1. Describe the EEPROM in Arduino Uno w.r.t. volatility, size, etc.?
   1. EEPROM stands for Electrically Erasable Programmable Read-Only Memory. The microcontrollers used on most of the Arduino boards have either 512, 1024 or 4096 bytes of EEPROM memory built into the chip. This memory is non-volatile, which means that the data doesn’t get erased when the board loses power.
   2. You can look at the EEPROM on Arduino as an array where each element is one byte. When reading from and writing to this memory, you specify an address which in the Arduino world is equivalent to an array index.
   3. This means when can use this short term to remember states from before powering down. For example whether or not the LED was on or off.
   4. EEPROM has a total lifetime of ~100,000 write cycles. Be careful when writing code so that you don’t write to EEPROM too often! Remember that erasing memory also is a writing operation.
2. Write a small program to show the commands to read and write from the built in EEPROM.

//Program to save state of LED

#include <EEPROM.h>

const int buttonPin = 8; // pushbutton pin

const int ledPin = 4; // LED pin

int ledState; // variable to hold the led state

int buttonState; // the current reading from the input pin

int lastButtonState = LOW; // the previous reading from the input pin

// the following variables are long's because the time, measured in miliseconds,

// will quickly become a bigger number than can be stored in an int.

long lastDebounceTime = 0; // the last time the output pin was toggled

long debounceDelay = 50; // the debounce time; increase if the output flickers

void setup() {

// set input and output

pinMode(buttonPin, INPUT);

pinMode(ledPin, OUTPUT);

// set initial LED state

digitalWrite(ledPin, ledState);

// initialize serial monitor

Serial.begin (9600);

//check stored LED state on EEPROM using function defined at the end of the code

checkLedState();

}

void loop() {

// read the state of the switch into a local variable

int reading = digitalRead(buttonPin);

if(reading != lastButtonState) {

// reset the debouncing timer

lastDebounceTime = millis();

}

if((millis() - lastDebounceTime) > debounceDelay) {

// whatever the reading is at, it's been there for longer

// than the debounce delay, so take it as the actual current state:

// if the button state has changed:

if(reading != buttonState) {

buttonState = reading;

// only toggle the LED if the new button state is HIGH

if(buttonState == HIGH) {

ledState = !ledState;

}

}

}

// set the LED state

digitalWrite(ledPin, ledState);

// save the current LED state in the EEPROM

EEPROM.update(0, ledState);

// save the reading. Next time through the loop,

// it'll be the lastButtonState

lastButtonState = reading;

}

// Prints and upates the LED state

// when the Arduino board restarts or powers up

void checkLedState() {

Serial.println("LED status after restart: ");

ledState = EEPROM.read(0);

if(ledState == 1) {

Serial.println ("ON");

digitalWrite(ledPin, HIGH);

}

if(ledState == 0) {

Serial.println ("OFF");

digitalWrite(ledPin, LOW);

}

}

1. Describe a few possible use cases for the built in EEPROM.
   1. Storing temperature data, or remember the directions in a particular maze for a robot. Anything that needs that extra short term memory.
2. How many I2C devices can you connect on a single bus? Explain why?
   1. In the best case scenario, using boards with pull-up resistors of 10kΩ, you could connect 10 boards together, resulting in a total pull-up resistance of 1kΩ, which would be fine for a VCC of 3.3V. For a VCC of 5V however, you could connect 6 boards with pull-up resistors of 10kΩ each, resulting in a total resistance of 1.67kΩ.
   2. When we connect multiple breakout boards together, we are actually connecting these resistors in parallel to each other, decreasing the total resistance. Even connecting two boards with pull-up resistors of 2.2kΩ would reduce the overall resistance to 1.1kΩ. That would still be fine for a power supply of 3.3V, but would be lower than the minimum value of 1.53kΩ for a VCC of 5V.
3. Describe a few characteristics of the I2C bus, for example: speed, type of devices connected to it, number of lines used, limitations, benefits, etc.
   1. WHY USE I2C?
      1. Far easier to use than SPI.
      2. Uses fewer connections than SPI.
      3. Supports a multi-master system, rather than SPI’s one master, infinite-slaves system.
      4. 100% compatible with our Arduino Uno.
   2. I²C uses 2 signals for transferring data between devices, this halves the SPI’s 4 wire communication protocol. The signals we will use are called:
      1. Data signal (SDA) – This is where all of our control information and data is sent/received. Everything is sent on this signal and can be sent anywhere on the bus.
      2. A clock signal (SCL) – This is purely for regulating the speed the data is sent. The slave devices will sample a bit from the data stream in synchronization with this signal.
   3. Devices that use I²C are “open drain” on the data signal. This means that they are only able to pull the line low, eliminating the risk of bus contention which can damage components. Every signal line has an internal pull-up resistor that drive the line back to high when a device is using it.