Remote Automated Vehicle Heating System

B code	Name (first middle last)
B00739157	Jack Gibson

Background

The use of Artificial Intelligence (AI) to resolve complex problems within automated vehicles has soared in recent years. It's use to solve everyday problems needs further development e.g. convenience of defrosting a windscreen, especially if medical conditions impacts physical ability, or driving a work vehicle in extreme weather.

Idea in brief

AI and Deep Learning is used with low power microcontrollers to voice activate the heating system and defrost a windscreen. In the home a user automates a temperature controlled system for defrosting their vehicle using its external sensors (i.e. temp sensor) and AI (in the form of a wake-word). The microcontrollers communicate to activate external peripherals (heater element and temperature sensor) using Internet of Things (IOT). Server stored data can be shared and used for performance analysis. An algorithm calculates fuel consumption from DC motor running time.

Objectives

1. To use low power microcontrollers and IOT to automate a temperature controlled environment in a vehicle in an internet zone. The DC motor (fan and heated element) switches on if temp sensor read $\leq 0^{\circ}C$. **Obj.1 has been met**. Threshold temperature was increased to $4^{\circ}C$ aligning with vehicle's ice risk warning. Additional functionality was added in the form of a wake-word. Deep Learning Neural Networks trained a model to deploy to the microcontroller, starting the system. **2.** To evaluate the systems operational time and fuel efficiency over 7 days, **Obj.2 has been largely met.** Fuel consumption (0.5L per hour at idle-car) was calculated using running time of the DC motor. Limitations of ThingSpeak allow only1.4 hours of collective data space (at 20 sec intervals) a larger server would enable objective 2 to be met in full.

No system today uses integrated remote control and WIFI from inside a home, or implements a voice activated heating system within vehicles. A start-up company for this product has huge relevance in industry and will market to industry leading companies.

Implementation

Initially the wake-word "Start" was modelled with software Edge Impulse, it failed to run on Arduino IDE with numerous fatal errors in file size. Research from *TinyML* (Warden.P,

2019) shows how to light LEDs for words “Yes”/ “No” using a Deep Learning algorithm. “On”/“Go” were chosen as raw audio from the Google Speech Commands Dataset. Python script was altered to detect the new words.

See stages for retraining the model in Figure 24, Appendix B. “On”/“Go” needed to replace “Yes”/“No” to reference the correct dataset files. Learning Rate and Training Steps were defined for later use in training the model. Features of the words were extracted using spectrogram analysis in training the model. The trained model was converted to a TensorFlow Model and then to a TensorFlow Lite model which is fully Quantized for use on Arduino with a model size of less than 20kbites. 97% accuracy was achieved when the model was tested against unseen data, (see graph in Figure 7, Appendix A for accuracy and cross entropy) allowing a high probability of wake-word detection without over fitting. Conversion to a C source file allowed the finished TensorFlow Lite model to be deployed. Following installation of the TensorFlow Lite library (ensuring compatibility with Arduino IDE) modifications to “microspeech” example were made by replacing the existing model with the new C source file. Wake-word labels changed from “Yes,”/“No” to “On,”/“Go, ”in the model_settings and the Command Responder file modified to light an external LED for 10 seconds if either wake-word is heard. See block diagram, Figure 9, Appendix A for explanation of how Arduino Nano 33 BLE Sense references its Built in Microphone for input, it’s Model for inference and its LED for Output.

Arduino Nano 33 BLE Sense, (located in the control panel) was not compatible with the ThingSpeak Server due to its m-bed architecture. It was necessary to connect this wake-word Arduino to a second Arduino (MKR1000) allowing data to be sent and received from the server (covered later in the report).

A third Arduino, Nano 33 IOT, in the vehicle reads wake-word status from the control panel and turns ON/OFF a heating element (represented by a DC motor). Temperature data is uploaded to the ThingSpeak server via a TMP36GZ sensor mounted externally (e.g. under the bumper). Arduino Nano 33 IOT uses its internal Wi-Fi module to send and receive data from the server using the WIFININA Library. This board has a power supply of 3.3V (Vcc) and is connected to ground (GND). Pin A1 captures temperature data from pin Vout on TMP36GZ sensor. Digital pin 9 sets the state of a MOSFET transistor module which powers a DC motor. Power is supplied via 3.3V breadboard power supply module, connected to a 9V battery (See Figure 10, Appendix A for breadboard schematic).

The TMP36GZ Temperature Sensor, range -45˚C-125˚C was chosen over the DHT11, range 0˚C-50˚C, reflecting the temp range applicable to this project. Voltage drop is measured on Vs of the sensor. Quantization and Calibration processes enable temperature to be calculated from voltage. After initialising Vout pin to A1 of the Arduino, analog reference is set to default (Nano 33 IOT default is 3.3V). Analog read resolution is set to 8 depicting the 8 bit ADC (Analog to Digital Converter). Local floats are set for assigning Voltage and Temperature values. A local integer is defined as Qlevel allowing the Quantisation level to be defined. Voltage is calculated by multiplying Quantization level by 3.3 and dividing by 255 (number of levels in 8 bits). Temperature is calculated from voltage using data sheet and process, outlined in Figures 12 and 13, Appendix A. Temperature accuracy was confirmed by cross referencing with those displayed on the internal thermostat of the room it was tested in.

A 3V DC motor with MOSFET IRF520N transistor represents the vehicle's heating element and fan allowing installation of a larger motor drawing more current if required. Transistor Source is connected to Ground (GND), Drain to first line of DC motor (with second line to 3.3V) and Gate (controlling state of DC motor) to digital pin 9. A diode connects from Drain to Ground (GND) ensuring a unidirectional flow of current.

Arduino Nano 33 IOT sends temperature and DC motor status data to the server using the WIFININA library, connecting to the internet and assigning data to a channel using the ThingSpeak library. Integers are assigned to Wi-Fi Name (ssid), Passwords, Read and Write API keys and channel number of the server. In setup, Baud Rate =9600 and ThingSpeak library is connected to WiFiClient. Within the loop, a local integer is set for Z and a local float defined for Temperature. Z and Temperature values update every 20 seconds with the latest sent to the server. From the control panel, the program reads the integer defined as Z to identify wake-word status. If Z = 0 DC Motor is OFF, if Z = 1 DC Motor is ON.

The Control Panel (located indoors) comprises Arduino Nano 33 BLE Sense, external LED, Arduino MKR1000, photoresistor, LCD display and two state LEDs. The MKR1000 takes input from the Arduino Nano 33 BLE Sense using an external LED (change in light intensity detected by photoresistor) and receives and sends data to ThingSpeak. Temperature is checked from the server (Sent by Arduino Nano 33 IOT). If temperature is ≤4˚C and wake-word is detected, integer A =1.

For the photoresistor, pin 1 is connected to pin A2 of the Arduino, pin 2 connected to 3.3V and pin 3 connected to ground (GND). Integer "val" stores the Light Intensity value from

pin A2 and is used in a series of “If” statements to determine whether a signal should be sent to turn on the DC motor (A = 1 or 0). i.e. if val <1010 && temp is ≤4, A = 1, if val>1010 && temp > 4, A = 0. Integer A is sent to the server enabling Arduino Nano 33 IOT to determine if its motor should be on to heat the vehicle.

The 16x2 LCD and two State LEDs are the user interface. The LCD was chosen for reliability and ease of use with HD44780 Hitachi Chipset and compatibility with the Liquid Crystal library. (See Figure 3 Appendix A for LCD and State LED wiring). VSS and VDD are both connected to Ground (GND) and 3.3V respectively. Vo is connected to the potentiometer reference pin and controls display contrast. RS (Register Select) is connected to Arduino pin 12 and controls character to display or commands. RW (Read/Write) is connected to Ground (GND) as Arduino is writing to the LCD not reading from it. E(Enable pin) connects to Arduino pin 11, allowing the LCD to process incoming data. D4, D5, D6 and D7 connect to Arduino pins 5, 4, 3 and 2 respectively, set up to allow for a 4 bit transfer. LCD can be configured for 8 bit data transfer if speed is required. A connects via a 220Ω resistor to 3.3V and K connects to Ground (GND) representing anode and cathode respectively, powering the backlight. The red and green LED anodes are connected to Arduino pins 6 and 7 respectively and cathodes connected through 220Ω resistors to Ground(GND).

The Liquid Crystal Library simplifies the code required to print outputs to the LCD. In the code all LCD and LED pins are defined as constant integers. When the power is on but Wi-Fi not yet connected, the LCD prints “Wait for Wi-Fi Connection” Once connected, the LCD prints “Connected State Command” notifying the user that the system is ready to receive spoken input. Based on the input from the microphone (wake-word && ≤4) if integer A=1, LCD displays “Frost!! Heating ON” and Green LED is on. Conversely if A=0 LCD displays “Temp is ‘T’ deg No Heating” and Red LED is on. These are displayed for 10 seconds then reset screen is shown “Program Reset Press Restart” instructing user to manually press the reset button on the Arduino resetting the internal timer to 0.

Data from the server is shown in the ThingSpeak channel as line graphs. These can be exported to MATLAB and an algorithm used to convert time DC motor=ON to amount of fuel consumed (script file Figure 26, Appendix B).

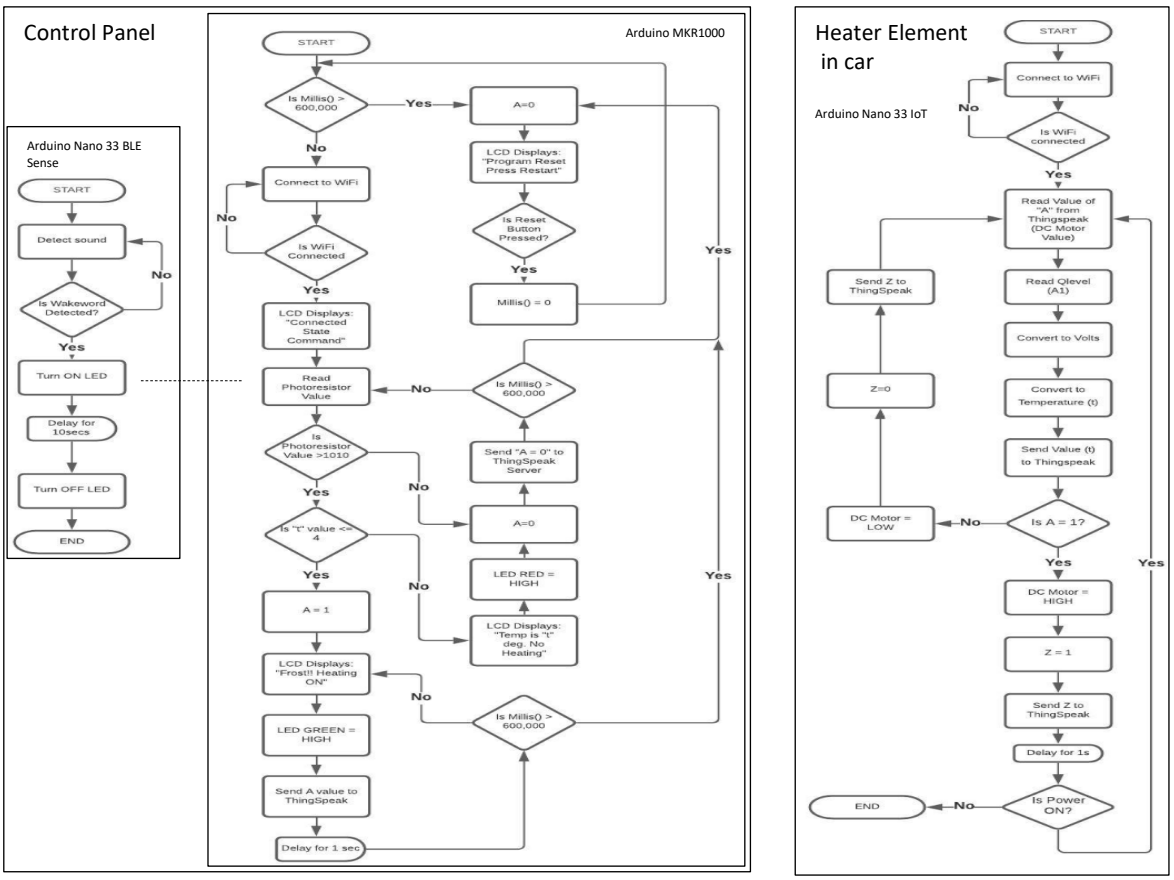


Figure 1: Flow chart of the Remote Car Heating System

Block diagram of the implemented system

Arduino Nano 33 BLE Sense sends wakeword input to Arduino MKR1000 to check temperature from ThingSpeak server sent by Arduino Nano 33 IoT with Temp Sensor. If temp ≤4 DC Motor and Green LED both on with LCD displaying “Frost!! Heating On”. If Temp>4 DC motor is off, Red LED on and LCD displays “Temp is ‘T’ deg No Heating”.

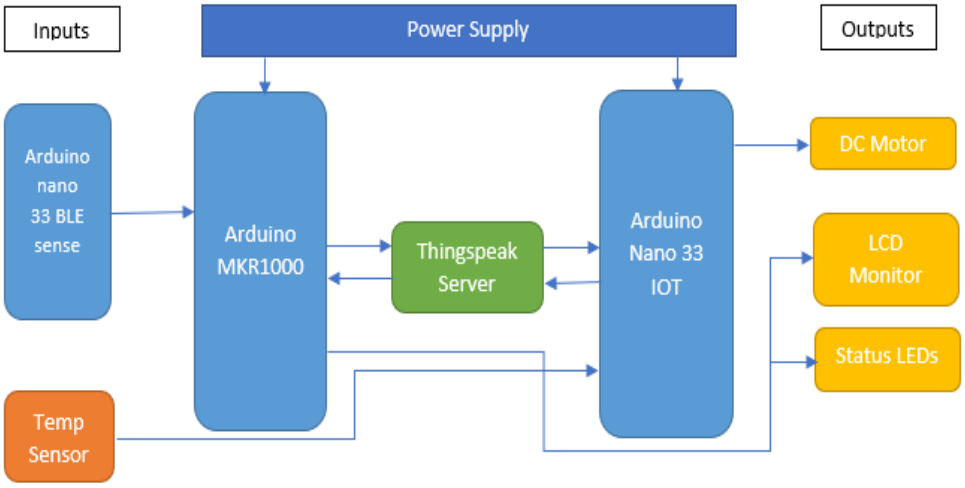


Figure 2: Block diagram of the Remote Car Heating System

Appendix A: Circuits

Control Panel (Fritzing)

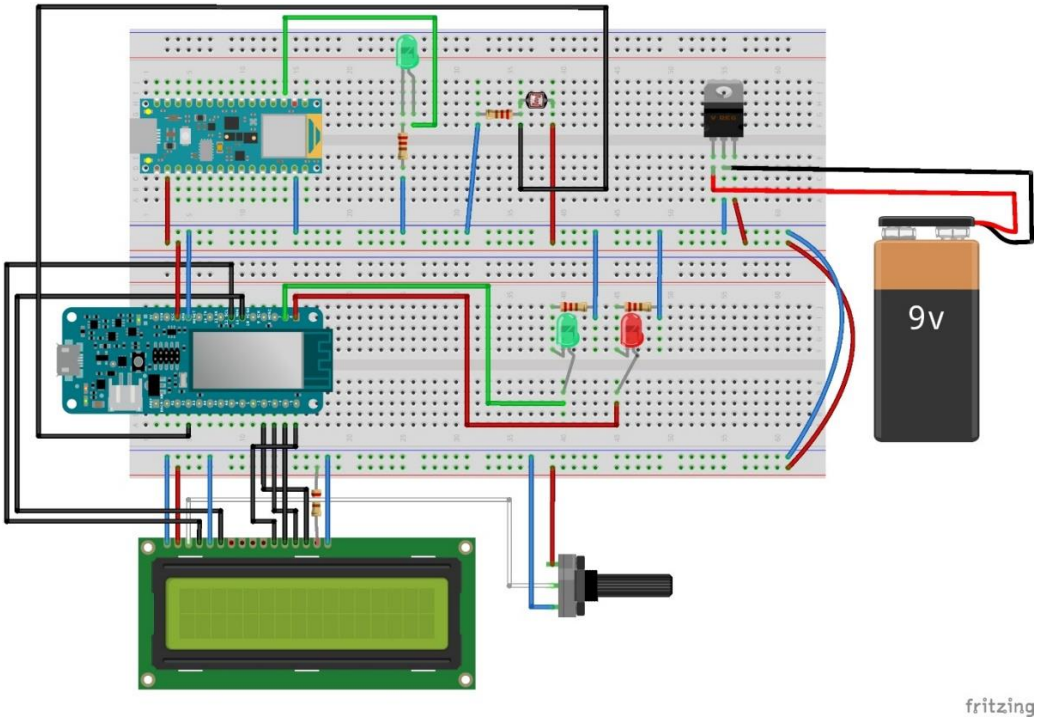


Figure 3 Control Panel Circuit Diagram - *Fritzing does not support breadboard power supply module and so a 9V battery has been added together with a voltage regulator to simulate the same function.)*

Control Panel (Implemented Circuit)

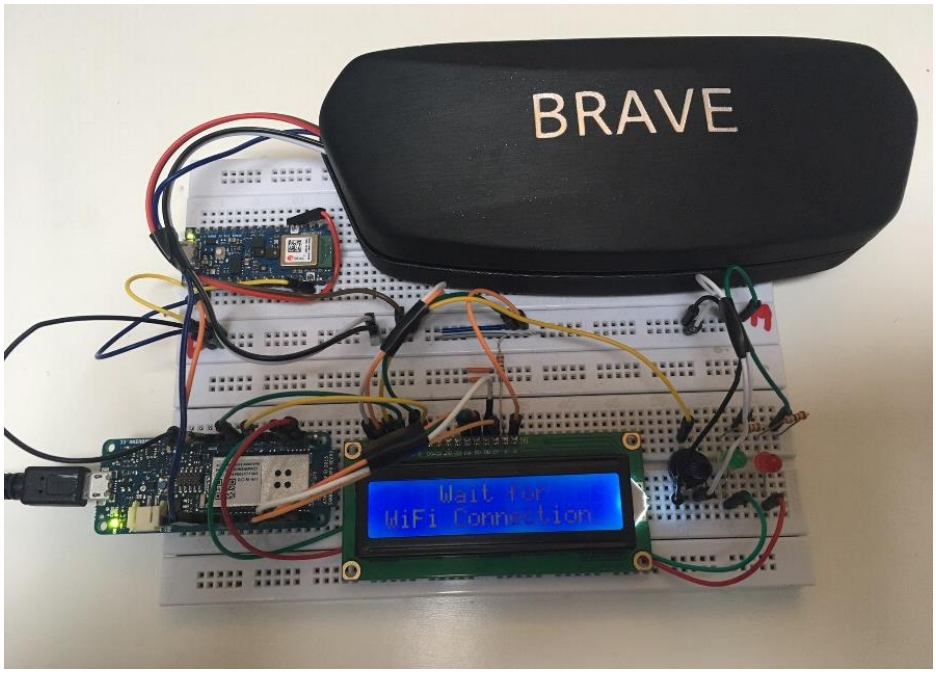


Figure 4 Control Panel Circuit- Case used to house photoresistor and LED

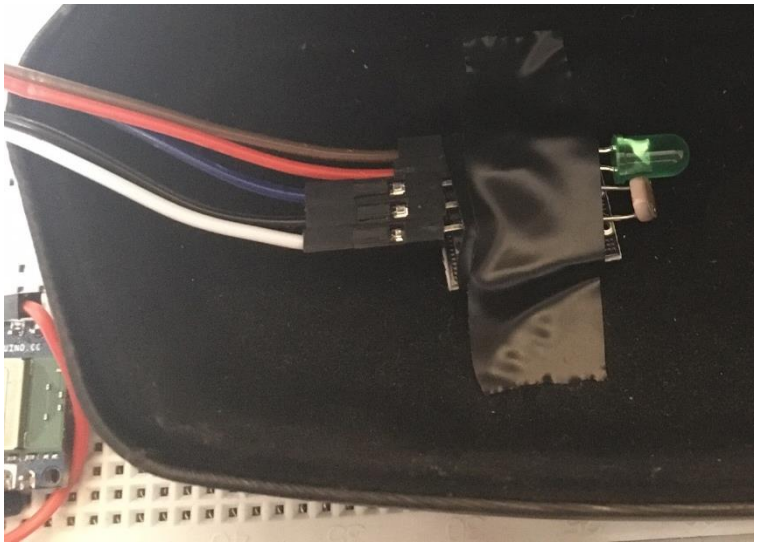


Figure 5: LED and Photoresistor in Case

Figure 6: 16x2 LCD output messages on Control Panel

Arduino Nano 33 BLE Sense (Wake-word)

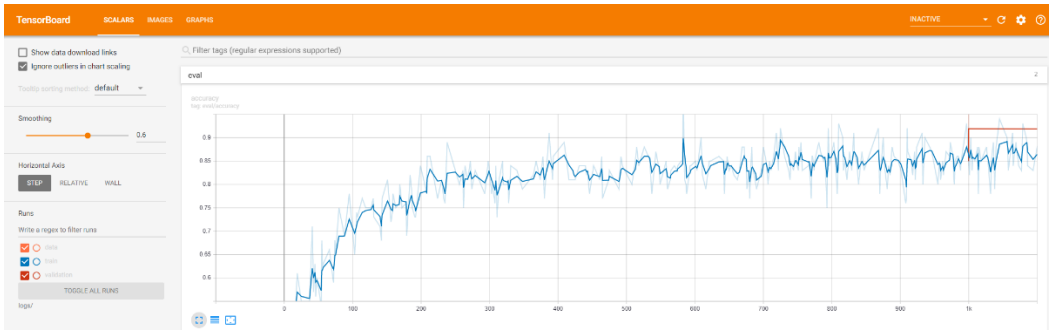


Figure 7: Accuracy of Wakeword detection during Training

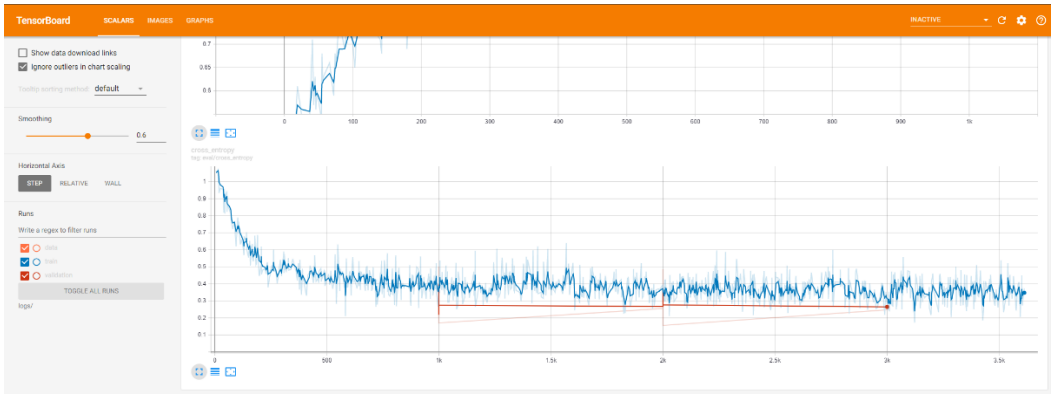


Figure 8: Cross Entropy of Wakeword during Training

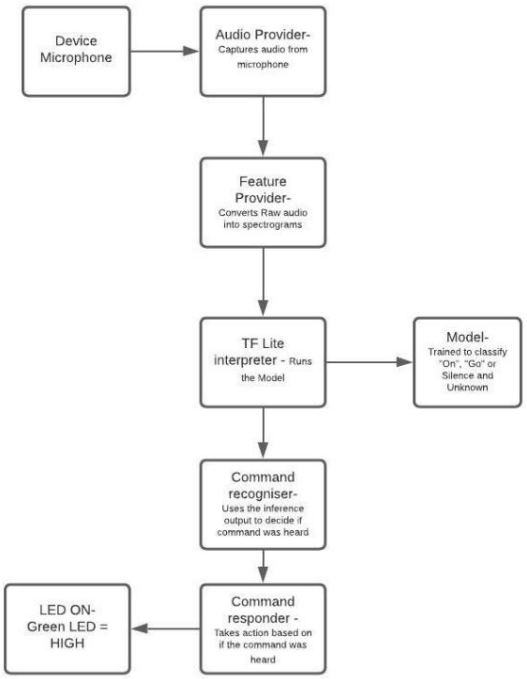


Figure 9: How Arduino Nano 33 BLE Sense References the Microphone for Input, Model for Inference and LED as Output.

Vehicle Heating and Temperature Element (Fritzing)

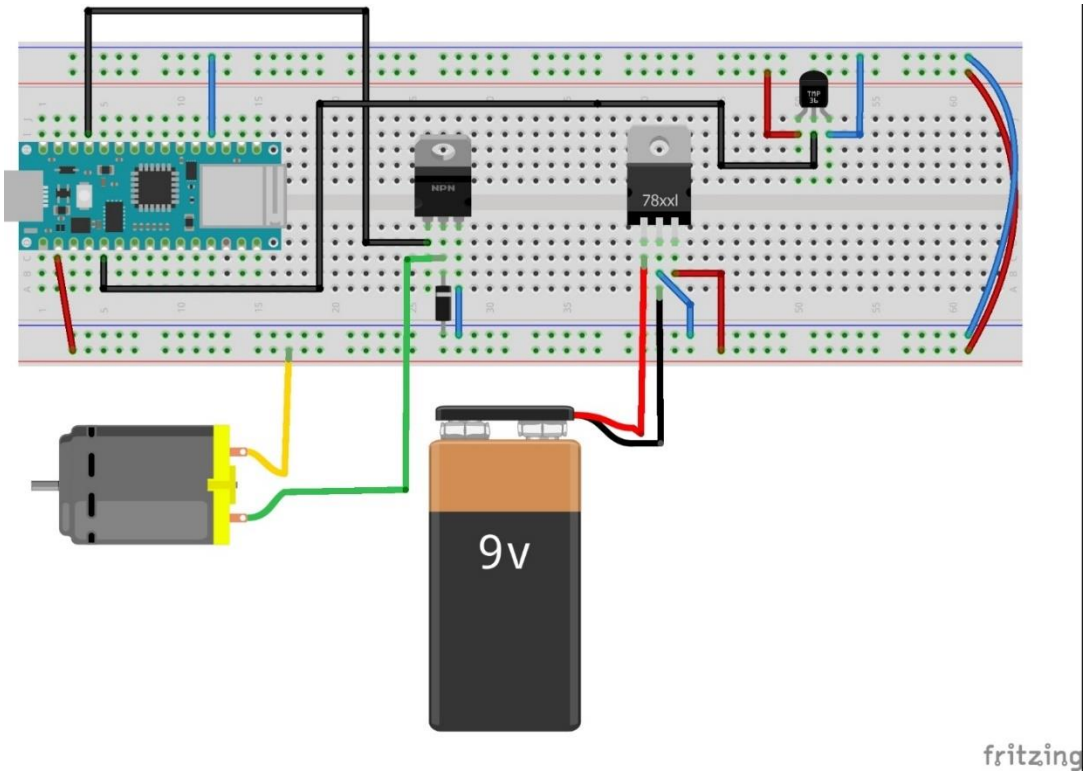


Figure 10: Vehicle Heating and Temp Element Circuit Diagram - *Fritzing does not support breadboard power supply module and so a 9V battery has been added together with a voltage regulator to simulate the same function.)*

Vehicle Heating and Temperature Element (Implemented Circuit)

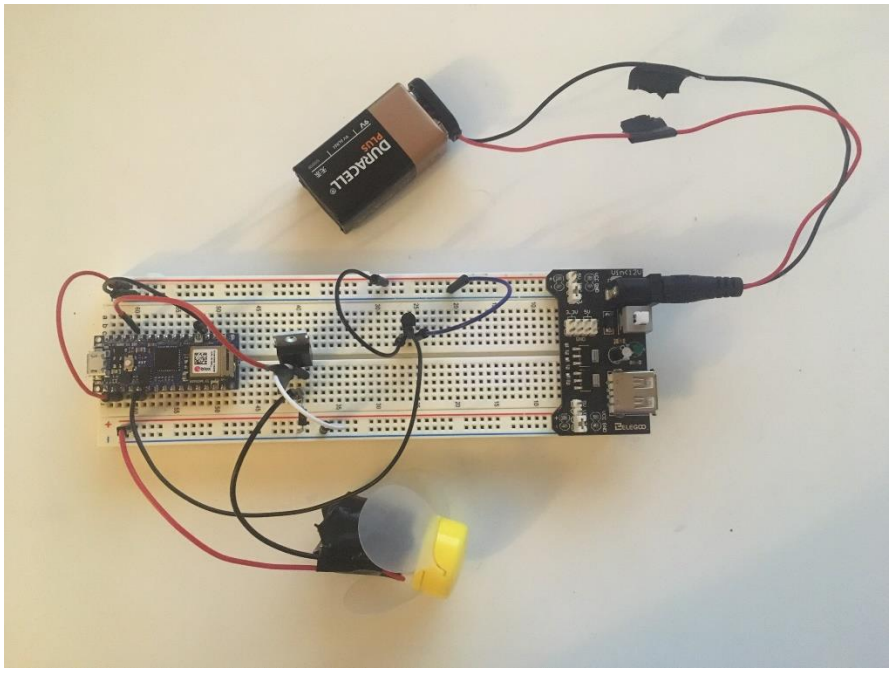


Figure 11: Vehicle Heating and Temp. Element Circuit

Calibration of TMP36GZ Sensor

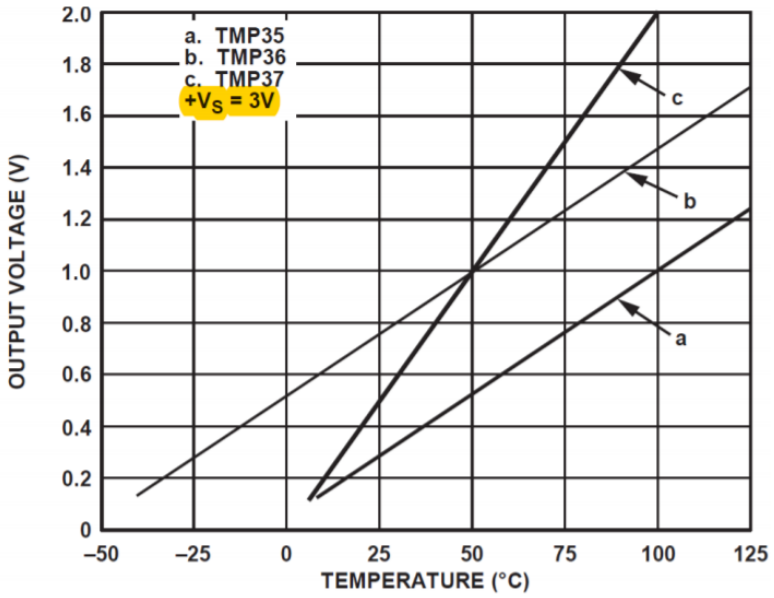


Figure 12: Correlation Between Temp and Output Voltage For TMP36 sensor taken from Data sheet

Take 4 points from Figure 12 - *(T1,V1)(T2,V2)*

T1 = -40

V1=0.1

T2= 125

V2=125

$$m = \frac{V2 - V1}{T2 - T1} = 0.01$$

$$V - V_2 = m(T - T_1)$$

$$V = 0.01T + 0.5$$

$$T = \frac{(V - 0.5)}{0.01}$$

[Key: V in Volts, T in ˚C]

Figure 13: Calculation to convert Voltage to Temperature

Components List

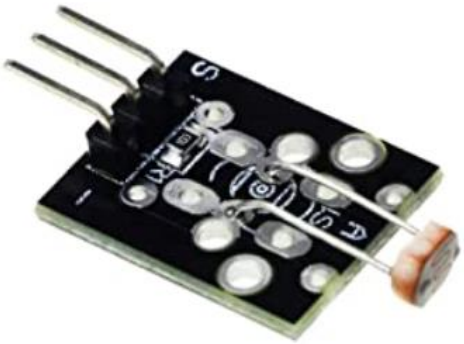


Figure 14: Photoresistor

Figure 15: Red and Green LEDs

Figure 16: 16x2 LCD Monitor

Figure 17: TMP36GZ Temperature Sensor

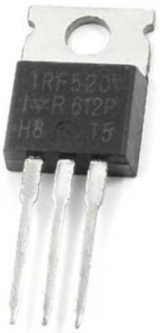


Figure 18 MOSFET IFR520N Transistor

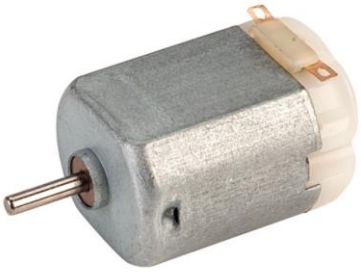


Figure 19: 3V DC Motor

Data Analysis using ThingSpeak

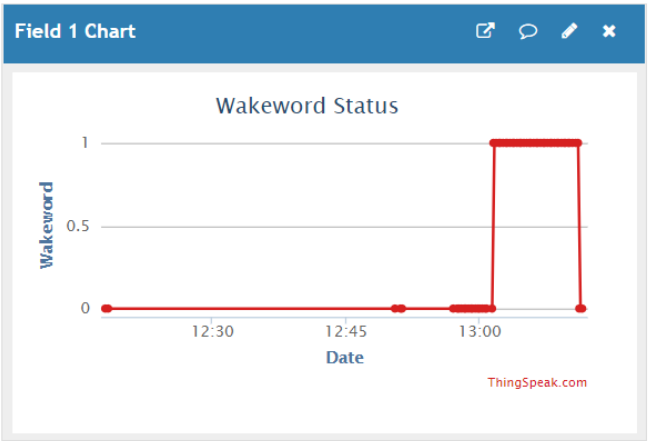
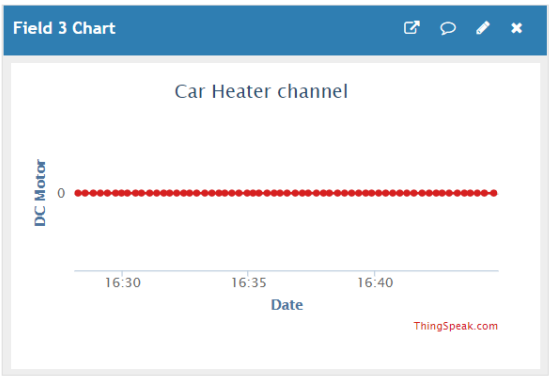


Figure 20: Wakeword Status in Wakeword Channel



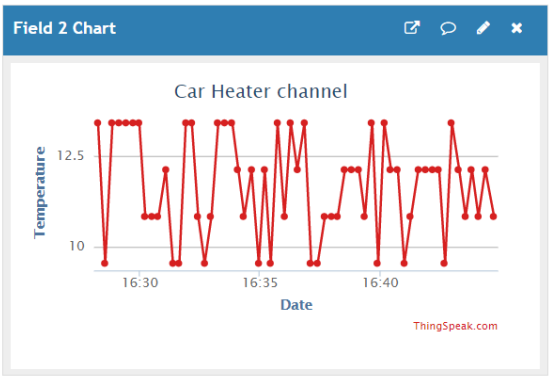


Figure 21: DC Motor and Temperature values in Vehicle Channel

Appendix B: Code

Figure 22: Control Panel MKR1000 Code

```
// Created :  2 / 11 / 2020
// Written by: Jack Gibson for EEE527 Project
// Description: Code for Arduino MKR 1000 - Module mounted to wall in
case in the home
//                                        - Reads Data from external LED
from Arduino Nano 33 BLE Sense and compares with temperature values sends
output to Thingspeak server
//                                        - If LED value is HIGH (<1010)
AND temperature read from Thingspeak <= ThresholdValue, Send HIGH value
(1) to Wakeword Channel on Thingspeak
//                                        - Uses output of Wakeword
status to display messages to the user via LCD


#include <SPI.h>              // SPI library used to process data from
photoresistor sensor and send it over WiFi-101
#include <WiFi101.h>          // WiF-101 Library used to connect MKR 1000
to Wifi
#include<ThingSpeak.h>        // ThingSpeak used to analyse data
#include<LiquidCrystal.h>     // Liquid Crystal library used to simplify
code required to output messeges to the LCD screen

const char ssid[] = "BTBHub6-MXH2";         //  WiFi Name (ssid) defined
const char password[] = "mwYTE9Fxu9kc";     // Wifi Password defined
int status = WL_IDLE_STATUS;                // status initially defined as
idle (available for connection)
WiFiClient client;                          // WiFi client defined as
client for use in WiFi-101 library

unsigned long wakewordChannelNumber =  1221517;     // set Channel ID for
Wakeword
const char * myWriteAPIKey = "IRXL27JIMVHMU1BB";    // Set Write API Key
for Wakeword Channel

unsigned long CarHeaterChannelNumber = 1224137;     // Channel number for
Car heater
unsigned int temperatureFieldNumber = 2;            // Temp values stored
in Field no. 2
const char * myReadAPIKey = "1CAZ6CZYEX1GBLKY";     // API key for Car
Heater Channel

const int  ledG = 7;          // Green LED at pin 7
const int ledR = 6;           // Red LED at pin 6
int lightSensor = A2;         // Light Sensor data to pin A2
int val = 0;                  // Initial Value for photoresistor set to
0
static byte A = 0;            // used to show status of Trigger (wakeword)
1-ON,  0-OFF
unsigned long start_time;     // start time defined for timer
int ThresholdVal = 4;        // Threshold value set for temperature
readings

const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;     // define
pins of LCD
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);                      //
configure pins to ports on LCD
```

```
void setup() {

  pinMode(ledG, OUTPUT);   //Green LED defined as OUTPUT
  pinMode(ledR, OUTPUT);   // Red LED defined as OUTPUT
  start_time = millis();   // Start_time set to Arduino's internal timer

  lcd.begin(16, 2);                // dimension of LCD defined as [16 x
2] and library begin
  lcd.print("    Wait for    ");
  lcd.setCursor(0,1);              // Prints "Wait for WiFi Connection"
across 2 lines in LCD while Wifi isn't connected
  lcd.print("WiFi Connection");
  Serial.begin(9600);              // Baud Rate set to 9600
/*
  while ( status != WL_CONNECTED) {
    Serial.print("Attempting to connect to WPA SSID: ");  // Attempt to
connect to Wifi network:
    Serial.println(ssid);
*/
    status = WiFi.begin(ssid, password);    // Connect to WiFi network:
    delay(10000);                           // Wait 10 seconds for
connection:

  ThingSpeak.begin(client);                              // Begin
Thingspeak Library
  Serial.println("You're connected to the network");    // Print to
Serial Monitor once connected to WiFi Network

  lcd.setCursor(0,0);
  lcd.print("   Connected!   ");    // Print message to LCD across 2 lines
to start heater
  lcd.setCursor(0,1);
  lcd.print(" State Command ");

  }


void loop() {

  int statusCode = 0;    // Define initial statusCode to 0 to show
correct error code in debugging or display correct value from server

float temperatureInC = ThingSpeak.readFloatField(CarHeaterChannelNumber,
temperatureFieldNumber);    //Temperature value defined by reading from
Car Heater Channel on Thingspeak server

 statusCode = ThingSpeak.getLastReadStatus();    // Read the last value
sent to the server in the Car Heater Channel
 if(statusCode == 200){
   Serial.println("Car Temperature is  " + String(temperatureInC) + " deg
C");    // If Status code is 200, Print "Car temp. is (temp value) deg C"
in Serial Monitor
 }
 else{
   Serial.println("Problem reading channel. HTTP error code " +
String(statusCode));    // print the error code if channel can't be read
correctly
 }
```

```
    val = analogRead(lightSensor);  // Read value of photoresistor
/*  Serial.print(val);
    Serial.print("\n");
  */

  if (val < 1010 && temperatureInC <= (ThresholdVal)) {     // Check if
LED is lit on BLE side (if HIGH analog value will be < 1010) AND
temperature is less than or equal to threshold value
    digitalWrite(ledG, HIGH); // turn Green LED on          // Turn on
Green LED
    lcd.setCursor(0,0);
    lcd.print("    FROST!!    ");                           // Print to
LCD - "FROST!!   Heating ON" across 2 lines
    lcd.setCursor(0,1);
    lcd.print("  Heating ON   ");
    (A = 1);                                               // Set A
(wakeword status) to 1
  }

  if (val < 1010 && temperatureInC > (ThresholdVal)) {          // check
if LED is lit on BLE side (if HIGH analog value will be < 1010) AND
temperature is greater than threashold value
    digitalWrite(ledR, HIGH); // turn Red LED on               // Turn on
Red LED
    lcd.setCursor(0,0);
    lcd.print("Temp is " + String(temperatureInC) + "deg");     // Print
to LCD - "Temp is [temperatureInC] deg No Heating" across 2 lines
    lcd.setCursor(0,1);
    lcd.print("  No Heating   ");
    (A = 0);                                                   // Set A
(wakeword status) to 0
  }

  if(millis()-start_time > 600000) {     // If internal timer passes 10
mins (600000ms)
   start_time = millis();
   A = 0;                                // Reset A to 0
   lcd.setCursor(0,0);
   lcd.print(" Program reset ");         // Print to LCD "Program Reset
Press Restart" across 2 lines
   lcd.setCursor(0,1);
   lcd.print(" Press Restart ");
   digitalWrite(ledG, LOW);              // Set Green LED to LOW
   digitalWrite(ledR, LOW);              // Set Red LED to LOW
   }

  Serial.print(A);       // Print A (wakeword status) to Serial Monitor
  Serial.print("\n");    // Take a new line

  ThingSpeak.setField(1,A);                                        //
Write Wakeword Status (A) to field 1 in Channel
  ThingSpeak.writeFields(wakewordChannelNumber, myWriteAPIKey);    //
Define Channel number and API Key to Write value to

  delay(1000);    // Delay for 1 sec to update serial monitor every
second and every 15th value is sent to ThingSpeak server

}
```

Figure 23:Control Panel Nano 33 BLE Sense Code

Model Settings

```
// Modified by: Jack Gibson for EEE526 Project
// Original file from (TinyML 2019)
// Description: This file defines Labels for the 4 possible audio
detections
//              (silence, unknown, on or go)
#include "micro_features_micro_model_settings.h"

const char* kCategoryLabels[kCategoryCount] = {
    "silence",
    "unknown",
    "on",
    "go",
};
```

Micro Features Model

```
// Modified by: Jack Gibson for EEE526 Project
// Original file from (TinyML 2019)
// Description: This file loads the model trained in google colab using
python script
// This is a standard TensorFlow Lite FlatBuffer model file that has been
// converted into a C data array, so it can be easily compiled into a
binary
// for devices that don't have a file system. It was created using the
command:
// xxd -i model.tflite > model.cc

#include "micro_features_model.h"

// We need to keep the data array aligned on some architectures.
#ifdef __has_attribute
#define HAVE_ATTRIBUTE(x) __has_attribute(x)
#else
#define HAVE_ATTRIBUTE(x) 0
#endif
#if HAVE_ATTRIBUTE(aligned) || (defined(__GNUC__) && !defined(__clang__))
#define DATA_ALIGN_ATTRIBUTE __attribute__((aligned(4)))
#else
#define DATA_ALIGN_ATTRIBUTE
#endif



const unsigned char g_model[] DATA_ALIGN_ATTRIBUTE = {
  0x20, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x00, 0x00,

/////////////      (1500 Line Model Continued)    ///////////////////////

  0x03, 0x00, 0x00, 0x00
};
const int g_model_len = 18712;
```

Command Responder

```
// Modified by: Jack Gibson for EEE526 Project
// Original file from (TinyML 2019)
// Description: This file defines output for Arduino if wakeword is
detected (ON,GO)

#if defined(ARDUINO) && !defined(ARDUINO_ARDUINO_NANO33BLE)
#define ARDUINO_EXCLUDE_CODE
#endif  // defined(ARDUINO) && !defined(ARDUINO_ARDUINO_NANO33BLE)
#ifndef ARDUINO_EXCLUDE_CODE
#include "command_responder.h"
#include "Arduino.h"

const int GREEN = 2;    // green external LED defined



// Toggles the built-in LED every inference, and lights a colored LED
depending
// on which word was detected.
void RespondToCommand(tflite::ErrorReporter* error_reporter,
                      int32_t current_time, const char* found_command,
                      uint8_t score, bool is_new_command) {
  static bool is_initialized = false;
  if (!is_initialized) {
    pinMode(LED_BUILTIN, OUTPUT);
    pinMode(GREEN, OUTPUT);
    // Pins for the built-in RGB LEDs on the Arduino Nano 33 BLE Sense
    pinMode(LEDR, OUTPUT);
    pinMode(LEDG, OUTPUT);
    pinMode(LEDB, OUTPUT);
    // Ensure the LED is off by default.
    // Note: The RGB LEDs on the Arduino Nano 33 BLE
    // Sense are on when the pin is LOW, off when HIGH.
    digitalWrite(LEDR, HIGH);
    digitalWrite(LEDG, HIGH);
    digitalWrite(LEDB, HIGH);
    digitalWrite(GREEN, LOW);

    is_initialized = true;
  }
  static int32_t last_command_time = 0;
  static int count = 0;
  static int certainty = 220;

  if (is_new_command) {
    TF_LITE_REPORT_ERROR(error_reporter, "Heard %s (%d) @%dms",
found_command,
                         score, current_time);
    // If we hear a command, light up the appropriate LED
    if (found_command[0] == 'o' ) {
      last_command_time = current_time;
      digitalWrite(LEDG, LOW);      // Green for "on"
      digitalWrite(GREEN, HIGH);
    }

 if (found_command[0] == 'g' ) {
      last_command_time = current_time;
      digitalWrite(LEDG, LOW);  // Green for "go"
      digitalWrite(GREEN, HIGH);
```

```
    }
    if (found_command[0] == 'u') {
      last_command_time = current_time;
      digitalWrite(LEDR, LOW);  // Red for unknown

    }
  }

  // If last_command_time is non-zero but was >10 seconds ago, zero it
  // and switch off the LED.
  if (last_command_time != 0) {
    if (last_command_time < (current_time - 10000)) {
      last_command_time = 0;
      digitalWrite(LED_BUILTIN, LOW);
      digitalWrite(LEDR, HIGH);
      digitalWrite(LEDG, HIGH);
      digitalWrite(LEDB, HIGH);
      digitalWrite(GREEN, LOW);
    }
    // If it is non-zero but <10 seconds ago, do nothing.
    return;
  }

  // Otherwise, toggle the LED every time an inference is performed.
  ++count;
  if (count & 1) {
    digitalWrite(LED_BUILTIN, HIGH);
  } else {
    digitalWrite(LED_BUILTIN, LOW);
  }
}

#endif  // ARDUINO_EXCLUDE_CODE
```

*Remaining 20 unmodified Arduino code files can be sourced from TensorFlow's GitHub Repository:

https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/micro/examples/micro_speech

References

TinyML [Book] / auth. Warden.P Situnayake.D. - CA : O'Reilly, 2019.

Figure 24: Vehicle Heating and Temperature Element Code

```
// Created : 10 / 10 / 2020
// Modified : 29/11/2020
// Written by: Jack Gibson for EEE527 Project
// Description: Code for Arduino Nano 33 IOT - Module in Car
//                                           - Reads Data from DHT and
logs to thingspeak server
//                                           - If Wakeword HIGH value (1)
read from thingspeak server turn DC Motor on
//                                           - Logs Value of DC Motor
ON(1) or OFF(0) to thingspeak server


#include <SPI.h>            // SPI library used to process data from DHT
sensor and send it over WiFiNINA
#include <WiFiNINA.h>       // WiFiNINA Library used to connect Nano 33
IOT to Wifi
#include<ThingSpeak.h>      // ThingSpeak used to analyse data



const char ssid[] = "BTBHub6-MXH2";        //  WiFi Name (ssid) defined
const char password[] = "mwYTE9Fxu9kc";    // Wifi Password defined
int status = WL_IDLE_STATUS;               // status initially defined as
idle (available for connection)
WiFiClient client;                         // WiFi client defined as
client for use in WiFiNINA library

int tempPin = A1;   // A1 defined as Vout pin of TEMP36GZ sensor

unsigned long CarHeaterChannelNumber =  1224137;   // set Channel ID for
Car Heater
const char * myWriteAPIKey = "1CAZ6CZYEX1GBLKY";   // Set Write API Key
for Car Heater channel

unsigned long wakewordChannelNumber = 1221517;     //Wakeword channel
number
unsigned int wakewordFieldNumber = 1;              // field 1 in channel
const char * myReadAPIKey = "IRXL27JIMVHMU1BB";    //API key for Wakeword
Channel

const int transistorPin = 9;    // transistor pin used for controlling DC
motor defined


void setup() {

  pinMode(transistorPin, OUTPUT); //Transistor used for running DC motor
defined as OUTPUT
  Serial.begin(9600);             // Baud rate defined as 9600

  while ( status != WL_CONNECTED) {
    Serial.print("Attempting to connect to WPA SSID: ");  // Attempt to
connect to Wifi network:
    Serial.println(ssid);

  status = WiFi.begin(ssid, password);   // Connect to WPA/WPA2 network:
  delay(10000);                          // Wait 10 seconds for
connection:
```

```
  ThingSpeak.begin(client);                                  // Begin
Thingspeak library
  analogReference(AR_DEFAULT);          // AREF has been set to default
3.3V
  analogReadResolution(8);              // ADC bits have been set to 8

}
}

void loop() {

 int statusCode = 0;     // Define initial statusCode to 0 to show correct
error code in debugging or display correct value from server

 int wakewordStatus = ThingSpeak.readIntField(wakewordChannelNumber,
wakewordFieldNumber);     //Wakeword Status defined by reading from
Wakeword Channel on Thingspeak server

 int Qlevel;                         // local int Qlevel defined
 float volt, temperature ;           // Local float value set up for Voltage
passing though Vs and Vout and calculated Temperature value


 statusCode = ThingSpeak.getLastReadStatus();       // Read the last value
sent to the server in the Wakeword Channel
 if(statusCode == 200){
   Serial.println("Wakeword Status is  " + String(wakewordStatus));    //
If Status code is 200, Print "Wakeword Status is (Wakeword value)"
 }
 else{
   Serial.println("Problem reading channel. HTTP error code " +
String(statusCode));  // print the error code if channel can't be read
correctly
 }


  int outputValueON = 200;           // Set speed of DC motor to 200 (ON)
  int outputValueOFF = 0;            // Set Speed of DC motor to 0 (OFF)
  int Z;                             // Set Z as integer to define output
state of motor to send to Thingspeak server

if (wakewordStatus == 1){
  analogWrite(transistorPin, outputValueON);     // If Wakeword Value is
HIGH (1) Set DC motor ON
  (Z = 1);                                       // Define int Z as 1
 }
 if (wakewordStatus == 0) {
  analogWrite(transistorPin, outputValueOFF);    // If Wakeword Value is
LOW (0) Set DC motor OFF
  (Z = 0);                                       // Define int Z as 0
 }



Qlevel = analogRead(tempPin);                   // Q level is read from
Vout of the Temp. sensor
volt = (( Qlevel * 3.3) / 255);                 // Voltage is calculated
by multiplying the Qlevel by the input voltage (3.3V) and dividing by the
number of levels for 8 bits
temperature = ((volt - 0.5)/0.01);              // Temperature defined by
simple algorithm derived from data sheet of TEP36QZ
```

```
Serial.print("Temperature value is:  ");       // Print Temperature value
is:
Serial.print(temperature);                     // Print the temperature
value
Serial.print('\n');                            // Take a new line



  Serial.print(" Motor Binary Val:  ");
  Serial.print(Z);                          //Print Binary Value of DC
Motor: (Z) and take new line
  Serial.print('\n');


ThingSpeak.setField(2,temperature);                                 // Write
Temperarture (t) to Field 2 in Channel
ThingSpeak.setField(3,Z);                                           // Write
Motor State  (Z) to Field 3 in Channel
ThingSpeak.writeFields(CarHeaterChannelNumber, myWriteAPIKey);  // Define
Channel number and API Key to Write values to

 delay(1000);    // Delay for 1 sec, Only need 1 new value for each time
the sever updates (every 15 or 20 seconds)

}
```

Figure 25: Python Code for Training Wakeword Model

Configure Defaults

MODIFY the following constants for your specific use case.

```
[ ]  # A comma-delimited list of the words you want to train for.
     # The options are: yes,no,up,down,left,right,on,off,stop,go
     # All the other words will be used to train an "unknown" label and silent
     # audio data with no spoken words will be used to train a "silence" label.
     WANTED_WORDS = "on,go"

     # The number of steps and learning rates can be specified as comma-separated
     # lists to define the rate at each stage. For example,
     # TRAINING_STEPS=12000,3000 and LEARNING_RATE=0.001,0.0001
     # will run 12,000 training loops in total, with a rate of 0.001 for the first
     # 8,000, and 0.0001 for the final 3,000.
     TRAINING_STEPS = "12000,3000"
     LEARNING_RATE = "0.001,0.0001"

     # Calculate the total number of steps, which is used to identify the checkpoint
     # file name.
     TOTAL_STEPS = str(sum(map(lambda string: int(string), TRAINING_STEPS.split(","))))

     # Print the configuration to confirm it
     print("Training these words: %s" % WANTED_WORDS)
     print("Training steps in each stage: %s" % TRAINING_STEPS)
     print("Learning rate in each stage: %s" % LEARNING_RATE)
     print("Total number of training steps: %s" % TOTAL_STEPS)
```

DO NOT MODIFY the following constants as they include filepaths used in this notebook and data that is shared during training and inference.

```
[ ]  # Calculate the percentage of 'silence' and 'unknown' training samples required
     # to ensure that we have equal number of samples for each label.
     number_of_labels = WANTED_WORDS.count(',') + 1
     number_of_total_labels = number_of_labels + 2 # for 'silence' and 'unknown' label
     equal_percentage_of_training_samples = int(100.0/(number_of_total_labels))
     SILENT_PERCENTAGE = equal_percentage_of_training_samples
     UNKNOWN_PERCENTAGE = equal_percentage_of_training_samples

     # Constants which are shared during training and inference
     PREPROCESS = 'micro'
     WINDOW_STRIDE = 20
     MODEL_ARCHITECTURE = 'tiny_conv' # Other options include: single_fc, conv,
                          # low_latency_conv, low_latency_svdf, tiny_embedding_conv

     # Constants used during training only
     VERBOSITY = 'WARN'
     EVAL_STEP_INTERVAL = '1000'
     SAVE_STEP_INTERVAL = '1000'

     # Constants for training directories and filepaths
     DATASET_DIR =  'dataset/'
     LOGS_DIR = 'logs/'
     TRAIN_DIR = 'train/' # for training checkpoints and other files.

     # Constants for inference directories and filepaths
     import os
     MODELS_DIR = 'models'
     if not os.path.exists(MODELS_DIR):
       os.mkdir(MODELS_DIR)
     MODEL_TF = os.path.join(MODELS_DIR, 'model.pb')
     MODEL_TFLITE = os.path.join(MODELS_DIR, 'model.tflite')
     FLOAT_MODEL_TFLITE = os.path.join(MODELS_DIR, 'float_model.tflite')
     MODEL_TFLITE_MICRO = os.path.join(MODELS_DIR, 'model.cc')
     SAVED_MODEL = os.path.join(MODELS_DIR, 'saved_model')

     QUANT_INPUT_MIN = 0.0
     QUANT_INPUT_MAX = 26.0
     QUANT_INPUT_RANGE = QUANT_INPUT_MAX - QUANT_INPUT_MIN
```

Setup Environment

Install Dependencies

```
[ ] %tensorflow_version 1.x
    import tensorflow as tf
```

DELETE any old data from previous runs

```
[ ] !rm -rf {DATASET_DIR} {LOGS_DIR} {TRAIN_DIR} {MODELS_DIR}
```

Clone the TensorFlow Github Repository, which contains the relevant code required to run this tutorial.

```
[ ] !git clone -q --depth 1 https://github.com/tensorflow/tensorflow
```

Load TensorBoard to visualize the accuracy and loss as training proceeds.

```
[ ] %load_ext tensorboard
    %tensorboard --logdir {LOGS_DIR}
```

Training

The following script downloads the dataset and begin training.

```
[ ] !python tensorflow/tensorflow/examples/speech_commands/train.py \
    --data_dir={DATASET_DIR} \
    --wanted_words={WANTED_WORDS} \
    --silence_percentage={SILENT_PERCENTAGE} \
    --unknown_percentage={UNKNOWN_PERCENTAGE} \
    --preprocess={PREPROCESS} \
    --window_stride={WINDOW_STRIDE} \
    --model_architecture={MODEL_ARCHITECTURE} \
    --how_many_training_steps={TRAINING_STEPS} \
    --learning_rate={LEARNING_RATE} \
    --train_dir={TRAIN_DIR} \
    --summaries_dir={LOGS_DIR} \
    --verbosity={VERBOSITY} \
    --eval_step_interval={EVAL_STEP_INTERVAL} \
    --save_step_interval={SAVE_STEP_INTERVAL}
```

Generate a TensorFlow Model for Inference

Combine relevant training results (graph, weights, etc) into a single file for inference. This process is known as freezing a model and the resulting model is known as a frozen model/graph, as it cannot be further re-trained after this process.

```
[ ] !rm -rf {SAVED_MODEL}
    !python tensorflow/tensorflow/examples/speech_commands/freeze.py \
    --wanted_words=$WANTED_WORDS \
    --window_stride_ms=$WINDOW_STRIDE \
    --preprocess=$PREPROCESS \
    --model_architecture=$MODEL_ARCHITECTURE \
    --start_checkpoint=$TRAIN_DIR$MODEL_ARCHITECTURE'.ckpt-'{TOTAL_STEPS} \
    --save_format=saved_model \
    --output_file={SAVED_MODEL}
```

Generate a TensorFlow Lite Model

Convert the frozen graph into a TensorFlow Lite model, which is fully quantized for use with embedded devices.

The following cell will also print the model size, which will be under 20 kilobytes.

```
[ ] import sys
    # We add this path so we can import the speech processing modules.
    sys.path.append("/content/tensorflow/tensorflow/examples/speech_commands/")
    import input_data
    import models
    import numpy as np
```

```
[ ] SAMPLE_RATE = 16000
    CLIP_DURATION_MS = 1000
    WINDOW_SIZE_MS = 30.0
    FEATURE_BIN_COUNT = 40
    BACKGROUND_FREQUENCY = 0.8
    BACKGROUND_VOLUME_RANGE = 0.1
    TIME_SHIFT_MS = 100.0

    DATA_URL = 'https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.02.tar.gz'
    VALIDATION_PERCENTAGE = 10
    TESTING_PERCENTAGE = 10
```

```
[ ] model_settings = models.prepare_model_settings(
        len(input_data.prepare_words_list(WANTED_WORDS.split(','))),
        SAMPLE_RATE, CLIP_DURATION_MS, WINDOW_SIZE_MS,
        WINDOW_STRIDE, FEATURE_BIN_COUNT, PREPROCESS)
    audio_processor = input_data.AudioProcessor(
        DATA_URL, DATASET_DIR,
        SILENT_PERCENTAGE, UNKNOWN_PERCENTAGE,
        WANTED_WORDS.split(','), VALIDATION_PERCENTAGE,
        TESTING_PERCENTAGE, model_settings, LOGS_DIR)
```

```
[ ]  with tf.Session() as sess:
       float_converter = tf.lite.TFLiteConverter.from_saved_model(SAVED_MODEL)
       float_tflite_model = float_converter.convert()
       float_tflite_model_size = open(FLOAT_MODEL_TFLITE, "wb").write(float_tflite_model)
       print("Float model is %d bytes" % float_tflite_model_size)

       converter = tf.lite.TFLiteConverter.from_saved_model(SAVED_MODEL)
       converter.optimizations = [tf.lite.Optimize.DEFAULT]
       converter.inference_input_type = tf.lite.constants.INT8
       converter.inference_output_type = tf.lite.constants.INT8
       def representative_dataset_gen():
         for i in range(100):
           data, _ = audio_processor.get_data(1, i*1, model_settings,
                                              BACKGROUND_FREQUENCY,
                                              BACKGROUND_VOLUME_RANGE,
                                              TIME_SHIFT_MS,
                                              'testing',
                                              sess)
           flattened_data = np.array(data.flatten(), dtype=np.float32).reshape(1, 1960)
           yield [flattened_data]
       converter.representative_dataset = representative_dataset_gen
       tflite_model = converter.convert()
       tflite_model_size = open(MODEL_TFLITE, "wb").write(tflite_model)
       print("Quantized model is %d bytes" % tflite_model_size)
```

Testing the TensorFlow Lite model's accuracy

Verify that the model we've exported is still accurate, using the TF Lite Python API and our test set.

```
[ ]  # Helper function to run inference
     def run_tflite_inference(tflite_model_path, model_type="Float"):
       # Load test data
       np.random.seed(0) # set random seed for reproducible test results.
       with tf.Session() as sess:
         test_data, test_labels = audio_processor.get_data(
             -1, 0, model_settings, BACKGROUND_FREQUENCY, BACKGROUND_VOLUME_RANGE,
             TIME_SHIFT_MS, 'testing', sess)
       test_data = np.expand_dims(test_data, axis=1).astype(np.float32)

       # Initialize the interpreter
       interpreter = tf.lite.Interpreter(tflite_model_path)
       interpreter.allocate_tensors()

       input_details = interpreter.get_input_details()[0]
       output_details = interpreter.get_output_details()[0]

       # For quantized models, manually quantize the input data from float to integer
       if model_type == "Quantized":
         input_scale, input_zero_point = input_details["quantization"]
         test_data = test_data / input_scale + input_zero_point
         test_data = test_data.astype(input_details["dtype"])

       correct_predictions = 0
       for i in range(len(test_data)):
         interpreter.set_tensor(input_details["index"], test_data[i])
         interpreter.invoke()
         output = interpreter.get_tensor(output_details["index"])[0]
         top_prediction = output.argmax()
         correct_predictions += (top_prediction == test_labels[i])

       print('%s model accuracy is %f%% (Number of test samples=%d)' % (
           model_type, (correct_predictions * 100) / len(test_data), len(test_data)))
```

```
[ ]  # Compute float model accuracy
     run_tflite_inference(FLOAT_MODEL_TFLITE)

     # Compute quantized model accuracy
     run_tflite_inference(MODEL_TFLITE, model_type='Quantized')
```

Generate a TensorFlow Lite for MicroControllers Model

Convert the TensorFlow Lite model into a C source file that can be loaded by TensorFlow Lite for Microcontrollers.

```
[ ]  # Install xxd if it is not available
     !apt-get update && apt-get -qq install xxd
     # Convert to a C source file
     !xxd -i {MODEL_TFLITE} > {MODEL_TFLITE_MICRO}
     # Update variable names
     REPLACE_TEXT = MODEL_TFLITE.replace('/', '_').replace('.', '_')
     !sed -i 's/'{REPLACE_TEXT}'/g_model/g' {MODEL_TFLITE_MICRO}
```

Deploy to a Microcontroller

Follow the instructions in the micro_speech README.md for TensorFlow Lite for MicroControllers to deploy this model on a specific microcontroller.

Reference Model: If you have not modified this notebook, you can follow the instructions as is, to deploy the model. Refer to the `micro_speech/train/models` directory to access the models generated in this notebook.

New Model: If you have generated a new model to identify different words: (i) Update `kCategoryCount` and `kCategoryLabels` in `micro_speech/micro_features/micro_model_settings.h` and (ii) Update the values assigned to the variables defined in `micro_speech/micro_features/model.cc` with values displayed after running the following cell.

```
[ ]  # Print the C source file
     !cat {MODEL_TFLITE_MICRO}
```

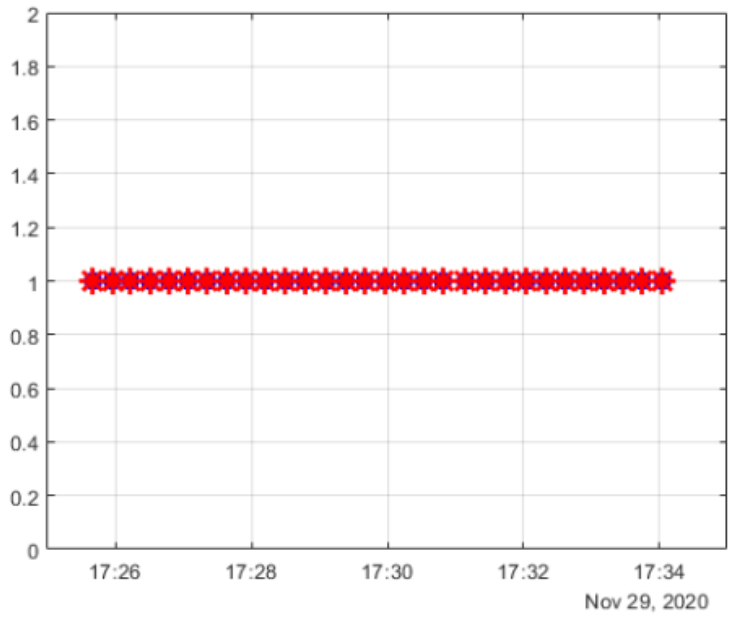
Figure 26: MATLAB code used for data analysis of Motor Running Time

```
readChannelID = 1224137;
fieldID1 = 3;

%% Read Data %%
[data, time] = thingSpeakRead(readChannelID, 'Field', fieldID1, 'NumPoints',
30, 'ReadKey', readAPIKey);

%% Visualize Data %%
plot(time, data, 'bo-', 'LineWidth', 2, 'MarkerSize', 7);
grid on;
% threshold defined.
threshold = 0.5;
moreThanThreshold = data > threshold; % Logical indexes.
% Extract those over the threshold into new arrays.
over_x = time(moreThanThreshold);
over_y = data(moreThanThreshold);
% plot over with red stars over the first set of points.
hold on;
plot(over_x, over_y, 'r*', 'LineWidth', 2, 'MarkerSize', 13);

num = sum(over_y) % sum of points over threshold i.e.=1
TimeOnHour = (num * 20) /3600 % number of points * average time between
points, convert from secs to hours
AvgFuelConsumption = 0.5; % Liters per Hour
Fuelconsumed_L = TimeOnHour * AvgFuelConsumption
Fuelconsumed_ml = Fuelconsumed_L*1000
```




```
num = 30
TimeOnHour = 0.1667

Fuelconsumed_L = 0.0833
Fuelconsumed_ml = 83.3333
```