual_Basic_Program.zip
9	Extract the file to C:\ …Documents\Visual Studio 2010\Projects. This will create folder Visual_Basic_Program. Inside the folder, double click on "Arduino_VB.sln" to open the VB2010 program.
10	Execute the program by pressing "F5" or by clicking the "start debugging" icon. Introduce the values, select the schedule and click "continue". Confirm the data entered by clicking "continue".

Note. To generate concurrent schedules of reinforcement with two responses download "Arduino_Program_Concurrent.zip" file (http://analisisdelaconducta.net/wp-content/uploads/2014/11/Arduino_Program_Concurrent.zip) in Step 3, and download "Visual_Basic_Program_Concurrent" folder (http://analisisdelaconducta.net/wp-content/uploads/2014/11/Visual_Basic_Program_Concurrent.zip) in Step 8.

and changes the state of the corresponding device in the operant chamber. In the following example, if character "R" is read, the feeder is activated for 40 ms (comments in Arduino® code are italicized and follow //).

```
char Event = Serial.read();
// The variable "Event" takes the value read from the serial communication port.
switch (Event) {
// This instruction is used to specify code that should be executed when the value of a case statement matches the value of the variable "Event".
case 'R':
// If the value of "Event" is equal to "R" (the symbolic code for reinforcer delivery) execute the code in the following lines before the break instruction.
digitalWrite(feeder,HIGH); // Activates the feeder
delay(40); //This delay keeps the feeder on for 40 ms.
digitalWrite(feeder,LOW); // Turns off the feeder.
break; // required to terminate the code when the value of "Event" is "R".
```

In the last section of the main loop, the state of the digital input is polled and its value (1 or 0) is sent to the serial communication port using "Serial.println(digitalRead(Response))". After the state of input is read, a 4-ms delay is added to debounce the input. If duplicate responses are recorded, this delay could be lengthened.

VB2010 Program

Two programs created in VB2010 can be downloaded from the authors' website to generate simple and concurrent schedules of reinforcement (see Table 1, Step 8). One program, "Arduino_VB.sln" within the

"Visual_Basic_Program" folder, is used for basic schedules of reinforcement from a single key or lever: fixed ratio (FR), fixed interval, variable ratio, and variable interval (VI) schedules[3], with or without added delays to reinforcement. The other program, within the "Visual_Basic_Program_Concurrent" folder, is used for two-input concurrent schedules of reinforcement using combinations of the four basic schedules. Each folder is stored in a compressed zip file. The programs use a graphic interface that allows users to set the values of common variables in operant research.

At the end of each session, the VB2010 program creates and saves an output file with the date, time of session, subject name, session number, schedule, an array of experimental events, and summary data including the total number of responses, total number of reinforcers, and seconds elapsed. The array of events consists of a list of values corresponding to the time stamp of each event in ms and the code of the event separated by a period (a header in every output file describes the code for each event). The output file can be imported from Excel® using the period to separate the values into columns.

After downloading one of the two compressed files containing one VB2010 program from the authors' website, the file must be extracted to the "Projects" folder in the "Visual Studio 2010" folder that can be found within "Documents". A double-click on the "Arduino_VB.sln" file within the folder opens the VB2010 program (see Table 1, Step 9). It is important to note that the "Visual_Basic_Program" folder contains the program that runs in conjunction with the program stored as "Arduino_Program.ino", and the "Visual_Basic_Program_Concurrent" folder contains the program that runs in conjunction with the program stored as "Arduino_Program_Concurrent.ino" file.

To communicate with the Arduino® program, the serial port class functions must be enabled with the instruction "Imports System. IO.Ports". Afterwards, the same communication port used by the Arduino® board (as specified in the "Device Manager") has to be declared and initialized as a serial port. In the following example, COM4 was used. Comments in Visual Basic code are italicized and follow '.

Public Comunication_Port As String = "COM4"

' *Declares the variable "Communication_Port" as a string with the name "COM4".*

Public Arduino As SerialPort

' *Declares the variable "Arduino" as a serial port object.*

Arduino = New SerialPort(Comunication_Port, 9600)

' *Establishes communication with "COM4" at a 9600 baud rate.*

Changes in input values are detected with a function that reads the serial communication port.

If Arduino.BytesToRead > 0 Then

'*Arduino was previously defined as SerialPort. The following code will be executed only if there is a value in the serial communication port.*

Arduino_string = Arduino.ReadLine()

' *"Arduino_String" takes the value read from the serial communication port.*

Actual_Response = Convert.ToInt32 (Arduino_string)

' *"Actual Response" takes the value in "Arduino_String" converted to an integer.*

End If

Each reinforcement schedule is generated with few lines of code in the function "Schedule_of_Reinforcement()" within the "Session" form. For example a fixed-interval schedule is generated with:

If Time () >= (Schedule_Value * 1000) And Actual_Response <> 1 Then

' *"Schedule_Value" is defined by the user as the duration of the schedule in seconds. Thus, if the time elapsed (measured in the Time() function, not shown) is longer than the schedule value in milliseconds and if a response is detected, execute the following code.*

Reinforce() ' *This instruction executes a function (see below).*

End If

Activating an output device is accomplished by sending a character to the serial communication port. For example, in the function "Reinforce()" the character assigned to the reinforcer event is sent to the serial communication port where the Arduino® program reads it and interprets it.

Function Reinforce() As Integer ' *Declares a function*

Arduino.WriteLine("R")

' *When the function is executed, the character "R" is sent to the serial communication port.*

[3] Variable-interval durations and the progression of responses for variable-ratio schedules are generated using Fleshler and Hoffman's (1962) equation.

```
TXT.WriteLine(Time_in_milliseconds() & ".500")
' Writes the time stamp of the event followed by a period and the event marker in the output file.
Total_Reinforcers = Total_Reinforcers + 1
' Reinforcer counter incremented by one.
End Function
```

Multiple sessions in different chambers, each connected to an Arduino® board, can be controlled simultaneously from the same VB2010 program. It is necessary to enter the corresponding COM port, session name, subject name for each session, and click the "continue" button to open the new session. The code necessary to open more than one session simultaneously can be found in the "Confirm" form. In this form, the instruction "Imports System.Threading" initializes threads in VB2010. Each session is considered a new thread. The instruction "Arduino.Close()" is needed before loading a new session using the same COM port. Custom schedules of reinforcement could be written by replacing the code within the "Schedule_of_Reinforcement ()" function or by creating a new project and transferring all Arduino®-related functions and variables.

Test Results

The precision and accuracy of standalone Arduino® boards in behavioral research have been tested previously (e.g., D'Ausilio, 2012). However, using these boards in combination with VB2010 warrants further testing. One type of test analyzed the speed of the interface at presenting a stimulus after detecting a response. A second, standalone Arduino® board was used to simulate a response and it also served as a microsecond stopwatch (Arduino® boards provide a 4-µs resolution timer). During the test, the following cycle of events took place for 1000 trials: 1) the standalone board started its stopwatch and generated a 20-ms pulse; 2) the pulse transition from off to on was detected and recorded by the main Arduino®-VB2010 interface; 3) the VB2010 program recorded the presentation of a "reinforcer" and generated a symbolic code that was read and executed by its associated Arduino® board; 4) the code activated a solid-state relay which in turn stopped the stopwatch of the second Arduino® board; a 1-s interresponse time followed before the next cycle began. The tests were conducted under conditions that simulated a variety of tasks conducted simultaneously in the host PC with an Intel Core i7 processor and 8 GB of RAM.

Table 2 shows the results. When CPU consuming tasks (e.g., antivirus scan) were prevented and the VB2010 and Arduino® programs controlled a simple schedule of reinforcement in the foreground, the latency from the onset of the 20-ms pulse to the stopping of the stopwatch averaged 12.56 ms.

Table 2

Latency tests

One response recorded (fixed ratio 1)

CPU load (%)	Boards	Executed in background	Mean (ms)	SD (ms)	Range (ms)	Latencies > 100 ms
7–17	1	No	12.56	2.21	9.59 – 17.73	0
15 – 21	2	No	13.11	2.30	9.70 – 17.68	0
19 – 23	3	No	12.83	2.25	9.71 – 17.68	0
60 – 70*	3	Yes	12.64	2.29	9.60 – 20.89	0
100**	3	No	15.17	4.48	9.92 – 41.33	0
100**	3	Yes	76.70	253.15	9.74 –2103.05	9%
Two responses recorded (concurrent VI VI)						
7 – 17	3	No	21.03	6.18	11.77 –32.04	0
60 – 70*	3	Yes	23.34	5.72	12.59 – 23.34	0
100**	3	No	48.89	47.68	11.88 – 251.67	11%

Note. Tests were conducted by recording one response in a simple schedule of reinforcement (FR 1), and two responses in concurrent schedules of reinforcement (VI 1 s VI 1 s) using computers running Windows 7®. The tests were conducted under three conditions that involved different CPU loads, with one, two, and three Arduino® boards connected simultaneously. The VB2010 program ran in the foreground and background.*These loads were generated by running simultaneously common programs that yielded high CPU usage (e.g., Norton® virus scan, Internet Explorer®, Word®, and Excel®). ** This CPU load was reached using JAM software Heavy Load® 3.3.

When the VB2010 and Arduino® programs controlled a concurrent schedule, the mean latency increased to 21.03 ms. The speed in both tests, however, was not systematically affected when more than one Arduino® board was used (simulating the control of up to three operant conditioning chambers polling one or two responses). Running antivirus scan tasks and other common programs in the foreground produced slightly longer latencies but had no clear effect on general performance. Additional test results (not shown in the tables) showed that performance was not noticeably affected when three responses in concurrent schedules were polled simultaneously, when generic hubs were used to connect the Arduino® boards, when high memory and high disk usage were simulated, or when the VB2010 ran in another PC (Intel Core i3 with 4 GB RAM). Latencies longer than 100 ms were observed only when CPU load was forced to reach 100% (see Table 2). The longest latencies (> 1 s) were observed when the VB2010 program executed in the background with a CPU load of 100%. Under such conditions, the interface would be inadequate for operant research. It is important to note that each Arduino® board increases CPU load by approximately 3%. Therefore, increasing the number of Arduino® boards, within practical limits, should have no substantial effect on performance.

Another type of test examined the maximum number of responses recorded per second. In this test, a stand-alone Arduino® Uno board turned on and off a solid-state relay (a simulated response) 1000 times in each of 10 consecutive trials. Transitions of the state of the relay from off to on, and from on to off were detected and recorded in an output file with the Arduino®-VB2010 interface connected to a host PC. The duration of the simulated response (from on to off) and the interresponse time (IRT) (from off to on) were set at 1 ms and lengthened until 1000 responses were recorded in each trial. Results showed that the minimum response duration detected accurately was 4 ms with an IRT of 4 ms. Therefore, up to 125 responses were recorded per second. However, because the VB2010 timer has a 15-ms resolution, the output file showed up to three responses with the same time stamp. This characteristic also restricts the accurate measurement of response duration in the output file to those lasting over 15 ms.

When the VB2010 and Arduino® programs recorded two or more responses in concurrent schedules, the minimum response duration detected was 12 ms with an IRT also of 12 ms. These durations are shorter than the 15-ms responses that were generally recorded using solid-state and electromechanical equipment in operant research during the 1960s and 1970s with pigeons as subjects. Furthermore, pulse

Table 3

Components and prices

Quantity	Component Description	Total Price (USD)
1	Arduino® Uno	25.00
1	Arduino® Mega	46.00
1	Itead Proto screw Shield (for Arduino® Uno)	6.50
1	Itead Mega proto screw shield (for Arduino® Mega)	8.00
1	USB Cable (A to B)	4.00
1	Power Supply 0–30 V DC, 5 A* (Mastech HY3005D)	120.00
2	Solid-state relays (Panasonic Model AQZ102)	29.00
2	220 Ohm carbon film 1/2 W resistor	2.00
2	Electromechanical relays (5 V DC coil, contact rating > 1 A at 30 V DC) (Sun Hold, Model THD-0501L)	10.00
1	Integrated circuit ULN2803A	2.00
10	Jumper wires (male to male)	4.00
1	Solderless breadboard	6.00
1	Solderable breadboard	10.00

Note. All the components mentioned in this note are included. See text for details on the specific components required for each configuration. *Any 28-V DC regulated power supply can be used. Each standard operant chamber draws approximately a maximum of 1.5 A of current.

formers were required to activate reliably the electromechanical equipment. The pulse former transformed each 15-ms response into a 30- or 40-ms pulse. During this pulse no responses were recorded. Therefore, it can be suggested that the Arduino®-VB2010 interface exceeds the requirements of typical operant research.

Conclusions

Combined with VB2010, Arduino® boards are a reliable low-cost alternative to commercial equipment for operant-conditioning research. An important advantage over other inexpensive systems is that the interface can be used with most desktop or portable computers running Windows® with an available USB port. Another advantage is that a variety of sensors (e.g., buttons, switches, photoresistors, force sensors) and output devices (e.g., LEDs, servo motors, step motors), which could be adapted for human and nonhuman animal research, can be connected with relative ease to the Arduino® boards. Additionally, the system could also be used for recording human behavior in natural settings.

Table 3 lists the interface components and their current prices. Even with an Arduino® Mega board and solid-state relays, the cost of the components, without the power supply, is approximately 100 USD. Although the cost is considerably lower than that of commercial equipment, the comparison may not be fair because commercial equipment consists of integrated systems of operant chambers, control equipment, and state-notation programming. The Arduino®-VB2010 interface requires connecting an array of relays to commercial or custom-made operant chambers, and VB2010 programming. In any case, the low price of the components is also important for maintenance. When an Arduino board malfunctions, it can be replaced and reconnected within minutes. It is also worth mentioning that after testing numerous Arduino® boards in a variety of environments during extended use, we found that not a single board has stopped working. The inexpensive, readily available microcontroller boards in combination with free software could help establishing or expanding operant-conditioning laboratories around the world.

Endnotes

References

Cabello,F., Barnes-Holmes, D., O'Hora, D., & Stewart, I. (2002). Using visual basic in the experimental analysis of human behavior: A brief introduction. *Experimental Analysis of Human Behavior Bulletin, 20,* 18–21.

D'Ausilio, A. (2012). Arduino: A low-cost multipurpose lab equipment. *Behavior Research Methods, 44,* 305–313.

Dixon, M. R. & MacLin, O. H. (2003). *Visual Basic for behavioral psychologists.* Reno, NV: Context.

Escobar, R., & Lattal, K. A. (2010). Low-cost interface using a parallel port and Visual Basic. *Mexican Journal of Behavior Analysis, 36,* 7–21.

Fleshler, M., & Hoffman, H. S. (1962). A progression for generating variable-interval schedules. *Journal of the Experimental Analysis of Behavior., 5,* 529–530.

Gollub, L. R. (1991). The use of computers in the control and recording of behavior. In I. H. Iversen & K. A. Lattal (Eds.), *Techniques in the behavioral and neural sciences: Experimental analysis of behavior* (Part 2, pp.155–192). Amsterdam: Elsevier..

Hoffman, A. M., Song, J., & Tuttle, E. M. (2007). ELOPTA: A novel microcontroller-based operant device. *Behavior Research Methods, 39,* 776–782.

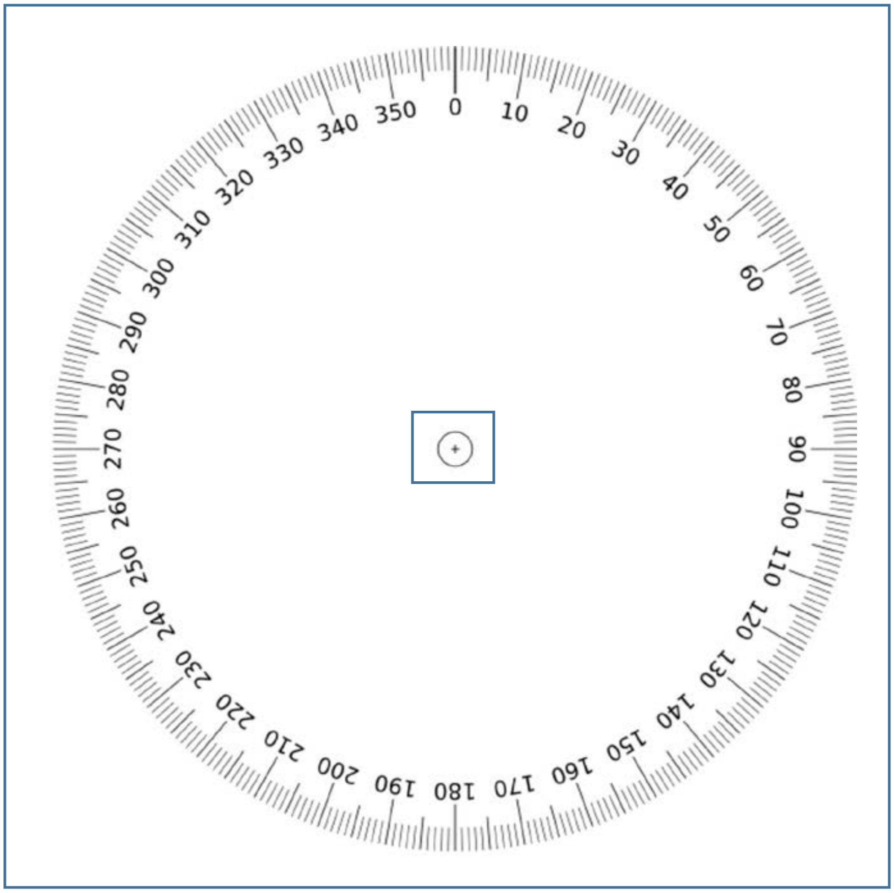
Palya, W. L., & Walter, D. E. (1993). A powerful, inexpensive experiment controller or IBM PC interface and experiment control language. *Behavior, Research Methods, Instruments & Computers, 25,* 127–136.

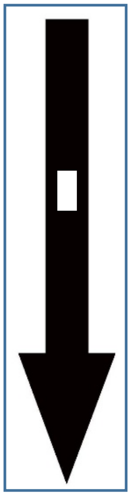
Schubert, T., D'Ausilio, A., & Canto, R. (2013). Using Arduino microcontroller boards to measure response latencies. *Behavior Research Methods, 45,* 1332–1346.

Received: June 16, 2014
Final Acceptance: December 5, 2014

Stepper 360 Dial

To measure the movement of the 28BYJ-48 stepper motor in degrees, cut out the meter face, affix it to the motor, and then fit the pointer onto the shaft of the motor.





Note the rectangles in the dial and pointer. Be sure to cut these out to allow the shaft of the motor poke through the dial and to allow the pointer to be fixed to the shaft.

Stepper motor with dial and pointer attached (the dial blocks your view of the most of the motor). You can see the shaft coming through the pointer.

Quick Reference

Input/Output (I/O) pins on the Arduino Uno. Some are capable of Pulse Width Modulation (PWM) (but need not be used that way). Some are capable of analog input (but need not be used that way).

Pin	Label	Use for Digital I/O?	Use for PWM Output?	Use for Analog Input?	Notes
0	0	Yes			1
1	1	Yes			
2	2	Yes			
3	~3	Yes	Yes		2
4	4	Yes			
5	~5	Yes	Yes		2
6	~6	Yes	Yes		2
7	7	Yes			
8	8	Yes			
9	~9	Yes	Yes		2
10	~10	Yes	Yes		2
11	~11	Yes	Yes		2
12	12	Yes			
13	13	Yes			3
14	A0	Yes		Yes	4
15	A1	Yes		Yes	
16	A2	Yes		Yes	
17	A3	Yes		Yes	
18	A4	Yes		Yes	
19	A5	Yes		Yes	

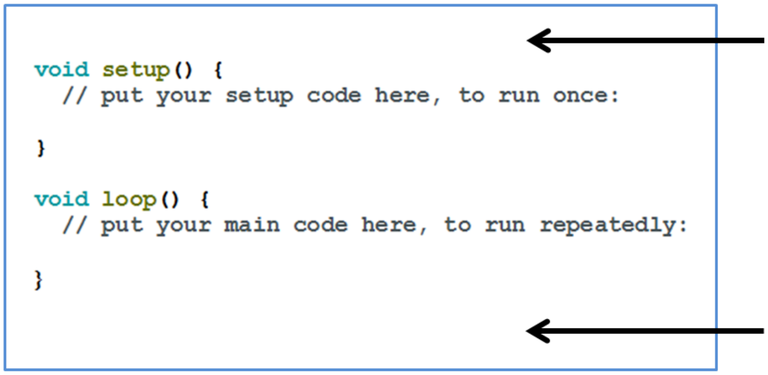
1. In sketches that involve serial communication, Pins 0 and 1 are occupied. If possible, don't use these two pins in your sketches. You have 18 other pins you can use for digital I/O.

2. Pins 3, 5, 6, 9, 10, and 11 are capable of Pulse Width Modulation (PWM) when configured as outputs. You don't have to use them for PWM; if you don't need PWM, you can use these pins as you would any other digital pin.

3. Do not use Pin 13 in INPUT_PULLUP mode. It is connected to an LED and a large diode which interefere with this mode.

4. Although these six pins are labeled with an "A" for "analog," they can be configured as digital inputs, digital outputs, or analog inputs. In sketches, you can refer to them with their labels (A0, A1, etc., which are treated as integer constants by the compiler) or you can refer to them by pin number (e.g., 14, 15).

Basic Structure of an Arduino Sketch



```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Add library references and declare global variables up here.

If you add your own functions, put them down here.

Variables, Constants, Arrays

int: Can store an integer (a whole number) between -32,768 and 32,767.

long: An integer that can vary from -2,147,483,648 to 2,147,483,647.

float: Floating-point numbers, that is, numbers with a decimal point.

String: Text, that is, a string of characters (note the upper-case ‘S”).

boolean: Can hold either of two values: true or false.

Declare a variable by stating the data type followed by the name, e.g.: `int respCount;`
To declare a constant, add const to the beginning and assign a value, e.g.: `const int ITI = 5000;`

An *array* is a collection of variables with a single name, differentiated by an index number. The most common way to declare an array is illustrated by this statement which would create an array of 100 integers, with index values from 0 to 99:

```
int responseLatency[100];
```

Arithmetic Operators

= Assignment operator, e.g. x = 3 assigns x the value of 3. Not to be confused with == which is a comparison operator.

+ Addition

- Subraction

* Multiplication

/ Division

% Modulo. Returns the remainder when one integer is divided by another. If x = 17 and y = 5 then x % y returns 2.

Math Functions

`abs(x)` Returns the absolute value of x.

`constrain(x,a,b)` Constrains x to the range from a to b inclusive. For example, if x = 106 then contrain(x, 0, 100) would return 100.

`map(val, fromMin, fromMax, toMin, toMax)` Takes “val” which can have a low of “fromMin” and a high of “fromMax”, and interpolates into a new range with a low of “toMin” and and a high of “toMax”. All values are integers. This functiont may come in handy when dealing with analog input.

`max(x,y)` Returns the higher of two numbers. If x = 5 and y = 1 then max(x, y) returns 5.

`min(x,y)` Returns the lower of two numbers. If x = 5 and y = 1 then min(x, y) returns 1.

`pow(b,e)` Raises b to the e^{th} power. If b = 10 and e = 3 then pow(b, e) will return 1000. Note that e can be a fraction: If b = 10 and e = .5 then pow(b, e) = 3.16.

`sqrt(x)` Returns the square root of x.

`int(x)`, `long(x)`, `float(x)` converts x to an int, a long, or a float, respectively.

`string.toFloat()` Converts string to a float.

`string.toInt()` Converts string to an int.

Comparison Operators

==	Equal to. Not to be confused with = which is the arithmetic assignment operator.
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

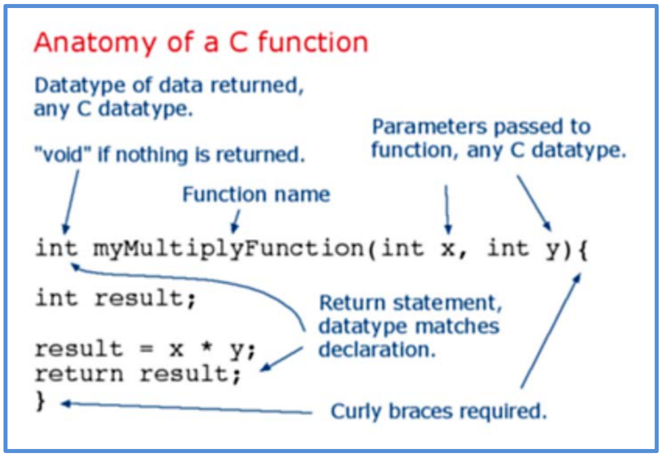
Boolean Operators

X	Y	X && Y	X || Y	! X
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

&&	Logical "and"
||	Logical "or" (these characters are typed by pressing your keyboard's back-slash key with the Shift key held down)
!	Logical "not" (negation)

Functions

A *function* is a block of code that performs some well-defined duty (it carries out a function) and can be called to action by other code within the sketch.


Anatomy of a C function
Datatype of data returned, any C datatype.
"void" if nothing is returned.
Function name
Parameters passed to function, any C datatype.
Return statement, datatype matches declaration.
Curly braces required.

```
int myMultiplyFunction(int x, int y){
int result;
result = x * y;
return result;
}
```

Serial Communication

`Serial.begin(`*`baud`*`);` Sets up the communications link between the Arduino and the PC. The “baud” parameter is replaced with a number representing the desired baud rate (e.g., 9600). It should appear in the setup section of your sketch.

`Serial.setTimeout(`*`milliseconds`*`);` Sets the amount of time, in milliseconds, that the sketch will spend reading the serial port before moving on. This function should appear in the setup section of your sketch. The default is 1000, a long time to wait. If you anticipate short strings, the time limit can be a few milliseconds. Try different limits to see what suits your purpose.

`Serial.available()` Returns the number of characters available to be read from the serial port. If the result is 0, then there is nothing in the buffer, that is, nothing to read.

`Serial.readString()` Tries to read a string of characters from the serial port. If will persevere until it reaches the time limit imposed by the “Serial.setTimeout()” function.

`Serial.print(`*`string`*`);` and **`Serial.println(`*`string`*`);`** Both functions send a string of characters from the Arduino through the serial port to the PC. They differ in one respect: “print” simply sends the string whereas “println” sends the string and follows with a newline character that causes the next string to be printed on the next line.

Timing

`delay(`*`milliseconds`*`);` Pauses (suspends) the sketch for the specified milliseconds.

`delayMicroseconds(`*`microseconds`*`);` Pauses the sketch for the specified microseconds.

`millis()`Returns the time in milliseconds since the Arduino began running the current sketch. This number will overflow (go back to zero) after approximately 50 days. Use a long variable, rather than an int, to store the result

`micros()` Returns the time in microseconds since the Arduino began running the current sketch. This number it returns will overflow (go back to zero), after approximately 70 minutes. Because micros() can return such a large number, use a long variable, rather than an int, to store the result.

Digital Input

`pinMode(`*`pin`*`, INPUT_PULLUP);` Configures the designated pin as an input and activates the internal pullup circuit so that the pin is held high in its resting state. It should appear in the setup section of your sketch.

`digitalRead(`*`pin`*`)`Returns the current state of the pin, LOW or HIGH.

Digital Output

`pinMode(`*`pin`*`, OUTPUT);` Configures the designated pin as an output.

`digitalWrite(`*`pin`*`, HIGH);` and **`digitalWrite(`*`pin`*`, LOW);`** Changes the state of the output pin; setting it HIGH allows 5V to flow and setting it LOW grounds it.

Analog Input

`analogRead(pin)` Reads the voltage on the designated pin and returns a number between 0 (if the pin is connected to GND) and 1,023 (if the pin is at 5V). The pin has to be one of the six on the Arduino Uno that are connected to analog-to-digital conversion channels. These pins are labeled A0 through A5 on the Arduino; your sketch can refer to them by these their labels (which are treated as integer constants by the Arduino compiler) or by their actual pin number (14 through 19).

Analog Output

`analogWrite(pin,dutyCycle);` The Arduino Uno does not support true analog output, but it can mimic it with this function. Here "pin" must be one of the pins capable of Pulse Width Modulation: 3, 5, 6, 9, 10, 11. "dutyCycle," is a value between 0 and 255 that expresses the part of the pin's normal cycle during which the pin will be HIGH. For example, a dutyCycle value of 63 is 25% of the way between 0 and 255. This value will cause the pin to be HIGH for 25% for the cycle (the "duty" part) and LOW for the other 75%.

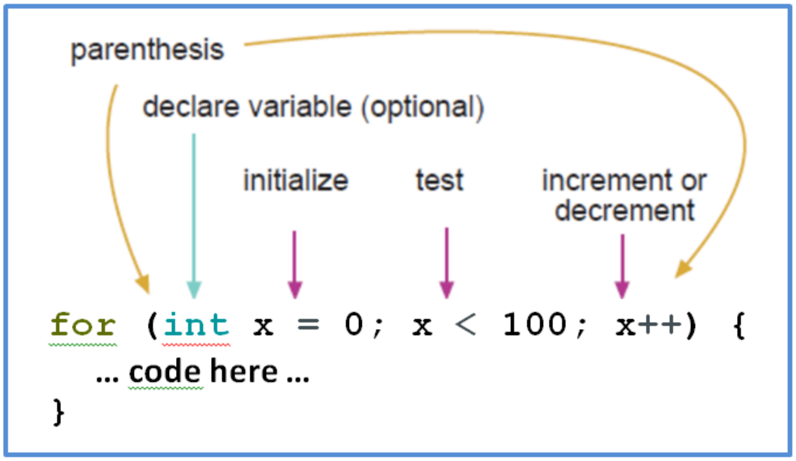
Branching

```
if (expression){
   … code that will be executed if the expression is true…
 }
```

```
if (expression) {
  … code that will be executed if the expression is true…
 }
   else {
  … code that will be executed if the expression is false…
 }
```

```
 switch (variable) {
    case 1:
     … code that will be executed if variable == 1…
      break;
    case 2:
     … code that will be executed if variable == 2…
      break;
    case 3:
     … code that will be executed if variable == 3…
      break;

    … etc., etc., as many cases as you need…

    default:
     … code that will be executed if no match is found…
      break;
 }
```

Looping



```
for (int x = 0; x < 100; x++) {
    … code here …
}
```

The for loop's header has three parts:

Initialization: Declare an integer type variable (if it hasn't been declared already) and set a starting value.

Test: A comparison involving the variable. The loop will execute until this comparison is false.

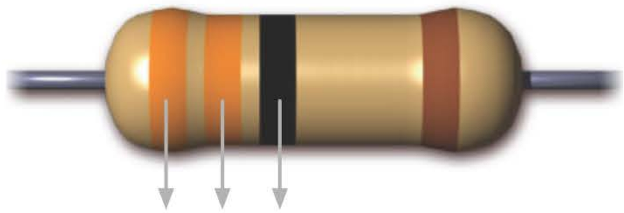
Increment or Decrement: An expression that increases or decreases the value of the variable.

In the example above, the code in the loop will run repeatedly, incrementing x after each iteration, until the comparison 'x < 100' is false.

Sound

`tone(`*`pin, frequency, duration`*`);` Generates a square wave on the designated pin, at the designated frequency (50% on, 50% off), for the specified duration in milliseconds. The last parameter is optional; if omitted, the tone will play continuously. The minimum frequency is 31 and the maximum is 65,535. On the Arduino Uno, this function will interfere with Pulse Width Modulation [i.e., with the analogWrite() function] on Pins 3 and 11.

`noTone(`*`pin`*`);` Turns off the square wave generated by a previous tone() function on the designated pin.



Match the **third band** to a chart below.
Then match the **first two bands** and read the value.

0	0	Ω	0	00	Ω	0	.0	kΩ	0	0	kΩ	0	00	kΩ	0	.0	MΩ	0	0	MΩ
1	1	Ω	1	10	Ω	1	.1	kΩ	1	1	kΩ	1	10	kΩ	1	.1	MΩ	1	1	MΩ
2	2	Ω	2	20	Ω	2	.2	kΩ	2	2	kΩ	2	20	kΩ	2	.2	MΩ	2	2	MΩ
3	3	Ω	3	30	Ω	3	.3	kΩ	3	3	kΩ	3	30	kΩ	3	.3	MΩ	3	3	MΩ
4	4	Ω	4	40	Ω	4	.4	kΩ	4	4	kΩ	4	40	kΩ	4	.4	MΩ	4	4	MΩ
5	5	Ω	5	50	Ω	5	.5	kΩ	5	5	kΩ	5	50	kΩ	5	.5	MΩ	5	5	MΩ
6	6	Ω	6	60	Ω	6	.6	kΩ	6	6	kΩ	6	60	kΩ	6	.6	MΩ	6	6	MΩ
7	7	Ω	7	70	Ω	7	.7	kΩ	7	7	kΩ	7	70	kΩ	7	.7	MΩ	7	7	MΩ
8	8	Ω	8	80	Ω	8	.8	kΩ	8	8	kΩ	8	80	kΩ	8	.8	MΩ	8	8	MΩ
9	9	Ω	9	90	Ω	9	.9	kΩ	9	9	kΩ	9	90	kΩ	9	.9	MΩ	9	9	MΩ

Resistor Decoder by Bret Victor

http://worrydream.com/ResistorDecoder/

Note: The fourth band is the resistor’s tolerance. Gold = 5%. Silver = 10%.