CODE

Arduino Code layout

Snippet:

```
//Global constants here


//Global variables here


void setup () {
//Setup pins etc here, runs only once
    }
void loop() {
//Main code goes here, runs all the time
}


//Functions here
```

Function:

This a template for laying out your code in a consistent manner, which aids it's development and understanding.

The lines that beginning // are comments to give information on what is happening at this point in your code.

CODE

How to delay for a set time

Snippet:

Syntax

```
delay(ms);
```

Parameters

ms: the number of milliseconds to pause (unsigned long)

Return

- None

Function:

Pauses the program for the amount of time (in miliseconds) specified as parameter. (There are 1000 milliseconds in a second.)

CODE

Reading a Digital input

Snippet:

Syntax

```
var = digitalRead(pin);
```

Parameters

- pin: the number of the digital pin you want to read

Return

- HIGH or LOW

Function:

Reads the value from a specified digital pin, either **HIGH** or **LOW**.

CODE

Create a variable

Snippet:

Syntax

```
type name = value;
```

Parameters

- type: variable type - int, byte, boolean, word, char
- name: meaningful name for you variable
- Value: the value you wish to store in the variable

Return

- None

Function:

This is the method use to set up variables, they need to be initialised at the start of your code - see the Arduino code layout card.

CODE

Configure your Pins

Snippet:

Syntax

```
pinMode (pin, type);
```

Parameters

- pin: is the number of the pin you are configuring
- type is the mode for that pin:
  - INPUT
  - INPUT_PULLUP
  - OUTPUT

Return

- None

Function:

This command is used to set the operation of individual pins. **INPUT** and **OUTPUT** are obvious, the other mode **INPUT_PULLUP** switches on internal pullup resistors so that external ones aren't required with switches.

CODE

Operate a Digital Output

Snippet:

Syntax

```
digitalWrite(pin, value);
```

Parameters

- pin: the pin number
- value: HIGH or LOW

Return

- None

Function:

If the pin has been configured as an **OUTPUT** with **pinMode()**, its voltage will be set to the corresponding value: 5V (or 3.3V on 3.3V boards) for **HIGH**, 0V (ground) for **LOW**.

CODE

Operate an Analog Output

Snippet:

Syntax

```
analogWrite(pin, value)
```

Parameters

- pin: the pin to write to
- value: the duty cycle: between 0 (always off) and 255 (always on)

Return

- None

Function:

Writes an analog value (**PWM wave)** to a pin. Can be used to light a LED at varying brightnesses or drive a motor at various speeds. After a call to analogWrite(), the pin will generate a steady square wave of the specified duty cycle until the next command on the same pin,.

CODE

Create a constant

Snippet:

Syntax

```
const type name = value;
```

Parameters

- const: indicates a fixed value variable
- type: variable type - int, byte, boolean, word
- name: meaningful name for you variable
- Value: the value you wish to store in the variable

Return

- None

Function:

This is the method use to set up constants, these have fixed values and **cannot** be changed. They need to be initialised at the start of your code - see the Arduino code layout card. Good for named values such as time delays, if you alter the value, then all occurrences are updated.

CODE

Variable types

Snippet:

The Arduino code has the following variable types;

- Boolean - holds one of two values, true or false.
- Byte - stores an 8-bit unsigned number, from 0 to 255.
- Char - stores an ASCII character value
- Int - stores a 16-bit value -32,768 to 32,767
- Word - stores a 16-bit value 0 to 65535
- String - stores text eg “Hello world”
- Array - a collection of variables

Function:

These are the key variable types that you might need to use for most code applications.

CODE

Creating a **Boolean** variable

Snippet:

Example

```
boolean name = flag;
```

Parameters

- name: meaningful name for you variable
- flag: true (yes) or false (no) (1 bit)

Return

- None

Function:

Boolean's are used as flag to indicate whether some action, state or function is on/off, done/not done etc.

CODE

Create a **Byte** variable

Snippet:

Example

```
byte name = value;
```

Parameters

- name: meaningful name for you variable
- value: 0 to 255 (8 bits)

Return

- None

Formatting

- Binary value B11001001, 201 decimal
- Hex value 0xC9, 201 decimal

Function:

Bytes are used to store small numbers.

CODE

Create an **Int** variable

Snippet:

Example

```
int name = value;
```

Parameters

- name: meaningful name for you variable
- Value: -32768 to 32767 (16 bits)

Return

- None

Formatting

- Binary value B11001001, 201 decimal
- Hex value 0xC9, 201 decimal

  Note preceeding 0’s aren’t required

Function:

Int (integers) are the standard value storage for the Arduino

CODE

Create a **Char** variable

Snippet:

Example

```
char name = 'A';
char name = 65;
```

Parameters

- name: meaningful name for you variable
- Letter in single quotes or ASCII value

Return

- None

Function:

A char (character) is used to store a single text letter/character.

CODE

Create a **Word** variable

Snippet:

Example

```
word name = value;
```

Parameters

- name: meaningful name for you variable
- Value: 0 to 65535 (16 bits)

Return

- None

Formatting

- Binary value B11001001, 201 decimal
- Hex value 0xC9, 201 decimal

  Note preceeding 0’s aren’t required

Function:

A word is used to store a 16 bit value which is not signed ie has no negative values. Has the same meaning as unsigned int.

CODE

Create a **String** variable

Snippet:

Example

```
Char name[] = “Arduino”;
```

Parameters

- name: meaningful name for you variable
- List of characters enclosed in speech marks

Return

- None

Function:

A string is used to store text that we might print, display or send to another device - computer, printer or file

CODE

Create an **Array**

Snippet:

Example

```
type name[] = {2,4,2,5,6};
```

Parameters

- type: variable type - int, byte, boolean, word, char
- name: meaningful name for you variable
- List of values separated by commas

Return

- None

Function:

An array is a collection of variables that are accessed with an index number (pointer). A pointer is a variable used to select which value you require from the array. Arrays are **zero indexed**, that is, the first element of the array is at index 0

CODE

Looping for a **While**

Snippet:

Syntax

```
while(expression){
  // statement(s)
}
```

Parameters

- expression: a statement that evaluates to true or false

Example

```
var = 0;
while(var < 200){
 // do something repetitive 200 times
 var++;
}
```

Function:

while loops will loop continuously, and infinitely, until the expression inside the brackets, () becomes false. Something must change the tested variable, or the while loop will never exit. This could be in your code, such as an incremented variable, or an external condition, such as testing a sensor.

CODE

Looping **For** a number of times

Snippet:

Syntax

```
for (init; test; inc/dec) {
    //statement(s);
}
```

Parameters

- init: create a loop variable
- test: a statement that evaluates to true or false
- Inc/dec: adjust the variable by adding / subtracting a value

Example

```
for (int i=0; i <= 255; i++){
    analogWrite(PWMpin, i);
    delay(10);
}
```

Function:

The **for** statement is used to repeat a block of statements inside the curly braces. The init happens once. Each time through the loop, the condition is tested; if it's true, the statement block, and the inc/dec are executed. This is done until condition is false (not true), the loop then ends.

CODE

Do for a while

Snippet:

Syntax

```
do
{
    // statement block
} while (expression);
```

Parameters

- expression: a statement that evaluates to true or false

Example

```
do
{
delay(50); // wait for sensors
  x = readSensors(); // check sensors
} while (x < 100);
```

Function:

The **do** loop works in the same manner as the while loop, with the exception that the condition is tested at the end of the loop, so the do loop will always run at least once.

CODE

If decision making

Snippet:

Syntax

```
if (someVariable > 50)
{
    // do something here
}
```

Parameters

- Comparisons: ==, !=, >, <, <=, >=

Example

```
if (x > 120)
{
  digitalWrite(LEDpin1, HIGH);
  digitalWrite(LEDpin2, HIGH);
}
```

Function:

if, is used in conjunction with a comparison to tests whether a certain condition has been reached, such as an input being above a certain number. if the comparison is true, the statements inside the brackets are run. If not, the program skips over the code.

CODE

If ... Else decision making

Snippet:

Syntax

```
if (pinFiveInput < 500)
{
  // action A
}
else
{
  // action B
}
```

Parameters

- Comparisons: ==, !=, >, <, <=, >=

Example

- See **if** card

Function:

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.

CODE

Switch .. Case decision making

Snippet:

Syntax

```
switch (var) {
    case 1:
      //do this when var equals 1
      break;
    case 2:
      //do this when var equals 2
      break;
    default:
      //nothing else matches, do this
      //default is optional
    break;
  }
```

Parameters

- Var: the variable you are testing against

Function:

Like **if** statements, **switch...case** controls the flow of a program. 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.

CODE

Mapping to a range

Snippet:

Syntax

```
y = map(value, fromLo, fromHi, toLo, toHi);
```

Parameters

- value: number to remap
- fromLo: old lowest value
- fromHi: old highest value
- toLo: new low value
- toHi: new highest value

Example

```
val = analogRead(0);
val = map(val, 0, 1023, 0, 255);
analogWrite(9, val);
```

Function:

Re-maps a number from one range to another. That is, a value of fromLo would get mapped to toLo, a value of fromHi to toHi, values in-between to values in-between, etc.

CODE

Basic maths

Snippet:

Arithmetic Operators

The basic maths operations are as follows:

- + addition
- -Subtraction
- * multiplication
- / division
- % modulo

Examples

- val1 = val + 2
- val1 = val - 2
- val1 = val * 2
- val1 = val / 2
- val1 = val % 2 (val1 = remainder)

Function:

The maths operations follow standard maths procedures, brackets can be used to ensure the correct order of calculation (precedence)

CODE

Inc'ing & **Dec**'ing variables

Snippet:

Syntax

```
variable ++; //incs by 1
Variable - -; //decs by 1
```

Parameters

- None

Example

```
index ++; //inc index by 1
pointer - -; //dec pointer by 1
```

Function:

Often in our code we wish to add 1 or subtract 1 from a variable, this is known as **incrementing & decrementing**.

Many coding languages have a quick method for this and Arduino is no exception.

CODE

Playing musical **Tone**s

Snippet:

Syntax

```
tone(pin, frequency, duration)
```

Parameters

- pin: the pin on which to generate the tone
- frequency: the frequency of the tone in hertz
- duration: the duration of the tone in milliseconds (optional)

Returns

- None

Note

- Min frequency 31Hz, max 65535Hz

Function:

Generates a square wave of the specified frequency on a pin. A duration can be specified, otherwise the wave continues until a call to **noTone**(). The pin can be connected to a piezo buzzer or other speaker to play tones.

CODE

Choosing a **Random** number

Snippet:

Syntax

```
random(max)
random(min, max)
```

Parameters

- min: lowest number (optional)
- max: highest number

Returns

- a random number between min and max-1

Example

```
// print a random number from 0 to 299
randNumber = random(300);
Serial.println(randNumber);
```

Function:

The random function produces pseudo random numbers. For a sequence of different values each time the sketch is run. Use **randomSeed**() to initialize the random number generator with a fairly random input, such as **randomSeed**(**analogRead**(0));

CODE

Setting up the **Serial** port

Snippet:

Syntax

```
Serial.begin(baudrate);
```

Parameters

- baudrate: speed of transmission usually 9600

Return

- None

Example

```
void setup() {
    Serial.begin(9600); //set serial
port
}
```

Function:

This function sets up the serial port, which allows the Arduino to communicate with the computer, using the serial monitor which is a part of the Arduino IDE.

The serial monitor is found by clicking the rightmost icon on the toolbar, it looks like a magnifying glass.

CODE

Send data back to the computer

Snippet:

Syntax

```
Serial.println(val)
```

Parameters

- val: a variable or a piece of text in speech marks

Returns

- number of bytes sent (using the number is optional)

Example

```
// read the analog input on pin 0:
analogValue = analogRead(0);
// print it out in many formats:
Serial.println(analogValue);
```

Function:

Prints data to the serial port as ASCII text followed by a carriage return character and a newline character.

When using the serial monitor the data sent from the Arduino will appear in the serial monitor window.

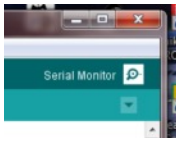
CODE

Setting up the **Serial Monitor**

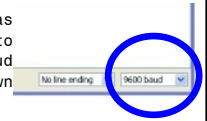
Snippet:

Setup

To set up the serial monitor in the Arduino IDE you will need to click on the rightmost icon on the toolbar:



Once the window has appeared you will need to select the matching baud rate from the drop-down box, usually 9600.



Function:

Once you have setup your serial monitor it now can send and receive data to/from your attached Arduino.

CODE

Stopping **notes** playing

Snippet:

Syntax

```
noTone(pin);
```

Parameters

- pin: the pin on which to stop generating the tone

Return

- None

Example

```
noTone(4);
```

Function:

Stops the generation of a note triggered by **tone()**. Has no effect if no tone is being generated.

In the example any tone playing on pin 4 will be stopped.

CODE

Reading **Data** from an Array

Snippet:

Syntax

```
var = name[index];
```

Parameters

- var: variable to store the value from the array into
- name: name of the required array
- index: variable used to point to the value you require, note arrays start from 0 (first item)

Example

```
Colour = ColTable[whichOne];
```

Function:

In the example, the variable **colour** is set to the value in the **ColTab** array *pointed* to by the variable **whichOne.**

By adding 1 to the **whichOne** variable, you can step through the array. *Pointers* can be used with more than one array, if you need more than one value.

CODE

Reading an **Analog** pin

Snippet:

Syntax

```
var = analogRead(pin)
```

Parameters

- var: variable to store the value into
- pin: the number of the analog input pin to read from (0 to 5 for Uno)

Return

- int (0 to 1023), (0V to 5V)

Example

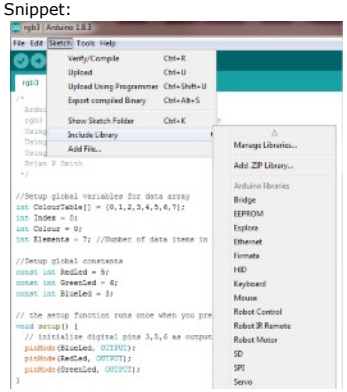
```
val = analogRead(0);
```

Function:
Reads the value on a specified analog pin. The Arduino Uno has 6 channels, 10-bit analog to digital converter. This means that it will map input voltages between 0 and 5 volts into integer values between 0 and 1023. This yields a resolution between readings of: 5 volts / 1024 units or, .0049 volts (4.9 mV) per unit.

CODE

Adding a Library to your code

Snippet:



Function:
Any of the supplied code Libraries can be added to your code by going to **Sketch>Include Library** and then clicking on the Library you need to add. The appropriate code will be inserted into your code.

Libraries are added before your **Globals.**

CODE

The **Servo** Library functions

Snippet:

Create Servo object

- Servo myservo; //myservo is the object

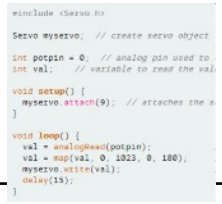
Setup Servo object

- myservo.attach(Pin); //Pin number

Set Servo position

- myservo.write(Angle); //Angle 0 to 180

Example



Function:
This library allows an Arduino board to control RC (hobby) servo motors. Standard servos allow the shaft to be positioned at various angles, usually between 0 and 180 degrees. Continuous rotation servos allow the rotation of the shaft to be set to various speeds.

CODE

Constrain a value

Snippet:

Syntax

```
var = constrain(x, a, b)
```

Parameters

- var: variable to store the value into
- X: variable to constrain
- A: lower / minimum limit
- B: upper / max limit

Return

- A value between a and b only

Example

```
input = constrain(input, min,max);
```

Function:

Constrains a number to be within a range, if the value has a value less than the lower limit, it is set to that value, similarly if it is larger than the upper limit it is set to that value.

CODE

MASKING bits in a variable

Snippet:

Syntax

```
var = var & mask
```

Parameters

- var: variable to store the value into
- mask: bit mask - usually in binary

Return

- A value ANDed (&) with the bit mask

Example

```
keys = keys & 0b00001111
//keeps bits 0 to 3, rest set to 0
```

Function:
Used to *mask out* bits ie set to 0 irrespective of current state. Used to set bits to 0 we aren't interested in or aren't required, can be one or more bits at the same time. It is best to set the *mask value in binary* so you can see exactly which bits you are masking out.

CODE

MERGING bits in a variable

Snippet:

Syntax

```
var = var | mask
```

Parameters

- var: variable to store the value into
- mask: bit mask - usually in binary

Return

- A value ORed (&|) with the bit mask

Example

```
keys = keys & 0b1000
//Sets bit 3 to 1, rest unchanged
```

Function:
Used to *merge in* bits ie set to 1 irrespective of current state. Used to set bits to 1 we are interested in, can be one or more bits at the same time. It is best to set the *merge value* in binary so you can see exactly which bits you are merging in.

CODE

SHIFTING bits left

Snippet:

Syntax

```
var = var << ntimes
```

Parameters

- var: variable to store the value into
- ntimes: number of shifts to perform

Return

- A value with bits shifted left ntimes

Example

```
Code = code << 4
//code bits shifted left 4 times
```

Function:
This operator shifts the bits in a variable left by the required number of times. Any bit in the left most position will be lost and 0 is injected into bit 0 position. It is used to move bits into a position you need or multiple by 2 for each shift.

CODE

SHIFTING bits right

Snippet:

Syntax

```
var = var >> ntimes
```

Parameters

- var: variable to store the value into
- ntimes: number of shifts to perform

Return

- A value with bits shifted right ntimes

Example

```
Code = code >> 4
//code bits shifted right 4 times
```

Function:
This operator shifts the bits in a variable right by the required number of times. Any bit in the right most position will be lost and 0 is injected into highest bit position. It is used to move bits into a position you need or divide by 2 for each shift.

CODE

Create a **FLOAT** variable

Snippet:

Syntax

```
Float name = value;
```

Parameters

- name: meaningful name for your variable
- Value: -3.4 x 10+38 to 3.4 x 10+38

Return

- none

Example

```
float myfloat;
float sensorCalbrate = 1.117;
```

Function:

The float variable is used to store floating point numbers, they are stored as 32 bits (4 bytes) of information. Floating point math is also much slower than integer math in performing calculations, so should be avoided if at all possible.

CODE

Create a simple **FUNCTION**

Snippet:

Syntax

```
Void name()
{
   Your code here
}
```

Parameters

- None

Return

- None

How to call

- Use function name();

Function:

A function is a small block of code, which can be used many times, it also has it's own local variables.

This form of function has no parameters or return value.

CODE

A **FUNCTION** with parameters

Snippet:

Syntax

```
Void name(int variable)
{
   Your code here which can use variable
}
```

Parameters

- Variable name

Return

- None

How to call

- Use function name(value);

Function:

Works as the simple function, but the value in the brackets is passed to the local variable named in the definition. It can be a value or a variable name it's current value is passed into the function.

It can be used with parameters to create flexible functions.

CODE

A **FUNCTION** with return value

Snippet:

Syntax

```
int name()
{
   Your code here to assign a value to f
   return f;
}
```

Parameters

- Prefix function with variable type

Return

- F is the return parameter

How to call

- Use function variable = name();

Function:

Works like the other the other functions except it returns a value, which is calculated etc in the function. The name of the returned value must be a local variable within the function.

It can be used with parameters to create flexible functions.

CODE

Setting a **PORT DDR** register

Snippet:

Syntax

```
DDRy = BXXXXXXXX;
```

Parameters

- Y = B or D
- X = 0 or 1

Return

- none

Example

```
DDRD = B11110000;  //Bits 0-3 = inputs,
                   4-7 = outputs
```

Note: 0 = input, 1 = output

Function:

The Arduino normally uses individual bits, but you can use them as two multi-bit ports if needed. Port D uses bits 0 to 7, and Port B uses bits 8 to 13. To use them you must set them up using DDR instructions, to ensure they are set to input or output. They can be any combination of in's and out's.

CODE

Writing to a **PORT**

Snippet:

Syntax

```
PORTy = BXXXXXXXX;
```

Parameters

- Y = B or D
- X = 0 or 1

Return

- none

Example

```
PORTD = B11110000; //also 240 dec
PORTB = 15;  //B00001111
```

Note: decimal, hex or binary numbers can be used.

Function:

This will turn on/off the pins controlled by the port. It is very fast and can control all the port pins at the same time.

Individual bits can be altered by using masking & merging or the **BIT** commands.

CODE

Reading from a **PORT**

Snippet:

Syntax

```
Var = PINy;
```

Parameters

- Var = name of byte variable
- Y = B or D

Return

- Value of the port as a byte

Example

```
Buttons = PINB
PORTB = 15;  //B00001111
```

Note: Port D is 8 bits, port B is 6 bits

Function:

This command will read all the pins of a port and return a byte number. Remember that port B is 6 bits, so the value is between 0 and 63. Port D is 0 to 255.

Individual bits can be read by using the **BIT** commands.

CODE

Measure a **pulseIn** time

Snippet:

Syntax

```
Var = pulseIn(pin, option);
```

Parameters

- Var = name of long variable
- pin = pin number
- Option = HIGH or LOW

Return

- Value of time at HIGH or LOW

Example

```
Duration = pulseIn(3, HIGH);
```

Function:

The command measures the time at a particular logic level and returns a value in ms.

If the option HIGH is used, then timing starts when the level on the pin changes from low to high and stops timing when it changes back to low. LOW operates in the opposite sense.

CODE

Pin based **INTERRUPTS**

Snippet:

Syntax

```
Attachinterrupt(digitalPintoInterrupt(pin), ISR, mode);
```

Parameters

- pin = pin number 2 or 3 only
- ISR = name of function to call
- Mode = LOW, CHANGE, RISING, FALLING

Return

- None

Example

```
Attachinterrupt(digitalPintoInterrupt(pin), ISR, HIGH);
```

Function:

This command is used to setup an interrupt using the state of a pin. When the pin changes state as defined by the mode, the main code is stopped and the ISR function is called, once that function is completed the main code is resumed. This is very useful for safety critical operations.

CODE

Turn off pin **INTERRUPTS**

Snippet:

Syntax

```
detachInterrupt(digitalPintoInterrupt(pin));
```

Parameters

- Pin = pin used for interrupt (2 or 3)

Return

- None

Example

```
detachInterrupt(digitalPintoInterrupt(2));
```

Function:

This command is used to turn off a pin based interrupt, once done more interrupt calls will be made if the pins value changes.

CODE

Timer based **INTERRUPTS**

Snippet:

Syntax

```
Timer1.initialize(time);
Timer1.attachInterrupt(ISR);
```

Parameters

- Time = repeat every time in ms
- ISR = name of function to call

Return

- None

Example

```
Timer1.initialize(500);
Timer1.attachInterrupt(toggleOutput);
```

Function:

This command is used to setup an interrupt using a timer. To use this type of interrupt a library called **TimerOne** must be used.

In the example an interrupt occurs every 500ms or 2Hz and calls the toggleOutput function.

CODE

Turn off timer **INTERRUPTS**

Snippet:

Syntax

```
detachInterrupt();
```

Parameters

- None

Return

- None

Example

```
detachInterrupt();
```

Note: This method only works with the TimerOne library.

Function:

This command is used to turn off a timer based interrupt, created by the TimerOne library, once done more interrupt calls will be made when the timer times out.

CODE

Write to **EEPROM** memory

Snippet:

Syntax

```
EEPROM.write(addr, val);
```

Parameters

- Addr = address 0 to 1023
- Val = byte value 0 to 255

Return

- none

Example

```
EEPROM.write(0, 255);
```

Note: your code must include the EEPROM library, which is installed as standard

Function:

This command writes a byte value to the address in the command. The Arduino Uno has 1024 bytes of EEPROM, addressed from 0 to 1023.

The example write 255 to address 0.

CODE

Read from **EEPROM** memory

Snippet:

Syntax

```
Var = EEPROM.read(addr);
```

Parameters

- Addr = address 0 to 1023
- Var = byte variable

Return

- none

Example

```
Val = EEPROM.read(0);
```

Note: your code must include the EEPROM library, which is installed as standard

Function:

This command reads a byte value from the address in the command and stores in a variable. The Arduino Uno has 1024 bytes of EEPROM, addressed from 0 to 1023.

The example reads a byte value from address 0.