

# Microcontroller

A microcontroller is a one-chip computer. In a single package, it has the computing core and memory (or various types) needed to store and run small programs. There will be connections for power (obviously) and a serial communications channel for moving program information onto the controller and receiving information from the program.

In addition a microcontroller has a number of other input and output connections.

- **Digital inputs** from external sources. These will detect whether the voltage on the input is “high” or “low”, which correspond to logic level 1 and logic level 0. The low voltage is usually 0 V and the high either 3.3 V or 5 V, depending on the particular controller.
- **Digital outputs** that drive external components. These can be set to either a high or low voltage to activate (or de-activate) components attached to the output.
- **Analog inputs** that measure the measure voltage of a component attached to the input. The voltage must be between 0 and 5 V, typically.



# Microcontroller

## Other connections

- **Analog outputs** produce an analog voltage that can be applied to an external component. (Not all controllers have these.)
- **Pulse-width modulation (PWM)** produces a high/low square-wave signal where the ratio of the high time to the low time can be varied. PWM is usually produced by digital output pin. PWM can be used as a “fake” analog output for controllers that do not have real analog outputs.

Many of the pins are multi-purpose and can be configured to do different jobs. For example, a digital pin can be configured to be either an input or an output. The analog output pins can be set up to work in a digital fashion if needed.

These various input / output connections provide an interface between the digital core and the outside world, allowing for measurement of external effects and allowing for the microcontroller to make changes in response to external stimuli.



# Microcontroller

This combination of digital computation coupled with the connections to the outside world allow for the microcontroller to be the “brains” of a automatic control system. Since the tiny computer is an integral part the control system along with sensors and actuators, we call systems that use micro controllers “embedded systems”. In today’s world, embedded systems are everywhere:

- cell phones
- automobiles
- home appliances
- factories
- aircraft
- and on and on.

The “Internet of Things” takes isolated embedded systems and connects them to the internet, allowing for massive collection of data coming from embedded systems and massive control of remote devices. (And massive potential for security problems.)



# Analog vs. Digital

A key concept in beginning to design and use micro controllers in embedded systems is understanding the difference between analog and digital signals.

**Analog** - a continuously varying energy or quantity of material. Much of how we interact with the surrounding physical world is analog in nature – sound, sight, smell, taste. We use sensors to convert physical analog quantities to analog voltages or currents. (Microphones, photosensors, temperature sensors, etc.) Just like the physical quantity, the voltage or current is defined at every point in time. Detecting an analog voltage or current requires precisely measuring the volts or amps at a particular time. A reasonable example of an signal is a sine wave. (Although there isn't much information in a pure sine wave.)



**Digital** - The signal is either high or low (on/off or true/false or 0 /1). There are not many digital signals in the physical world. However, in terms of encoding information into signals, digital is much more robust. There is more leeway in interpreting whether a voltage is high or low. “High” means simply being “high enough” — above some threshold level. “Low” means simply being “low enough” — below some threshold level. Interpreting the information from a digital signal is less susceptible to errors because the voltages don’t have to be known precisely.

The systems that we have developed to collect information work best by collecting the analog information from the surrounding physical environment, converting it digital form, processing the information digitally, and then converting back to analog (if needed).



# Sensors

A key part of many embedded systems is measuring the surrounding environment. Sensors provide this type of input. There are many different kinds of sensors for measuring:

- voltage, current
- temperature (electronic, thermocouple, thermistor)
- humidity / pressure
- sound (microphone)
- light (photoresistor, photodiode, bolometer)
- distance (sonar, radar, lidar)
- mechanical motion (accelerometer, gyroscope)
- magnetic fields (coil, Hall-effect)
- chemical
- RFID tags



# Microcontrollers

- **Microchip** was the original microcontroller company. The first controller was the PIC1650 in 1976. (The company at that time was General Instruments. Microchip was spun out as a separate company in the late 1980s.) PIC originally stood for Peripheral Instrumentation Controller. Microchip sells more than 1 billion microcontrollers per year.
- **Atmel** was founded in 1984, specifically to compete in the microcontroller market. Their major product line is the AVR family of controllers. Earlier this year Atmel was bought by Microchip.
- **Texas Instruments (TI)** is one of the original electronics companies. TI engineer Jack Kilby was one of the co-inventors of the integrated circuit. The MSP430 is one of their primary microcontroller products.
- Many other companies

**ARM** is separate microcontroller architecture owned by ARM Holdings (from Great Britain) that is licensed by many companies. It is used in all smartphones and finds increasing use elsewhere.



# Microcontrollers

There are literally hundreds of microcontrollers available from the various companies. They differ on the number of bits (8, 16, 32, 64), memory (a few kilobytes to 1 gigabyte), clock speed (10s of kilohertz to 1 gigahertz), price (a few cents to 10s of dollars) and the number of inputs and outputs (3 or 4 up to 100 or more).

Start with 1 example - the Atmega328 from Atmel. (This is what is in the most popular Arduino.)

- 8-bit AVR architecture
- 8 – 20 MHz clock frequency (16 MHz is typical)
- 32 kbytes of flash memory (program storage)
- 2 kbytes of static memory
- 28-pin or 32-pin package
- 14 digital I/O
- 6 analog input (no analog out)
- 6 PWM outputs (part of the 14 digital I/O)



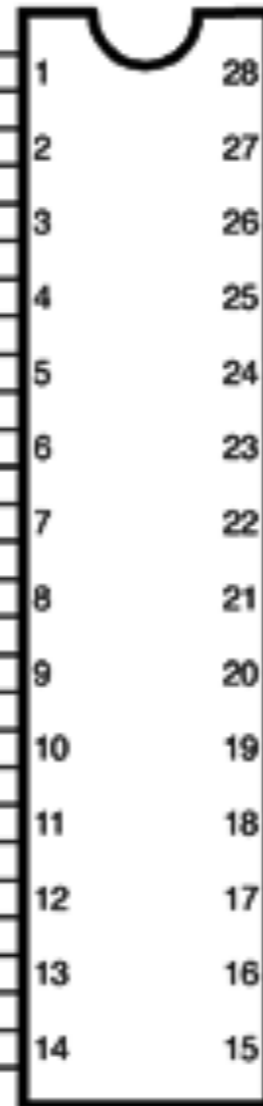
# Pin assignment for Atmega 168/328

## Atmega168 Pin Mapping

### Arduino function

reset  
 digital pin 0 (RX)  
 digital pin 1 (TX)  
 digital pin 2  
 digital pin 3 (PWM)  
 digital pin 4  
 VCC  
 GND  
 crystal  
 crystal  
 digital pin 5 (PWM)  
 digital pin 6 (PWM)  
 digital pin 7  
 digital pin 8

(PCINT14/RESET) PC6 □ 1  
 (PCINT16/RXD) PD0 □ 2  
 (PCINT17/TXD) PD1 □ 3  
 (PCINT18/INT0) PD2 □ 4  
 (PCINT19/OC2B/INT1) PD3 □ 5  
 (PCINT20/XCK/T0) PD4 □ 6  
 VCC □ 7  
 GND □ 8  
 (PCINT6/XTAL1/TOSC1) PB6 □ 9  
 (PCINT7/XTAL2/TOSC2) PB7 □ 10  
 (PCINT21/OC0B/T1) PD5 □ 11  
 (PCINT22/OC0A/AIN0) PD6 □ 12  
 (PCINT23/AIN1) PD7 □ 13  
 (PCINT0/CLKO/ICP1) PB0 □ 14



28 □ PC5 (ADC5/SCL/PCINT13)  
 27 □ PC4 (ADC4/SDA/PCINT12)  
 26 □ PC3 (ADC3/PCINT11)  
 25 □ PC2 (ADC2/PCINT10)  
 24 □ PC1 (ADC1/PCINT9)  
 23 □ PC0 (ADC0/PCINT8)  
 22 □ GND  
 21 □ AREF  
 20 □ AVCC  
 19 □ PB5 (SCK/PCINT5)  
 18 □ PB4 (MISO/PCINT4)  
 17 □ PB3 (MOSI/OC2A/PCINT3)  
 16 □ PB2 (SS/OC1B/PCINT2)  
 15 □ PB1 (OC1A/PCINT1)

### Arduino function

analog input 5  
 analog input 4  
 analog input 3  
 analog input 2  
 analog input 1  
 analog input 0  
 GND  
 analog reference  
 VCC  
 digital pin 13  
 digital pin 12  
 digital pin 11 (PWM)  
 digital pin 10 (PWM)  
 digital pin 9 (PWM)

Digital Pins 11, 12 & 13 are used by the ICSP header for MOSI, MISO, SCK connections (Atmega168 pins 17, 18 & 19). Avoid low-impedance loads on these pins when using the ICSP header.

Note: Atmega328 has same pin configuration as Atmega 168.



# Developing embedded systems

What used to be needed to get started? (In the “old” days.)

- To make the hardware operational, the controller chip will need: a power supply, a clock oscillator crystal, a serial communications connection, and physical connections for all of the digital and analog pins. Typically, this is provided by a prototyping platform available from the microcontroller supplier. Of course, developers can also build their own prototype hardware, but this is usually not an effective use of resources.
- Integrated Development Environment (IDE). Software for the controller must be written on a regular computer (Windows/MacOS/Linux) and then transferred to the controller. Typically, the IDE is not particularly user friendly — some prior software experience is probably needed. Also, a special hardware dongle (a programmer) is needed for the transfer.

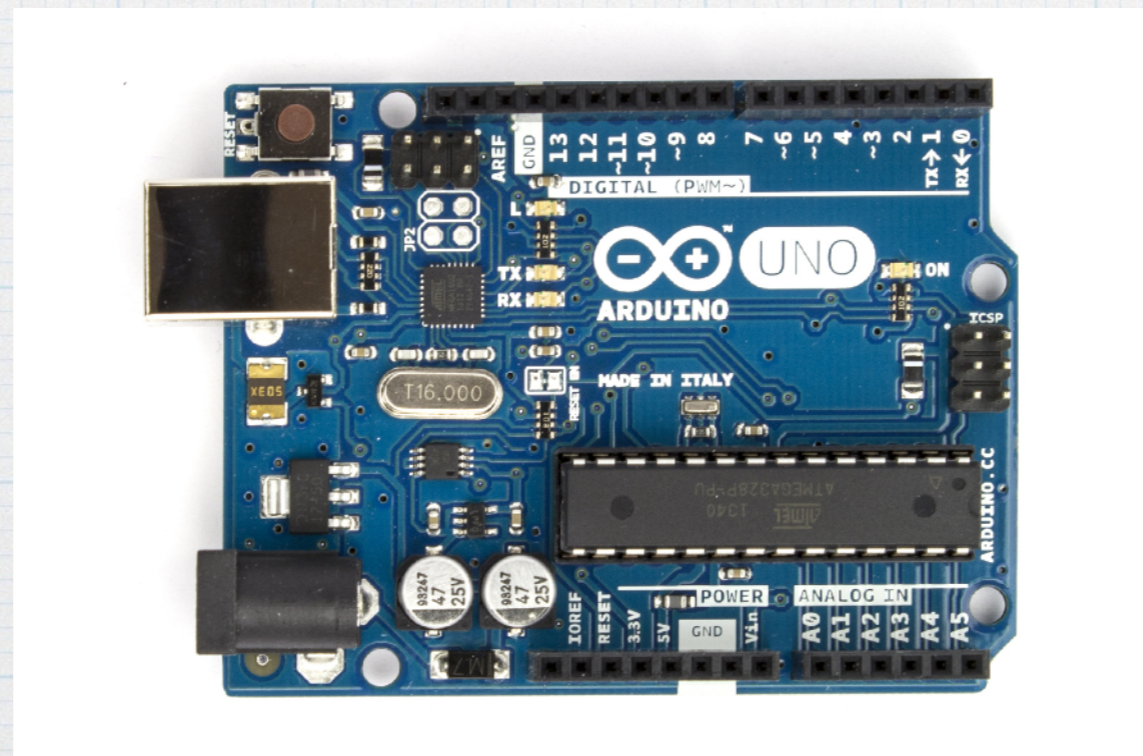
Upshot: Developing embedded systems is only for “professionals”.  
Hobbyists or “makers” need not apply.



# Enter Arduino

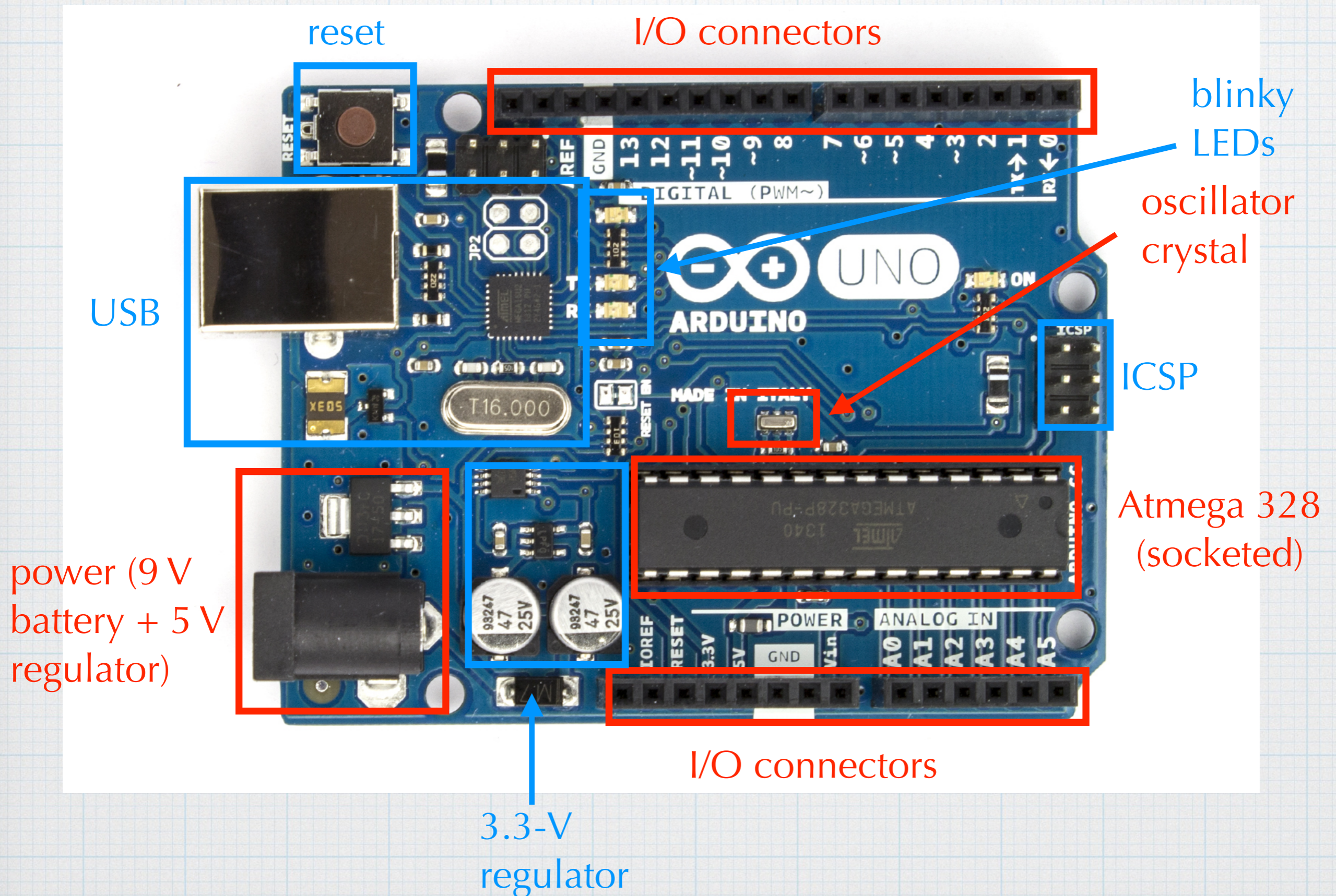
In 2005, a group at the Interactive Design Institute Ivrea in Ivrea, Italy put together a hardware / software platform that would make it much easier to build embedded systems prototypes. (The titular leader of the group was — and still is — Massimo Banzi, although there were four other founders.) They named the platform *Arduino*, after a bar that they frequented. They were focusing on making a system on which students and hobbyists could learn easily and would be affordable to non-professionals. Their efforts unleashed the current “maker” movement.

There have been several iterations on the basic platform over the years. The current standard hardware arrangement is the Arduino Uno.



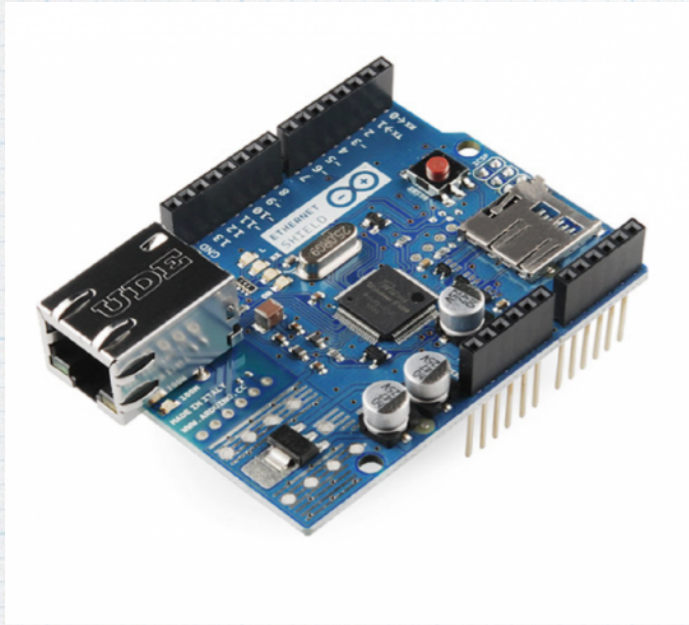


# Arduino Uno hardware

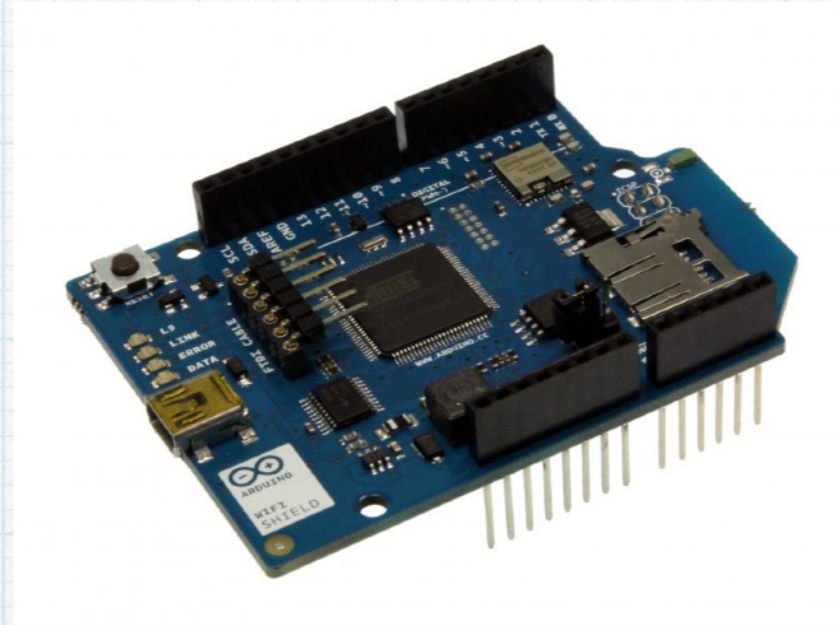




# Hardware expansion through "shields"



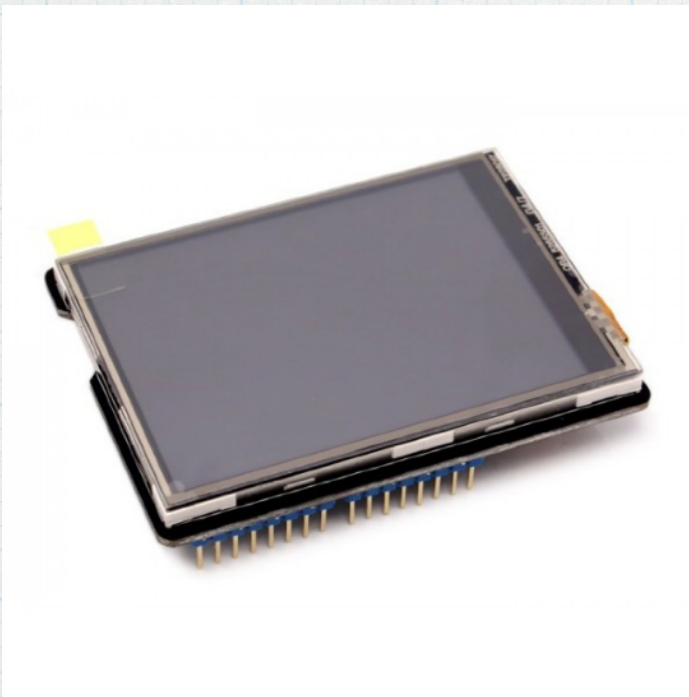
Ethernet



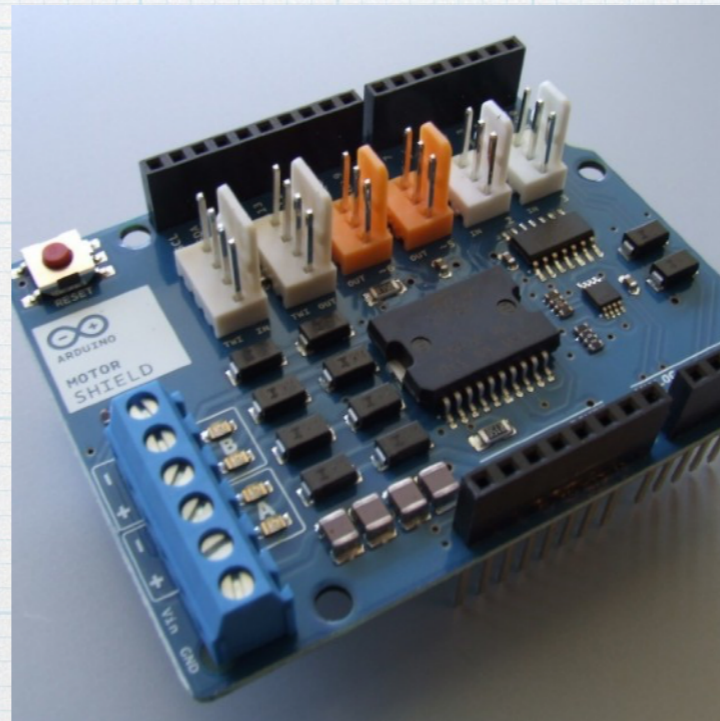
WiFi



Bluetooth



Touch display



Motor





# Arduino Uno software

- The IDE for the Arduino is a Java application (runs identically on Windows / Mac / Linux).
- The programs (called sketches) are written in a subset of C. (Good for us!) There some special commands for defining the operation of I/O and for receiving/sending information to those pins.
- There is a USB interface to facilitate transferring programs from the IDE on the computer to the controller. (No hardware dongle!) The USB interface also conveniently provides power when connected to a computer.
- There is an interface analogous to the console for sending information and receiving information from the controller. (serial monitor)
- Note: The controller is meant to operate in a stand-alone mode. It is not intended to be tethered to the host computer indefinitely. But when developing the software, it is important to be able to “see” what is happening with the program.
- Everything is open-source. (Including the hardware design.)



# What is needed to get started?

**An Arduino Uno board - \$25**

[arduino.cc](http://arduino.cc)

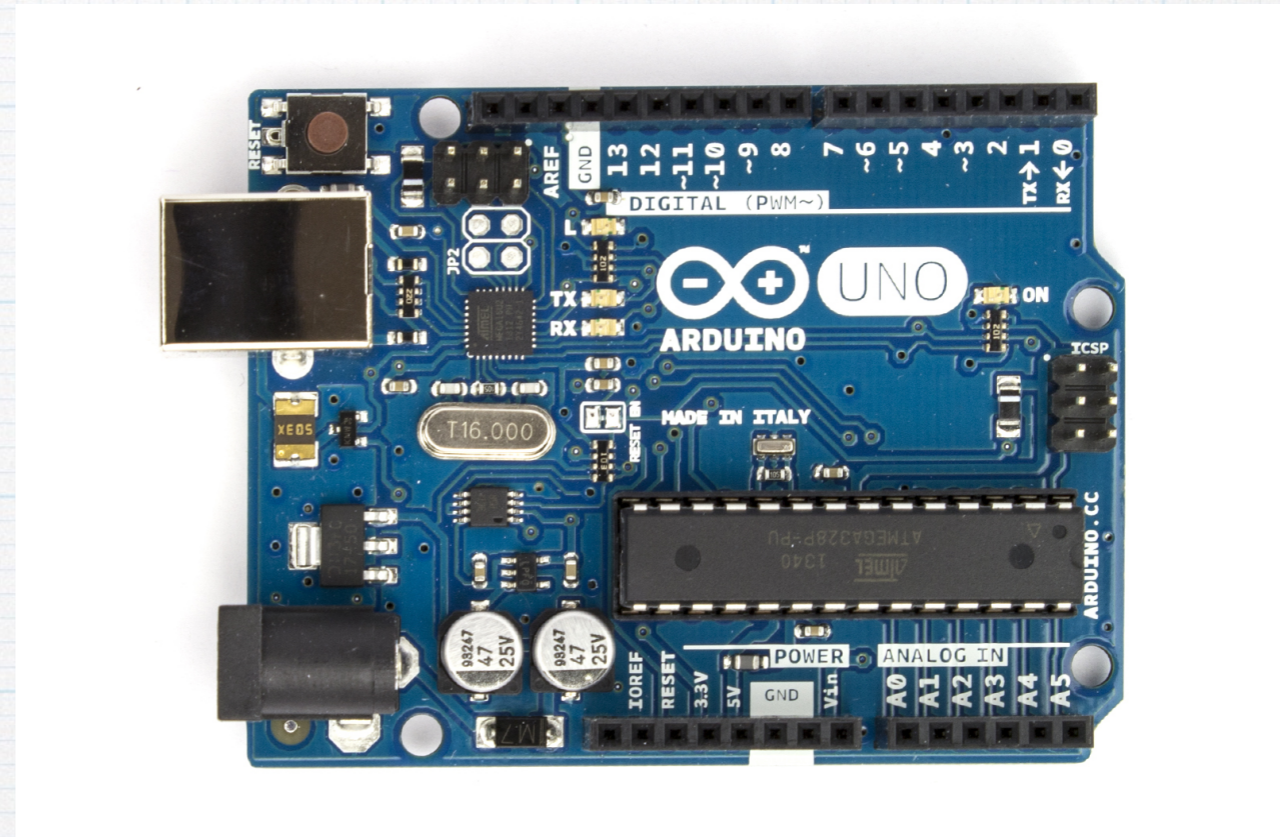
[amazon.com](http://amazon.com)

[sparkfun.com](http://sparkfun.com)

[adafruit.com](http://adafruit.com)

[element14.com](http://element14.com)

Digikey, Mouser, Jameco, etc  
(Beware of knock-offs.)



**USB - A to B cable**

(Standard printer connection.)



**Download Arduino IDE**

<https://www.arduino.cc/en/Main/Software>



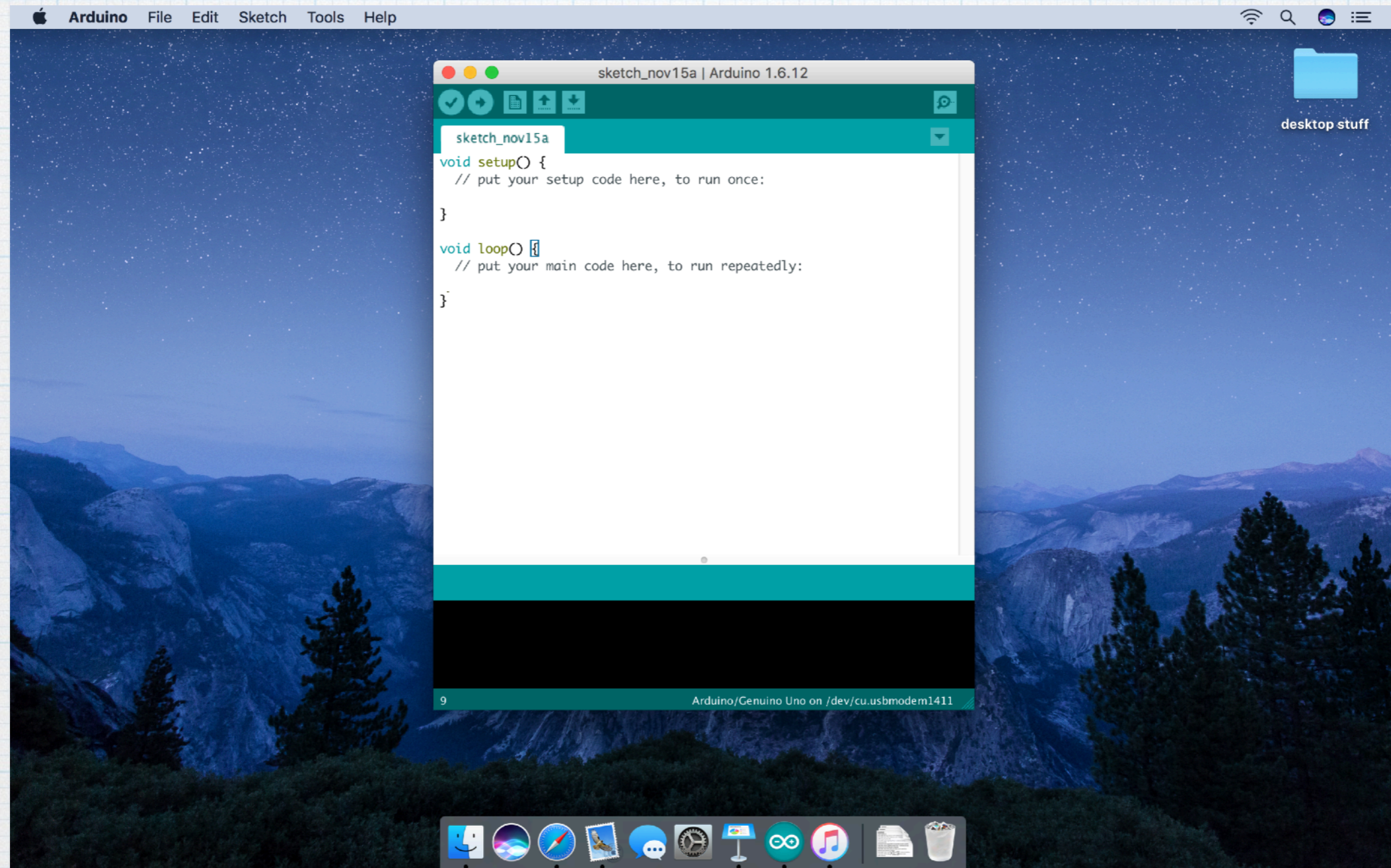
# Structure of an Arduino program

An Arduino program (sketch) is slightly different from our typical C program. In a microcontroller, there is no `main` function. Instead there are two separate functions — one called `setup()` and a second called `loop()` runs in a continuous loop.

- The `setup` function behaves just like a regular C program. Program flow starts at the top and progresses through to the last line. (If there are any loops, they behave according to the conditionals like we would expect.) As the name implies, the purpose of `setup` is to define variables, set the function of the pins, and take care of any preliminary housekeeping. When `setup()` is complete, program flow passes to `loop()`.
- The program continues with the first line of `loop` and then progresses through to the last line. But, as the name implies, the program goes back to the first line of `loop` and runs again. And again and again. Forever. (Or until the power is removed.) It is as there is an implied `while` loop whose conditional is always true.



Once you have downloaded the Arduino software and installed it\*. When you launch it, you will get a blank, generic program. The setup and loop functions are empty.

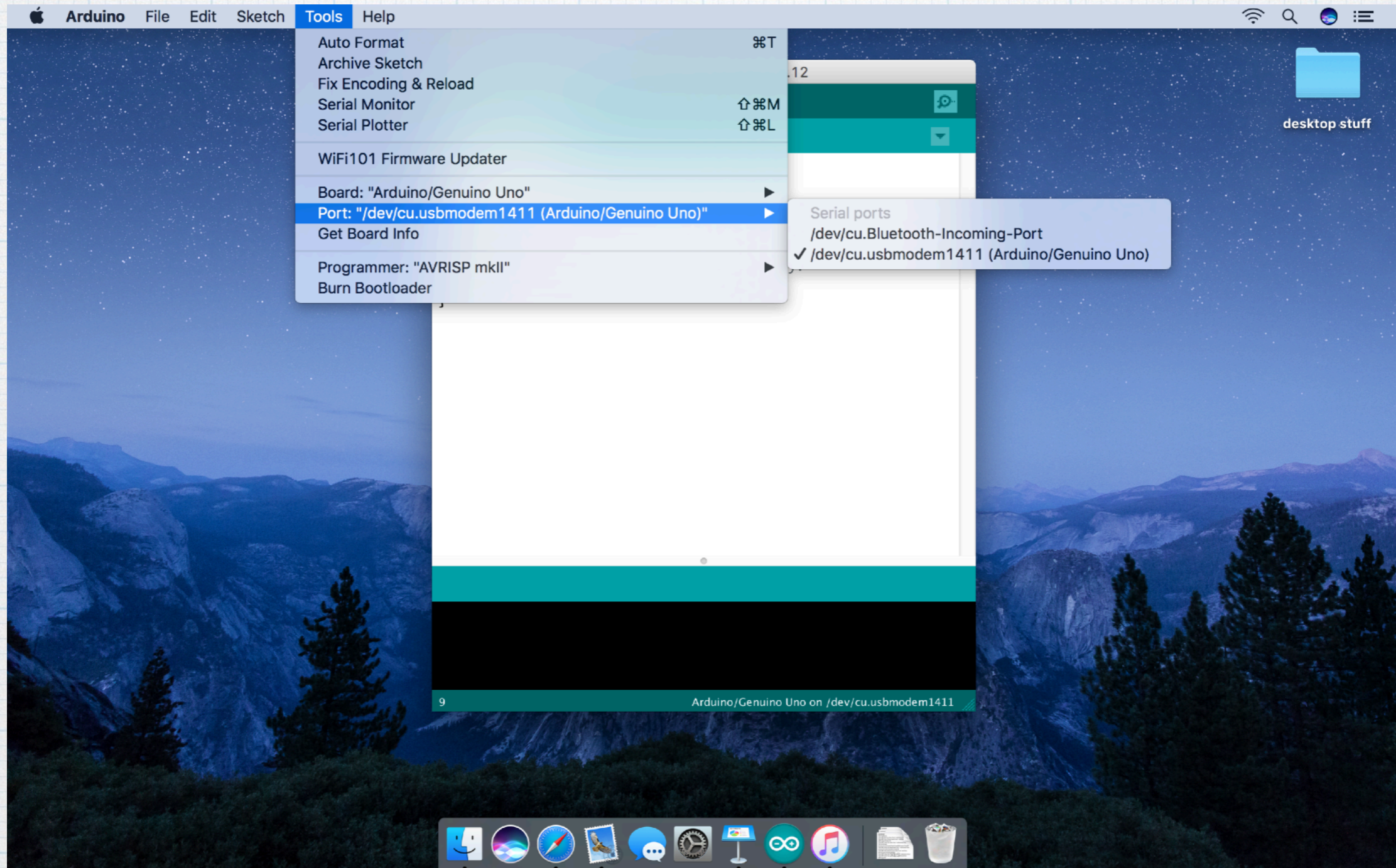


\*MacOS does not have Java installed by default. When you try to run the Arduino IDE for the first time, you will be asked if you want install Java. Follow the instructions to do so.



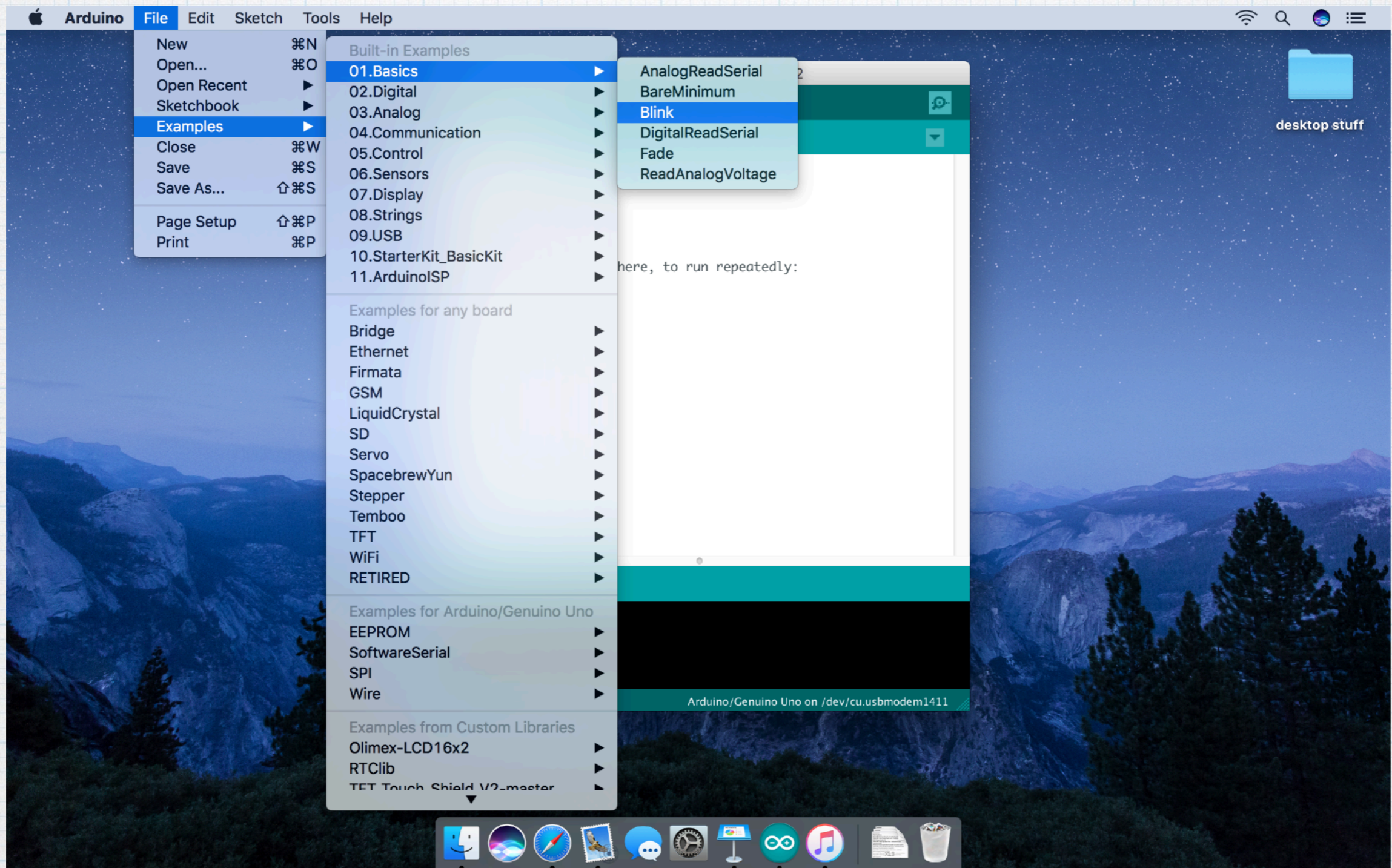
Plug the Arduino board into a USB port.

You will need to choose the correct USB “port”. (Might require a bit of trial and error to get the right port. A wrong port will generate an error when you try to upload a program to the Arduino.)

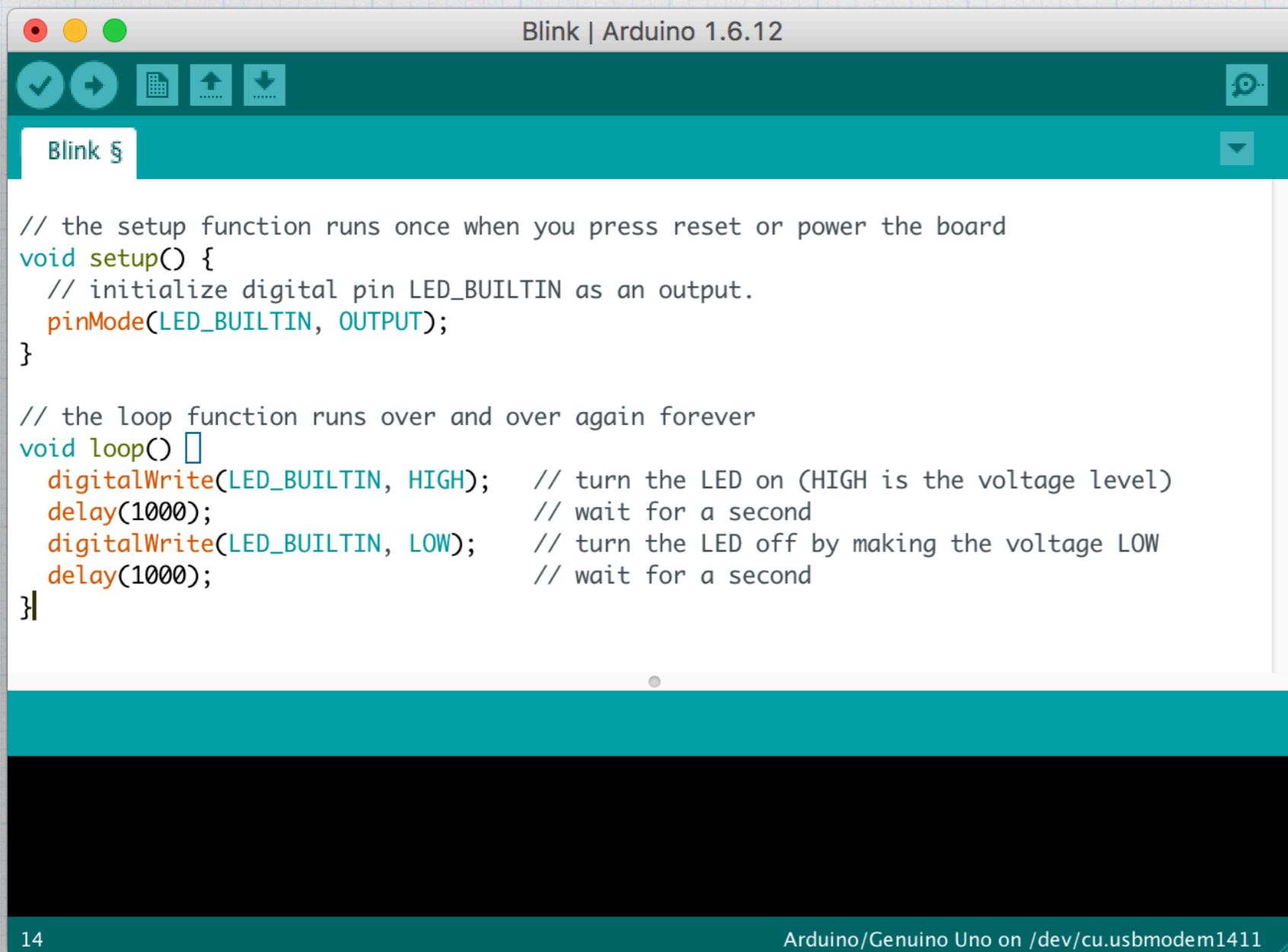




There are many example programs that you can use to help in learning the various commands. The simplest is a program called “Blink” that will make one of the light-emitting diodes on the board blink on and off. Choose it from the menu.





A screenshot of the Arduino IDE interface. The window title is "Blink | Arduino 1.6.12". The code editor shows the following code:

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000); // wait for a second
  digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW
  delay(1000); // wait for a second
}
```

The status bar at the bottom shows "14" on the left and "Arduino/Genuino Uno on /dev/cu.usbmodem1411" on the right.

Note: A bunch of comments at the top of the program have been removed to save space.

Inside `setup()`: The `pinMode` command defines the pin connected to the built-in LED (also known as pin 13) to be a digital output.

Inside `loop()`: The first command sets the voltage of the LED pin to the high value (5V), turning on the LED. The second command is a 1 s (1000 ms) delay. (Basically, do nothing for 1 second.) The third command sets the voltage of the LED to the LOW value (0 V), turning off the LED. Then there is another 1000 ms delay. Then everything repeats — the LED will repeatedly blink on and off.