Introduction to Arduino Programming

Binghuan W Li
binghuan.li19@imperial.ac.uk

Dr Christopher Rowlands
c.rowlands@imperial.ac.uk

March 23, 2022

1 Introduction

Today, we will delve a bit into Arduino programming. Originating from Italy, Arduino is an open-source[1] electronics prototyping platform based on flexible, easy-to-use hardware and software. It builds on the idea of *using less expensive devices for controlling interactive electronic projects*, supporting rapid prototyping of microcontroller hardware by electronic designers and students. Typically, an Arduino (Nano, Uno) consists a +3.3V power rail, a +5V power rail and two ground(GND) terminals, which can be used to power up most small pieces of external hardware (say, sensors or motors). It also has multiple analogue(A) and digital (D) input/output terminals, which can efficiently control the I/O of your device using signals that you can program youself.

A detailed pinout of an Arduino Uno is shown below, but more details can be found in the datasheet, available at here.

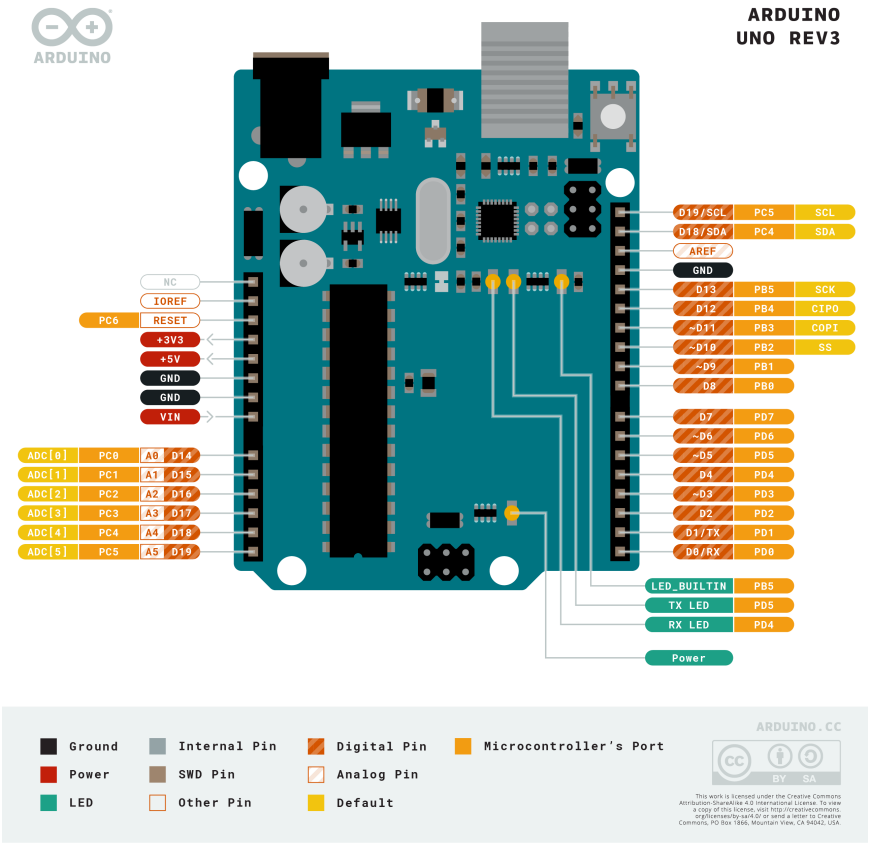

Figure 1: Arduino Uno pinouts. Adopted from https://content.arduino.cc/assets/Pinout-UNOrev3_latest.png

[1]The schematics of every board are available to view. You are free to make you own Arduino using standalone components - but you still need to buy them!

2 From `Blink` to Arduino Programming Syntax

2.1 Arduino IDE

So, how can you communicate with your Arduino board? Luckily, a 'ferry boat', the Arduino IDE, is already there for you, to help you bridge the gap between your computer and the board. Arduino IDE is an integrated development environment introduced by Arduino official, multiple versions are available to download for both PC and Mac users. Of course, you are required to write your program in a syntax that is specially designed for Arduino - your Arduino code, or **sketches**, are written in a variant of C++ called the Arduino Programming Language.

Once you have installed Arduino IDE successfully on your PC/Mac, the following graphical user interface should be available to you.

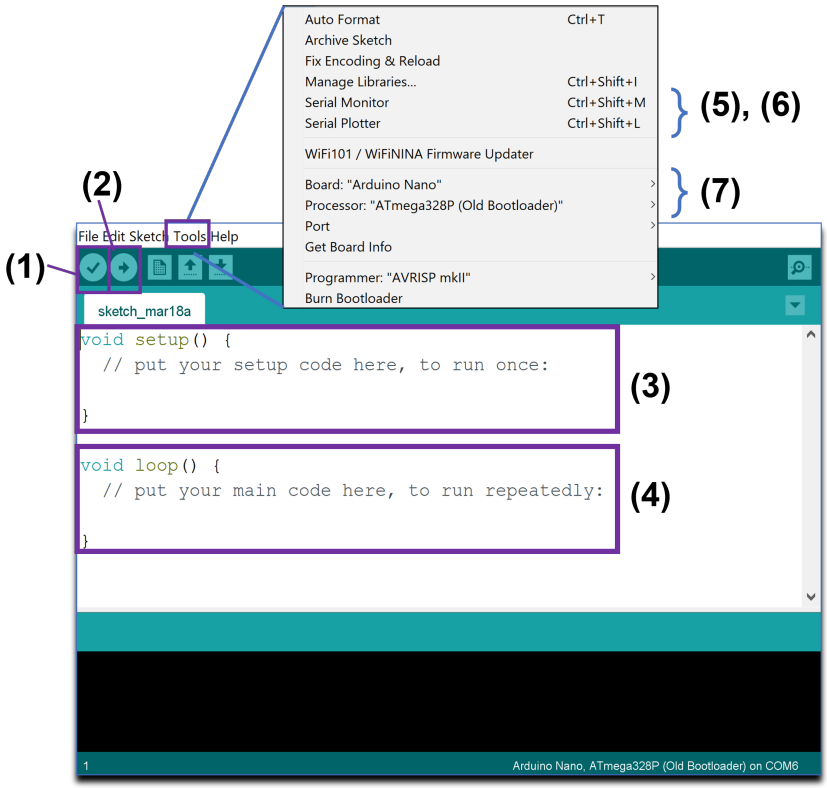

Figure 2: Arduino IDE user interface

1. **Verify**: by clicking this button, Arduino IDE checks the code for errors, then compiles it, ready for uploading to the Arduino. Click this first.
2. **Upload**: when you click this button, Arduino IDE uploads the code to the connected Arduino board.
3. `void setup`: in an Arduino program, statements within this function will only be executed once. Therefore, this function block is commonly used to set up system parameters, etc.
4. `void loop`: in an Arduino program, statements within this function will be executed periodically, similar to if the statements were placed in a `while True` loop.
5. **Serial monitor**: similar to the console in Python IDLE's IDE, the serial monitor is the 'tether' between the computer and your Arduino. It allows you send and receive text messages, handy for debugging and also controlling the Arduino from a keyboard. Unlike the console however, your program needs to be programmed to know how to respond to your text messages!
6. **Serial plotter**: similar to the oscilloscopes you've used in the electronics labs, the serial plotter is commonly used to display the data read from the Arduino board. It receives the data (e.g. temperature, humidity) from the hardware sensors and plots the data as one or more waveforms.

7. **Board/processor/port selection**: these options are used to select board model, microcontrollers (by default, ATMega328P) and communication ports in your Arduino IDE. Make sure they match the board you connected to your computer. **Check these settings every time you connect your Arduino board to your computer.**

2.2 Blink!

Similar to `print("Hello, world!")` in Python and other programming languages, `Blink` is a very basic Arduino sketch for testing purposes, and to get new learners quickly familiarized with Arduino programming.

This program blinks the on-board light-emitting diode (LED), as well as the LED that is externally connected to digital output pin 13 (if there is one!), with a fixed period of one blink every 2 seconds.

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  Most Arduinos have an on-board LED you can control. On the Uno and
  Leonardo, it is attached to digital pin 13. If you're unsure what
  pin the on-board LED is connected to on your Arduino model, check
  the documentation at http://arduino.cc

  This example code is in the public domain.

  modified 8 May 2014
  by Scott Fitzgerald
 */


// the setup function runs once when you press reset or power the board
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
}
```

Listing 1: Blink.ino

The corresponding wiring schematic is shown below. The resistor, $R = 220\Omega$.

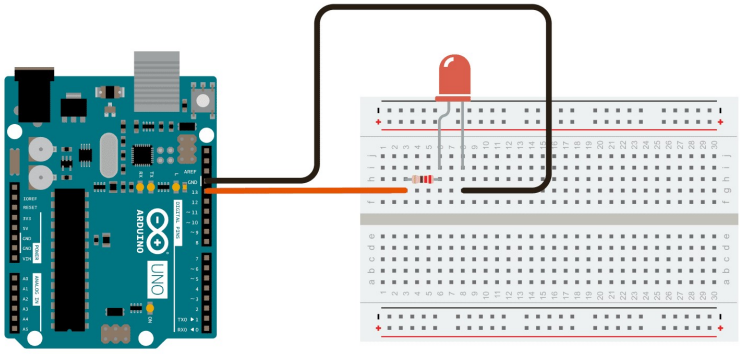

Figure 3: Schematic for Blink

A detailed code breakdown:

1. `void setup()` - `setup` is the name of the function which runs once, whenever the board is turned

on (or reset). `void` means it returns no varaiables, and the empty brackets mean that it takes no arguments. Statements within this function will only be executed once.

In C++/C-like programming languages, when returning a value from a function, you are required to tell your compiler what data type that value is (such as `int`, `char`, `float`, or any other data type). In this case, the return type is `void`, which just tells the compiler that there is no return value.

2. `pinMode(13, OUTPUT)` - this command initializes pin 13 on your Arduino board as an output pin.

   The general format of this command is `pinMode(pin, mode)`; it takes two arguments, which specify the `mode` (`OUTPUT` or `INPUT`) of a specific pin.

3. `void loop()` - similar to `void setup()`, the function `loop` does not return any value. Statements within this function will be executed periodically while the board is active.

4. `digitalWrite(13, HIGH)` - this command controls your Arduino board, telling it to turn the LED on by supplying a +5V voltage (V_{cc}) to pin 13 (and thus to the anode of your LED). The same is true for `digitalWrite(13, LOW)`, which will turn off the LED, taking the voltage supply back to 0V.

5. `delay(1000)` - this command tells the board to do nothing for 1000 milliseconds, or 1 second. It is similar to the `time.sleep` function in Python.

Other general comments on coding syntax:

6. **Semicolons** - In Arduino sketches or C++/C-like programming languages, a semicolon **;** is required to end your statements. Usage of Semicolons in C will remove ambiguity and confusion while looking at the code. However, this is optional in Python.

7. **Comments** - In Arduino sketches or C++/C-like programming languages, **multiline comments** are inserted between **/*...*/** while **single line comments** are inserted after double backslashes **//**. Comments will be ignored by the compiler.

2.3 Variables and Arithmetic Expressions

In this section, we are going to see the different data types and operators that you may find useful in your sketches. The following syntax can also be applied to other C-like languages, including C and C++.

2.3.1 Variable declaration

In all C-like programming languages, variables must be **declared** before they are used. A declaration consists of a *type name* and *variable(s)*. You can assign a value to the variable right after you declare it. For example:

```
int my_variable_1 = 25; \\ declare an integer type variable, assign a value to it.
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'}; \\ strings are terminate by \0.
```

Primary data types include `int` (integer, 16 bits), `float`(floating point, 32 bits), `char`(character, 8 bits), `double`(double precision floating point) etc.

Moreover, **modifiers** - `signed`, `unsigned`, `short`, and `long` are used to modify default properties of primary data types. For example, by applying `unsigned` modifier to the `int` type, the variable can only store values greater than or equal to zero.

A clear and neat summary of C data types in a table can be found here. A 'friendly' explanation to C data types and modifiers can be found here.

2.3.2 Arithmetic expressions

A large set of operators are available in the Arduino Programming Language. Here are a few quick cheat sheets you can use in your Arduino programming.

Operator	Meaning	operator	Meaning
+	addition	-	subtraction
*	multiplication	/	division
%	reminder after division		

Table 1: Arithmetic operators

Operator	Meaning	operator	Meaning
=	a=b	+=	a+=b equivalent to a=a+b
-=	a-=b equivalent to a=a-b	*=	a*=b equivalent to a=a*b
/=	a/=b equivalent to a=a/b	%=	a%=b equivalent to a=a%b

Table 2: Assignment operators

Operator	Meaning	operator	Meaning
==	equal to, *e.g.* 5==3⇒0	>	greater than
<	smaller than	!=	not equal to
>=	greater than or equal to	<=	less than or equal to

Table 3: Relational operators

Operator	Meaning
&&	Logical AND. True only if all operands are true
||	Logical OR. True only if either one operand is true
!	Logical NOT. True only if the operand is 0

Table 4: Logical operators

2.4 Control flow

In this section, we will look at the syntax of common control flow constructs - `for` loops, `if...else...` statements and `while` loops.

While we are only covering these three control flow statements here, note that there are more useful statements supported by C-like languages: `do...while` loops, `switch` case, `break` and `continue` statements.

2.4.1 `for` loop

`for` loops in Arduino sketches are following the structure below - three arguments are given in the order of initial condition-final condition-action, as shown in the parenthesis following the keyword `for`.

A quick example is `for(int i=0, i<=5, i++)`.

```
for (initial_condition, final_condition, action){
    statement_1;
    statement_2;
    ...
}
```

Listing 2: for loop

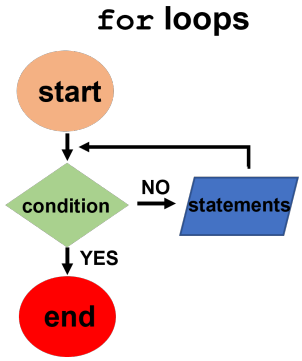

Figure 4: `for` loop

Note the differences from Python's `for` loop syntax : There is no syntactic whitespace. C-like languages place the code within the loop inside of two curly brackets . It is however very good practice to still indent the statements in the loop, so you can read the code more easily. Don't forget you still need a semicolon at the end of each statement in the loop!

2.4.2 `if...else...` statements

`if...else-if...else...` statements in Arduino sketches share a very similar syntax to Python, except `elif` in Python is written as `else if` in an Arduino sketch.

```
if (condition_1){
    statement_block_1;
}
else if (condition_2){
    statement_block_2;
}
else{
    statement_block_3;
}
```

Listing 3: if else statements

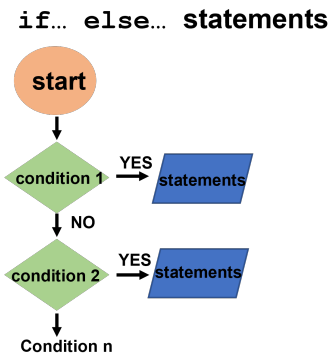

Figure 5: `if... else...` statements

2.4.3 `while` loop

`while` loops in Arduino sketches also share a similar syntax to those in Python. In a `while` loop, a condition check is given in the parentheses, statements are expressed between the curly brackets. The while loop will execute so long as the condition check evaluates to `true`; the check will be performed each time the loop repeats, so make sure it will evaluate to `false` at some point or your program will get stuck!

```
while (true_condition){
    statement_1;
    statement_2;
    ...
    statement_increment_to_true_condition;
}
```

Listing 4: while loop

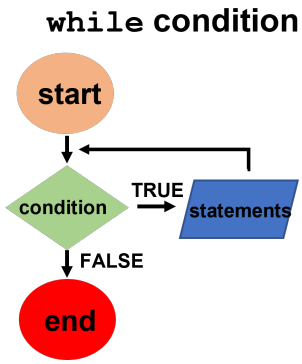

Figure 6: `while` loop

2.5 Functions

Like Python, Arduino Programming Language (and, in fact, any C-like language) allows you to define functions. A function definition in an Arduino sketch follows the structure below:

```
return_type function_name (arg1, arg2, ..., argN) {
    statement_1;
    statement_2;
    ...
    statement_N;
    return value;
}
```

All functions must have a function return type that is specified prior to the function name. This type can be `int`, `float`, `char*` [2] or simply `void` (return nothing).

The function name and arguments come after the return type, and at this point there is fundamentally nothing different from Python's function definition. As before however, there is no syntactic white space; all statements should be placed in between the curly brackets. Although indentation is not compulsory in C-like languages, they improve your code readability - you should definitely use it.

A `return` statement is used to return values/variables from the function. It is optional. Sometimes, even there is no need to return any variables from the function, people use `return 0` to indicate that the function executed correctly, and use non-zero returns (e.g, `return -1`) to indicate errors.

[2] The asterisk declares a pointer. The pointer stores the starting memory location of the variable it is next to, in this case a string.

3 Serial communication at a glance

Well, after going through some basic syntax which will be useful in Arduino sketches (ah, finally!), you are now a shining expert in Arduino coding! Wait... you may ask, are there any specific features for Arduino *only*, but not for other C-like languages? The answer is "yes", and one of the most important features is **serial communication**.

Serial communication is used for data transfer between the Arduino board and your device (let us say, your PC), by sending bite-sized data sequentially over a **communication port** (COM). This is how your compiled sketch can be uploaded and executed on the board, and is also how the data collected from an external device (let us say, a temperature sensor), can be sent back to your PC.

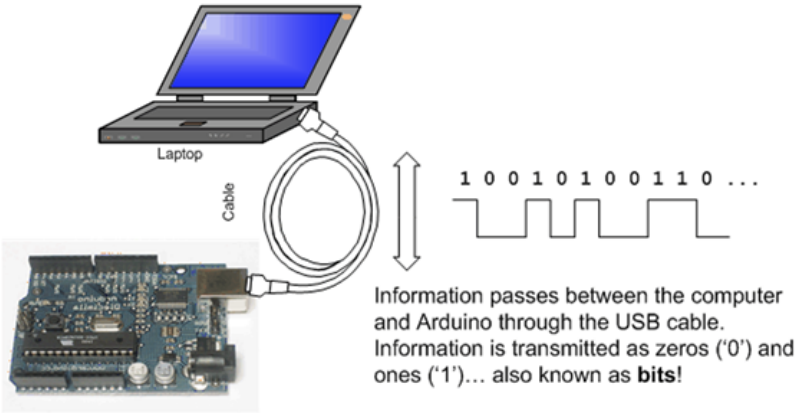

Figure 7: Serial communication. Adopted from https://www.ladyada.net/learn/arduino/lesson4.html

Look carefully to your Arduino board, there are two helpful built-in LEDs, labeled with RX and TX, that will blink when the Arduino is receiving data (RX) as well as transmitting data (TX). You can use them whenever you are communicating with the board, to check that everything is working properly.

Now, we are going to look at five useful functions which are related to serial communication. However, this is but a small selection of the functions available; more can be found here.

3.1 `Serial.begin()`

The first function related to the serial communication is `Serial.begin(`*`speed`*`)`. This function takes one compulsory argument, `speed`, which sets the data rate (baud, or bits per second) for serial data transmission.

By convention, we take the data transmission rate to 9600 baud (i.e. `Serial.begin(9600)`), as this is the default rate for the Arduino. You should be aware that your Arduino (and other devices) can communicate at different speeds, so make sure you check the manual for whatever you are trying to communicate with.

DO include this command in your `void setup()` function, as most likely you do not want this command to be executed periodically!

3.2 `Serial.print()`

Similar to Python, the `print` function is particularly useful to print data to a particular output device. Unlike in your Python code however, we don't have a console to print to, so instead we print to the serial port. This function takes 1 compulsory argument. For example, `Serial.print('Hello, world!');` gives 'Hello, world!'.

The second argument is optional, it is used to specify the base of the value to print. For example, `Serial.print(78, BIN);` gives '1001110', as it converts a decimal number into the binary form.

3.3 `Serial.println()`

This function is *almost* identical to `Serial.print()`, but every time we call this function, it will start with printing with a new line.

How does your program achieve this function...? Basically, an **escape sequence**, `\n`, will be appended to your string, which tells your program to start a new line. You may remember this from the ASCII and Unicode slides in the first lecture of the course.

3.4 `Serial.read()`

The function `Serial.read()` is used for reading incoming serial data. It does not take any arguments, but returns the first byte of incoming serial data with an `int` data type. You should notice that this helps explain why in C (and even in Python), variables have a type. The compiler or interpreter knows how to interpret the sequence of 1s and 0s so that they represent what the programmer wants them to represent (a number, or a string, or a boolean for example). Here, we don't know what the data type returned from the serial port is, so we just guess that it is an `int`. We might be wrong!

Usually, the use of `Serial.read()` is accompanied with another function `Serial.available()`, which returns how many bytes of data have arrived in the serial buffer, and therefore are ready to be read. Make sure you don't wait too long before reading from the buffer, as it can overflow and you will lose data!

```
1 void loop() {
      if (Serial.available() > 0) {
3         int incomingByte = Serial.read();
      }
5 }
```

3.5 `Serial.write()`

The final function we are going to analyse is `Serial.write()`. It writes binary data to the serial port. If the argument is a value, the data will be sent as a byte; if the argument is a string, the it will be sent as a series of bytes.

4 Analog? Digital?

In subsection 2.2, we had a brief exposure to Arduino digital pins as well as the digital output function `digitalWrite()`. In this section, we will delve a bit deeper into the digital/analog output/input functions on the Arduino boards. First, to use the functions below, remember to initialize `pinMode()` in `void setup()`!

4.1 Analog signals - `analogRead` and `analogWrite`

Arduinos can both input (read) and output (write) analog signals. To *read* an input analog signal, the function `analogRead(pin)` is useful; it tells the Arduino's built-in analog-to-digital converter (ADC) to convert an analog signal into a digital value and return an integer between 0 and 1023. This digital value is is proportional to that of a reference voltage, either 5V or 3.3V. You should look up the specifications for the board you are using to know what the reference voltage is.

To *write* an analog signal, although no digital-to-analog converters (DACs) are pre-built into Arduino Uno or Arduino Nano[3], Arduinos can **pulse-width modulate** (PWM) a digital signal to achieve an analog output, through the function `analogWrite(pin, value)`. This function takes two arguments - the first argument specifies the output pin, the second argument is an integer value between 0 to 255 that is proportional to the **duty cycle** of the signal.

To illustrate how PWM works in an Arduino board, refer to Figure 8. PWM works by very rapidly switching between two states (on and off), spending more time in one state than the other. The signal is then low-pass filtered to get an 'average' value, which will be somewhere between V_{on} and V_{off}. When value = 0, the signal is always OFF; when value = 255, the signal is always ON. By altering the value, a digital signal can be used to 'imitate' an analog signal. To determine the value based on your desired voltage, the following equation is used:

$$value = 51 \times u$$

where u is your desired analog voltage.

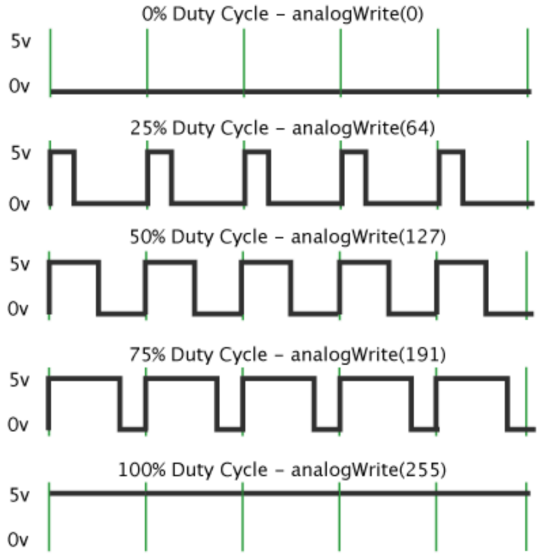

Figure 8: Pulse-width modulation. Adopted from https://www.arduino.cc/en/Tutorial/Foundations/PWM

Note: NOT all pins support PWM! On most Arduino boards, the PWM function is available on pins 3, 5, 6, 9, 10, and 11. The frequency of the PWM signal on most pins is approximately 490 Hz. On the Uno and similar boards, pins 5 and 6 have a frequency of approximately 980 Hz.

[3] If you need a DAC you can either use a separate DAC chip, or the Arduino Due has two built in.

4.2 Digital signals

Analogous to `analogRead(`*pin*`)` and `analogWrite(`*pin*`, `*value*`)`, there are equivalent functions for digital signals, `digitalRead(`*pin*`)` and `digitalWrite(`*pin*`, `*value*`)`. These perform input and output for digital signals, respectively. There is a minor difference however - the second argument of the function `digitalWrite(`*pin*`, `*value*`)` can only take two expressions: ‘HIGH’ or ‘LOW’.

Appendix A From sketches to actions - compilation

When you click the 'upload' button in your Arduino IDE, an sequence of important events have to happen to pass your sketch to your Arduino board. These events are complicated, but to break them down into a manageable outline, below is a very brief description. Nevertheless, a full, comprehensive explanation can be found here.

1. **Pre-processing**: during this stage, your `.ino` sketches will be processed into a C++ program (`.cpp`). This step is unique in Arduino Programming Language.
2. **Compilation**: at this stage, the C++ program that is obtained from the last step will be compiled into machine-readable instructions by the avr-gcc compiler.

   To take a closer look the compilation process: first, an object file (`.o`) is generated. Your compiler will link[4] this object file to the standard Arduino libraries. The standard Arduino libraries enable us to use those Arduino built-in functions, such as `Serial.begin()` and `digitalWrite()`.

   Meanwhile, your compiler will try to locate the dependencies (say, external libraries, `.h` files) if you have added any `#include` statements in your sketch. This process is known as **separate compilation**.
3. **Uploading**: the final output of the compilation is a `.hex` file. The `.hex` file is uploaded to the board by a uploader/downloader utility, Avrdude, over the USB or serial connection.

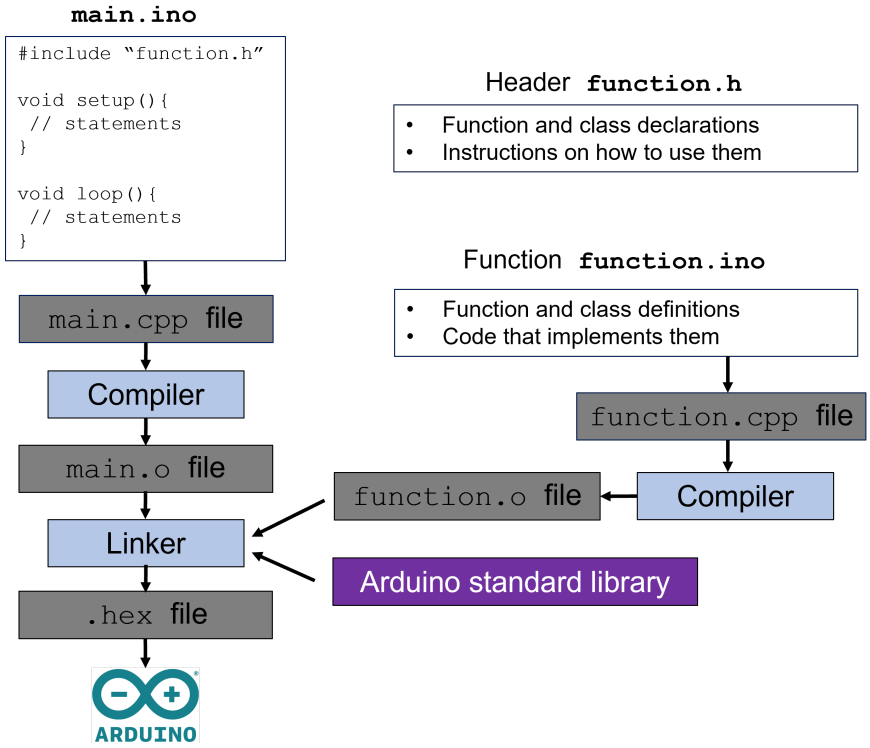

Figure 9: Arduino pre-processing, compilation and uploading flowchart

C/C++ language shares a very similar compilation process to the one shown above!

[4] Linking is when the 'linker' finds all the extra code from the libraries and functions you call in your code, and puts the actual code for those functions into your program

Appendix B Pointers

"Wait what...? Pointers? Point to where?", you may ask. The best answer to this question (ever!) is the formal definition: **A pointer is a variable that contains the *address* of a variable**. In other words, instead of storing a number or a character, a pointer stores a memory address of another variable.

Wierd? It should be - this concept isn't found in Python, so you haven't had to work with it before. Let us begin with a simple picture of how memory of your program is organised. A typical machine has an array of consecutively numbered or addressed memory cells. Let us say,

- one cell can hold the memory of one byte - that is, the size of a `char` variable;
- a pair of one-byte cells can be treated as a `short int` variable;
- two pairs of adjacent one-byte cells represent a `long int` type variable.

A pointer is a group(two or four) of memory cells that can hold an address.

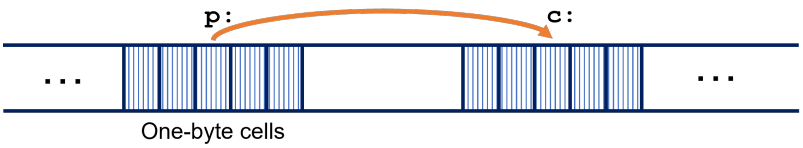

Figure 10: Memory location and pointer

If `c` is a `char` type variable, `p` is a pointer that points to the memory location of `c`, the situation can be presented in Figure 10.

In C and C-like languages, two specific operators, & and *, are used to **reference** and **dereference** a pointer.

1. The operator & gives the address of an object.

```
p = &c;
```

   which assigns the address of `c` to the variable `p`, say, "`p` points to `c`".

2. The operator * dereferences the address. When applied to a pointer, it accesses the object the pointer points to.

```
int *ptr;           // declare a pointer
int a = 1;          // declare a int variable and assign 1 to it
ptr = &a;           // assign the address of 'a' to 'ptr'
printf("%d", *ptr); // deference the pointer, access to 'a'
```

   The last command `printf("%d", *ptr)` prints out the value of the variable `a` by derefencing the pointer `ptr`. Note that, `%d` in the function `printf` is a format specifier in C language, it just means the data printed is an integer.

Why use pointers? Firstly, pointers allow functions to change the specific variables sent to them - not copies of the values, but the original memory locations of the variables themselves. This is useful when you are working on a large dataset, for example; you may not have enough memory to make a copy every time you pass the data to a function, but with a pointer, you can work on the data itself, without needing to copy it. Secondly, pointers are used to implement arrays. Especially when working with dynamic memory allocation. Thirdly, pointers make code faster - copying large datasets every time you call a function takes time. And finally, pointers are used in object-oriented programming.

Why not use pointers? Pointers allow you to directly access memory addresses, which makes them very **dangerous**. If you do not use your pointers correctly you can access garbage data, overwrite important parts of your data or your program itself, or you can leave the pointers dangling. Another product of incorrect usage may be memory leaks.

Appendix C (A warm-hearted) Checklist

This is a general checklist generated by Mr Paschal Egan, NOT specifically related to the questions that YOU have encountered. Adapted from *Appendix D, Arduino First-year Task Sheet* (2021).

1. Have you set in tools correct Arduino and com port (highest) – a particular problem with shared computers it is saves last user's set up.
2. If the Arduino powered? – Sure? If have a known good/ spare always a good idea to swap.
3. Is the device being recognized as a USB device? (Look at device manager plugging in and out)
4. Try another USB port – on some laptops the left-hand side USB ports are USB2 and the right are USB3 and the driver is only installed for one or other.
5. If you have a Nano which needs the CH340 driver – is it installed? Google how to install the driver if needed – the original nano and the UNO use the FTDI chip whose driver is part of windows.
6. Regression is a powerful idea. If fails to compile go back to earlier version (even as far as back to "Blink" – if that does not work what has changed since it last did?)
7. Double check selected correct Arduino version, and correct COM port (generally highest one)
8. Hardware swap out. Try another PC, another cable, another Arduino. Ideally have a fully working parallel system and bit by bit morph one into the other.
9. If going on a forum for help, give as much details as possible: version of the IDE, version of the hardware, what operating system used, where the problem is (compiles / but does not load? / device not detected? / does not compile? / unexpected behaviour? etc.)

Appendix D Interesting links, fantastic ideas

1. 20 Awesome Arduino UNO Projects:
   https://www.seeedstudio.com/blog/2020/01/16/20-awesome-arduino-projects-that-you-must-try-2020/
   (I love the idea of auto plant watering system so much!)
2. OKdo Blog - a regularly-updated web for your Arduino project inspirations: https://www.okdo.com/okdo-blog/
3. Build your own Arduino from scratch using ATMega328P:
   https://www.instructables.com/Make-Your-Own-Arduino-ArduinoISP-Learn-to-Burn-Boo/