Programming the Arduino Uno

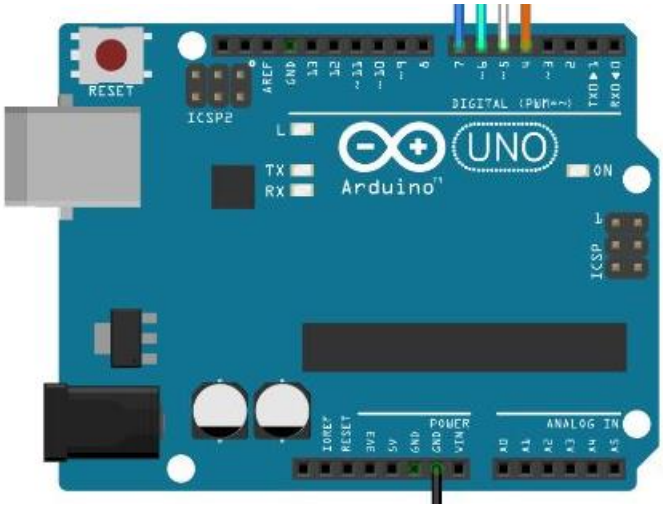
Introduction

Why bother with an Arduino Uno when there are ready made products to control our model railways? Well, the challenge of doing it for yourself, the cost, (many components are available very cheaply), but most important of all the ability to identify and build a control system for my railway were, for me, the three main reasons. I was able to buy a complete starter kit including the Arduino Uno for about £18. The Arduino Uno, (I bought one recently for £3.99), is an open source electronics platform. The program to control it is a free download and there is a great deal of support on the Arduino web site, (www.arduino.cc) and on YouTube. Also, additional support was available from a group called the Micro Electronics Railway Group (MERG), of which I am a member.

I started with simple operations; such as switching on a light, creating a traffic light system and switching on and off an electric motor to gain an understanding of the programming structure. After that it became a bit of a 'dark art' as far as the electronics and programming were concerned. I had no idea really of how a stepper motor differed from a servo or an ordinary electric motor and, how it might be controlled from the Arduino board. So here is a record my journey.

The Arduino Uno R3 board

Let's start with the board itself. It's a small collection of input and output sockets of different sorts, voltage supplies, and a storage area for the program that you have to write to in order to make use of a variety of peripherals. It has a USB socket to connect it to a computer and an external power supply socket which allows it to be run independently of a computer. From the diagram shown below you will see the groups of sockets it has. a group called Power, a group called Analog IN and a large Digital output group of different kinds.



The Arduino Software

As previously mentioned, working with the Arduino board requires you to obtain a free downloadable program which is straight forward to use but it's necessary for you to know a little of the programming language known as 'C'. The Arduino program provides a simple window into which you type various bits of code that can then be tested for mistakes and once corrected can be uploaded to the Arduino. The programming process works as follows:

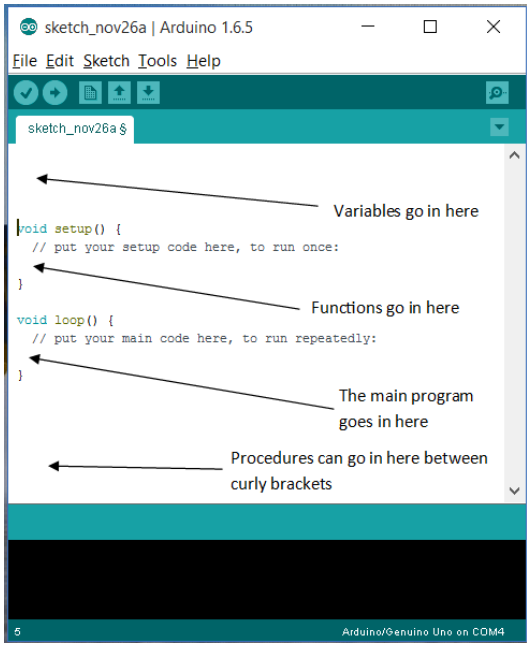
First you have to name and give a value to any constants that you are going to use.

e.g. int button_1 = 2 (the **int** part identifies it as a number or **int**eger)

Next you set up the necessary functions that use those variables from the previous section and declare which of them is an input or output.

e.g. pinMode(button_1, INPUT); (the semi colon at the end is important)

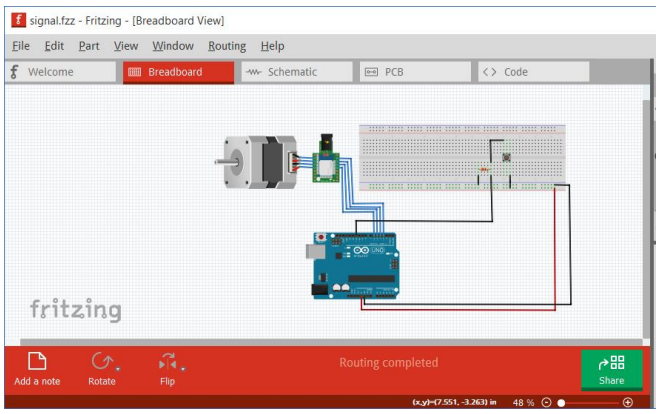
Finally you build up a program in a loop that has a defined structure to control the Arduino board. It calls on procedures to control lights, motors, servos, steppers etc. in specific ways.



The Arduino program screen comes with certain areas already identified for you like the illustration shown here. At the beginning you would normally identify and put any constants you will be using in the main programming area. The setup area, called 'void setup()' is where the functions are inserted.

The program area is called 'void loop()' and is where the organising structure of the program goes. This is the structure of the programming language you will have to use at its simplest.

Next, a quick look around the program window. The icon bar at the top of the window has 6 icons on it. The first one tests your program and allows you to verify that all is correct. If it finds a mistake, a message appears at the bottom of the screen to let you know what it thinks is incorrect. Usually this is the result of a typo or missing off a bracket or semi colon at the end of a line. The second icon compiles and uploads the code to the Arduino board. The third icon opens up a new window where you can write a fresh program. The fourth opens an old file, the fifth saves the current file you are working on. The last icon on the extreme right hand side of the bar, (the magnifying glass), is used to further test your code by opening a small separate window that can read feedback from your code, (this has to be setup with specific commands).



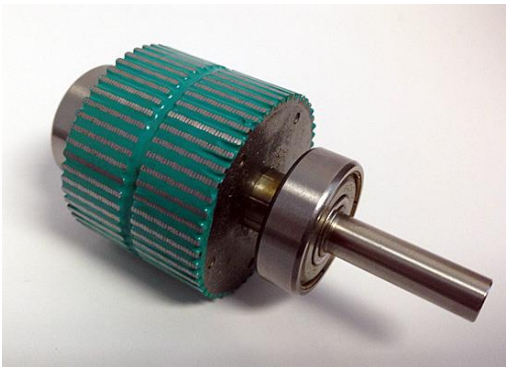
A further piece of free software that may also be useful is known as Fritzing, (http://fritzing.org/home/). It is a graphics program that allows you to build electronic circuits around the Arduino and other boards. If only used to copy a breadboard layout it is a useful program as shown here. It contains libraries of icons used in building electric circuits.

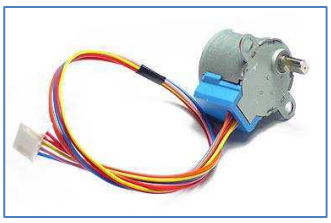
DC motors, servos and stepper motors

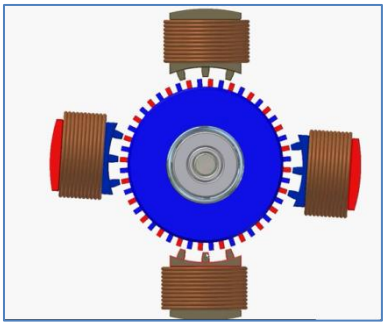
The purpose of all this research and purchase of components was to define a system that would motorise the wagon tippler on my railway. I was not sure if an ordinary DC motor with some sort of gearing would suffice, or other options using a stepper or a servo would be a better. Servos are really only designed to rotate a small amount and my tippler required much more that 180˚ turn. The DC motor with gear box didn't have enough strength to lift the tippler and the gearbox took up a lot of room that was difficult to mask in a small building. That left the stepper motor. In order to inform my decision to use one, I looked at the stepper motor itself to see how it functioned as opposed to an ordinary DC electric motor.

This illustration shows the main component of a DC motor. The armature has coils on it and it rotates within a permanent magnetic field produced by a couple of magnets surrounding it. The commutator on the armature is designed such that each electromagnet changes polarity as it rotates within the magnetic field. Thus it is in a constant state of change that results in the armature rotating.

Unlike DC motors, a stepper motor does not continuously rotate! Instead, it moves in minute steps which can be grouped together to provide precise rotation. A stepper motor has a permanent magnet rotor such as the one shown below which is attracted to electromagnets that reside in the surrounding part of the motor, this is the opposite to the DC motor. The rotor 'teeth' on the front of the rotor are offset from those on the back. Between the two rotor halves is a strong permanent magnet that ensures all the teeth on the front of the rotor are magnetic North while those on the back are South poles. This combination used with the electromagnetic surround produces the tiny pulses that move the position of the rotor slightly and hence cause it to rotate. I used the stepper motor that came with the Arduino kit, (a unipolar stepper). This stepper has 5 cable connections, (2 paired and one common), and is arranged as shown here, (to buy one costs about £3 including its control board). The arrangement of the cable connections is important as this will be used in the program that drives the stepper motor. It is therefore necessary to know how they are attached to the electromagnets inside the stepper motor housing.







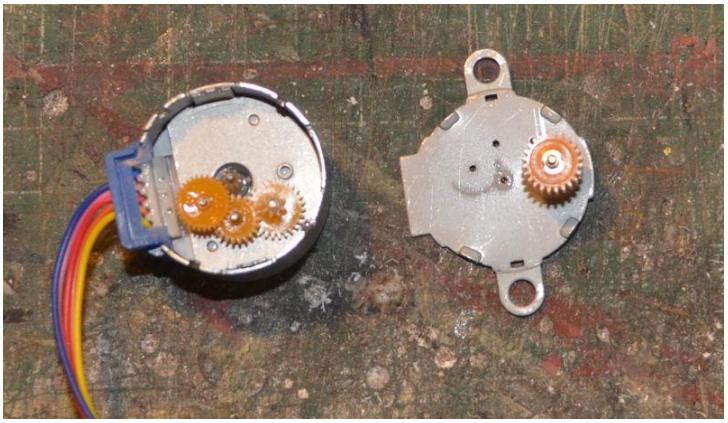
The coils can be charged individually or in pairs. In this diagram you will see that the coils are slightly offset so that

switching from the vertical coils to the horizontal coils will produce continuous adjustment, (or pulses that cause movement). These will be very small amounts of movement, hence they are referred to as pulses, but they can be measured accurately. Also by having many more pairs of coils and a gear system, the torque can be increased to make this motor a powerful little machine. One point to make though is that the motor will generate heat when pulsing continually, even at rest so it's important to switch off pulses if nothing is happening.

To control the stepper, apply voltage to each of the coils in a specific sequence.

Step	wire 1	wire 2	wire 3	wire 4
1	High	low	high	low
2	low	high	high	low
3	low	high	low	high
4	high	low	low	high

The sequence would go something like this and be repeated many times very quickly, to maintain rotation in a certain direction.



To reverse the rotation the sequence would need to be reversed.

So what does a stepper motor look like inside? Well here is one, (28BYJ-48), I dismantled. This first image shows the lid removed and turned over to reveal the gears used.

With the gears removed from the body you can see the magnetic core and 8 spikes on a plate surrounding it. These would normally be activated in pairs as described above and would be known as 4 phase.

The stepper motor unleashed

Controlling the Wagon Tippler

So the plan, after much research, of what I needed to do was as follows:

The aim was to use a stepper motor, as other options didn't seem to fit the bill. I would need to setup the stepper motor to raise the wagon platform, pause it while the wagon emptied its coal and then return it to the track base to allow the wagon to be removed.

Organising this as a single button press seemed OK at the outset, but then I realised that if the stepper motor were stopped for any reason, (a power outage or coal getting stuck in the gearing)

then the process would lose its synchronisation and some form of corrective adjustment would be needed.

To retain a simple approach, I decided that two push buttons, (the push to make type), could be used instead of one. One to raise the wagon platform and one to lower it. As they were push to make buttons they would have to be held down to make each work, (not press for on then press for off). This would then be controlled by the operator and line of sight.

To my mind this would allow me to keep the programming simple and straight forward and certainly within my capabilities.

Programming the Arduino

What follows, is a detailed description of the program that I fed into the Arduino to let it control the wagon tippler. You will notice the use of curly brackets, (which are always in pairs) and semi colons which are important components in the programming language. Comments can be included using double forward slashes at the beginning of each line of text. I have used comments to explain each part of the program.

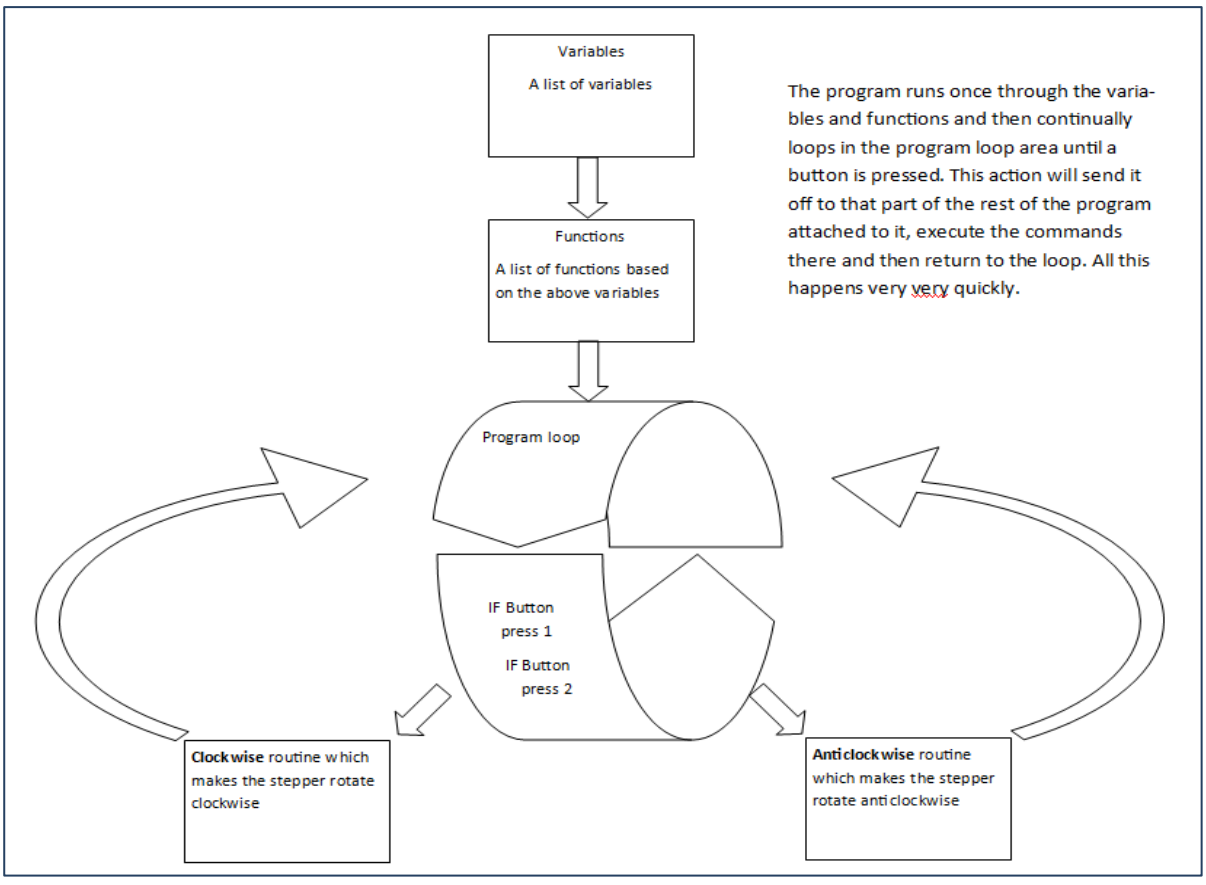
```
// Stepper Motor Direction Control Using 2 push to make buttons with the Arduino Uno
// Setup up the required values for buttons 1 and 2 and motor pins 8,9,10 and 11 here
// also setup a variable for the speed of the motor and initial values for the buttons

int button_1 = 2;
int button_2 = 3;
int motorPin1 = 8;
int motorPin2 = 9;
int motorPin3 = 10;
int motorPin4 = 11;
int motor_Speed = 2;
int val1 = 0;
int val2 = 0;

// Use the button and motor variables above and set them up as the input or output functions
// within the section below known as void setup

void setup()
{
pinMode(button_1, INPUT);
pinMode(button_2, INPUT);
pinMode(motorPin1, OUTPUT);
pinMode(motorPin2, OUTPUT);
pinMode(motorPin3, OUTPUT);
pinMode(motorPin4, OUTPUT);
}


//In the next section which is the power house of the program it is the program loop that
//monitors the button presses and calls the appropriate sub routine if one occurs. This is called
//void loop and it repeats itself over and over again. Button readings
//will remain in a low state, until they are pressed, which makes them go high
```

```
void loop()
{
 val1 = digitalRead(button_2);
 val2 = digitalRead(button_1);

   if (val2 == HIGH)
      { clockwise(); }

   if (val1 == HIGH)
      {  anticlockwise(); }
}


// Once button 1 is pressed in the void loop the program is directed here for the anticlockwise
//sub routine
//a subroutine is identified by the void command, then its name, followed by empty brackets
// the commands in a subroutine sit inside curly brackets

void anticlockwise()
  {
   digitalWrite(motorPin4, LOW);
  digitalWrite(motorPin3, LOW);
  digitalWrite(motorPin2, HIGH);
  digitalWrite(motorPin1, HIGH);
  delay(motor_Speed);
  digitalWrite(motorPin4, LOW);
  digitalWrite(motorPin3, HIGH);
  digitalWrite(motorPin2, HIGH);
  digitalWrite(motorPin1, LOW);
  delay(motor_Speed);
  digitalWrite(motorPin4, HIGH);
  digitalWrite(motorPin3, HIGH);
  digitalWrite(motorPin2, LOW);
  digitalWrite(motorPin1, LOW);
  delay(motor_Speed);
  digitalWrite(motorPin4, HIGH);
  digitalWrite(motorPin3, LOW);
  digitalWrite(motorPin2, LOW);
  digitalWrite(motorPin1, HIGH);
  delay(motor_Speed);

// finally make sure any high pins are set to low to stop pulsing before returning to void loop
  digitalWrite(motorPin4, LOW);
  digitalWrite(motorPin1, LOW);
  }


// If button 2 is pressed the clockwise sub routine as listed here is executed
```

```
void clockwise()
{
digitalWrite(motorPin4, HIGH);
digitalWrite(motorPin3, LOW);
digitalWrite(motorPin2, LOW);
digitalWrite(motorPin1, HIGH);
delay(motor_Speed);
digitalWrite(motorPin4, HIGH);
digitalWrite(motorPin3, HIGH);
digitalWrite(motorPin2, LOW);
digitalWrite(motorPin1, LOW);
delay(motor_Speed);
digitalWrite(motorPin4, LOW);
digitalWrite(motorPin3, HIGH);
digitalWrite(motorPin2, HIGH);
digitalWrite(motorPin1, LOW);
delay(motor_Speed);
digitalWrite(motorPin4, LOW);
digitalWrite(motorPin3, LOW);
digitalWrite(motorPin2,HIGH);
digitalWrite(motorPin1, HIGH);
delay(motor_Speed);

// finally make sure any high pins are set to low to stop pulsing before returning to void loop
  digitalWrite(motorPin1, LOW);
  digitalWrite(motorPin2, LOW);
  }
```

Notice how the pin sequences here are different from the other sub routine.

A diagrammatical view of how the program behaves looks something like this:



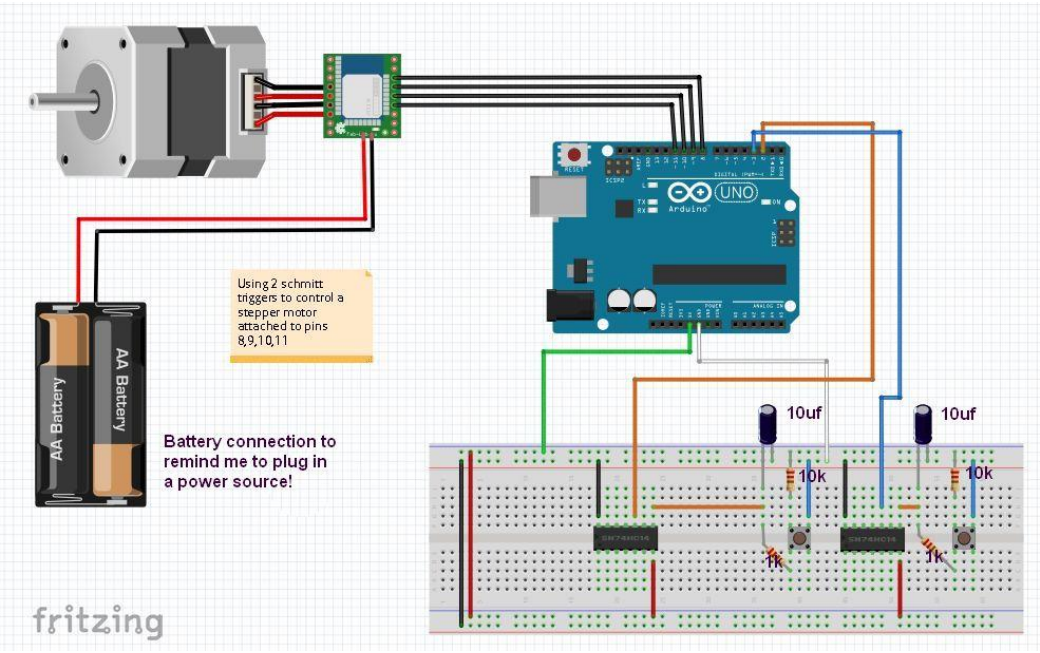
Connecting the stepper motor to the Arduino board

The Arduino Uno communicates with the stepper motor by means of an electronic circuit that contains the buttons. What this looks like is down to a certain amount of 'speculation'. On the one hand it is said that a very simple circuit is all that is needed. But there is a belief that a more complex circuit is necessary particularly as the type of switch in use on its own, may cause problems.

So let's look first at the switches in use. The push button switch effectively closes a circuit. While the circuit is open it is deemed to be low when it is closed it is deemed to be high. In ordinary terms the action of looking at the switches for a press happens extremely quickly over and over again in the looping part of the program. Therefore if the switch is a crude one and for any of several reasons does not perform 'cleanly' it may not appear to work at all. This is known as debouncing. This can make it unreliable or twitchy and over time makes it becomes less and less likely to work as intended. To alleviate this, the circuit in which it sits should cater for this likelihood. So the issue is how to make and maintain the action of switching, a definite 'on' or 'off'? It can be done as a software or hardware solution. I chose to investigate a hardware solution to keep the programming simple!

Using a debouncing circuit

Without going into theory too much there are 'chips' on the market that are deemed to help in this area. One such 'chip' is called a Schmitt trigger. Used in conjunction with a capacitor and resistors it can maintain a steady 'on/off' situation for a physical switch and as such make it much more reliable in use.



Here is a diagram of my switch circuit. You will notice there are two Schmitt triggers in use. This is because of me rather than proper procedure. A Schmitt trigger can control up to 6 switches in theory but my lack of knowledge of circuitry prohibits me from developing a circuit using a single Schmitt trigger, (and they only cost 30p each or less), so the above is the result. It works and it works consistently. I am led to believe that the triggers are not necessary and that would make the circuit a

lot simpler but it depends which camp you find yourself in as to whether they are really necessary or not. Those of us who maintain and run a railway of any size are only too aware of the problems simple electronic circuits can pose in day to day running!

The component parts used were as follows:

2 x 1k resistors

2 x 10k resistors

2 x 10µf capacitors

2 x Texas Instruments SN74LS14N Hex Schmitt Trigger Inverters DIP14

2 x push button switches

The circuit theory in simple terms

The purpose of the capacitor is to smooth out the debounce signal when the button is pressed but it takes a little time to discharge and recharge during the process. The Schmitt trigger component changes this to a more appropriate almost instant change but inverts the signal in the process. So these two components are really just an add-on to a simple switch circuit.

What else is required?

Power supplies: One for the Arduino, (this can be a battery pack), or a mains power supply of 9v. The stepper motor board also requires a power supply of around 5v. Although the Arduino can supply 5v or 12v it is preferable to use this just for the electronics and power the stepper independently.

I left my circuitry on the breadboard which then fitted neatly into a Camden Boss enclosure along with the Arduino board. The two press button switches were mounted on this enclosure as well which meant that the only space needed near the tippler was for the stepper motor and its control board. These fitted easily into a hut placed to the side of the tippler as shown below.

As with most things there are many roads to success and no one road may be the right one. If you spend some time watching YouTube videos on the subject as I did, you might be unsure as to what the correct thing to do is and possibly which road is for you. But I wanted to take on the challenge and so the above is my take on it, right or wrong or unnecessary, it works for me and more importantly the wagon tippler!

Sources:

Hardware:

Ebay, (search for Arduino kit, etc. the cheapest kit comes from China)

Software:

Arduino Uno at: www.Arduino.cc

Fritzing at: http://fritzing.org