

Microcontroller Programming with the Arduino

Contents

Intro.....	3
Audience	3
Your kit	3
Development Environment.....	5
Connecting the Arduino	5
Setting up your programming environment	5
Ubuntu	5
Windows 7/8/10, OS X, Linux (other)	5
Interface basics	6
Programming the Arduino	7
Most basic Program (Sketch)	7
Running the program	7
Blinking Pin 13.....	8
Your first module: The buzzer.....	9
Talking with your Arduino.....	10
Installing Libraries	12
The Liquid Crystal Displace (LCD) Shield	13
Digital Multi-Function Shield	15
LED control.....	15

Button Click Detection	16
Driving the 7-segment display	16
Sound Detector	19
Obstacle Sensor	20
Ultrasonic Sensor	21
Digital Compass.....	23
IMU: Internal Measurement Unit	25

Intro

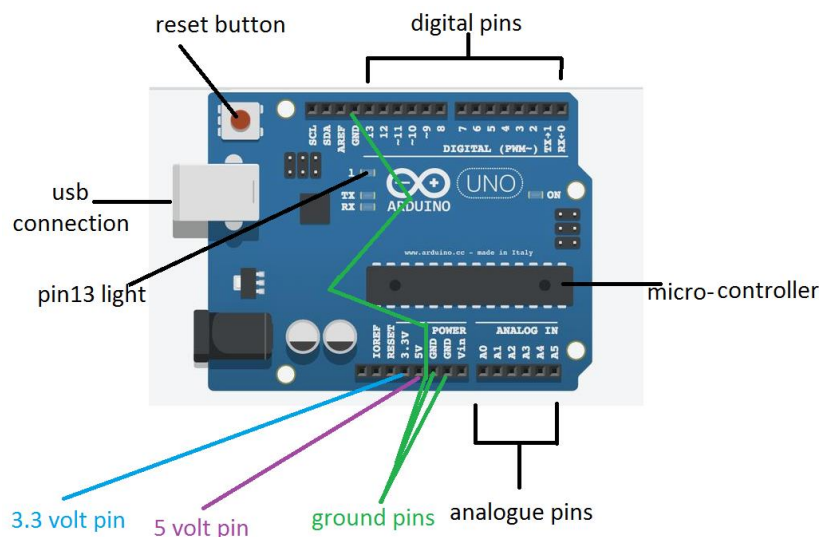
Audience

This tutorial pdf is intended for those students currently studying the first year programming course in C. It will teach you the basics of programming the arduino microcontroller without any assumptions that the reader knows anything about electronics, all basic concepts are learnt along the way.

Your kit

As part of your course you should have access to the arduino and the ICs (integrated circuits allow you to extend the functionality of your arduino), ask your teacher about these if you haven't got a hold of them yet (you may have to pay a small fee for the kit).

Here is a small description of each element (don't worry if you don't have a particular item, it may have been taken out for repairs or been removed from the course).



The Arduino microcontroller

- The USB connection is where you will plug your Arduino into your desktop or laptop.
- The reset button resets the program currently running on the Arduino.
- The digital pins can read and write output/input 1s and 0s (bits) as high (5 volt) and low (less than 3 volts) voltages. They can also simulate analogue output in the form of something called pulse width modulation, which is a fancy name for flipping between high and low voltage at different speeds so that the output is high for some percentage of time to simulate analogue output.
- The pin13 light is an LED (light emitting diode), if you write a 1 to pin13, the light will turn on.
- The micro-controller processes your program to control the pins on the Arduino.
- The 3.3 volt and 5 volt pin supply voltage to devices connected to the Arduino.
- The ground pins complete the circuit from devices connected to the digital/analogue output or 5v/3.3v pins.
- The analogue pins can read analogue input. They can also do general I/O too.



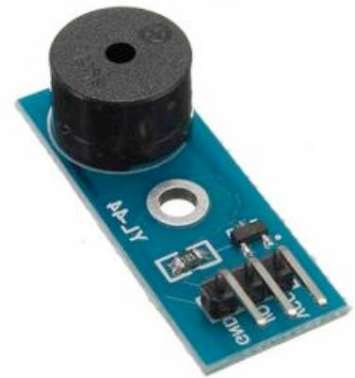
digital multi-function shield

- Is connected to the top of your Arduino
- Has a reset button too
- Has 3 buttons which you can read input
- Has 4 LED lights
- Has a 7 segment (digital alarm clock) display



LCD shield

- Can write text to LCD screen
- Has several buttons
- Turn the screw on the blue box to adjust screen brightness



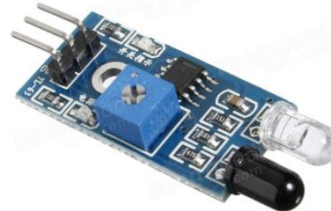
Buzzer

- Can make annoying sounds at different pitches



Sound Sensor

- Can detect sound
- Turn the screw to adjust sensitivity



Obstacle sensor

- Detects if an object is within a defined range
- Turn the screw to change the range



Ultrasonic sensor

- Detects the distance between the object it faces and itself



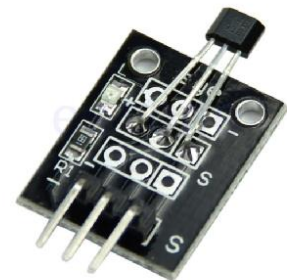
Digital compass

- Detects the axis it is rotated at



IMU – Internal Measurement Unit

- Detects the axis it is rotated at
- Detects accelerations it is put through



Temperature sensor

- Can detect the temperature of the current environment

Development Environment

Connecting the Arduino



The first step is to get your cable, connect one end to the Arduino and the other to the USB port of your laptop or Desktop. Use the picture above as a reference, some lights on the Arduino should light up.

Setting up your programming environment

We will be using the Arduino IDE (integrated development environment) to write and upload our programs to the Arduino device. The website for this IDE is: <https://www.arduino.cc/en/Main/Software> .

Installation instructions are as follows:

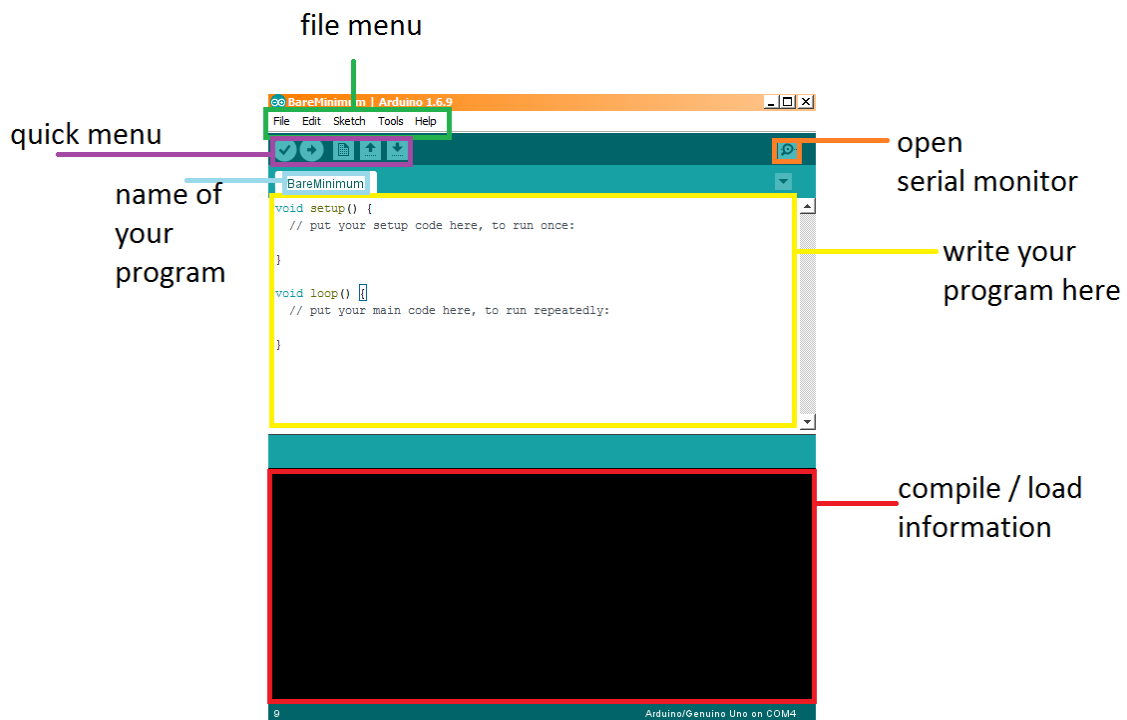
Ubuntu

1. Open the terminal (Ctrl + t)
2. Paste in: `sudo apt get install arduino arduino-core`
3. Follow prompts

Windows 7/8/10, OS X, Linux (other)

1. Go to <https://www.arduino.cc/en/Main/Software>
2. Download the correct installer
3. follow prompts

Interface basics



- In the file menu (green) you can do various operations such as:
 - File -> open/save your program (called a sketch)
 - Edit -> find/replace/copy and paste
 - Sketch -> compile/upload program to Arduino
 - Tools -> set the port the Arduino is connected to, set the board type (we are using the UNO)
- The text-editor (yellow) is where you will write your programs
- The compile/load information section (red) displays compile time information
- The quick menu has 5 buttons
 - The tick button compiles your program
 - The arrow button uploads the program to the Arduino (only if it compiles first)
 - The rest of the buttons are: Open a new file, open a specific file, save
- Serial Monitor open button (orange)
 - Picture below
 - Allows the developer to send information to the Arduino
 - Allows the developer to read information sent from the Arduino
 - If Auto scroll is checked, the latest message from the Arduino is focused on
 - 9600 baud rate is the rate at which the Arduino is communicating with your IDE, 9600 baud is 9600 bits per second



Programming the Arduino

Most basic Program (Sketch)

The language we will be using to program the Arduino will be a variation of C/C++.

Take a look at the most basic program for Arduino, it does nothing. There are two functions setup and loop. The setup function is called when the program begins and the loop function is called continually whilst the Arduino is running.

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

Figure 1 Most basic program

Essential the Arduino executes the following C program: “setup(); while(1) loop();”

Running the program

1. Connect your Arduino to your laptop/desktop
2. Enter the most basic program into the editor (should appear by default)
3. Click the compile button (the tick) to verify the program is legitimate C
4. Click the upload button (you may have to save the program first)
5. Stand back and watch as the Arduino blinks a few times to let you know a program is being uploaded before doing... nothing

Note: if the upload fails try:

1. going to tools->board and make sure the UNO is selected
2. going to the tools->port and select the right board (you may have a few options, try them until you find which USB port your Arduino is connected to.

Blinking Pin 13

```
void setup() {  
  // initialize digital pin 13 as an output.  
  pinMode(13, OUTPUT);  
}  
  
// the loop function runs over and over again forever  
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  
}
```

The digital pin 13 is special, not only can you connect other pins to it for input/output but it is also connected to an LED on the board (see the Your Kit section with the description for the Arduino board).

We can set this pin HIGH to light up the LED, and low to turn the LED off.

1. `pinMode` - The first thing we do is initialize pin 13 as an output pin using the `pinMode` function. This takes in the pin selected as the first argument and `OUTPUT` or `INPUT` as the second argument. We are going to be setting the output for this pin so we use `OUTPUT`.
2. Loop function:
 - a. `digitalWrite`: this function takes the pin id as the first argument and whether the pin should output `HIGH` (digital 1) or `LOW` (digital 0). Note: `HIGH` and `LOW` are pre-processor defines for `0x1` and `0x0` respectively.
 - b. `delay` causes the program to hang waiting for `delay` to return, for `x` milliseconds where `x` is the first argument.

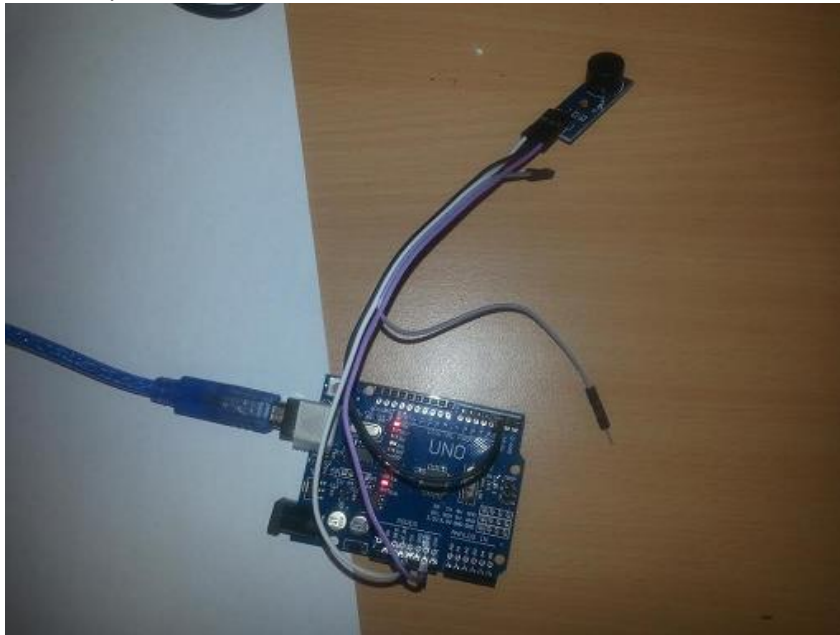
Compile and upload your program to your Arduino, the pin 13 LED should blink at a rate of 0.5 times per second (turn on, wait for 1 second, turn off, wait for 1 second).

Your first module: The buzzer

The buzzer module makes an awful sound, yay!

This IC has 3 pins to attach to the Arduino, a VCC pin which connects to any of the 5 volt pins on the Arduino, a GND pin which connects to any of the ground pins on the Arduino and an input pin (named I/O on the device) for sending through a 1 or a 0 as input data.

A picture below shows my wiring, not the colours of the pin cables (called male to female wire connectors) do not mean anything. The buzzer pins are marked on the IC (VCC, I/O, GND), for this tutorial I am using digital pin 3 for the I/O pin.



The way the IC works is as follows, the I/O pin on the buzzer must receive a 1 followed by a 0 at some frequency f . The frequency defines the frequency of the sound the buzzer makes.

```
int ioPin, frequency = 400;
void setup() {
  ioPin = 3;
  pinMode(ioPin, OUTPUT);
}
void loop() {
  digitalWrite(ioPin, LOW);
  delay(500 / (double)frequency);
  digitalWrite(ioPin, HIGH);
  delay(500 / (double)frequency);
}
```

In this program we setup 2 variables, ioPin specifies the pin we connected I/O to, and frequency is the frequency we change our input signal at. Note: as you can see we can initialize frequency/ioPin/any global variable in global space or in setup(). Then in our loop function, using the digitalWrite function we learnt, we set the pin LOW, then wait for $500 / \text{frequency}$ milliseconds, this equation converts our frequency variable to how long we should delay to write both a 0 and a 1 at frequency times per second (delay per 0 and 1 send = $1000\text{ms}/\text{frequency}$, delay per 0 or 1 send = $1000\text{ms}/\text{frequency}/2 = 500/\text{frequency}$).

Talking with your Arduino

Exchanging messages between yourself (through the Serial Monitor) and your Arduino does not require any modules. You can do it if you want to alter the logic or control of your program, view sensors' input and/or debug your program (see what's going on and why your program may not be working).

```
int loopCounter; //counts the number of loop() function calls
void setup() {
  Serial.begin(9600); //setup our Serial i/o @ 9600 bits per second
  loopCounter = 0;
}

void loop() {
  Serial.print("hello"); //prints hello with no new line character
  String arduinoString = " world"; //an Arduino string
  Serial.println(arduinoString); //prints the Arduino string with a new line character

  //C way to print-out loopCounter
  Serial.print("counter: ");
  char buffer[100];
  char * ptr = itoa(loopCounter, buffer, 10);
  char buffer2[100] = "C count: ";
  strcat(buffer2, ptr);
  Serial.println(buffer2); //can also do: Serial.print(buffer2); Serial.println(loopCounter);

  //Simpler C++/Arduino way
  Serial.println("C++/Arduino count: " + String(loopCounter));

  loopCounter++;
  delay(2000);
}
```

In the code above, we have an int variable called loopCounter which we will be using to count the number of times loop is completed. Notice in our setup function we have a line Serial.begin(9600), this sets up our serial communication to/from the Arduino and computer at a rate of 9600 bits per second (a frequently used comm rate for Arduino). According to Arduino supported baud rates include: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 31250, 38400, 57600, and 115200.

In our loop function we use the function Serial.print() and Serial.println() a few times. Serial.println() prints out its first argument with a trailing new line character, Serial.print() omits an end of line character. Both functions can take in c-strings, ints, floats, doubles etc... as well as the Arduino String (technically a C++ wrapper for c-strings). In the first few lines of loop we use Serial.print to print out a c-string "hello" and then we declare an Arduino String arduinoString to " world", and print the Arduino string with a trailing new line.

We then expose a C-style way to print out loopCounter using a buffer to write loopCounter in with the C itoa function, then we use another c-string buffer to hold: "C count: " before doing a string concatenation with the C function strcat to combine the two, then print the result.

The Arduino (technically C++ style) way of printing out our loopCounter is far more simpler and intuitive.

Note: to convert an Arduino String to an int, you can use the built in method toInt(), like String s = "5";
int sInt = s.toInt();

Run this Arduino program and click the Serial Monitor button to view the console.

In the next program the Arduino will be receiving data from our Serial Monitor window. If you open your monitor, you should see a text input widget next to a send button. You can send information to the Arduino using this.

Armed with that knowledge, check out the following program source code:

```
void setup() {  
  Serial.begin(9600);  
}  
  
void loop() {  
  if (Serial.available()){ //Serial.available() returns true if there is some data to read  
    String input = Serial.readString(); //readString reads the line of input the user typed  
    Serial.println("got a message: " + input);  
    if (input.length() >= 7){  
      if(input.substring(0, 5) == "print"){  
        int count = input.substring(6, input.length()).toInt();  
        for(int i = 0; i < count; i++)  
          Serial.println(i);  
        Serial.println("executed counting operation");  
      }  
    }  
  }  
  delay(10);  
}
```

Look at the loop function, here we are using a function `Serial.available()` which returns true if the user hit send with some data. If we do have some information coming in, we print it out. Using the `length()` method of our String, we check if the length is above 7. In this simple application we use the `substring` method to detect if the first 4 letters (sub-string indices 0 to 5 exclusive) say "print", if they do, we convert the rest of the string into an int, and print out the numbers from 0 to that input integer.

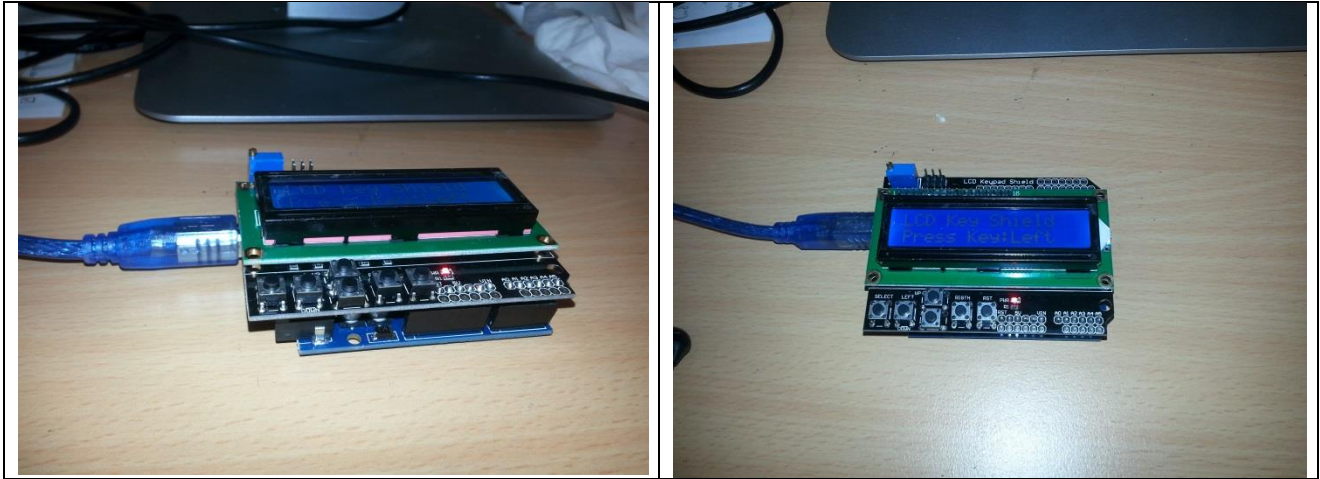
See if you can create some interesting applications by combining user i/o with the modules seen so far.

Installing Libraries

In the next tutorial we will be using the LCD shield for Arduino. We will be using the LiquidCrystal library. If you don't have this library in your windows machine "C:/Program Files (x86)/Arduino/libraries" directory or your Ubuntu "/home/yourUserName/Arduino/libraries" directory, you can download it from here: <https://github.com/luke611/ArduinoTutes/tree/master/libs>. All you need to do is simply copy the "LiquidCrystal" folder to this directory. All of the libraries required for these tutorials can be found at the site given above anyway.

The Liquid Crystal Displace (LCD) Shield

The pictures below show how to attach the LCD shield to the Arduino. You should be careful not to plug it too far into the input pins of the Arduino and connect the shield board with metallic parts of the Arduino as this can short some of the components (allow current to flow freely without resistance) which can cause the module not to work as expected.



No let's take a look at some code. This example shows how to use both the LCD display to output characters, as well as reading input from the select, left, up, down and right buttons.

```
#include <LiquidCrystal.h>
// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

void setup() {
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  // Print a message to the LCD.
  lcd.setCursor(0,0); //set the cursor to column 0, row 0
  lcd.print("LCD Key Shield");
  lcd.setCursor(0,1); //set the cursor to column 0, row 1
  lcd.print("Press Key:");
}

void loop() {
  int x;
  x = analogRead (0); //read button input from Analog pin 0
  lcd.setCursor(10,1); //move cursor to row 1, column 10
  if (x < 60)
    lcd.print ("Right ");
  else if (x < 200)
    lcd.print ("Up  ");
  else if (x < 400)
    lcd.print ("Down ");
  else if (x < 600)
    lcd.print ("Left ");
  else if (x < 800)
    lcd.print ("Select");
}
```

Here we include the LiquidCrystal.h library header file. In global space we create an LiquidCrystal object called lcd and provide a list of interface pins (these are the pins to use with this module and the Arduino Uno).

In our setup function, we call lcd.begin(16, 2), the begin function takes two parameters, the number of columns (our shield has space for 16 characters from left to right) and the number of rows (our lcd shield provides two rows). The method for writing to the screen includes setting the cursor location to a particular row and column, before calling the print function.

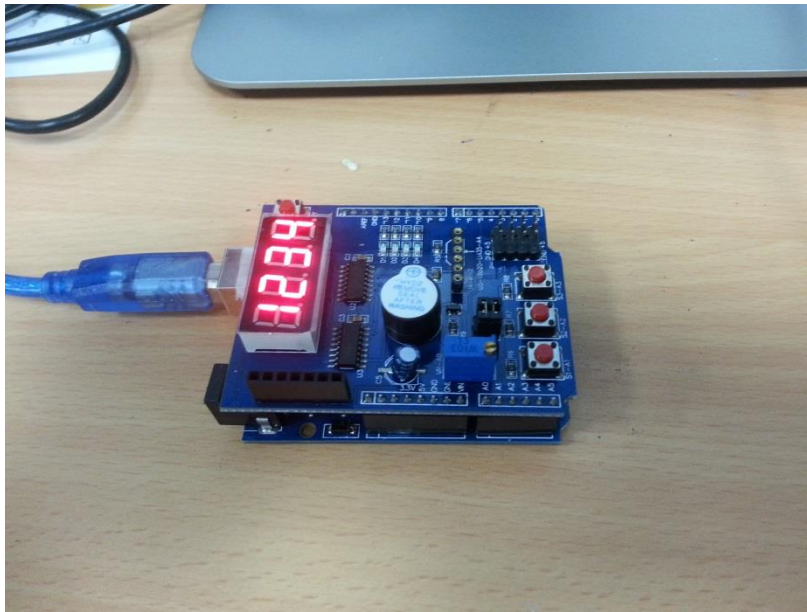
In the next lines we write lcd.setCursor(0, 0), to set our cursor to the top left of the screen. Then lcd.print("LCD Key Shield") prints this string to the top left hand side of our LCD screen. The setCursor function takes the column value then the row value as parameters, and the print function can take any Arduino string, char, byte, long or int.

In the loop function, we detect input from the various buttons on the shield. If a particular button is detected, we write which button was pressed to the lcd screen. To detect a button press, we perform an analogue read from analogue input pin 0. This returns a value between 0 and 1023. If the value is in the range [0-59] then the right button was pressed. The table below lists the ranges for different button presses.

Range (inclusive)	button
800-1023	
0-59	Right button
60-199	Up button
200-399	Down button
400-599	Left button
600-799	Select button

Digital Multi-Function Shield

In this tutorial we will be using the digital multi-function shield. The wiring involves a simple attachment similar to the lcd screen (shown below). This device is capable of driving a 7-segment display, detecting button pressed (there are 3 buttons) and driving 4 LEDs on the shield.



LED control

```
const byte LED[] = {13,12,11,10};
int highlightedPin = 0;

void setup()
{
  // initialize the digital pin as an output.
  /* Set each pin to outputs */
  for(int i = 0; i < 4; i++) pinMode(LED[i], OUTPUT);
}

void loop()
{
  for(int i = 0; i < 4; i++)
    if(i == highlightedPin) digitalWrite(LED[i], LOW);
    else digitalWrite(LED[i], HIGH);
  highlightedPin = (highlightedPin + 1) % 4;
  delay(500);
}
```

In this first tutorial, we will be creating a small animation among the 4 leds on the shield. The digital output pins used to drive these LEDs are 13, 12, 11 and 10. Notice in our code, we have a highlighted pin variable as well as an LED byte array for indexing out pins. The highlightedPin variable can be 0, 1, 2 or 3 to turn on the corresponding LED. In our setup function we use the pinMode function to setup each pin as an output pin using a for loop. In our loop function, we have another for loop, and turn pin i on if it is equal to our highlightedPin otherwise we turn it off. We then set highlightedPin to the next index, looping around back to 0 if need be.

Button Click Detection

To detect button clicks on the multi-shield, we have to read a digital input from Analoge pins A1, A2 or A3 (1 for each button). A digital read of 0 means the button is pressed, a digital read of 1 or HIGH means the button is not pressed.

In the code below, we setup out button pins as inputs, and our LED pins as outputs. If digitalRead(BUTTON1) returns LOW, we turn all our pins on, if we detect BUTTON2 was pressed, we turn all the pins off.

```
const byte LED[] = {13,12,11,10};

#define BUTTON1 A1
#define BUTTON2 A2
#define BUTTON3 A3

void setup()
{
  // initialize the digital pin as an output.
  /* Set each pin to outputs */
  for(int i = 0; i < 4; i++) pinMode(LED[i], OUTPUT);
  pinMode(BUTTON1, INPUT);
  pinMode(BUTTON2, INPUT);
  pinMode(BUTTON3, INPUT);
}

void loop()
{
  if(!digitalRead(BUTTON1))
    for(int i = 0; i < 4; i++) digitalWrite(LED[i], HIGH);

  if(!digitalRead(BUTTON2))
    for(int i = 0; i < 4; i++) digitalWrite(LED[i], LOW);
}
```

Driving the 7-segment display

In this example we will be using a new function called shiftOut(dataPin, clockPin, bit order, byte). This function writes a whole byte of data out on pin: dataPin. Below is a table indicating what each parameter means.

Index	Name	Description
0	dataPin	The pin in which data is sent out on
1	clockPin	The pin in which a clock signal is used, when we send data serially in electronics we usually use one pin (the clock) to signal a new bit is ready to be read and another pin (data) to indicate which bit should be read (1 or 0).
2	Bit order	Can be MSBFIRST or LSBFIRST indicating that the most significant bit should be written first or the least significant bit, this decides the order in which to send the 8 bits of out byte
3	byte	The byte to send

Next comes the code:

```
/* Define shift register pins used for seven segment display */
int  latch = 4;
int  clk  = 7;
int  data = 8;

/* bytes which id one of the 4 7 segment displays */
const byte segmentIds[] = {0xF1,0xF2,0xF4,0xF8};

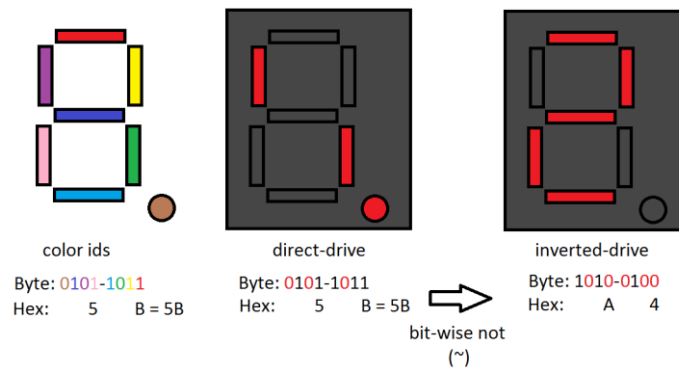
void setup ()
{
  /* Set DIO pins to outputs */
  pinMode(latch,OUTPUT);
  pinMode(clk,OUTPUT);
  pinMode(data,OUTPUT);
}

void drive7Segment(byte segment, int id){
  digitalWrite(latch,LOW);
  shiftOut(data, clk, MSBFIRST, segment);
  shiftOut(data, clk, MSBFIRST, segmentIds[id] );
  digitalWrite(latch,HIGH);
}

/* Main program */
void loop()
{
  /*write a number to each 7 segment*/
  drive7Segment(~0x06, 0);
  drive7Segment(~0x5B, 1);
  drive7Segment(~0x4F, 2);
  drive7Segment(~0x66, 3);
}
```

Here we define some pins in global space latch, data and clk. The data and clk (clock) pins are set as pins 8 and 7, and will be used in our shiftOut function for sending bytes to the shield. The latch pin lets the shield know whether serial data is coming in. In our setup function we setup these pins as outputs.

In our main loop, we are using a function drive7Segment which takes two parameters, the byte to drive a seven segment display (explained soon). And an index into global array segmentIds which contains the id codes for each 7 segment display (1 of 4). Now, for the first parameter which drives a seven-segment display is a representation in binary for which segments we want turned on/off. Take a look at the picture below:



We can see that for a given byte (8 bits) we can align those bits up (by colour in the picture) to one of the 7 segments, for example say we want to drive the number 2. We could take a look at which segments we want to be ON and which OFF. For the number 2, the on segments should be (dark blue, pink, light blue, yellow and red). If we turn those bits on, then we have the result in the middle picture (direct-drive). This is the opposite of what we want, on the multi-shield, the 7-segment display byte signal should contain a 0 at a particular bit location to turn the segment ON. If we invert our bits 0101-1011 (hex: 0x5B) to 1010-0100 (hex: 0xA4), we have the correct representation for 2.

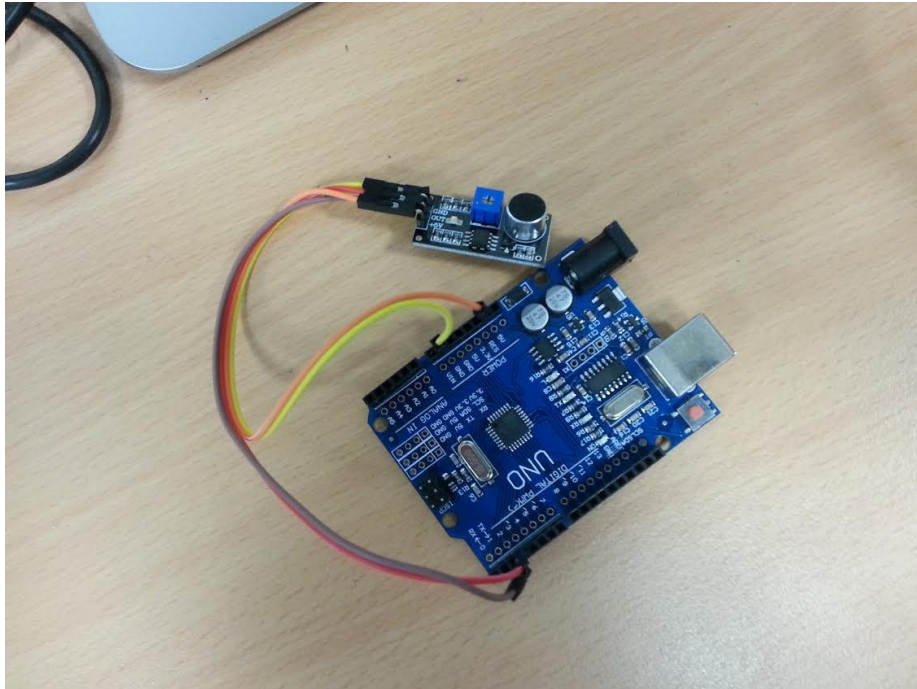
Back to our code, when we write `drive7Segment(~0x06, 1)`, we are taking 0x06 (0000-0110) and inverting it becomes 0xF9 (1111-1001) which is the correct drive for the number 1. See if you can figure out what the rest of the loop function does based on this pattern.

Finally we have our `drive7Segment` function, taking the byte segment pattern (discussed above) and the index to the `segmentIds` array to indicate which of the 4 7 segment displays we wish to alter. Before we send the shield any information we must set the latch output to LOW. After communication we set it high again. We then use the `shiftOut` function to send two bytes of data (uses 2 calls 1 for each byte). The first is our segment pattern, the second is the id of the 7-segment we wish to display.

See if you can program your own driver to display all of the Hex values: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f.

Sound Detector

The next module we are working with is the sound detector. The wiring job is shown below. If you want to alter the module's sensitivity to noise, simply turn the screw on the top of it.



I have connected the +5V pin on the module to the 5V Arduino pin, the GND to the Arduino GND and the out pin on the module to the digital 3 pin on the Arduino. Essentially this module drops the output pin LOW when noise is detected. Below we have an example program which counts the number of noises made and prints it to the serial monitor. No new programming concepts are present, read through the code to understand the tutorial.

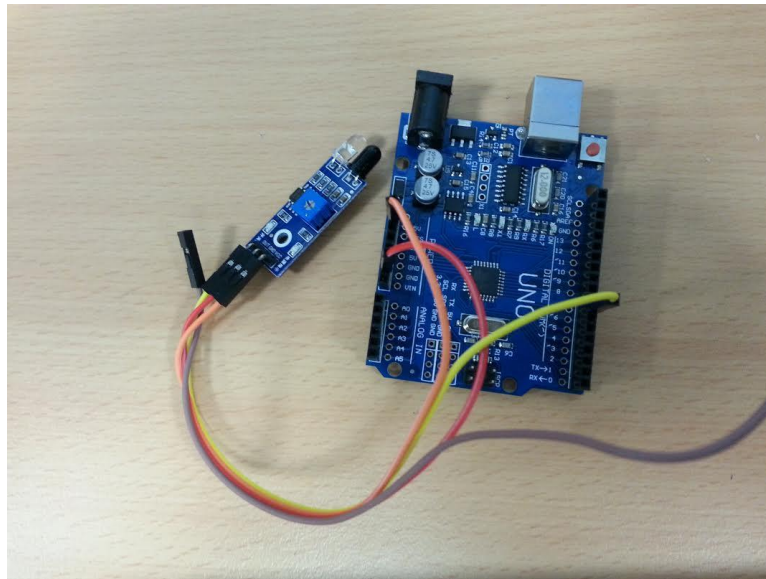
```
int inputPin = 3; //connect the output on the sensor to pin 3, gnd to gnd and vc5v to 5v
int noiseCount = 0;

void setup() {
  Serial.begin(9600); //we will write when we hear noise
  pinMode(inputPin, INPUT); //setup the pin
}

void loop() {
  int hasNoise = !digitalRead(inputPin); //read pin value
  if(hasNoise){ //if there is noise, alert
    Serial.println(String("noise made, count: ") + String(noiseCount)); //count number of noises
    noiseCount++;
    delay(1000);
  }
  delay(10);
}
```

Obstacle Sensor

The obstacle sensor is just as easy to use as the sound sensor. The wiring job is shown below. Here we connect the output from the module to the Arduino digital pin 7, and of course GND to GND and VCC to 5V.

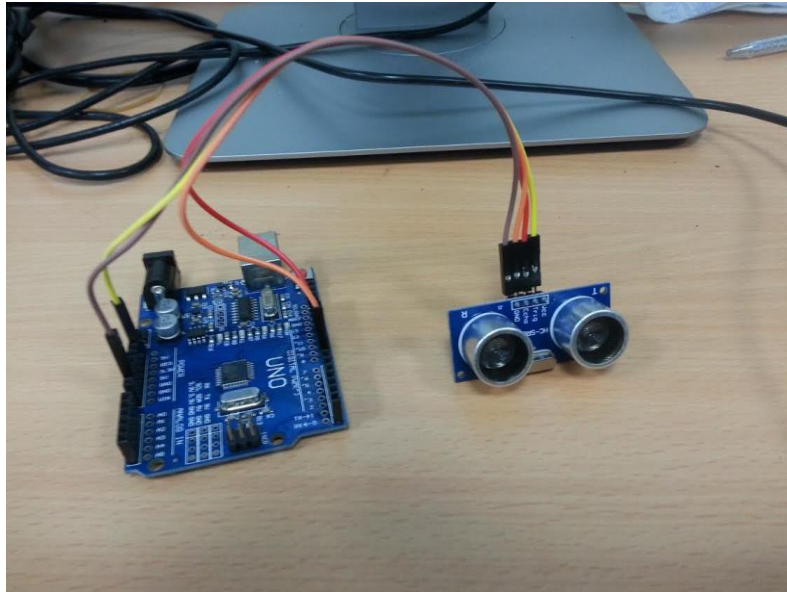


Like the previous module we have an input pin (we connect this one to digital 7) and when this is dropped low, the sensor has detected something, in this case: an object. Here, when we detect an object, we alert the user using the the serial monitor (covered in previous module examples).

```
//connect OUT pin to digital 7, gnd to gnd and vcc to 5v
int isObstaclePin = 7;
void setup() {
  pinMode(isObstaclePin, INPUT);
  Serial.begin(9600);
}
void loop() {
  if (!digitalRead(isObstaclePin))
    Serial.println("obstacle detected");
  delay(200);
}
```

Ultrasonic Sensor

The ultrasonic sensor is more complicated than reading a binary value. It not only must be driven using a trigger pin, but it's output comes in the form of a measurement of how long it drives one of the pins High. The wiring can be seen in the picture below.



Here, we connected: Vcc to 5v and GND to GND of course, but we also connect the Trig pin from the module to the Arduino digital 8 pin and the echo pin from the module to the Arduino's digital 7 pin. The Trig pin is an output pin, so we will be writing HIGH or LOW to it in order to initiate protocol with the module. The echo pin is an input pin, we will measure how long it remains high, and this length can be converted to cm in order to give us the distance of objects from the module, in centimetres.

The code is shown in the figure below. As stated, the trigger pin is pin 7 and the echo pin is pin 8. Here we define a minimum and a maximum range from 0 to 200cm. We also declare two variables, distance and duration. We will write the length of time in which the echo pin remains high into the duration variable, and the distance variable will be the object's distance in cm.

As usual in our setup function, we initialize our Serial communication object, and of course set the pin modes using the pinMode function, trigPin is an output and echoPin is an input.

To begin communication with the module, we drive trigPin LOW for 2 microseconds using the digitalWrite function and the delayMicroseconds function. The delayMicroseconds delays by the amount of microseconds passed as the first argument. After this we drive trigPin HIGH for 10 microseconds using the same strategy.

Now we begin to measure the distance our echo pin remains HIGH. First, we drive trigPin LOW, Next we use a function called pulseIn(x,y). The pin measures the time the x pin remains in state y. In our case we measure the duration for which echoPin remains HIGH. Using our duration variable, we then convert this to cm. If our distance is in the range we declared earlier, then we write it to the Serial Monitor.

```

/*
VCC to arduino 5v
GND to arduino GND
Echo to Arduino pin 7
Trig to Arduino pin 8
*/

#define echoPin 7 // Echo Pin
#define trigPin 8 // Trigger Pin

int maximumRange = 200; // Maximum range needed
int minimumRange = 0; // Minimum range needed
float duration, distance; // Duration used to calculate distance

void setup() {
  Serial.begin (9600);
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}

void loop() {
  //initialize communication protocol, using the trigger pin
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);

  //bring the pin low
  digitalWrite(trigPin, LOW);
  //pulse in records how long the echoPin remains high to obtain the input
  duration = (float)pulseIn(echoPin, HIGH);

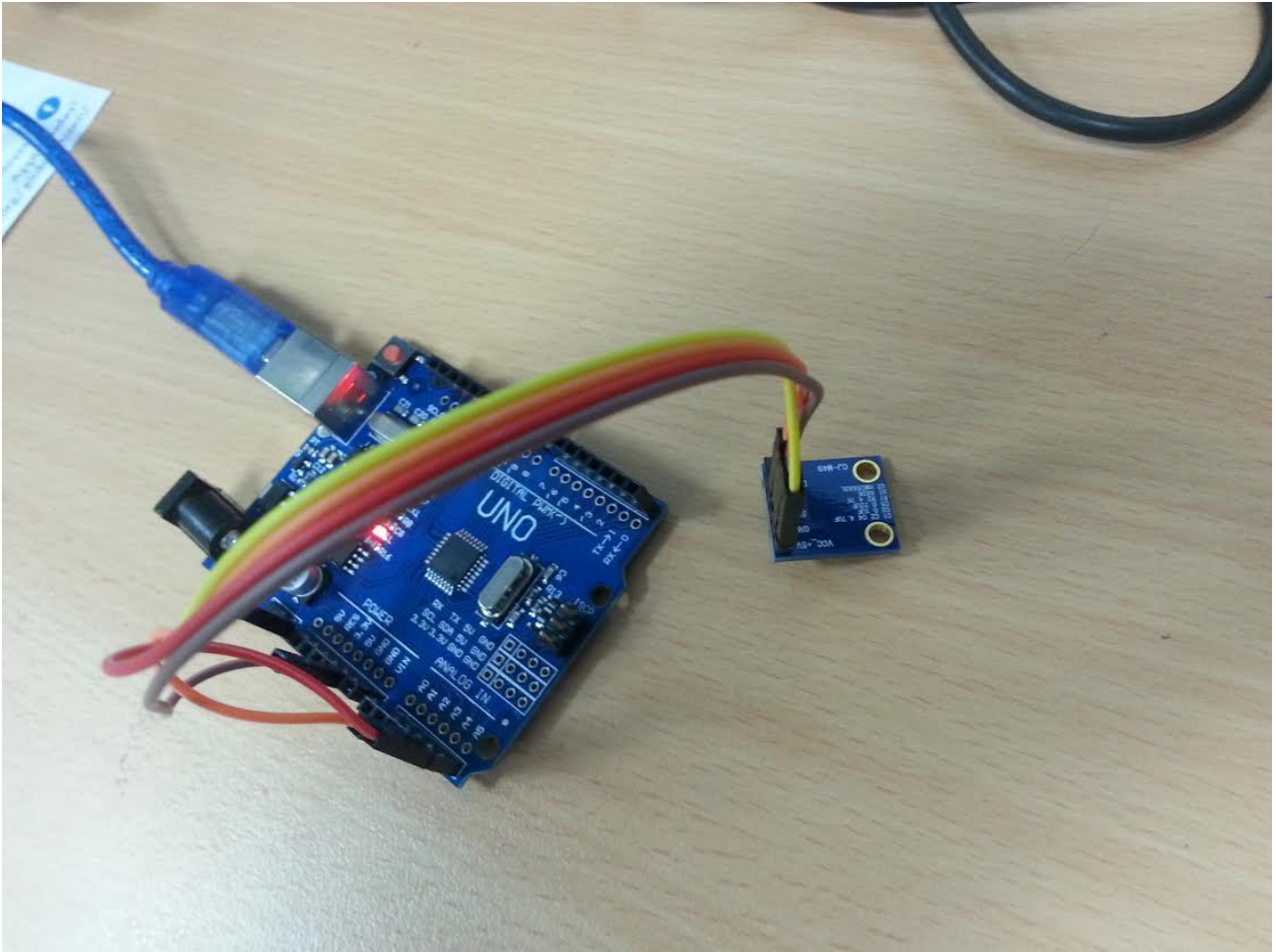
  //Compute the distance based on the duration
  distance = duration/58.2;

  if (distance <= maximumRange && distance > minimumRange) //if not in range
    Serial.println("object detected at: " + String(distance) + " cm.");
  delay(100);
}

```

Digital Compass

The digital compass module requires a few libraries, namely: Wire, Adafruit_Sensor and Adafruit_Adafruit_HMC5883_U. Here we connect the 5V to 5V and GND to GND pins as usual, but you should also connect the SCL pin to the Analogue 5 pin on the Arduino and the SDA pin to the Arduino Analogue 4 pin. The wiring is shown below.



First let's take a look at our setup function, global variables and includes:

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_HMC5883_U.h>

/* Assign a unique ID to this sensor at the same time */
Adafruit_HMC5883_Unified mag = Adafruit_HMC5883_Unified(12345);

void setup(void)
{
  Serial.begin(9600);
  /* Initialise the sensor */
  if(!mag.begin())
  {
    Serial.println("Ooops, no HMC5883 detected ... Check your wiring!");
    while(1);
  }
}
```


Of course we are including the necessary includes at the top. We will be using the Adafruit_HMC5883_Unified object from the Adafruit library in order to read the input from our compass. We name this variable mag, in our setup function we initialize our Serial object, and call begin() with our Adafruit object, it will return true if the initialization was correct. If it returns false we print an error message and go into an infinity while loop (never reaching our loop function).

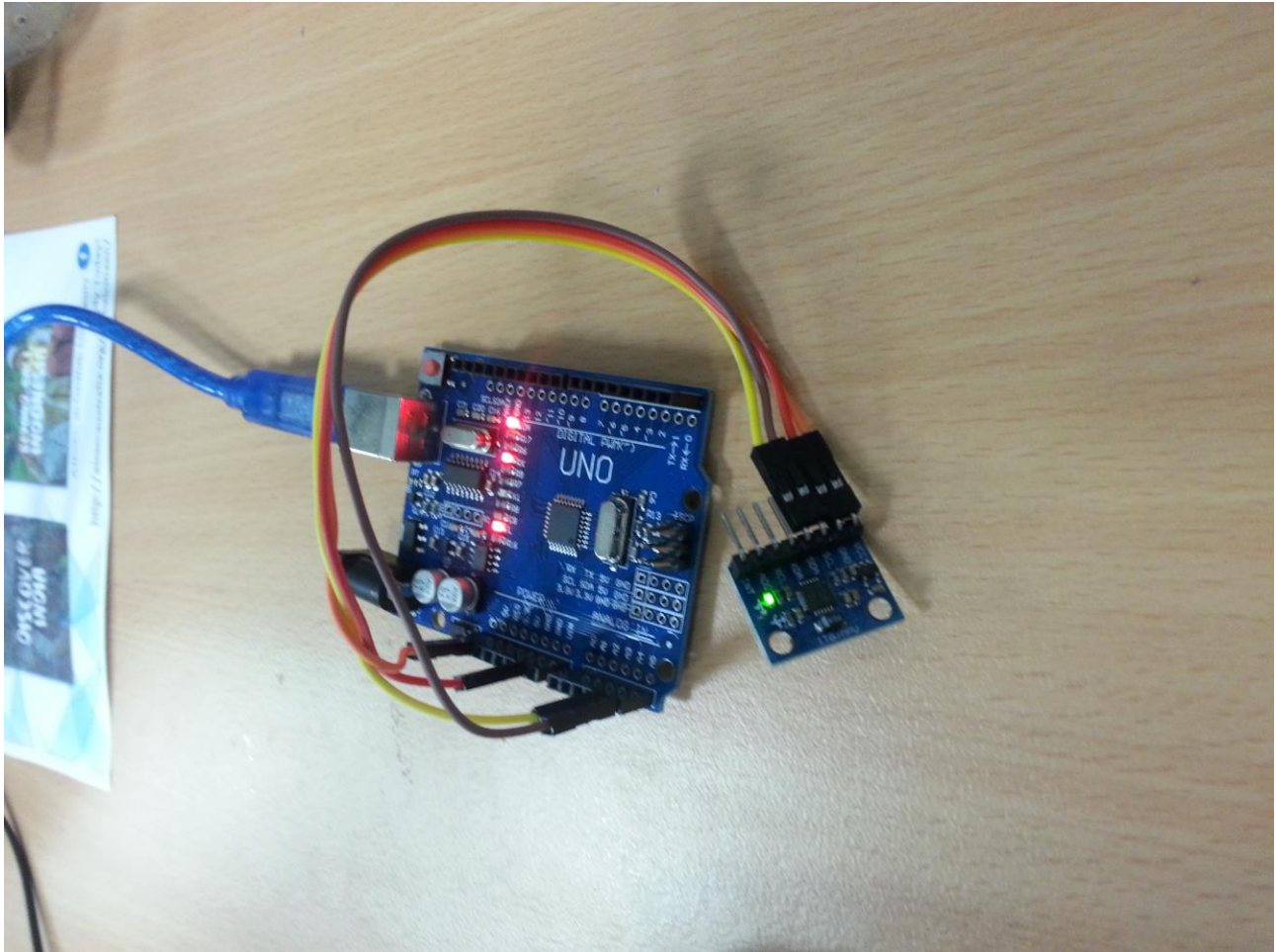
Below is the accompanying loop function. In the setup function we obtain a new event by passing a pointer of a local event object to our mag object. This provides us with our x,y,z axis through variables: event.magnetic.x, event.magnetic.y and event.magnetic.z. Here we simply print them out to the serial monitor. If your compass is facing upwards (flat with the pins pointing towards the sky) then you can use the heading measurement. This is based on the x and y components of our vector, we simply find the angle which these two represent using the atan2 function, as in most languages this returns radians, so we therefore adjust for degrees for a human readable format.

```
void loop(void)
{
  /* Get a new sensor event */
  sensors_event_t event;
  mag.getEvent(&event);
  /* Display the results (magnetic vector values are in micro-Tesla (uT)) */
  Serial.print("X: "); Serial.print(event.magnetic.x); Serial.print(" ");
  Serial.print("Y: "); Serial.print(event.magnetic.y); Serial.print(" ");
  Serial.print("Z: "); Serial.print(event.magnetic.z); Serial.print(" "); Serial.println("uT");

  // Hold the module so that Z is pointing 'up' and you can measure the heading with x&y
  // Calculate heading when the magnetometer is level, then correct for signs of axis.
  float heading = atan2(event.magnetic.y, event.magnetic.x);
  // Convert radians to degrees for readability.
  float headingDegrees = heading * 180/M_PI;
  Serial.print("Heading (degrees): "); Serial.println(headingDegrees);
  delay(500);
}
```


IMU: Internal Measurement Unit

In our final tutorial we will drive the IMU. The IMU provides the Arduino with acceleration and gyroscopic information as well as temperature information. This is provided in the form of two vectors `_acceleration` and `_direction`. The wiring is shown below, but the wiring is identical to the compass tutorial: vcc to vcc, gnd to gnd, SCL to A5 and SDA to A4.



In this tutorial, we will define our own Vector struct and use the Wire library module. Here are our includes, global variables and our Vector struct called Vec3:

```
#include<Wire.h>
const int MPU_addr=0x68; // I2C address of the MPU-6050

struct Vec3{
  int x, y, z;
  void init(int x, int y, int z){ this->x = x, this->y = y, this->z = z; }
  String toString(){ return String(x) + ", " + String(y) + ", " + String(z); }
};
```

We use an address id of 0x68 when communicating with the IMU. We also will be using a basic struct called Vec3, it has 3 member variables, x, y and z. It also has an `init(x,y,z)` function for setting these variables and a `toString` function which returns a string representation of the vector.

In our setup function, we initialize communication with the IMU by writing the address id which we are communicating through (0x68), before waking up the IMU by writing 0x6B and 0x00. We also setup our

Serial object baud rate. We have also included a function called read16bits, which uses the Wire module to do just that. It reads one byte, before shifting those bits over to the left 8 times, and does a bit wise or with the second byte it reads. It returns these 16 bits back as our standard 32 bit integer.

```
void setup(){
  Wire.begin();
  Wire.beginTransmission(MPU_addr);
  Wire.write(0x6B); // PWR_MGMT_1 register
  Wire.write(0);    // set to zero (wakes up the MPU-6050)
  Wire.endTransmission(true);
  Serial.begin(9600);
}

int read16Bits(){
  return Wire.read()<<8|Wire.read();
}
```

In our loop function (shown below) we continue transmission to the IMU by writing the destination address id once again, then requesting to read our vectors and temperature from it's registers.

Next we read the acceleration vector, the temperature and the direction (gyroscope) vector using our read16Bits function, each component of a vector is 16 bits and the temperature is just 16 bits.

So we declare our vectors and temperature variables as Vec3s and floats. And use the read16Bits function, passing the result to our init functions with the Vec3s or straight to our _temperature variable (since it is a singular value). Finally we use the toString functions in the Vec3s to print our vectors to the serial monitor.

```
void loop(){
  Wire.beginTransmission(MPU_addr);
  Wire.write(0x3B); // starting with register 0x3B (ACCEL_XOUT_H)
  Wire.endTransmission(false);
  Wire.requestFrom(MPU_addr,14,true); // request a total of 14 registers

  //initialize our vectors and temperature
  Vec3 _acceleration, _direction;
  float _temperature;

  _acceleration.init(read16Bits(), read16Bits(), read16Bits());
  _temperature = ((float)read16Bits())/340.00+36.53;
  _direction.init(read16Bits(), read16Bits(), read16Bits());

  Serial.print("direction: " + _direction.toString());
  Serial.print(",\tacceleration: " + _acceleration.toString());
  Serial.println(",\ttemperature: " + String(_temperature));

  delay(300);
}
```