

Arduino mini-course

Lesson 1: Connections, inputs, processes, outputs.

Vocabulary

Voltage, current, short.

Function, variable, variable type, threshold, syntax.

Components

UNO, USB cable, breadboard, light dependent resistor, LED, jumper wires.

Symbolic diagrams and realistic representations

[SYMBOLIC DIAGRAM OF THIS CIRCUIT] [REALISTIC REPRESENTATION INCLUDING BREADBOARD]

We supply both in this lesson but in future you might only see one or the other.

Connections

Connecting the UNO to USB as power supply and programming cable. Disconnecting during circuit-building to avoid shorts.

Using a breadboard

Busses are five pins connected to each other inside the breadboard; let's call these "rows." Also there are longer busses called "rails" at top and bottom of breadboard usually for connecting high and low voltage ends of power supply; let's call these "five volts" and "ground."

Components like a light-dependent resistor can straddle two busses so that each lead can be connected to a different part of the circuit.

[DIAGRAM]

Programming

Arduino UNOs are programmed with a variation of the C+

+ language.

Arduinos require two core functions, and from within them you can call variables and functions defined outside them, as well as built-in constants and functions. The two core functions are `setup()` which is run once every time the Arduino is reset, and `loop()`, which restarts every time the program pointer reaches the end of `loop()`.

We will use the built in flow-control function `if()`, which requires a certain syntax:

```
if ([LOGICAL CONDITION]) {[FUNCTION(S) TO  
BE EXECUTED IF CONDITION IS TRUE];}  
[OPTIONAL ELSE:]  
else {[FUNCTION(S) TO BE EXECUTED IF  
CONDITION IS FALSE];}
```

We will use some built-in operation-functions like `digitalRead()` and `digitalWrite()`.

We will define our own functions to do things we want like:

```
void turn_led_on()  
{digitalWrite(LED_BUILTIN,HIGH);}   
void turn_led_off()  
{digitalWrite(LED_BUILTIN,LOW);} 
```

Note the syntax of functions having parentheses, their definitions being enclosed by curly braces, and each operation is followed with a semicolon, unlike flow control `if()` which is not followed by a semicolon.

We can set variables, defining them outside the core functions so they can be used inside both. Notice we have to tell what kind of information is in the variable, in this case it is an integer between -32,768 and 32,767, called "int":

```
int threshold = 400;
```

Arduino pins can be used in different ways, so have to be

programmed explicitly to work as input or output:

```
pinMode(LED_BUILTIN, OUTPUT);
```

Here's a complete starter program for reading an input and controlling an output based on the level of the input:

```
int threshold = 400;
void turn_led_on()
{digitalWrite(LED_BUILTIN, HIGH);}
void turn_led_off()
{digitalWrite(LED_BUILTIN, LOW);}
setup(){
pinMode(LED_BUILTIN, OUTPUT);
}
loop(){
int a1 = analogRead(A1);
if(a1>threshold) {turn_led_on();}
else {turn_led_off();}
}
```

Lesson 2: Logical flow and extension

Amending program

Say you have a project where a light turns on when the sensor is darkened. You can reverse the behavior of the project either by reversing the circuit, in the case of a voltage divider, or by changing the logical conditions in the program. Before we had

```
if (light_sensor_v() > 400)
{turn_led_on();}
```

and we can simply flip the condition to get the opposite reaction:

```
if (light_sensor_v() < 400)
{turn_led_on();}
```

Or we can flip the reaction:

```
if (light_sensor_v() > 400)
{turn_led_off();}
```

Adding inputs

```
if (light_sensor_v() > 400 and
temp_sensor_c > 100) {turn_led_on();}
```

Lesson 3: Smartening

Handling engineering constraints

Power usage

User experience

Cost of materials

Time

How to use internal clock function `millis()`. In Arduino's language, `millis()` returns a long integer indicating milliseconds since the program began. At some point around 50 days, the clock will reach the limits of its memory (4,294,967,295 is the max of 32 bits) and start back at zero. You can handle this pretty well by carefully choosing order of operations when calculating a difference in time: Say `millis()` returns 4,294,967,000 for the start of an event, then a second later it would return 5. If the Arduino subtracted 5 from 4,294,967,000 the result would be 4,294,966,995, which is 4,294,967 seconds. But if the Arduino subtracted 4,294,967,000 from 5, the subtraction itself would wrap around and result would be 1000, which is one second. So:

```
time_start = millis();
```

```
...//DO SOME STUFF OR WAIT
```

```
time_elapsed = millis() - time_start;
```

If you have to measure an interval longer than 50 days, then you need to look for and count the wraps of `millis()`.

Discriminating levels of inputs

Adding levels to outputs

Instead of just turning an LED on or off, we can change its brightness. The value of `a1` in our original program could be as high as 1023, but `analogWrite()` only takes values up to 255, so we use the bitshift operator to essentially divide by 2^2 .

```
analogWrite(9, a1 >> 2);
```

Comparing one input to another

We can add a potentiometer, a way of letting a user vary a voltage with the turn of a knob:

[DIAGRAM ADDING POTENTIOMETER CONNECTED TO 5V, A4, GND]

```
int a1 = analogRead(A1);  
int threshold = analogRead(A4);  
if(a1>threshold) {turn_led_on();}
```