

Arduino Microcontroller

1 Introduction

1.1 Overview

The Arduino microcontroller is an easy to use yet powerful single board computer. This guide covers the Arduino Uno. With the Arduino board, you can write programs and create interface circuits to read and control switches and other sensors and actuators.

This is what the Arduino board looks like.



The board features an Atmel ATmega328 microcontroller operating at 5 V with 2 Kb of RAM, 32 Kb of flash memory for storing programs and 1 Kb of EEPROM for storing parameters. The clock speed is 16 MHz, which translates to about executing about 300,000 lines of C source code per second. The board has 14 digital I/O pins and 6 analog input pins. There is a USB connector for talking to the host computer and a DC power jack for connecting an external 6-20 V power source, for example a 9 V battery, when running a program while not connected to the host computer. Headers are provided for interfacing to the I/O pins using 22 g solid wire or header connectors. For additional information on the hardware, see <http://arduino.cc/en/Main/ArduinoBoardUno>.

The Arduino programming language is a simplified version of C/C++. You do not need to worry if you do not have much experience with C, as only a few commands are needed to perform useful functions.

An important feature of the Arduino is that you can create a control program on the host PC, download it to the Arduino and it will run automatically. Remove the USB cable connection to the PC, and the program will still run from the top each time you push the reset button. In particular, you connect the board to the host PC to develop and debug your program, but once that is done, you no longer need the PC to run the program.

1.2 What You Need for a Working System

1. Arduino board
2. USB programming cable (A to B)
3. Solderless breadboard for external circuits, and 22 g solid wire for connections
4. Host PC running the Arduino development environment. Versions exist for Windows, Mac and Linux

For stand-alone operation a 9V battery or external power supply is needed.

1.3 Installing the Software

Follow the instructions on the Getting Started section of the Arduino web site, <http://arduino.cc/en/Guide/HomePage>. Go all the way through the steps to where you see the pin 13 LED blinking. This is the indication that you have all software and drivers successfully installed and can start exploring with your own programs. Note that the software is already installed on the EIETL workstations.

1.4 Connecting a Battery

For stand-alone operation, the board is powered by a battery rather than through the USB connection to the computer. While the external power can be anywhere in the range of 6 to 24 V, a standard 9 V battery is convenient. This should be connected to the power jack on the board via a DC power plug, as e.g. shown in the diagram.



Warning: Watch the polarity as you connect your battery to the snap as reverse orientation could blow out your board.

1.5 Moving On

Connect your Arduino to the computer with the USB cable. You do not need the battery for this handout. The green PWR LED will light. If there was already a program burned into the Arduino, it will run.

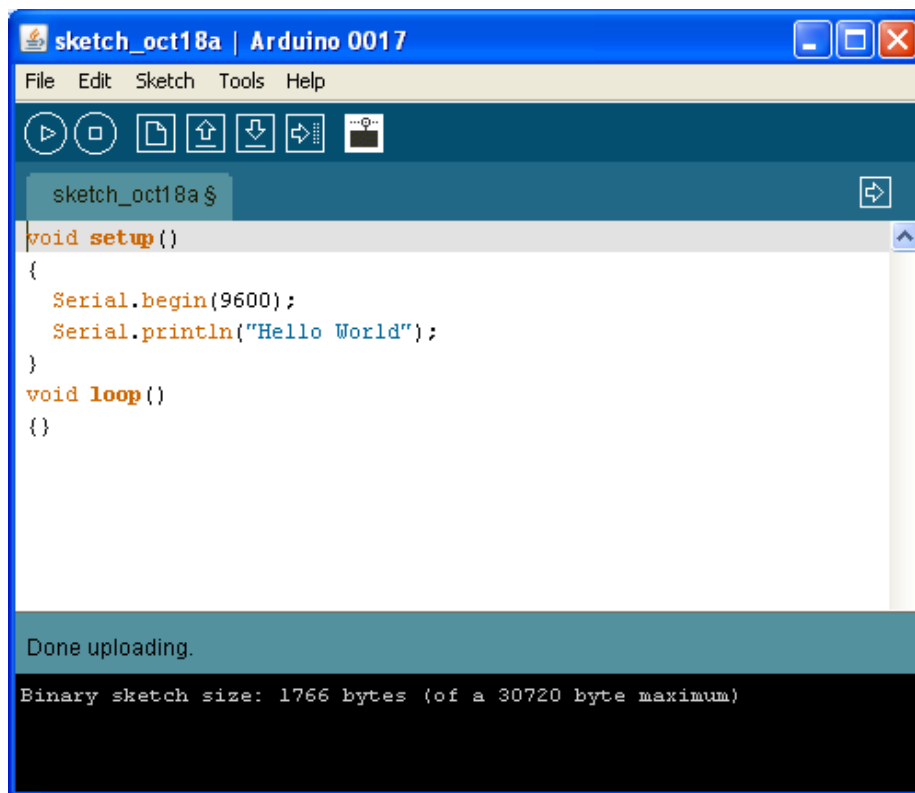
Warning: Do not put your board down on a conductive surface; you will short out the pins on the back!



Start the Arduino development environment.

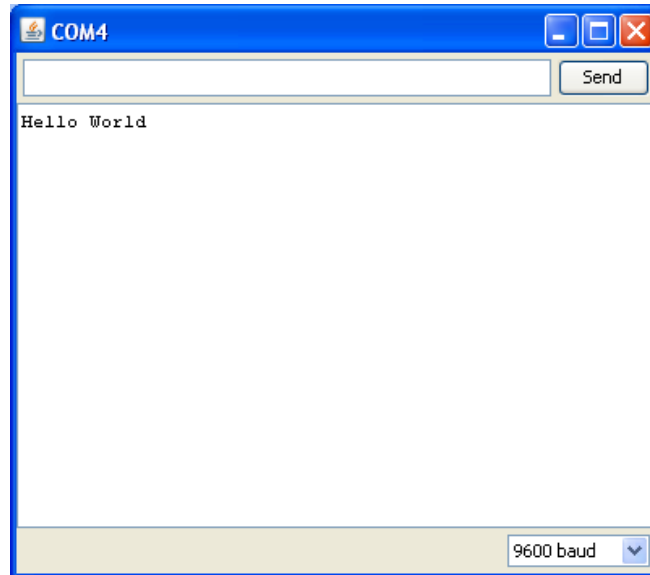
In the editing window that comes up, enter the following program, paying attention to where semi-colons appear at the end of command lines.

```
void setup()  
{  
  Serial.begin(9600);  
  Serial.println("Hello World");  
}  
void loop()  
{}
```


Your window will look something like this (note that there could be slight variations in the appearance of this window depending on the version of the development environment and the operating system of the host computer)



Click the Upload button  or Ctrl-U to compile the program and load on the Arduino board. Click the Serial Monitor button . If all has gone well, the monitor window will show your message and look something like this



Congratulations; you have created and run your first Arduino program!
Push the Arduino reset button a few times and see what happens.

Hint: If you want to check code syntax without an Arduino board connected, click the Verify button  or Ctrl-R.

Hint: If you want to see how much memory your program takes up, Verify then look at the message at the bottom of the programming window.

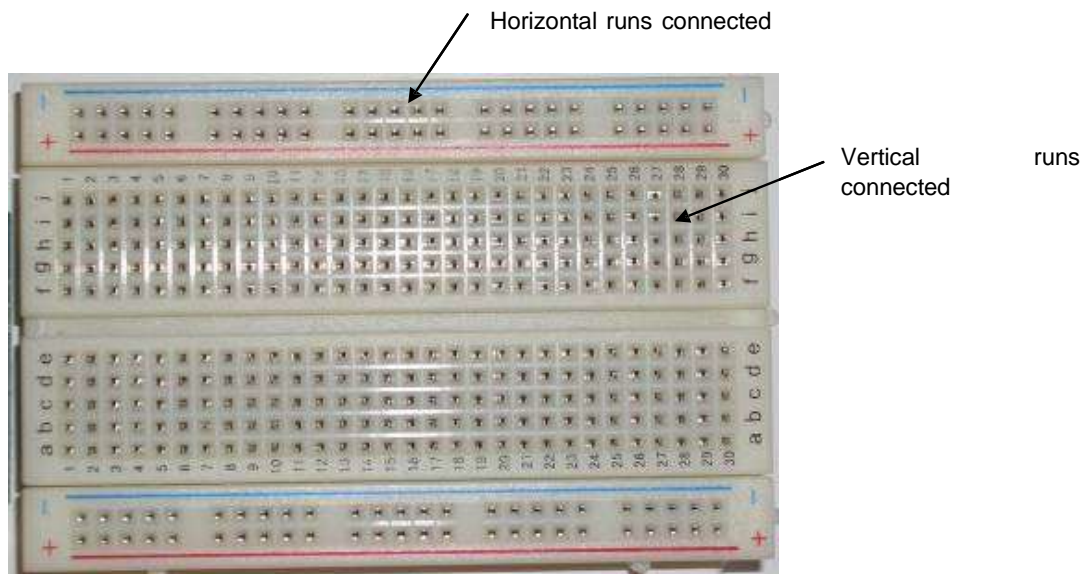
1.6 Troubleshooting

If there is a syntax error in the program caused by a mistake in typing, an error message will appear in the bottom of the program window. If you continue to have problems, try these ideas

- Run the Arduino program again
- Check that the USB cable is secure at both ends.
- Reboot your PC because sometimes the serial port can lock up
- Ask a friend or demonstrator for help

1.7 Solderless Breadboards

A solderless breadboard is an essential tool for rapidly prototyping electronic circuits. Components and wire push into breadboard holes. Rows and columns of holes are internally connected to make connections easy. Wires run from the breadboard to the I/O pins on the Arduino board. Make connections using short lengths of 22 g solid wire stripped of insulation about 0.25" at each end. Here is a photo of a breadboard showing which runs are connected internally. The pairs of horizontal runs at the top and bottom are useful for running power and ground. Convention is to make the red colored run +5 V and the blue colored run Gnd. The power runs are sometimes called "power buses".

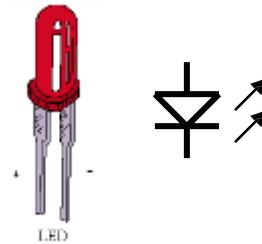


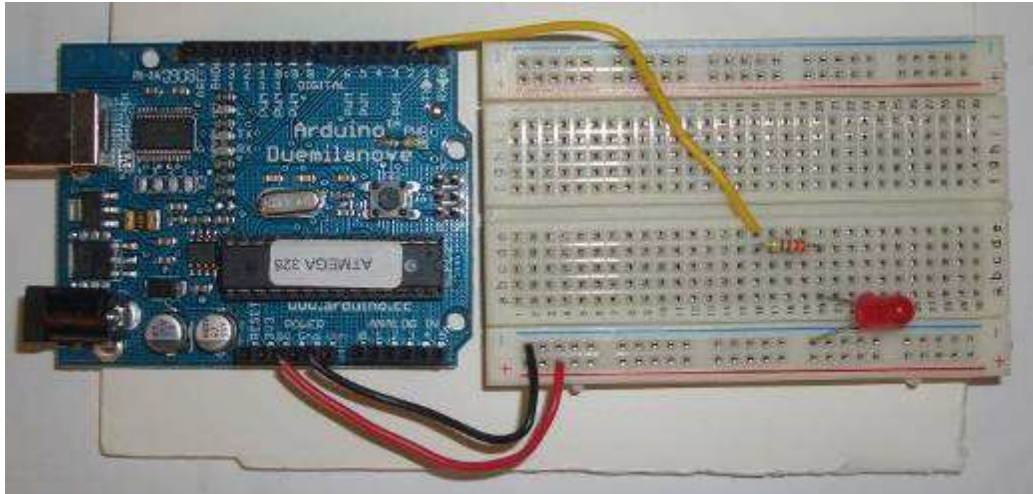
Warning: Only use solid wire on the breadboard. Strands of stranded wire can break off and fill the holes permanently.

Hint: Trim wires and component leads so that wires and components lie close to the board.

2 Flashing an LED

Light emitting diodes (LED's) are handy for checking out what the Arduino can do. For this task, you need an LED, a 330 ohm resistor, and some short pieces of 22 or 24 g wire. Using 22 g solid wire, connect the 5V power pin on the Arduino to the bottom red power bus on the breadboard and the Gnd pin on the Arduino to the bottom blue power bus on the breadboard. Connect the notched or flat side of the LED (the notch or flat is on the rim that surrounds the LED base; look carefully because it can be hard to find) to the Gnd bus and the other side to a free hole in main area of the breadboard. Place the resistor so that one end is in the same column as the LED and the other end is in a free column. From that column, connect a wire to digital pin 2 on the Arduino board. Your setup will look something like this





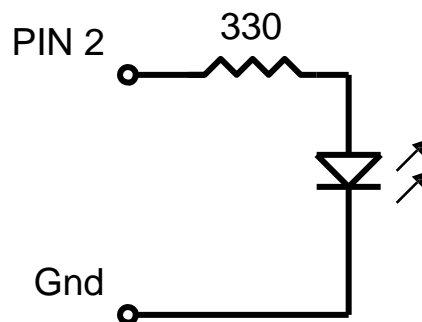
To test whether the LED works, temporarily disconnect the wire from pin 2 on the Arduino board and touch to the 5V power bus. The LED should light up. If not, try changing the orientation of the LED. Place the wire back in pin 2.

Create and run this Arduino program

```
void setup()
{
  pinMode(2,OUTPUT);

  digitalWrite(2,HIGH);
  delay(1000);
  digitalWrite(2,LOW);
}

void loop()
{}
```



Did the LED light up for one second? Push the Arduino reset button to run the program again.

Now try this program, which will flash the LED at 1.0 Hz. Everything after the // on a line is a comment, as is the text between „/*” and „*/” at the top. It is always good to add comments to a program.

```
/*-----
Blinking LED, 1.0 Hz on pin 2
-----*/
```

```

void setup()                // one-time actions
{
  pinMode(2,OUTPUT);        // define pin 2 as an output
}

void loop()                 // loop forever
{
  digitalWrite(2,HIGH);      // pin 2 high (LED on)
  delay(500);                // wait 500 ms
  digitalWrite(2,LOW);       // pin 2 low (LED off)
  delay(500);                // wait 500 ms
}

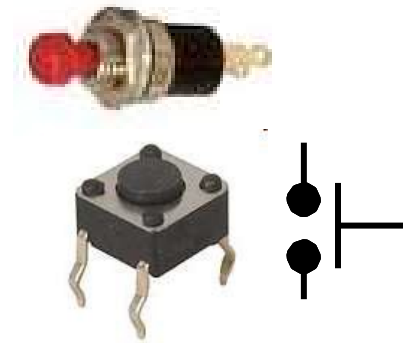
```

The `pinMode` command sets the LED pin to be an output. The first `digitalWrite` command says to set pin 2 of the Arduino to HIGH, or +5 volts. This sends current from the pin, through the resistor, through the LED (which lights it) and to ground. The `delay(500)` command waits for 500 msec. The second `digitalWrite` command sets pin 2 to LOW or 0 V stopping the current thereby turning the LED off. Code within the brackets defining the `loop()` function is repeated forever, which is why the LED blinks.

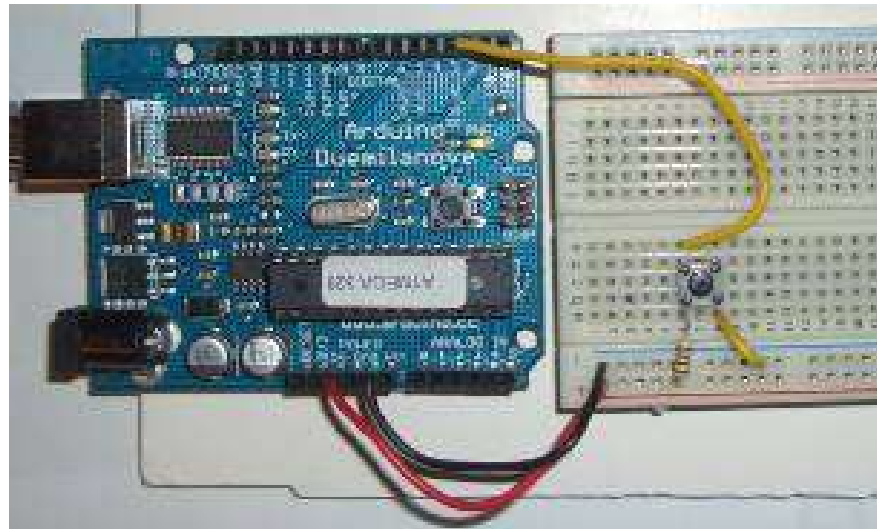
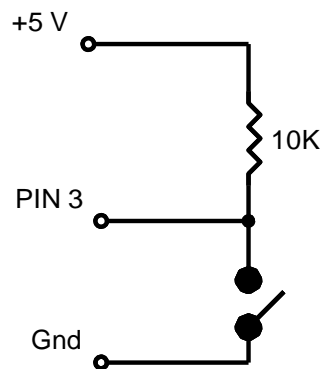
This exercise shows how the Arduino can control various devices connected to it. These could be motors, relays, solenoids, electromagnets, pneumatic valves or any other on-off type device.

3 Reading a switch

The LED exercise shows how the Arduino can control the outside world. Many applications require reading the state of sensors, including switches. The figure to the right shows a picture of a pushbutton switch and its schematic symbol. Note that the symbol represents a switch whose contacts are normally open, but then are shorted when the button is pushed.



For this exercise, the Arduino will read the state of a normally-open push button switch and display the results on the PC using the `serial.println()` command. You will need a switch, a 10 k Ω resistor and some pieces of 22 g hookup wire. If a switch is not available, substitute two wires and manually connect their free ends to simulate a switch closure. The figure below shows the schematic for the circuit on the left and a realization on the right.



Create and run this Arduino program

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

void loop()
{
  Serial.println(digitalRead(3));
  delay(250);
}
```

Open the Serial Monitor window. When the switch is open, you should see a train of 1's on the screen. When closed, the 1's change to 0's. On the hardware side, when the switch is open, no current flows through the resistor. When no current flows through a resistor, there is no voltage drop across the resistor, which means the voltage on each side is the same. In your circuit, when the switch is open, pin 3 is at 5 volts which the computer reads as a 1 state. When the switch is closed, pin 3 is directly connected to ground, which is at 0 volts. The computer reads this as a 0 state.

Now try this program which is an example of how you can have the computer sit and wait for a sensor to change state.

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

void loop()
{
  while (digitalRead(3) == HIGH)
  {
    ;
    Serial.println("Somebody closed the switch!");
  }
}
```



```

while (digitalRead(3) == LOW)
;
Serial.println("The switch is now open!");
}

```

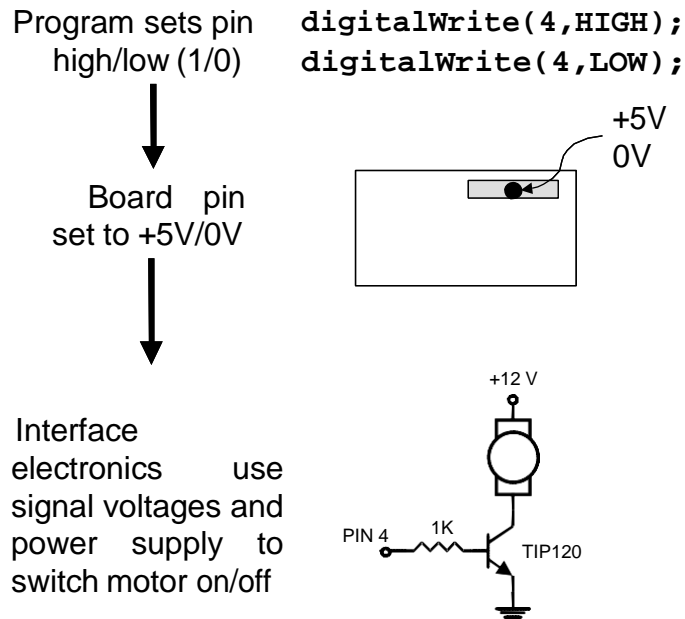
Watch the activity in the Serial Monitor window as you press and release the switch.

4 Arduino Hardware

The power of the Arduino is its ability to interact with the outside world through its input-output (I/O) pins. The Arduino has 14 digital I/O pins labeled 0 to 13 that can be used to turn actuators on and off and read the state of sensors or switches.

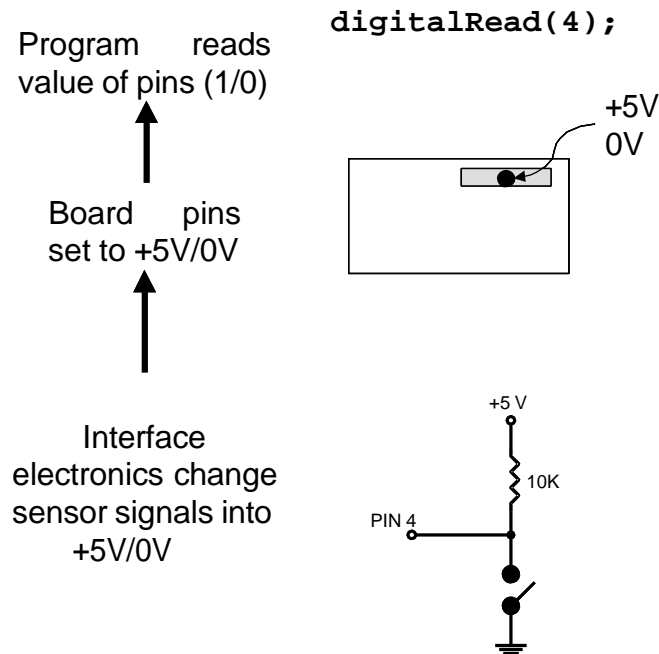
Each digital pin can sink or source about 40 mA of current. This is more than adequate for interfacing to most devices, but does mean that interface circuits are needed to control devices other than simple LED's. In other words, you cannot run a motor directly using the current available from an Arduino pin, but rather must have the pin drive an interface circuit that in turn drives the motor.

To interact with the outside world, the program sets digital pins to a high or low value using C code instructions, which corresponds to +5 V or 0 V at the pin. The pin is connected to external interface electronics and then to the device being switched on and off. The sequence of events is shown in this figure.



To determine the state of switches and other sensors, the Arduino is able to read the voltage value applied to its pins as a binary number. The interface circuitry translates the sensor signal into a 0 or +5 V signal applied to the digital I/O pin. Through a program command, the Arduino

interrogates the state of the pin. If the pin is at 0 V, the program will read it as a 0 or LOW. If it is at +5 V, the program will read it as a 1 or HIGH. If more than +5 V is applied, you may blow out your board, so be careful. The sequence of events to read a pin is shown in this figure.



Interacting with the world has two sides. First, the designer must create electronic interface circuits that allows devices to be controlled by a low (1-10 mA) current signal that switches between 0 and 5 V, and other circuits that convert sensor readings into a switched 0 or 5 V signal. Second, the designer must write a program using the set of Arduino commands that set and read the I/O pins.

When reading inputs, pins must have either 0 or 5V applied. If a pin is left open or "floating", it will read random voltages and cause erratic results. This is why switches always have a 10K pull up resistor connected when interfacing to an Arduino pin.

Note: The reason to avoid using pins 0 and 1 is because those pins are used for the serial communications between the Arduino and the host computer.

The Arduino also has six analog input pins for reading continuous voltages in the range of 0 to 5 V from sensors such as potentiometers.

5 Arduino Programming Language

The Arduino runs a simplified version of the C programming language, with some extensions for accessing the hardware. In this guide, we will cover the subset of the programming language. For more information on the Arduino language, see the Language Reference section of the Arduino web site, <https://www.arduino.cc/reference/en/>.

All Arduino instructions are one line. The board can hold a program hundreds of lines long and has space for about 1,000 two-byte variables. The Arduino executes programs at about 300,000 source code lines per sec.

5.1 Creating a Program

Programs are created in the Arduino development environment and then downloaded to the Arduino board. The compiler will catch and flag syntax errors before download. Sometimes the error message can be cryptic and you have to do a bit of hunting because the actual error occurred before what was flagged.

Although your program may pass cleanly through the syntax checker, it still might not do what you wanted it to. The best way to catch these errors is to read the code line by line.

5.2 Program Formatting and Syntax

Programs are entered line by line. Code is case sensitive which means "myvariable" is different than "MyVariable".

Statements are any command. Statements are terminated with a semi-colon. A classic mistake is to forget the semi-colon so if your program does not compile, examine the error text and see if you forgot to enter a colon.

Comments are any text that follows “//” on a line. For multi-line block comments, begin with “/*” and end with “*/”

Constants are fixed numbers and can be entered as ordinary decimal numbers (integer only) or in hexadecimal (base 16) or in binary (base 2) as shown in the table below

Decimal	Hex	Binary
100	0x64	B01100100

Labels are used to reference locations in your program. They can be any combination of letters, numbers and underscore (_), but the first character must be a letter. When used to mark a location, follow the label with a colon. When referring to an address label in an instruction line, don't use the colon. Here's an example

```
repeat: digitalWrite(2,HIGH);  
        delay(1000);  
        digitalWrite(2,LOW);  
        delay(1000);  
        goto repeat;
```

Use labels sparingly as they can actually make a program difficult to follow and challenging to debug. In fact, some C programmers will tell you to never use labels.

Variables are allocated by declaring them in the program. Every variable must be declared. If a variable is declared outside the braces of a function, it can be seen everywhere in the program. If it is declared inside the braces of a function, the variable can only be seen within that function.

Variables come in several flavors including byte (8-bit, unsigned, 0 to 255), word (16-bit, unsigned, 0 to 65,536), int (16-bit, signed, -32,768 to 32,767), and long (32-bit, signed, -2,147,483,648 to 2,147,483,647). Use byte variables unless you need negative numbers or numbers larger than 255, then use int variables. Using larger sizes than needed fills up precious memory space.

Variable declarations generally appear at the top of the program

```
byte i;  
word k;  
int length;  
int width;
```

Variable names can be any combination of letters and numbers but must start with a letter. Names reserved for programming instructions cannot be used for variable names and will give you an error message

Symbols are used to redefine how something is named and can be handy for making the code more readable. Symbols are defined with the "#define" command and lines defining symbols should go at the beginning of your program. Here's an example without symbols for the case where an LED is connected to pin 2.

```
void setup()  
{  
  pinMode(2,OUTPUT);  
}  
  
void loop()  
{  
  digitalWrite(2,HIGH);    // turn LED on
```

```

    delay(1000);
    digitalWrite(2,LOW);    // turn LED off
    delay(1000);
}

```

Here is the same using a symbol to define "LED"

```

#define LED 2    // define the LED pin

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

void loop()
{
    digitalWrite(LED,HIGH);
    delay(500);
    digitalWrite(LED,LOW);
    delay(500);
}

```

Note how the use of symbols reduces the need for comments. Symbols are extremely useful to define for devices connected to pins because if you have to change the pin that the device connects to, you only have to change the single symbol definition rather than going through the whole program looking for references to that pin.

5.3 Program Structure

All Arduino programs have two functions, `setup()` and `loop()`. The instructions you place in the `setup()` function are executed once when the program begins and are used to initialize. Use it to set directions of pins or to initialize variables. The instructions placed in `loop` are executed repeatedly and form the main tasks of the program. Therefore every program has this structure

```

void setup()
{
    // commands to initialize go here
}

void loop()
{
    // commands to run your machine go here
}

```

The absolute, bare-minimum, do-nothing program that you can compile and run is

```

void setup() {} void loop() {}

```

The program performs no function, but is useful for clearing out any old program. Note that the compiler does not care about line returns, which is why this program works if typed all on one line.

5.4 Math

The Arduino can do standard mathematical operations. While floating point (e.g. 23.2) numbers are allowed if declared as floats, operations on floats are very slow so integer variables and integer math is recommended. If you have byte variables, no number, nor the result of any math operation can fall outside the range of 0 to 255. You can divide numbers, but the result will be truncated (not rounded) to the nearest integer. Thus in integer arithmetic, $17/3 = 5$, and not 5.666 and not 6. Math operations are performed strictly in a left-to-right order. You can add parenthesis to group operations.

The table below shows some of the valid math operators. Full details of their use can be found in the Arduino Language Reference.

Symbol	Description
+	addition
-	subtraction
*	multiplication
/	division
%	modulus (division remainder)
<<	left bit shift
>>	right bit shift
&	bitwise AND
	bitwise OR

6 The Simple Commands

This section covers the small set of commands you need to make the Arduino do something useful. These commands appear in order of priority. You can make a great machine using only digital read, digital write and delay commands. Learning all the commands here will take you to the next level.

If you need more, consult the Arduino language reference page at <http://arduino.cc/en/Reference/HomePage>.

pinMode

This command, which goes in the setup() function, is used to set the direction of a digital I/O pin. Set the pin to OUTPUT if the pin is driving an LED, motor or other device. Set the pin to INPUT if the pin is reading a switch or other sensor. On power up or reset, all pins default to inputs. This example sets pin 2 to an output and pin 3 to an input.

```
void setup()
{
  pinMode(2,OUTPUT);
  pinMode(3,INPUT);
}
void loop() {}
```

Serial.print

The `Serial.print` command lets you see what's going on inside the Arduino from your computer. For example, you can see the result of a math operation to determine if you are getting the right number. Or, you can see the state of a digital input pin. When your interface circuits or program does not seem to be working, use the `Serial.print` command to shed a little light on the situation. For this command to show anything, you need to have the Arduino connected to the host computer with the USB cable.

For the command to work, the command `Serial.begin(9600)` must be placed in the `setup()` function. After the program is uploaded, you must open the Serial Monitor window to see the response.

There are two forms of the print command. `Serial.print()` prints on the same line while `Serial.println()` starts the print on a new line.

Here is a brief program to check if your board is alive and connected to the PC

```
void setup()
{
  Serial.begin(9600);
  Serial.println("Hello World");
}
void loop() {}
```

Here is a program that loops in place, displaying the value of an I/O pin. This is useful for checking the state of sensors or switches and to see if the Arduino is reading the sensor properly. Try it out on your Arduino. After uploading the program, use a jumper wire to alternately connect pin 2 to +5V and to Gnd.

```
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  Serial.println(digitalRead(2));
  delay(100);
}
```

If you wanted to see the states of pins 2 and 3 at the same time, you can chain a few print commands, noting that the last command is a `println` to start a new line.

```
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  Serial.print("pin 2 = ");
  Serial.print(digitalRead(2));
  Serial.print("    pin 3 = ");
  Serial.println(digitalRead(3));
}
```



```

    delay(100);
}

```

You may have noticed when trying this out that if you leave one of the pins disconnected, its state follows the other. This is because a pin left floating has an undefined value and will wander from high to low. So, use two jumper wires when trying out this example.

Here's one that checks the value of a variable after an addition. Because the calculation is done just once, all the code is in the `setup()` function.

```

int i,j,k;
void setup()
{
    Serial.begin(9600);
    i=21;
    j=20;
    k=i+j;
    Serial.flush();
    Serial.print(k);
}
void loop() {}

```

digitalWrite

This command sets an I/O pin high (+5V) or low (0V) and is the workhorse for commanding the outside world of lights, motors, and anything else interfaced to your board. Use the `pinMode()` command in the `setup()` function to set the pin to an output.

```

digitalWrite(2,HIGH);    // sets pin 2 to +5 volts
digitalWrite(2,LOW);     // sets pin 2 to zero volts

```

delay

Delay pauses the program for a specified number of milliseconds. Since most interactions with the world involve timing, this is an essential instruction. The delay can be for 0 to 4,294,967,295 msec. This code snippet turn on pin 2 for 1 second.

```

digitalWrite(2,HIGH);    // pin 2 high (LED on)
delay(1000);             // wait 500 ms
digitalWrite(2,LOW);     // pin 2 low (LED off)

```

if

This is the basic conditional branch instruction that allows your program to do two different things depending on whether a specified condition is true or false.

Here is one way to have your program wait in place until a switch is closed. Connect a switch to pin 3 as shown in Section 3. Upload this program then try closing the switch

```

void setup()

```

```

{
  Serial.begin(9600);
}

void loop()
{
  if (digitalRead(3) == LOW) {
    Serial.println("Somebody closed the switch!");
  }
}

```

The if line reads the state of pin 3. If it is high, which it will be for this circuit when the switch is open, the code jumps over the Serial.println command and will repeat the loop. When you close the switch, 0V is applied to pin 3 and its state is now LOW. This means the if condition is true so this time around the code between the braces is executed and the message is printed.

The syntax for the if statement is

```

if (condition) {
  //commands
}

```

If the condition is true, the program will execute the commands between the braces. If the condition is not true, the program will skip to the statement following the braces.

The condition compares one thing to another. In the example above, the state of pin 1 was compared to LOW with ==, the equality condition. Other conditional operators are != (not equal to), > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to).

You can have the program branch depending on the value of a variable. For example, this program will print the value of i only when it is less than 30.

```

int i;

void setup()
{
  Serial.begin(9600);
  i=0;
}

void loop()
{
  i=i+1;
  if (i<30) {
    Serial.println(i);
  }
}

```

for

The for statement is used to create program loops. Here is an example that flashes an LED attached to pin 2 five times

```
int i;

void setup()
{
  pinMode(2,OUTPUT);
  for (i=0;i<5;i++) {
    digitalWrite(2,HIGH);
    delay(250);
    digitalWrite(2,LOW);
    delay(250);
  }
}
void loop() {}
```

The variable *i* is the loop counter. The for() statement has three parts: the initialization, the check and the increment. Variable *i* is initialized to zero. The check is to see if *i* is less than 5. If so, the commands between the braces are executed. If not, those commands are skipped. After the check, *i* is incremented by 1 (the *i++* command). While the for statement could read for (*i=1;i==5;i++*), it is convention to start the counter variable at zero and use less than for the condition check.

You can have the loop counter increment by two or by three or by any increment you want. For example, try this code fragment.

```
int i;
void setup()
{
  Serial.begin(9600);
  for (i=0;i<15;i=i+3) {
    Serial.println(i);
  }
}
void loop() {}
```

Loops can be nested within loops. This example will flash the LED 10 times because for each of the five outer loops counted by *i*, the program goes twice through the inner loop counted by *j*.

```
int i,j;
void setup()
{
  pinMode(2,OUTPUT);
  for (i=0;i<5;i++) {
    for(j=0;j<2;j++) {
      digitalWrite(2,HIGH);
      delay(250);
      digitalWrite(2,LOW);
      delay(250);
    }
  }
}
```

```

    }
    void loop() {}

```

while

The while statement is another branch command that does continuous looping. If the condition following the while is true, the commands within the braces are executed continuously. Here is an example that continuously reads a switch on pin 3, and then when the switch is pressed, the condition is no longer true so the code escapes the while command and prints.

```

void setup()
{
    Serial.begin(9600);
    while(digitalRead(3) == HIGH) {
    }
    Serial.println("Switch was pressed");
}
void loop() {}

```

goto

The goto statement commands the computer to jump immediately to another part of the program marked by an address label. The goto should be used sparingly because it makes the program hard to follow, but is handy for breaking out of nested loops or other complex control structures.

Here is an example

```

void setup()
{
    Serial.begin(9600);
    while(true) {
        if (digitalRead(3) == LOW) {
            goto wrapup;
        }
    }
wrapup:
    Serial.println("Switch was pressed");
}
void loop() {}

```

The while(true) statement runs continuously, checking the state of pin 3 each time. When pin 3 is low (pressed), the if condition is true and the goto statement executed, breaking out of the while loop.

functions

Functions are a powerful programming feature that are used when you want to set up an action that can be called from several places in the program. For example, let's say you wanted an LED connected to pin 2 to flash 3 times as an alert, but that you needed to execute the alert at three different places in the program. One solution would be to type in the flashing code at the three separate program locations. This uses up precious code space and also means that if you change the flash function, for example changing from 3 flashes to 4, you have to change the code in three places. A better solution is to write the flash function as a subroutine and to call it from the main body of the code. Here is an example

```

int i;
void setup()
{
  pinMode(2,OUTPUT);
  Serial.begin(9600);

  Serial.println("Welcome to my program");
  delay(1000);
  flasher(); // call flasher function
  Serial.println("I hope you like flashing");
  delay(1000);
  flasher(); // call flasher again
  Serial.println("Here it is one more time");
  delay(1000);
  flasher();
}
void loop() {}

void flasher()
{
  for(i=0;i<3;i++) {
    digitalWrite(2,HIGH);
    delay(250);
    digitalWrite(2,LOW);
    delay(250);
  }
}

```

Several things should be noted here. The function `flasher()` is defined outside the `setup()` and `loop()` functions. When the main program encounters a `flasher();` command, the program immediately jumps to the function and starts executing the code there. When it reaches the end of the function, the program returns to execute the command that immediately follows the `flasher();` command. It is this feature that allows you to call the subroutine from several different places in the code. Parameters can be passed to and returned from functions.

This concludes the section on basic program commands. There is much more that the Arduino can do and you are urged to read through the complete Arduino Language Reference page online.

7 Plotting graphs using Matlab (in real time)

Interfacing with Matlab is possible with the Arduino board which provides a convenient way of plotting data. The following matlab code plots data provided from the serial port in real time.

Arduino code (send data to serial port):

```

int i=0;
void setup()
{
  Serial.begin(9600);

```

```

    }

    void loop()
    {
        while(i<500){
            Serial.println(i);
            i++; }
        }

```

Matlab code that reads data from serial port and plots them in real time (make sure you are using the right COM port in the “serialport” command):

Note that for the code below to work the port used by the Arduino IDE needs to be changed (or the Arduino IDE software needs to be closed) as otherwise the serial port will be reserved, and Matlab will not be able to access it.

```

clear all
clc
Sard=serialport('COM3',9600);
fopen(Sard);
figure(1)
grid
x=[1:200];
axis([0 length(x) 0 length(x)])
    hold on
    for i=1:length(x)
        y(i)=fscanf(Sard,'%d');
        plot(x(i),y(i),'r*')
        drawnow
    end
    hold off
    fclose(Sard);
    delete(Sard);
    clear Sard

```

Also make sure that all serial sessions and corresponding matlab figures are closed (fclose, delete commands in the end) when the task is finished as otherwise this might affect the operation of the Arduino.

8 Plotting graphs using Python (in real time)

Analogous real-time plots of data sent via the serial port can also be plotted out using Python.

As in the previous section the Arduino code below generates some data that is sent to the serial port:

```

int j = 0;
void setup() {
    // put your setup code here, to run once:
    Serial.begin(9600);
}

void loop() {
    // put your main code here, to run repeatedly:
    for(int j = 0; j<150;j++){
        Serial.println(j);
    }
}

```

If using Anaconda on your own computer to run python scripts, the following command needs to be run at a cmd.exe terminal to install the appropriate package that will enable the monitoring of the serial port.

```
conda install -c anaconda pyserial
```

If you are using an EITEL computer in order to run python scripts and install packages you need to run the Jupyter notebook using the file “SF4_Jupyter_Launcher.bat” available in the Python folder of the moodle page of the project: <https://www.vle.cam.ac.uk/course/view.php?id=70291>.

NOTE that the file has to be downloaded to the desktop (or in one of the folders of the shared file system) and only then run it.

In order to install packages you have to launch an Anaconda prompt. This can be done by launching the Anaconda GUI and selecting CMD.EXE. Once you have started the command line then you can install packages via:

```
pip install NameOfPackage
```

For example, in order to run the python code in the next page that is a basic serial communication example, you **must install** the pyserial module via the command

```
pip install pyserial
```

The python code below reads data from the serial port and plots them in real time (make sure you are using the right com port in the “serial” command). The same code is also included in the Python folder of the moodle page of the lab, file “SF4_ArduninoCommunicationNotebook.ipynb”.

Note that for the code below to work the port used by the Arduino IDE needs to be changed (or the Arduino IDE software needs to be closed) as otherwise the serial port will be reserved, and python will not be able to access it. As noted above, check also that the com port in the serial command is the one to which the Arduino is connected. Furthermore, if copying code from the pdf file, please note that python code is sensitive to indentations, so these need to be corrected if they are altered when copying.


```

import datetime as dt
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import serial #note that this package needs to be installed via
#pip pyserial and NOT serial

# Enable interactive plot
%matplotlib notebook

# Create figure for plotting
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
xs = []
ys = []
arduinoSerialData = serial.Serial('com3',9600)
# This function is called periodically from FuncAnimation
def animate(i, xs, ys):
    StringData = arduinoSerialData.readline()
    temp_c = float(StringData)# Convert serial string into a number

    # Add x and y to lists
    xs.append(dt.datetime.now().strftime('%H:%M:%S.%f'))
    ys.append(temp_c)
    # Limit x and y lists to 20 items
    xs = xs[-20:]
    ys = ys[-20:]

    # Draw x and y lists
    ax.clear()
    ax.plot(xs, ys)
    # Format plot
    plt.xticks(rotation=45, ha='right')
    plt.subplots_adjust(bottom=0.30)
    plt.title('Data over Time')
    plt.ylabel('Data samples')
# Set up plot to call animate() function periodically
anim = animation.FuncAnimation(fig, animate, fargs=(xs, ys),
interval=1000)
plt.show()

```

9 Coding Style

Style refers to your own particular style for creating code and includes layout, conventions for using case, headers, and use of comments. All code must follow correct syntax, but there are many different styles you can use. Here are some suggestions:

- Start every program with a comment header that has the program name and perhaps a brief description of what the program does.
- Use indentation to line things up. Function name and braces are in column one, then use indents in multiples of 2 or 4 to mark code chunks, things inside loops and so on.
- Mark major sections or functions with a comment header line or two
- Have just the right number of comments, not too few and not too many. Assume the reader knows the programming language so have the comment be instructive. Here is an example of an instructive comment

```
digitalWrite(4,HIGH)    // turn on motor
```

and here is a useless comment

```
digitalWrite(4,HIGH)    // set pin 4 HIGH
```

You need not comment every line. In fact, commenting every line is generally bad practice.
- Add the comments when you create the code. If you tell yourself, "Oh, I'll add the comments when the code is finished", you will never do it.

10 Common Coding Errors

- Forgetting the semi-colon at the end of a statement
- Misspelling a command
- Omitting opening or closing braces

11 Appendix A

On the following page there is a summary of basic Arduino commands, with examples.

Arduino Programming Basics

Command	Description
<code>pinMode(<i>n</i>, INPUT)</code>	Set pin <i>n</i> to act as an input. One-time command at top of program.
<code>pinMode(<i>n</i>, OUTPUT)</code>	Set pin <i>n</i> to act as an output
<code>digitalWrite(<i>n</i>, HIGH)</code>	Set pin <i>n</i> to 5V
<code>digitalWrite(<i>n</i>, LOW)</code>	Set pin <i>n</i> to 0V
<code>delay(<i>x</i>)</code>	Pause program for <i>x</i> millisec, <i>x</i> = 0 to 65,535
<code>tone(<i>n</i>, <i>f</i>, <i>d</i>)</code>	Play tone of frequency <i>f</i> Hz for <i>d</i> millisec on speaker attached to pin <i>n</i>
<code>for()</code>	Loop. Example: <code>for (i=0; i<3; i++) {}</code> Do the instructions enclosed by {} three times
<code>if (expr) {}</code>	Conditional branch. If <i>expr</i> true, do instructions enclosed by {}
<code>while (expr) {}</code>	While <i>expr</i> is true, repeat instructions in {} indefinitely

Instructions in the `setup()` function are executed once. Those in the `loop()` function are executed indefinitely.

Examples

1. Turn on LED connected to Pin 2 for 1 s.

```
void setup() {
  pinMode(2, OUTPUT);
  digitalWrite(2, HIGH);
  delay(1000);
  digitalWrite(2, LOW);
}
void loop()
{}
```

2. Flash LED connected to Pin 2 at 1 Hz forever.

```
void setup() {
  pinMode(2, OUTPUT);
}
void loop() {
  digitalWrite(2, HIGH);
  delay(500);
  digitalWrite(2, LOW);
  delay(500);
}
```

3. Turn on motor connected to Pin 4 for 1 s.

```
void setup() {
  pinMode(4, OUTPUT);
  digitalWrite(4, HIGH);
  delay(1000);
  digitalWrite(4, LOW);
}
void loop()
{}
```

4. Play 440 Hz tone for one second on speaker connected to pin 5. Delay is needed because the program does not wait for the `tone()` command to finish but rather immediately goes to the command following `tone()`.

```
void setup() {
  pinMode(5, OUTPUT);
  tone(5, 440, 1000);
  delay(1100);
}
void loop()
{}
```

5. LED is on Pin 2 and switch is on Pin 6. Turns on the LED for one sec when switch is pressed.

```
void setup() {
  pinMode(2, OUTPUT);
  pinMode(6, INPUT);
  while (digitalRead(6) == HIGH)
    ;
  digitalWrite(2, HIGH);
  delay(1000);
  digitalWrite(2, LOW);
}
void loop()
{}
```

12 Appendix B

Additional reading material for more advanced operations is provided in the Moodle website and you are strongly advised to go through that as well.