Lab Report 1

Abstract

The objective of the week was to build a functioning human-machine interface (HMI) for a previously built car. The car was already capable of travelling 10 m in straight line, with the help of the encoders on two of the four motors.

A digital logic board that would interface the Arduino Nano with the H-bridge for each motor was constructed. The digital logic board will receive eight inputs from the Arduino Nano and convert them into 16 outputs for the H-bridges – each output corresponding to a switch on a H-bridge. The digital logic board design for a single motor (Figure 4) was derived from a truth table (Figure 3). The digital logic board was constructed. An error in the soldering caused a few faults, but once that was fixed the board worked as intended. The final board fulfilled the purpose of interfacing the Arduino Nano with the H-bridges.

For the HMI a 7-inch touchscreen with a resolution 800 x 480 pixels will be used. The coding language Python was paired with the Tkinter module. The final interface design (Figure 10) followed an intuitive forward, backwards, left and right layout. The user can make the car turn 90 ° or 45 ° and if repeated, the user can make the car turn more than 90 °. As the user inputs commands, it is displayed in the LOGO code text box as a list. However, the numbers were unlabelled, and a user would assume it to be a distance, but it was instead a delay. The delays should be replaced by a distance, via the encoders, and be displayed as labelled distances on the GUI.

I^2C was used as the serial communication method between the Arduino Nano and Raspberry Pi 3 due to its simple master/slave relationship. The SDA, SCL and ground pins on the Arduino Nano were connected to the SDA, SCL and ground pins on the Raspberry Pi 3. The Arduino Nano was configured to join the I^2C bus at address seven, the same as the Raspberry Pi 3, and communication was established. However, during the second half of the week the I^2C stopped working. The problem could have been a change in the code or a hardware error, but there wasn't enough time to troubleshoot the problem. A possible solution would have been to have multiple backups of code so rollbacks on code were available.

The full capabilities of the car, by the end of the week, only changed in that it was using the logic board instead of the basic motor connection board. A fully functional HMI was created, so the objective of week was fulfilled, but it could not be attached to the car as the Raspberry Pi and Arduino Nano could not communicate. The Arduino Nano still needed to be pre-programmed and heavy penalties were taken as a result.

Introduction

The main purpose of the week was to build a functioning human-machine interface (HMI), for a previously built car, and thus shift the programming stage from the Arduino Nano to the HMI. To achieve this a digital logic board to interface the Arduino Nano with the H-bridges; a HMI to interface with the user; a LOGO interpreter; and inter-board communication of the HMI with the Arduino Nano all need to be designed, constructed and tested. Half of the group would work on the digital logic board and the other half on the HMI, with the group coming together near the end of the week to assemble their work.

As a test the group will be presented with a route, with known distances, and will be expected to program the car, via the HMI, with the correct pre-programmed set of movements for it to navigate the course correctly. The first course is a 1 m square (Figure 1). This will test the car's accuracy by measuring how close the car returns to where it started. The second course (Figure 1) will be made of 60 cm straights and 90 ° left and right turns.

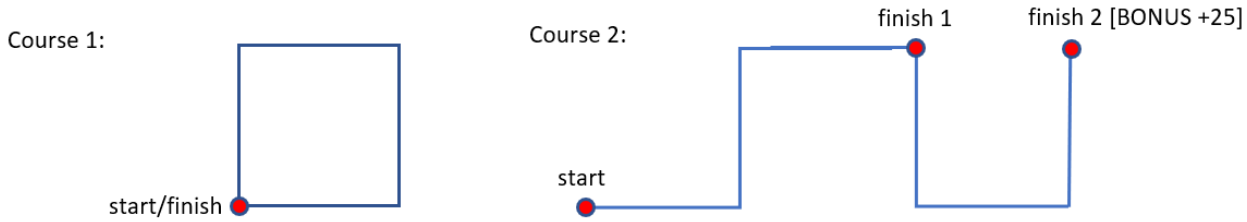

Figure 1 – The layout of course one and course two [1]

The previously built car consists of four H-bridges (Figure 2). Each H-bridge contains four switches. The direction the motor spins, anticlockwise or clockwise, depends upon the direction of current across the motor – which is controlled by the pair of switches that are on. If H+ and L- are switched on, the current flows from left to right, across the motor, and causes clockwise rotation. If H- and L+ are switched on, the current flows from right to left, across the motor, and causes anticlockwise rotation. Each H-bridge is connected to an independent motor. Two of the motors have encoders, which are placed diagonally opposite from each other so that all possible scenarios are available – comparing encoder values as left and right motors or comparing encoder values as front and back motors. The encoder transmits the number of rotations of the motor and sends that to the Arduino Nano on the mainboard, which can then be incorporated in to the code on the Arduino Nano. This allows the car to travel in a straight line for 10 m by self-correcting as it moves.

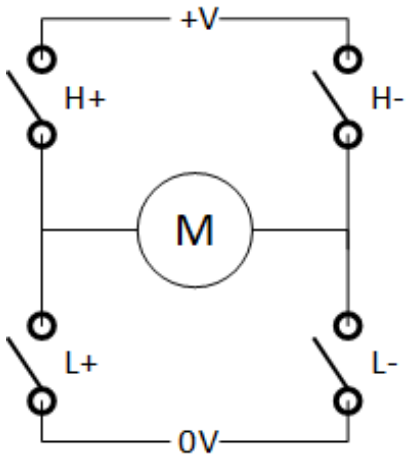

Figure 2 – The layout of a single H-bridge [2]

Digital Logic Board

Design

The digital logic board will be replacing the existing motor connection board, which was acting as wire connection between the Arduino Nano and H-bridges. The digital logic board will receive eight inputs from the Arduino Nano and convert them into 16 outputs for the H-bridges – each output corresponding to a switch on a H-bridge.

A truth table was created for a single motor (Figure 3), which would then be repeated for all other motors. Pulse width modulation (PWM) and direction were the inputs received from the Arduino Nano and the outputs are defined as the four switches on the H-bridge. PWM "controls the amount of power, in the perspective of the voltage component, that is given to a device by cycling the on-and-off phases of a digital signal quickly and varying the width of the "on" phase or duty cycle. To the device, this would appear as a steady power input with an average voltage value, which is the result of the percentage of the on time" [3]. This would allow efficient control of the motor speed from a digital input from the Arduino Nano.

Inputs		Outputs			
PWM	Direction	H+	H-	L+	L-
0	0	0	0	0	0
0	1	0	0	0	0
1	0	0	1	1	0
1	1	1	0	0	1

Figure 3 – The truth table for the digital logic board, where M = motor; H/L = high/low side of the H-bridge; and +/- = positive/negative motor terminals [4]

PWM should take precedence in the table, meaning if it is logic low, then whatever the direction signal, the motor should not spin. The direction has an effect only if PWM is logic high. When PWM is logic high and direction is logic low H- and L+ are switched on and cause anticlockwise rotation of the motor. When PWM is logic high and direction is logic high H+ and L- are switched on and cause clockwise rotation of the motor. The output combinations that would short circuit the motor are prevented with the table above – these combinations are all four switches on; H+ and H- on; and L+ and L- on.

The Boolean expressions obtained from the truth table were H+ = AB; L- = AB; H- = $A\overline{B}$; and L+ = $A\overline{B}$. H+ and L- share the same output, so they can receive it from the same gate. H- and L+ share the same output, so they can receive it from the same gate. Thus, the circuit was simplified from two NOT and four AND gates per motor to one NOT and two AND gates per motor (Figure 4).

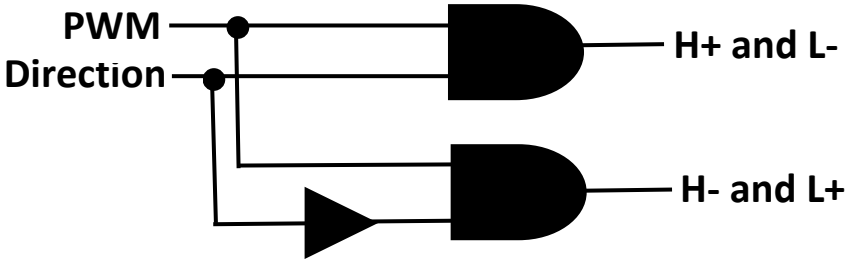

Figure 4 – Logic circuit for a single motor

For a logic low the logic ICs have a range of <0.8 V [5] for the AND gate and <0.7 V [6] for the NOT gate. So, it is very important that the inputs are grounded when they are supposed to be at a logic low and not grounded when they are supposed to be at a logic high. To achieve this, pull-down resistors were connected from each input to ground. A resistor value of 10 kΩ was used as it is common in many circuits and will work within tolerance for this application. When the input is supposed to be a logic low, the resistors will pull it down to the voltage at ground (0 V), and thus keeping it within the ICs range. However, when the inputs are at a logic high the pull-down resistor will have no effect. If the pull-down resistor wasn't included the voltage of the input would enter a range where the ICs couldn't discern if the input was a logic high or logic low and unpredictable errors would occur.

This design was then transferred to a breadboard. To simulate a logic high and logic low the inputs were connected to 5 V and ground respectively. The board itself was connected to ground and 5 V as well, as the logic ICs require power to function. The circuit in Figure 4 worked as the table in Figure 3 predicted. A dry fit design for one motor was tested on the board and adjusted until the design could be replicated for the other motors. The board was then soldered (Figure 5 and Figure 6).

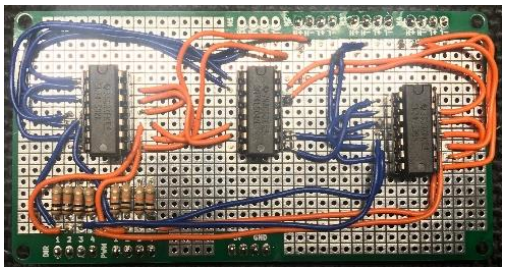

Figure 5 – Front side of digital logic board

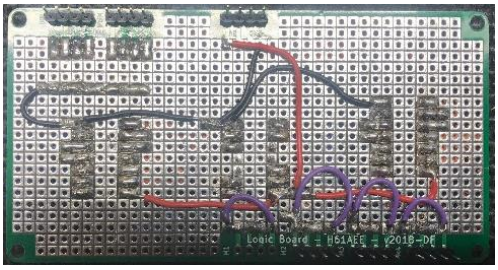

Figure 6 – Back side of digital logic board

Results

The soldered digital logic board was tested and found to have a total of 14 faults. The digital logic board was connected to a breadboard and all input scenarios were tested individually. All scenarios but one worked. For motor two an input of PWM logic high and direction logic low should give an output of logic high at H- and L+. Closer inspection of soldering revealed that the output of a NOT gate and the ground of the NOT logic IC were connected. So, the AND gate for H- and L+ received logic high and logic low instead of logic high and logic high, and thus always outputted a logic low. After separating the output and ground, all faults disappeared, and the board worked as intended.

Discussion

Once the fault of the soldering was found the digital logic board worked as expected from the design process.

The design could be improved by using four colours, instead of two, as this wold help understand the circuit quicker and so troubleshooting problems would be easier. Currently for the car to move forwards or backwards the left and right motors must turn in opposite directions – and it is essential that this is accounted for in the coding process. If this was accounted for in the logic board design, the coding process would have been simplified. This could have been achieved by defining a logic low direction as backwards and a logic high as forwards, instead of the current anticlockwise and clockwise definitions. However, the logic board design would become more complex and the probability of mistakes occurring would have increased.

Human Machine Interface (HMI)

Design

For the HMI a 7-inch touchscreen with a resolution 800 x 480 pixels will be used. A touchscreen is a very intuitive interface, as a result of the popularity of smartphones and tablets, so the learning curve for this HMI should be minimal. A Raspberry Pi 3 will be used due to the lack of a display port, of any type, on the Arduino Nano. There is greater flexibility in software (the graphical user interface) over hardware (buttons, rotary encoders, potentiometers, etc) as changes can be made quickly, and a complete overhaul of the GUI is always available. The coding language Python was paired with the Tkinter module, which is "the standard Python interface to the Tk GUI toolkit from Scriptics" [7].

Figure 7 shows how the forward button is created for the HMI. The button is defined as 'forwardbutton'. The 'tk.Button()' uses the Tkinter module to create a button. First 'text=' prints forwards underneath a centred arrow; then the dimensions of the button are given in height and width; then 'Forward' is the command that will be executed; and the '.grid()' defines the button's location on the screen. This code will be repeated for all the other buttons for the HMI (refer to appendix A).

```
forwardbutton = tk.Button(window, text="^\n| \n Forwards", height=5, width=7,
                    command=Forward).grid(row=2, column=3)
```

Figure 7 – Code for Forward button on the GUI

Figure 8 shows what the function 'Forward', that was called in Figure 6, is defined as. First 'line' is printed to the terminal for debugging purposes; then 'a' is printed to the terminal for debugging purposes; 'disp.insert' adds an F: in the centre text box and at the end of the current line. This type of function will be repeated for all functions that are called from a button press (refer to appendix A)

```
def Forward():
        global line
        print line
        a = '%d.end' %line
        print a
        disp.insert(a, "F:")
```

Figure 8 – Code for the Forward function

Figure 7 and Figure 8 demonstrate a type of LOGO interpreter. As the buttons are pressed the words are converted in to single letter commands and are displayed in the LOGO code text box. When the 'run' button is pressed, the 'run' function (Figure 9) converts the string of commands in to the integers that represent it in Unicode. This is then sent to the Arduino Nano to be executed.

```
def run():
        line1 = disp.get("1.0", "1.7")
        line1b = ConvertStringToByte(line1)
        print line1
        print line1b
        bus.write_i2c_block_data(i2c_address, i2c_cmd_write, line1b)
```

Figure 9 – Code for the Run function

Results

The final interface design (Figure 10) followed an intuitive forward, backwards, left and right layout. The user can make the car turn 90 ° or 45 ° and if repeated, the user can make the car turn more than 90 °. The numerical values included relate to delays instead of distances. As the user inputs commands, it is displayed in the LOGO code text box as a list. This will help remind the user of what instructions have already been inputted. To initiate the program the user will have to press the run button. The reset button will clear all inputs which can be useful if an error had been made in the coding process, or if the user wants to start again.

Discussion

As expected, the HMI was very intuitive and required no understanding of the code to operate.

However, the numbers were unlabelled, and a user would assume it to be a distance, but it was instead a delay. The delays should be replaced by a distance, via the encoders, and be displayed as labelled distances on the GUI.

Some features to improve the experience of interacting with the HMI include having more flexibility on rotation by using the encoder information to allow for more fine control over turning; having the LOGO code text box indicate which part of the list is being executed at any given time, so the user knows how much code is left to run; removing the return button and automatically placing a new command on a new line; having some way of being able to slowly increment the distance, instead of being stuck at the current fixed values; and rearrange the widgets to improve the aesthetic.

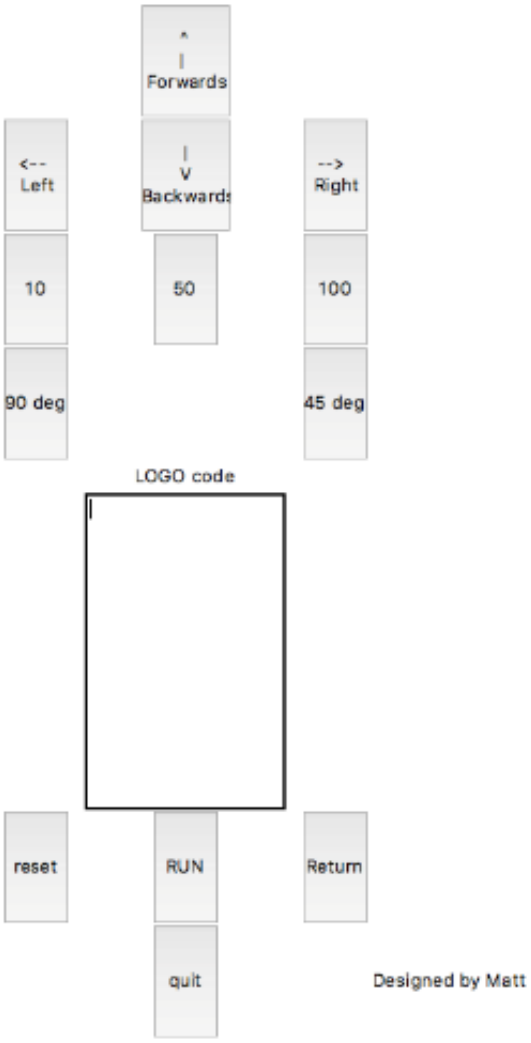

Figure 10 – Final GUI of HMI

Inter-Board Communications

Design

I^2C was used as the serial communication method between the Arduino Nano and Raspberry Pi 3 due to its simple master/slave relationship – evident in that only two bus lines required. The two lines are the serial data line (SDA) and the serial clock line (SCL), as I^2C is a synchronous protocol.

The SDA, SCL and ground pins on the Arduino Nano were connected to the SDA, SCL and ground pins on the Raspberry Pi 3. The Arduino Nano was configured to join the I^2C bus at address seven, the same as the Raspberry Pi 3, and communication was established.

When the Arduino Nano receives commands from the Raspberry Pi 3, a function called 'receiveEvent()' is called (refer to appendix B). This function first creates an array and sets all values within the array to zero. Then the 'while()' function reads the incoming commands until a full stop is seen, then stops reading data.

In early testing of the 'receiveEvent()' function, a mystery character '☒' would precede every command sent to the Arduino Nano. This would cause the 'switch()' function to fail as the mystery character would always be compared against each case instead of the command sent by the Raspberry Pi. An 'if()' statement, that checks the ASII code of every incoming character, was added to the 'while()' function. The range of ASCII codes that will be used for commands by the Raspberry Pi 3 is known. Therefore, a range is specified within the 'if()' statement that allows the characters of the commands to be allocated to the array, but not the mystery character '☒'. In testing this removed the mystery character and allowed the 'switch()' function to work as intended.

A confirmation message of 'All Read' is printed to the serial monitor. Then 'sscanf()' is used to extract, from the command, the letter that is to be used in the 'switch()' function and a parameter that will be used when a case is matched.

A message, confirming the completion of the command and a request for a new command, is sent from the Arduino Nano to the Raspberry Pi 3. This is repeated until all commands are completed.

In the second half of the week the I^2C stopped working and the Arduino Nano and Raspberry Pi 3 could no longer communicate. It was thought that the logical level voltage differences between the Arduino Nano (5 V) and the Raspberry Pi 3 (3.3 V) was now starting to cause problems. A bi-directional MOSFET voltage level shifter design [8] (Figure 11), that would mediate the differing voltage levels, was agreed upon and constructed using a MOSFET and 10 kΩ resistors.

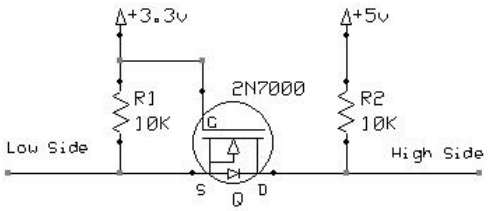

Figure 11 – Bi-Directional MOSFET Voltage Level Converter [8]

The voltage level shifter did not work, even after many hours of troubleshooting. The voltage level shifter was giving lower than expected readings – 3 V instead of 5 V and 2.2 V instead of 3 V. Another voltage level shifter was created, just in case poor soldering was the offender. On this occasion the voltage level shifter just gave the wrong reading all together.

Results

When the I^2C was functioning correctly it worked as follows. If a command of '☒F:100.' is received the command will be read until the full stop and the '☒' character will be removed to leave 'F:100'. This will be split in to 'F' and '100' by 'sscanf()'. The 'F' will be compared to the cases in the switch statement. Case F will be called. 'Forwards' and '100' will be printed to the serial monitor. The function fd(100) will then be called. Once fd(100) has been executed the 'switch()' will terminate, as well as the 'receiveEvent()' function.

However, when the I^2C stopped working the Arduino Nano gave remained idle and appeared of as though no command had been sent to it.

Discussion

For the first half of the week the I^2C worked correctly and allowed for inter-board communication. Commands could be sent via the HMI and would then be executed by the Arduino Nano. However, during the second half of the week the I^2C stopped working. The problem could have been a change in the code or a hardware error, but there wasn't enough time to troubleshoot the problem. A possible solution would have been to have multiple backups of code so rollbacks on code were available.

Conclusion

The final digital logic board fulfilled the aim of interfacing the Arduino Nano with the H-bridges. The board could have been improved by using different colours for each motor and if it accounted for the motors, on the left and right side of the car, having to rotate in opposite direction for the car to move forwards or backwards.

The HMI design satisfied the aim of being very intuitive and required no understanding of the code to interface with it. Changes could have been made to make coding the car easier and more efficient.

The inter-board communication worked for the first half of the week and if it stayed that way the whole car would have come together to fulfil all the aims. However, the inter-board communication stopped working in the second half of the week, which meant that the HMI could not be used.

The full capabilities of the car, by the end of the week, only changed in that it was using the logic board instead of the basic motor connection board. A fully functional HMI was created but it could not be attached to the car as the Raspberry Pi and Arduino Nano could not communicate. The Arduino Nano still needed to be pre-programmed and heavy penalties were taken as a result.

References

[1] *H61AEE Session 2 Challenges Document.*

[2] *H61AEE Session 2 Intro Lecture.*

[3] "PWM Definition," [Online]. Available: https://www.techopedia.com/definition/9034/pulse-width-modulation-pwm.

[4] *H61AEE Project Session 2 Lab Sheet.*

[5] "Quad 2 Input AND Gate Data Sheet," [Online]. Available: https://html.alldatasheet.com/html-pdf/12619/ONSEMI/74LS08/365/2/74LS08.html.

[6] "Hex Inverter Data Sheet," [Online]. Available: http://html.alldatasheet.com/html-pdf/5638/MOTOROLA/74LS04/518/2/74LS04.html.

[7] "Tkinter Module," [Online]. Available: http://effbot.org/tkinterbook/tkinter-whats-tkinter.htm.

[8] "MOSEFET Voltage Level Converter," [Online]. Available: http://www.hobbytronics.co.uk/mosfet-voltage-level-converter.

Appendix A

HMI Code

```
# ///////// HUMAN MACHINE GRAPHICAL USER INTERFACE //////////////
#


from Tkinter import *
import Tkinter as tk
import ttk as ttk
import smbus
import time
import os
import sys

bus = smbus.SMBus(1)

# i2c adress of arduino slave
i2c_address = 0x07
i2c_cmd_write = 0x01
i2c_cmd_read = 0x02

def ConvertStringToByte(src):
        converted = []
        for i in src:
                converted.append(ord(i))
        return converted


window = tk.Tk()

window.title("Human machine Interface")
window.geometry("500x850")


t = tk.Label(window, text="Human Machine Interface").grid(row=1, column=3)

global line
line = 1

#Button Functions

def Forward():
        global line
        print line
        a = '%d.end' %line
        print a
        disp.insert(a, "F:")

def Backwards():
        global line
        print line
        a = '%d.end' %line
        print a
        disp.insert(a, "B:")

def Right():
        global line
        print line
        a = '%d.end' %line
        print a
        disp.insert(a, "R:")
```

```
def Left():
        global line
        print line
        a = '%d.end' %line
        print a
        disp.insert(a, "L:")

def _10():
        global line
        print line
        a = '%d.end' %line
        print a
        disp.insert(a, "10.")

def _50():
        global line
        print line
        a = '%d.end' %line
        print a
        disp.insert(a, "50.")

def _100():
        global line
        print line
        a = '%d.end' %line
        print a
        disp.insert(a, "100.")

def _45deg():
        global line
        print line
        a = '%d.end' %line
        print a
        disp.insert(a, "45.")

def _90deg():
        global line
        print line
        a = '%d.end' %line
        print a
        disp.insert(a, "90.")

def Reset():
        disp.delete('1.0', "100.0")
        global line
        line = 1

def Return():
        global line
        line = line + 1
        print line
        a = '%d.end' %line
        disp.insert(a, "\n")

def run():
        line1 = disp.get("1.0", "1.7")
        line1b = ConvertStringToByte(line1)
        print line1
        print line1b
        bus.write_i2c_block_data(i2c_address, i2c_cmd_write, line1b)

def quit1():
        os._exit(0)
```

```
# Button initialiations

forwardbutton = tk.Button(window, text="^\n| \n Forwards", height=5, width=7,
command=Forward).grid(row=2, column=3)

rightbutton = tk.Button(window, text="--> \n Right", height=5, width=5,
command=Right).grid(row=3, column=4)

leftbutton = tk.Button(window, text="<-- \n Left", height=5, width=5, command=Left).grid(row=3,
column=2)

backbutton = tk.Button(window, text="|\n V \n Backwards", height=5, width=7,
command=Backwards).grid(row=3, column=3)

resetbutton = tk.Button(window, text="reset", height=5, width=5, command=Reset).grid(row=10,
column=2)

returnbutton = tk.Button(window, text="Return", state="normal", height=5, width=5,
command=Return).grid(row=10, column=4)

runbutton = tk.Button(window, text="RUN", state="normal", height=5, width=5,
command=run).grid(row=10, column=3)

quitbutton = tk.Button(window, text="quit", state="normal", height=5, width=5,
command=quit1).grid(row=12, column=3)

button10 = tk.Button(window, text="10", state="normal", height=5, width=5,
command=_10).grid(row=4, column=2)

button50 = tk.Button(window, text="50", state="normal", height=5, width=5,
command=_50).grid(row=4, column=3)

button100 = tk.Button(window, text="100", state="normal", height=5, width=5,
command=_100).grid(row=4, column=4)

button90deg = tk.Button(window, text="90 deg", state="normal", height=5, width=5,
command=_90deg).grid(row=5, column=2)

button45deg = tk.Button(window, text="45 deg", state="normal", height=5, width=5,
command=_45deg).grid(row=5, column=4)

canvs = tk.Canvas(window, height=5, width=80).grid(row=5, column=1)


t = tk.Label(window, text="LOGO code").grid(row=6, column=3)
disp = tk.Text(window, height=15, width=20, state="normal")
disp.grid(row=7, column=3)




t = tk.Label(window, text="Designed by Matt").grid(row=12, column=5)



window.mainloop()
```

Appendix B

Arduino Nano Code

```
#include <Wire.h>
#include <H61AEE_S01.h>

//Global Variables
Vehicle car;

//Functions for moving car

void fd(int distance) //function for moving car fd
{
  car.setDirection(LEFT, forwards);
  car.setDirection(RIGHT, forwards);
  car.setSpeed(LEFT, 50);
  car.setSpeed(RIGHT, 50);
  delay(distance); //This constant is yet to be worked out
  car.setSpeed(LEFT, 0);
  car.setSpeed(RIGHT, 0);
}

void bk(int distance)// function for moving car bk
{
  car.setDirection(LEFT, backwards);
  car.setDirection(RIGHT, backwards);
  car.setSpeed(LEFT, 50);
  car.setSpeed(RIGHT, 50);
  delay(distance); //This constant is yet to be worked out
  car.setSpeed(LEFT, 0);
  car.setSpeed(RIGHT, 0);
}

void rt(int angle)// function for turning rt
{
  car.setDirection(LEFT, forwards);
  car.setDirection(RIGHT, backwards);
  car.setSpeed(LEFT, 50);
  car.setSpeed(RIGHT, 50);
  delay(angle); //These constants are yet to be worked out
  car.setSpeed(LEFT, 0);
  car.setSpeed(RIGHT, 0);
}

void lt(int angle)
{
  car.setDirection(LEFT, backwards);
  car.setDirection(RIGHT, forwards);
  car.setSpeed(LEFT, 50);
  car.setSpeed(RIGHT, 50);
  delay(angle); //These constants are yet to be worked out
  car.setSpeed(LEFT, 0);
  car.setSpeed(RIGHT, 0);
}

void setup() {

  car.setupVehicle(); // configure car library
  car.enableMotors(true); // Disable motors so they don't move

  Wire.begin(7); //Join I2c bus at address 7
  Wire.onReceive(receiveEvent);
  Serial.begin(9600);
```

```
}

void loop() {
  delay(10);
}

//Function for when data is recieved from Raspberry Pi

void receiveEvent(int howMany)
{
  char datarr[200];
  for (int b = 0; b < 200; b++)
  {
    datarr[b] = 0;
  }
  char c = 'a';
  int i=0;
  int param=0;
  while ( (c = Wire.read()) != '.')
  {
    if ( ( c > 32 ) && ( c < 123 ))
    {
      datarr[i++]= c;
      Serial.print(c);
    }
  }
  Serial.print(" All read");


  // "F:100."
  sscanf(datarr,"%c:%d", &c, &param);

  switch (c)
  {
    case 'F' :  Serial.print("Forward");
                Serial.print(param);
                fd(param);
                break;

    case 'B' :  Serial.print("Backwards");
                Serial.print(param);
                bk(param);
                break;

    case 'R' :  Serial.print("Right Turn");
                Serial.print(param);
                rt(param);
                break;

    case 'L' :  Serial.print("Left Turn");
                Serial.print(param);
                lt(param);

                break;
  }
}
```