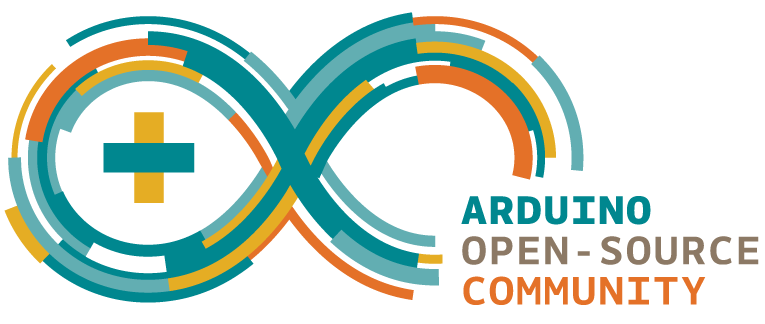
****

**Line following robot coding guide**



**Coding Structure**

The Arduino robots run on code written in the Arduino IDE (Integrated Development Environment). This is basically a text file of commands which the robot will run through in order. All Arduino programs have the same basic structure:

Declare variables; *– describe the hardware set in software so you can follow what the code is controlling, all code lines finish with either a semi-colon* ; *to state that a sentence is complete, or a curly bracket* { *to show that there is more that should be done.*

void setup(){ *– describe actions to be taken once at power on. The curly brackets* {} *are used to define ‘sections’ of code, so all of the setup instructions are contained within these brackets.*

}

void loop(){ *– describe the actions to be taken by the robot, these will be repeated over and over.*

Check where line is; *- this section contains commands to be run.*

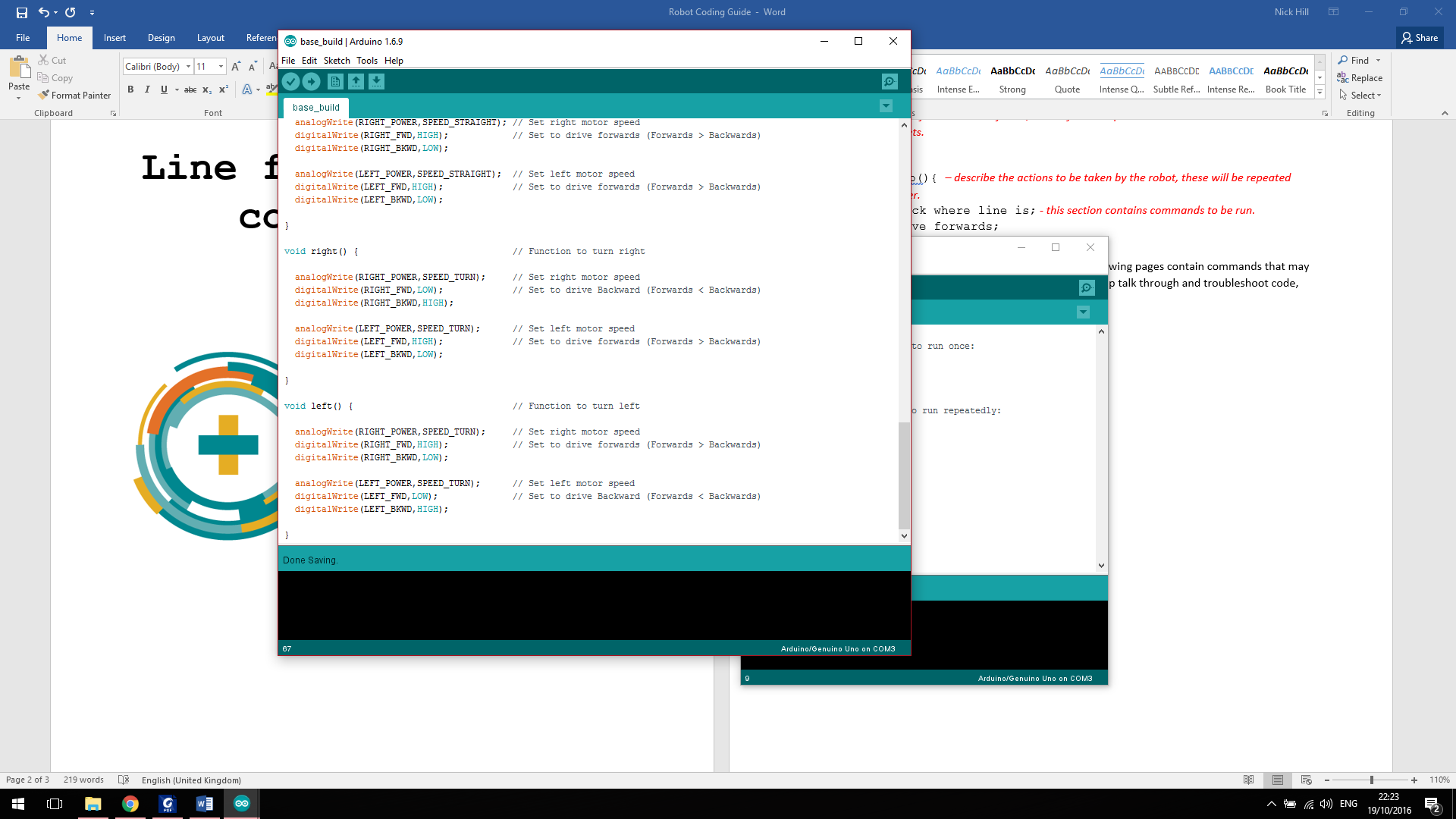
Drive forwards;

}

This is the basic structure of all Arduino programs. The following pages contain commands that may be of use in coding your robot. The staff are available to help talk through and troubleshoot code, but feel free to experiment and see what works!

**Uploading Code to the Robot**

Within the Ardunio IDE the top toolbar contains all you need to check and run your code:



This button uploads your code to the robot through the blue cable. The code will be run by the robot whenever it has power, but the motors will only run when the battery pack is turned on.

This button checks that your code will run on the robot and lets you know if there are any errors. **Check for errors before uploading to your robot!**

These are the fundamentals, with a walk through the existing code “base\_build” you’re ready to try changing things and see what happens! The rest of this document is reference material from the Arduino website.

**if (conditional) and ==, !=, <, > (comparison operators)**

if, which is used in conjunction with a comparison operator, tests whether a certain condition has been reached, such as an input being above a certain number. The format for an if test is:

if (someVariable > 50)

{

// do something here

}

The program tests to see if someVariable is greater than 50. If it is, the program takes a particular action. Put another way, if the statement in parentheses is true, the statements inside the brackets are run. If not, the program skips over the code.

The brackets may be omitted after an if statement. If this is done, the next line (defined by the semicolon) becomes the only conditional statement.

if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120)

digitalWrite(LEDpin, HIGH);

if (x > 120){ digitalWrite(LEDpin, HIGH); }

if (x > 120){

digitalWrite(LEDpin1, HIGH);

digitalWrite(LEDpin2, HIGH);

} // all are correct

The statements being evaluated inside the parentheses require the use of one or more operators:

Comparison Operators:

x == y (x is equal to y)

x != y (x is not equal to y)

x < y (x is less than y)

x > y (x is greater than y)

x <= y (x is less than or equal to y)

x >= y (x is greater than or equal to y)

Warning:

Beware of accidentally using the single equal sign (e.g. if (x = 10) ). The single equal sign is the assignment operator, and sets x to 10 (puts the value 10 into the variable x). Instead use the double equal sign (e.g. if (x == 10) ), which is the comparison operator, and tests whether x is equal to 10 or not. The latter statement is only true if x equals 10, but the former statement will always be true.

This is because C evaluates the statement if (x=10) as follows: 10 is assigned to x (remember that the single equal sign is the assignment operator), so x now contains 10. Then the 'if' conditional evaluates 10, which always evaluates to TRUE, since any non-zero number evaluates to TRUE. Consequently, if (x = 10) will always evaluate to TRUE, which is not the desired result when using an 'if' statement. Additionally, the variable x will be set to 10, which is also not a desired action.

**if / else**

if/else allows greater control over the flow of code than the basic if statement, by allowing multiple tests to be grouped together. For example, an analog input could be tested and one action taken if the input was less than 500, and another action taken if the input was 500 or greater. The code would look like this:

if (pinFiveInput < 500)

{

// action A

}

else

{

// action B

}

else can proceed another if test, so that multiple, mutually exclusive tests can be run at the same time.

Each test will proceed to the next one until a true test is encountered. When a true test is found, its associated block of code is run, and the program then skips to the line following the entire if/else construction. If no test proves to be true, the default else block is executed, if one is present, and sets the default behavior.

Note that an else if block may be used with or without a terminating else block and vice versa. An unlimited number of such else if branches is allowed.

if (pinFiveInput < 500)

{

// do Thing A

}

else if (pinFiveInput >= 1000)

{

// do Thing B

}

else

{

// do Thing C

}

**Addition, Subtraction, Multiplication, & Division**

Description

These operators return the sum, difference, product, or quotient (respectively) of the two operands. The operation is conducted using the data type of the operands, so, for example, 9 / 4 gives 2 since 9 and 4 are ints. This also means that the operation can overflow if the result is larger than that which can be stored in the data type (e.g. adding 1 to an int with the value 32,767 gives -32,768). If the operands are of different types, the "larger" type is used for the calculation.

If one of the numbers (operands) are of the type float or of type double, floating point math will be used for the calculation.

Examples

y = y + 3;

x = x - 7;

i = j \* 6;

r = r / 5;

Syntax

result = value1 + value2;

result = value1 - value2;

result = value1 \* value2;

result = value1 / value2;

Parameters:

value1: any variable or constant

value2: any variable or constant

Programming Tips:

Know that integer constants default to int, so some constant calculations may overflow (e.g. 60 \* 1000 will yield a negative result).

Choose variable sizes that are large enough to hold the largest results from your calculations

Know at what point your variable will "roll over" and also what happens in the other direction e.g. (0 - 1) OR (0 - - 32768)

**Boolean Operators**

These can be used inside the condition of an if statement.

&& (logical and)

True only if both operands are true, e.g.

if (digitalRead(2) == HIGH && digitalRead(3) == HIGH) { // read two switches

// ...

}

is true only if both inputs are high.

|| (logical or)

True if either operand is true, e.g.

if (x > 0 || y > 0) {

// ...

}

is true if either x or y is greater than 0.

! (not)

True if the operand is false, e.g.

if (!x) {

// ...

}

is true if x is false (i.e. if x equals 0).

Examples

if (a >= 10 && a <= 20){} // true if a is between 10 and 20

**switch / case statements**

Like if statements, switch...case controls the flow of programs by allowing programmers to specify different code that should be executed in various conditions. In particular, a switch statement compares the value of a variable to the values specified in case statements. When a case statement is found whose value matches that of the variable, the code in that case statement is run.

The break keyword exits the switch statement, and is typically used at the end of each case. Without a break statement, the switch statement will continue executing the following expressions ("falling-through") until a break, or the end of the switch statement is reached.

Example

switch (var) {

case 1:

//do something when var equals 1

break;

case 2:

//do something when var equals 2

break;

default:

// if nothing else matches, do the default

// default is optional

break;

}

Please note that in order to declare variables within a case brackets are needed. An example is showed below.

switch (var) {

case 1:

{

//do something when var equals 1

int a = 0;

.......

}

break;

default:

// if nothing else matches, do the default

// default is optional

break;

}

Syntax

switch (var) {

case label:

// statements

break;

case label:

// statements

break;

default:

// statements

break;

}

Parameters

var: the variable whose value to compare to the various cases

label: a value to compare the variable to

**Variables**

A variable is a place to store a piece of data. It has a name, a value, and a type. For example, this statement (called a *declaration*):

int pin = 13;

creates a variable whose name is pin, whose value is 13, and whose type is int. Later on in the program, you can refer to this variable by its name, at which point its value will be looked up and used. For example, in this statement:

pinMode(pin, OUTPUT);

it is the value of pin (13) that will be passed to the pinMode() function. In this case, you don't actually need to use a variable, this statement would work just as well:

pinMode(13, OUTPUT);

The advantage of a variable in this case is that you only need to specify the actual number of the pin once, but you can use it lots of times. So if you later decide to change from pin 13 to pin 12, you only need to change one spot in the code. Also, you can use a descriptive name to make the significance of the variable clear (e.g. a program controlling an RGB LED might have variables called redPin, greenPin, and bluePin).

A variable has other advantages over a value like a number. Most importantly, you can change the value of a variable using an *assignment* (indicated by an equals sign). For example:

pin = 12;

will change the value of the variable to 12. Notice that we don't specify the type of the variable: it's not changed by the assignment. That is, the name of the variable is permanently associated with a type; only its value changes. [1] Note that you have to declare a variable before you can assign a value to it. If you include the preceding statement in a program without the first statement above, you'll get a message like: "error: pin was not declared in this scope".

When you assign one variable to another, you're making a copy of its value and storing that copy in the location in memory associated with the other variable. Changing one has no effect on the other. For example, after:

int pin = 13;

int pin2 = pin;

pin = 12;

only pin has the value 12; pin2 is still 13.

Now what, you might be wondering, did the word "scope" in that error message above mean? It refers to the part of your program in which the variable can be used. This is determined by where you declare it. For example, if you want to be able to use a variable anywhere in your program, you can declare at the top of your code. This is called a *global*variable; here's an example:

int pin = 13;

void setup()

{

pinMode(pin, OUTPUT);

}

void loop()

{

digitalWrite(pin, HIGH);

}

As you can see, pin is used in both the setup() and loop() functions. Both functions are referring to the same variable, so that changing it one will affect the value it has in the other, as in:

int pin = 13;

void setup()

{

pin = 12;

pinMode(pin, OUTPUT);

}

void loop()

{

digitalWrite(pin, HIGH);

}

Here, the digitalWrite() function called from loop() will be passed a value of 12, since that's the value that was assigned to the variable in the setup() function.

If you only need to use a variable in a single function, you can declare it there, in which case its scope will be limited to that function. For example:

void setup()

{

int pin = 13;

pinMode(pin, OUTPUT);

digitalWrite(pin, HIGH);

}

In this case, the variable pin can only be used inside the setup() function. If you try to do something like this:

void loop()

{

digitalWrite(pin, LOW); // wrong: pin is not in scope here.

}

you'll get the same message as before: "error: 'pin' was not declared in this scope". That is, even though you've declared pin somewhere in your program, you're trying to use it somewhere outside its scope.

Why, you might be wondering, wouldn't you make all your variables global? After all, if I don't know where I might need a variable, why should I limit its scope to just one function? The answer is that it can make it easier to figure out what happens to it. If a variable is global, its value could be changed anywhere in the code, meaning that you need to understand the whole program to know what will happen to the variable. For example, if your variable has a value you didn't expect, it can be much easier to figure out where the value came from if the variable has a limited scope.

Functions

Segmenting code into functions allows a programmer to create modular pieces of code that perform a defined task and then return to the area of code from which the function was "called". The typical case for creating a function is when one needs to perform the same action multiple times in a program.

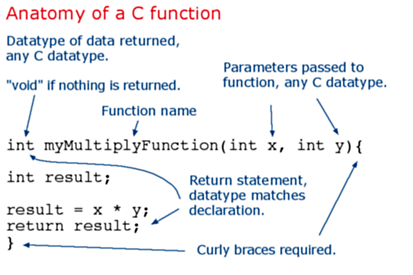
For programmers accustomed to using BASIC, functions in Arduino provide (and extend) the utility of using subroutines (GOSUB in BASIC).

Standardizing code fragments into functions has several advantages:

* Functions help the programmer stay organized. Often this helps to conceptualize the program.
* Functions codify one action in one place so that the function only has to be thought out and debugged once.
* This also reduces chances for errors in modification, if the code needs to be changed.
* Functions make the whole sketch smaller and more compact because sections of code are reused many times.
* They make it easier to reuse code in other programs by making it more modular, and as a nice side effect, using functions also often makes the code more readable.

There are two required functions in an Arduino sketch, setup() and loop(). Other functions must be created outside the brackets of those two functions. As an example, we will create a simple function to multiply two numbers.

Example



To "call" our simple multiply function, we pass it parameters of the datatype that it is expecting:

void **loop**(){  
int i = 2;  
int j = 3;  
int k;  
  
k = myMultiplyFunction(i, j); *// k now contains 6*  
}

Our function needs to be *declared* outside any other function, so "myMultiplyFunction()" can go either above or below the "loop()" function. The entire sketch would then look like this:

void **setup**(){  
  Serial.begin(9600);  
}  
  
void **loop**() {  
  int i = 2;  
  int j = 3;  
  int k;  
  
  k = myMultiplyFunction(i, j); *// k now contains 6*  
  Serial.println(k);  
  delay(500);  
}  
  
int myMultiplyFunction(int x, int y){  
  int result;  
  result = x \* y;  
  return result;  
}

Another example

This function will read a sensor five times with analogRead() and calculate the average of five readings. It then scales the data to 8 bits (0-255), and inverts it, returning the inverted result.

int ReadSens\_and\_Condition(){  
  int i;  
  int sval = 0;  
  
  for (i = 0; i < 5; i++){  
    sval = sval + analogRead(0);    *// sensor on analog pin 0*  
  }  
  
  sval = sval / 5;    *// average*  
  sval = sval / 4;    *// scale to 8 bits (0 - 255)*  
  sval = 255 - sval;  *// invert output*  
  return sval;  
}

To call our function we just assign it to a variable.

int sens;  
  
sens = ReadSens\_and\_Condition();