IoT Assignment 2 Group Report
SWE30011

IoT- Internet of Things

Team Members:
Thomas Wright (101470604)
Saud Alshamsi (101034097)
Apostolos Lafazanis (101360815)
Vasilios (Billy) Dasopatis (101156315)

Introduction

IoT Assignment 2 Group Report
SWE30011

For this group assignment, our team had to create a Smart Fan and a Memory Game using an Arduino and a Raspberry Pi board. To begin with, the Fan had to be connected to the Arduino and powered on and off based on a temperature collected from an Arduino temperature sensor. Then, the data collected from the Arduino would be sent over to a database to be stored so that we can properly utilize them. Furthermore, a website had to be created in which all the relevant fan information would be displayed and from which the user could manually set the temperature threshold that would determine whether the fan would be on or off.
In regard to the memory game, we had to develop this fun Arduino system the would produce a sound using a buzzer hooked onto the board and then the user would have to enter the number of beeps using an IR Remote. The user would also have a unique ID that would distinguish them from other players. The difficulty of the game would increase based on the number of correct answers the player has given (3 correct answers = +1 level), the correct answer would be displayed in a 7 segment display unit and the states of the game using LED lights. Finally, all the necessary data would be sent to a database and a web page hosted on the Raspberry Pi would neatly display the info on a table.

In this report, we will discuss our approach in developing the aforementioned systems, the challenges we faced and the issues we have encountered as well as the software and hardware tools we deployed for the creation of the project.

Hardware Used In Our Assignment:

Please note the following Product ID'S are from the vendor JAYCAR unless stated otherwise.

1. Arduino Uno R3 Board

The Arduino UNO is a low power microcontroller based on the ATmega32P8p. The role of the Arduino in our project was to manage the collection of data from the external sensors & transmit it back to our server node.

The pins on the Arduino were used as follows:

Analog
- 0: Temperature sensor input

Digital:
- 2: Buzzer output
- 3: IR sensor input
- 4: Led strip output

USB Type-B port:
- Serial Communication to Raspberry Pi

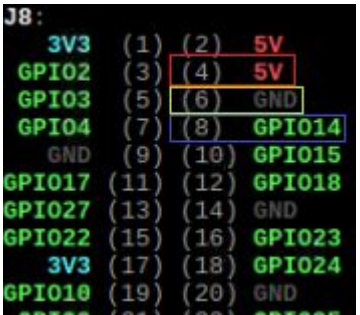
5v Pin
- Power supply to external sensors

Ground Pin
- Grounds external Sensors

2. Raspberry Pi 3B Board (Product ID: XC9000)

Digital:

Pin 4: 5V



Pin 6: Groud
Pin 8: GPIO 14

Pin 4 was used for the 5V power to power the fan, This was directly connected to the fan we had, this gave it sufficient power to turn on and spin

Pin 6: This pin was used to ground the left side of a transistor. Which when the transistor was sent a signal would ground the fan and complete the circuit thus tuning the fan on (like a light switch system)

Pin 8: which was the GPIO pin was connected to the middle rail of the transistor, this was used to turn the transistor ON or OFF which would turn the fan ON or OFF

3. Male to Male Cables (Product ID: WC6024)

We used Male to Male (M/M) cables to create the connection between the Arduino and the breadboard.

4. Male to Female Cables (Product ID: WC6026)

Those cables were used to connect the Raspberry Pi to the transistor which is used to control the fan.

5. Fan (5V) (Product ID: YX2500)

A really small fan that runs on a 5V power rail. This fan is ideal for this project since the raspberry pi can only provide current up to 5V a perfect amount for the fan…
This means we don't need any external batteries and keeps this whole project internal without having to replace any batteries in the long run

6. Temperature Sensor Module (Product ID: XC4494)

This temperature sensor is used to get the temperature in the room. It is comprised of 3 socket pins. The VCC pin from which it gets its power to run, the GND pin which grounds the circuit and the analog output pin from which we receive the temperature on the Arduino. The following function is used to convert the raw analog output (Calvin) to a double representing the temperature in degrees Celsius.

```
void getTemp()
{
  TEMP = analogRead(A0);
  TEMP = log(((10240000 / TEMP) - 10000));
  TEMP = 1 / (0.001129148 + (0.000234125 + (0.0000000876741 * TEMP * TEMP )) * TEMP);
  TEMP = TEMP - 273.15; // Convert Kelvin to Celcius
}
```

7. IR Sensor (Product ID: XC4427)

The IR sensor was used to receive user input via an infrared signal sent by an IR remote control.
The sensor has a ground, a 5v power in, and a digital output pin from which the data is read.

8. Mini IR Remote Controller (Product ID: XC3718)

The mini IR remote contains an infrared transmitter that is used to transmit an IR signal to the XC4427 IR receiver. The message encoded in the IR signal is determined by which button is pressed by the user.

9. 3-16V Piezoelectric buzzer (Prdocut ID: AB3462)

The buzzer is used to create a chain of sounds in quick succession for the user to count as part of the buzzer game.
The buzzer is switched on & off via the output of a 3v gpio pin.

10. 2N7000 N-Channel FET Transistor (Product ID: ZT2400

We got online and got this pretty simple n-FET transistor to serve as a switch for our fan. This transistor has 3 pins - the connector, base, and the emitter. The connector is used to supply power the base is to connect with the Raspberry Pi so that we can control the fan and the emitter is where the ground is.

11. RGB LED 8 Pixel Strip (Product ID: XC4380)

The 8 led strip was used in conjunction with the Adafruit_Neopixel library to display the current state of the buzzer game to the player.

12. Breadboard (Product ID: PB8815)

This is where we wired up all the circuitry, we kept it all on one board since this board is long, it makes transporting the product a lot easier. We used this board to also ground multiple circuits as well as share power with our power rails.

13. USB Cable A-B for Arduino (Product ID: FIT0056)

This cable was used to connect the Arduino with the Raspberry Pi so that the two units can communicate with each other using a serial interface.

Software/Languages:

1) Arduino (For the Arduino Board)

The Arduino UNO was exclusively programmed using the IDE & language.

2) Python

Python was used by the Raspberry Pi to retrieve and parse data sent by the Arduino from the serial buffer.
The parsed data decides which MySQL commands are executed by the pi using python scripts.

3) MySQL

As our project utilized a MariaDB database MySQL was used to alter /input/update/ delete data in the database that had to be altered. This was done using MySQL queries that were being executed from both Python scripts and PHP code.

4) Bash script

A simple 5 line bash script was used to run the two python scripts on a delayed loop. We could have set it up in our Python scripts but then we would have to manually start each python file every time, we decided to you a bash script centralize the process. One script ran a FULL system.

5) HTML5

HTML5 was used to build the web pages, this was paired with PHP / ajax.

Index.html - This page is just the index page that displays the welcoming information as well as group members and links to the two projects.

6) PHP

PHP was used to display data from the database as well as set the fan temperature.

FanStatus.php - This file retrieves the fan status from the fan status table and echo's it out (called by Ajax for dynamic page updates)

Weather.php - This page simply echo's out the current temperature, it is a bar chart with the temperature being written in the bar chart.

CurrentTemp.php - This file is similar to FanStatus where it retrieves the current temperature from the database we set and then echo's it out (called by Ajax for dynamic page updates)

MemoryGame.php - This is the main file for displaying the memory games page. It writes out a title then connects to the database where it echo's out all the rows in the database and organizes them neatly into a table.

fanProcess.php - This page is a big page. Firstly it validates the input from the FanTemp form field (setFan.php)
Table Format [Current Temperature | Temperature Limit | Fan Status]
The Page Runs a query to get the Temperature Limit which is set in a database. Before it prints out the Temperature limit in the table it makes a BLANK table cell which is a holder for the Current Temperature (ajax used to update)
It then prints out the Temperature Limit and lastly it prints out another BLANK table cell which is used for the Fan Status

setFan.php - This is where the user can set the fan Limit. This page makes a form where the can enter the fan temperature limit and is then processed by fanProcess.php

DataBaseCon.inc - This file contains all the database connection information and uses multiple times in different PHP files to connect to the database.

7) AJAX

Ajax is used to update the Fan Status as well as the Current Temperature, it simply points to the TWO BLANK table cells set up by fanProcess.php, runs both FanStatus.php and CurrentTemp.php, gets the output of these two files and places them in the correct cell, this is done every half a second for best accuracy.

8) CSS

Styling language is used to improve the aesthetics and clarity of the HTML structure of the webpage. There were some CSS files with basic CSS to make the pages look better than just plain black and white. In addition, there was some bootstrap framework used on multiple pages, such as the index page, to make them look modern.

```
<div class="jumbotron jumbotron-fluid">
  <div class="container">
    <h1 class="display-4">Group Assignment</h1>
    <p class="lead">This page contains two project smart fan and memory game</p>
  </div>
</div>
```

9) MariaDB DB Host

Open Source MySQL fork used to store data in relational tables.

We had 4 tables as follows
fan status - This was to store the current status of the fan (On or Off)
TempLimit - This was to set the fan limit in Celcius between -99 to 99 Celcius.

Players - this was for the player's data, it would contain the player id a 4 digit long number and the players current level.
temperatureDB - This stored the temperature from the Temperature Sensor.

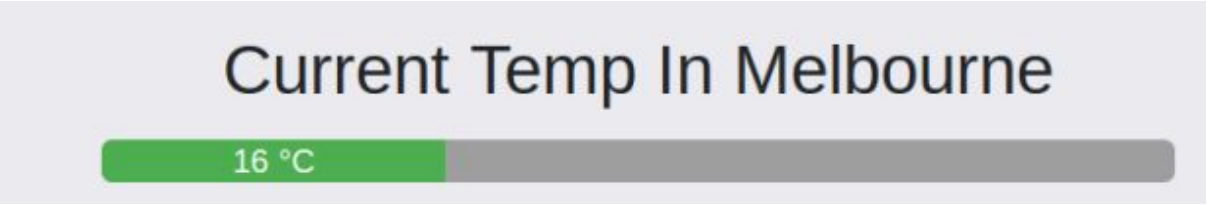
```
MariaDB [assignment2]> show tables;
+----------------------+
| Tables_in_assignment2 |
+----------------------+
| FanStatus            |
| TempLimit            |
| players              |
| temperatureDB        |
+----------------------+
4 rows in set (0.01 sec)

MariaDB [assignment2]> describe FanStatus;
+------------+------------+------+-----+---------+-------+
| Field      | Type       | Null | Key | Default | Extra |
+------------+------------+------+-----+---------+-------+
| id         | int(1)     | NO   | PRI | NULL    |       |
| FanOnOrOff | varchar(3) | YES  |     | NULL    |       |
+------------+------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

MariaDB [assignment2]> describe TempLimit;
+-----------+--------+------+-----+---------+----------------+
| Field     | Type   | Null | Key | Default | Extra          |
+-----------+--------+------+-----+---------+----------------+
| temp_id   | int(5) | NO   | PRI | NULL    | auto_increment |
| TempLimit | int(2) | YES  |     | NULL    |                |
+-----------+--------+------+-----+---------+----------------+
2 rows in set (0.00 sec)

MariaDB [assignment2]> describe players;
+--------------+---------+------+-----+---------+-------+
| Field        | Type    | Null | Key | Default | Extra |
+--------------+---------+------+-----+---------+-------+
| player_id    | char(4) | NO   | PRI | NULL    |       |
| current_level | char(1) | YES  |     | NULL    |       |
+--------------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)

MariaDB [assignment2]> describe temperatureDB;
+-------+------------+------+-----+---------+----------------+
| Field | Type       | Null | Key | Default | Extra          |
+-------+------------+------+-----+---------+----------------+
| id    | int(7)     | NO   | PRI | NULL    | auto_increment |
| Temp  | varchar(7) | NO   |     | NULL    |                |
+-------+------------+------+-----+---------+----------------+
2 rows in set (0.01 sec)
```

10) Apache2 Web Server

Quite simply the web host we used to host the web server. We redirected the base website from /var/www/html to /var/www/iot to make the file directory a little more obvious.

11) API

We used a “OpenWeatherMaps” API to display the current temperature in Melbourne This requires a simple registration and creation of an API token to be able to display the weather



Overview of the Smart Fan Project

So we specified the hardware we used to create the circuit and the software but how did we manage to communicate between the Raspberry Pi and the Arduino and managed to collect all the info from the sensor, store it in a database and in general establish that ongoing communication?

Well, let's take things one at a time.

Sensor Node, Raspberry Pi and Database Communication

To begin with, the temperature sensor on the Arduino collects the temperature data and sends it over through a cable to one of the USB ports on the Pi. Then inside a communication file, we wrote in Python called **comms.py**, we open a serial connection to utilize that same temperature data. we, later on, create the tables in which all of our data will be stored using a few MySQL queries. For the fan project, we created 3 tables: the First one is for the temperature limit, second for the temperature collected from the sensor and the third and final one for the status of the fan.

Now that the tables are created in the database we are able to start filling them up with useful data. So what we are doing next is we take the temperature we got from the serial connection and insert it into the temperature table, successfully establishing communication between the sensor, the Raspberry by and the database.

Web server and web application for the Fan

Firstly we installed a few required programs.

1) MariaDB

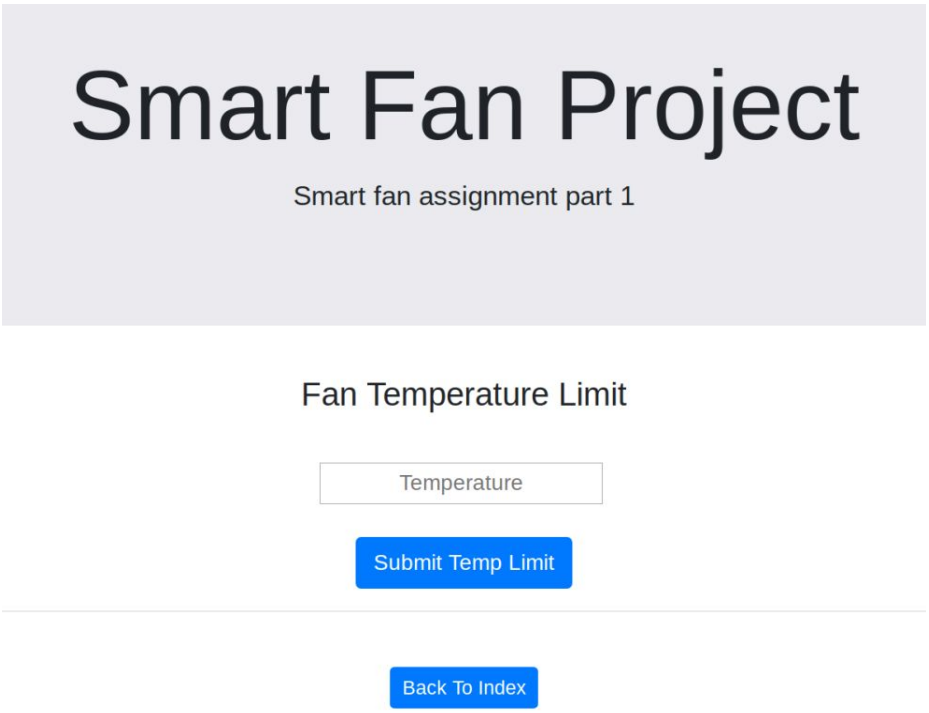
2) Apache2
3) Php7

Then, what we did is we went into the configuration of the Apache2 server and set the default web location from /var/www/html to /var/www/iot to add consistency to the file names so we don't get lost when looking for folders.

So the temperature is now in the database but we are still missing on the Temperature Limit value. This value is set by the user using a form that is located in the **setFan.php**. Once the user fills out the form and clicks the Submit Temp Limit button the values are submitted over to another file (**fanProcess.php** file) in which it will be inserted into the TempLimit table we have created in while ago using **MySQL update queries**. This table is **updated** every time you set a new temp limit **instead of adding a new row to the table** this is to reduce space space used in our Arduino and to stop useless data being saved.
It is worth mentioning that (because we have to access the database frequently in order to get or update data) the details of the connection to the database are stored in a file called **DataBaseCon.inc** which we include at the top of a file every time we want to make a connection. That makes our code less repetitive and more coherent adding modularity to our PHP system, if we never need to change databases or hosts its quite easy to do so, all we do is modify this one singular file and everything will automatically be created and inserted again.

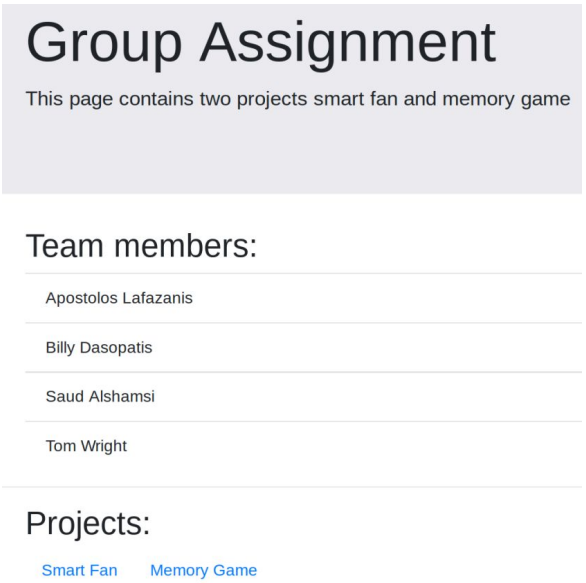
The **fanProcess.php** file is not only responsible for updating the temperature limit of the fan but also responsible for displaying all the relevant info (temperature of the sensor, state of the fan and temperature limit) on the webpage along with this the weather API which we are going to talk about later on.
Displaying the current temperature and the fan status is achieved through ajax with calls 2 separate PHP file (**CurrentTemp.php** and **FanStatus.php**). These 2 files are also mentioned and used in the **ajax.js** file which links them to the **fanProcess.php** and then refreshes them every half a second so the website can display up to date and accurate information.

The photo below depicts the temperature limit input box in which the user will put the desired temperature threshold.



The way someone can access that part of the site is through the home page which is contained in the index.html file. That page only contains the names and IDs of the people in our group and links to both of our projects (clicking the link that says Smart Fan will load the image above).

Here is an image of the home page below (index.html).



We also added a couple of CSS stylesheets to style each of our pages and tables. A CSS file called **index.css** was used to style the home page of our website, **setFan.css** to style the **setFan.php** page in the picture above and finally the **tables.css** file to style all the tables in both projects

Database, Raspberry Pi and Sensor Node (fan) Communication

Now that we've explained how we get the current temperature and set the temperature threshold, the only thing left to do is to process them and determine whether we want the fan to be on or off. This all takes place inside the **WebServerFanControll.py**

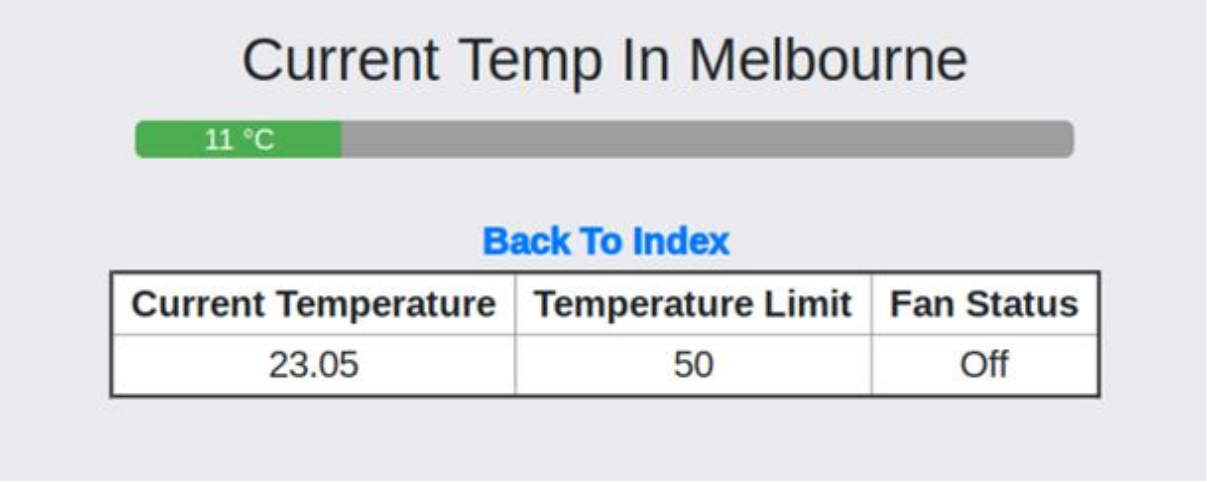
At the top of the file, we declare the GPIO pin of the Raspberry that is connected to the receiving part of the transistor which is connected to the fan. Then we get the values of the current temperature and the temperature limit from the database using MySQL queries so that we can compare them to each other. However, before we begin with the comparison it is worth noting that all data has been stripped off of any special characters like parentheses or commas or quotes with the use of certain functions and typecasted into integer or float. Continuing, we simply wrote an if statement that says: If the Current temperature is greater than the temperature limit then set the pin to HIGH which turns on the fan and set the value of the fan status in the database to ON, else set the pin to LOW which turns of the fan off and then set value of the fan status in the database to OFF. Finally, all that's left to do is to upload that fan status value to the database using a MySQL query so that it can be displayed when we load the website.

Please note we thought it would be better to connect the fan to the Arduino because we wanted to reduce the amount of communication via serial as you can drop bits/characters when communicating via serial. We currently have a single direction communication to reduce these issues

API communication Smart Fan

The API code we used for the current temperature is located in the **Weather.php** file and it is used on our website to display the temperature around Melbourne at the moment. So the way we went about was to first find a free API from the internet in the form of a URL.
We used Open Weather Maps and their API Key system.
This was in the form of JSON so we had to decode it by using a decode json function. After doing that we were now able to access the temperature and assign it to a variable which we could now use to display the temperature. We decided to display it as a temperature bar that takes a max value of 50 degrees Celsius right above the table where we display the fan information on the website.

The image below shows the actual bar:



Buzzer Game Overview

The buzzer game is a simple guessing game in which a user listens to a number of high pitch sounds from an electric buzzer & inputs the number of buzzes they heard using a mini-IR remote.
After three correct guesses, the user progresses a level & the difficulty of the game is increased by reducing the duration & pause between buzzes, such that they progressively increase in speed as the user progresses through the game.

Receiving Valid IR signals

Starting the buzzer game requires a signal that corresponds to the play/pause button on the IR remote. Rather than write code to interpret the raw IR input from the IR sensor the IRremote library was included to simplify the process. The IRremote library defines an 'IRrecv' object that contains methods to convert the raw IR input to a hexadecimal number, and a custom data type 'decode_results' that holds the raw IR input.

There are several main difficulties with this section of the task:

1. The IR sensor picks up a large amount of interference from its environment, effectively disrupting the valid signals being sent from the user
2. The system must **wait** for user input so that the results of the game are from the user alone
3. The IR signal receives requires representation inside the code that allows the developer to understand what values are expected, & easily convert the hex numbers into an int that can be used.

To eliminate IR interference a function 'waitForIRSignal()' is used. The function loops until a signal is received that corresponds to the exact IR output of one of the mini IR remote buttons. This solves the first two issues, as the program cannot progress until it receives valid user input.
The third issue was solved using a simple enumeration.

```
enum IR_SIGNAL {
  Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine,
  CHplus, CH, CHminus, PREV, NEXT, VOLdown, VOLup, EQ, HundredPlus, TwoHundredPlus, PLAYPAUSE
};
```

By starting from 0, the first 10 elements of the enumeration directly map onto the integer value displayed on the IR remote. This was worked into the waitForIRSignal function by making the function return an enumerated value that corresponds to the button pressed by the user:

```
switch (results.value)
{
  case 0xFF6897: return Zero;
  case 0xFF30CF: return One;
  case 0xFF18E7: return Two;
  case 0xFF7A85: return Three;
  case 0xFF10EF: return Four;
  case 0xFF38C7: return Five;
  case 0xFF5AA5: return Six;
  case 0xFF42BD: return Seven;
  case 0xFF4AB5: return Eight;
  case 0xFF52AD: return Nine;
  case 0xFFA25D: return CHplus;
  case 0xFF269D: return CH;
  case 0xFFE21D: return CHminus;
  case 0xFF22DD: return PREV;
  case 0xFF02FD: return NEXT;
  case 0xFFE01F: return VOLdown;
  case 0xFFA857: return VOLup;
  case 0xFF906F: return EQ;
  case 0xFF9867: return HundredPlus;
  case 0xFFB04F: return TwoHundredPlus;
  case 0xFFC23D: return PLAYPAUSE;
}
```

As a result of the main loop simply grabs input from the user and starts the game if it receives an enumeration equal to play/pause

```
USER_INPUT = waitForIRSignal(); // NO Code should be executed without a form of user input
void messageRecieved(); // Flash led to indicate valid IR signal received

switch (USER_INPUT)
{
  case PLAYPAUSE: // Player presses play/pause to start game
    // Serial.println('GAME INITIATED');
    setUserID();
    startGame();
    break;
}
```

Led feedback to the player

The manual for the XC4380 led strip recommends the Adafruit_Neopixel library for controlling the activation & of the LEDs. This simply requires you to create an Adafruit_Neopixel object with parameters that match the led you to have & then the Led's can be controlled with 3 simple library methods.

```
pixels = Adafruit_NeoPixel(ledCount, ledPin, NEO_RGB + NEO_KHZ800);
```

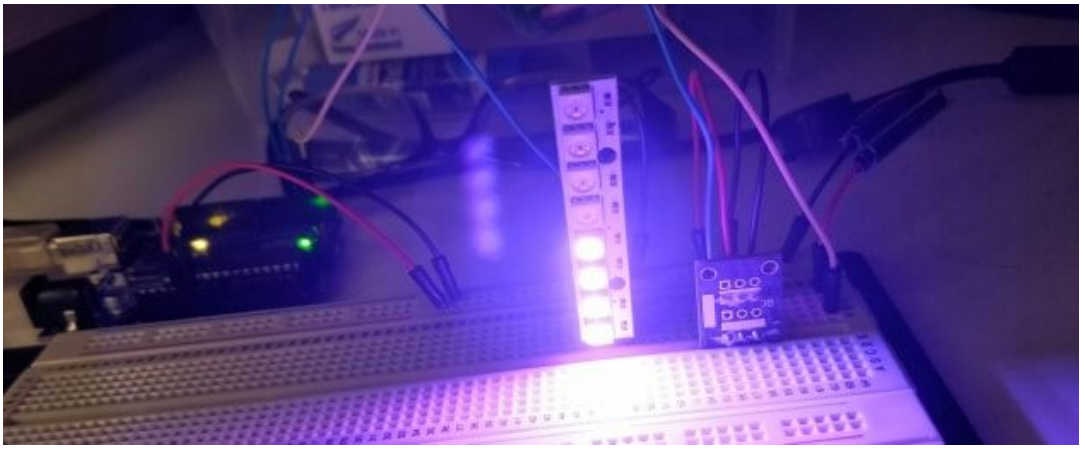
Pixels.Clear() is used to reset/turn off the LEDs on the strip. Pixels.setPixelColor() takes in the corresponding led you to wish to change & an RGB value and sets the pixel to display that color. Pixels.Show() turns on the led.

Very quickly we realized that coding sequences of flashing lights took up a large chunk of code space in our main file, and thus decided to store them within a custom class so that the methods no longer took up space in our Main.ino.

For example, to display the user's current level in our playLevel() function we call the led.displaylevel() function.

```
bool playLevel()
{
  bool GAME_OVER = false;
  // Level 1 = 1 full second, level 2 = 0.5 seconds etc...
  int msDelay = 1000 / LEVEL;
  int msLength = 1000 / (SCORE + LEVEL);
  int noOfBuzzes = 0;

  for (int i = 0; i < 3 && !GAME_OVER; i++)
  {
    leds.showLevel(LEVEL);
    updateThread.check(); // Check if its time to fire Thread function
```

This method simply takes in the global variable storing the users level and lights up LEDs from the bottom to the top.
For example when the users level is 4 the LEDs display four lights as such:



Setting the user ID

Before the player can begin they must have a user id by which they can be identified when their high score is submitted to the database.
The user ID is a global string variable that is initialized as empty at the start of the program & at the end of each game. It is only ever set whilst a game is in progress.

It is set by calling a loop four times (once for each character) that waits for the player to enter a 4 digit pin. The player is only allowed to use the first 10 digits on the IR remote so that the user id size remains constant.

Playing the game

The buzzer is connected to GPIO pin 2 of the Arduino Uno. It is activated via the buzz() method. This method takes in a random number of buzzes & activates the buzzer that many times. The length of the buzz & the delay between buzzes is equal to a second divided by the Level that the player is on. Therefore at level 1, the buzzes will be a full second, at level two they will be ½ a second, at 3 ⅓ of a second etc…

The number of buzzes is stored and compared to the users score. If the user guesses correctly their score increases and they continue, otherwise the LEDs' flash red and the game ends.

```
int noOfBuzzes = random(1, 9); // Give a random no. of tones to guess
buzz(noOfBuzz); // Play the tones for the player

// Serial.println("Buzzes: " + String(noOfBuzzes));
IR_SIGNAL guess = waitForIRSignal();

// Serial.println(guess);

if (guess == noOfBuzzes) // Get the user input == difficulty)
{
  leds.messageRecieved();
  SCORE++;
}
```

Serial Communication with Raspberry Pi

The USB type B connection between the Arduino & raspberry pi allowed for very simple one-way serial communication between the two devices.
The Serial.print() method is all that is required to transmit data over the serial connection from the Arduino:

```
// Functions for a timed action must be defined BEFORE the timed action variable
void transmitData()
{

  getTemp(); // Make sure we're transmitting latest data
  TEMP = 20.25;
  Serial.print(String(TEMP).length());
  Serial.print(TEMP);d

  if (USER_ID.length() == 4) // If we have a current player, update their score
  {
    Serial.print(1);
    Serial.print(USER_ID);
    Serial.print(LEVEL);
  }
  else
  {
    Serial.print(0); // If we dont have a player let the pi know
  }

  Serial.println(); // EOL or '\n' indicates to pi to stop reading
}
```

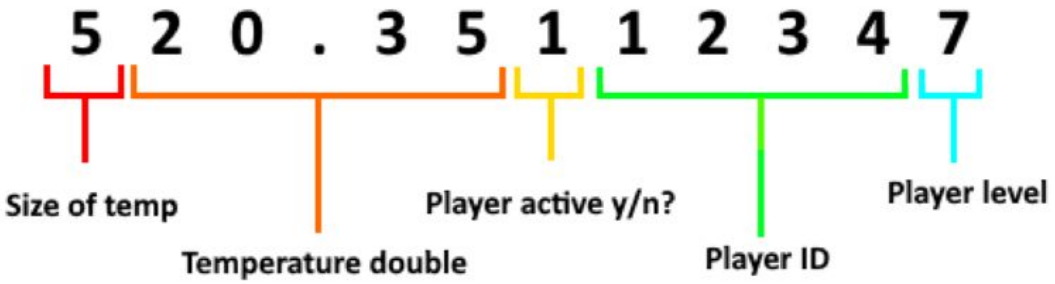
To receive the data on the raspberry pi, a very simple python script opens a connection to the serial port and then reads a line from the serial buffer.

```
# Serial commmunication
ser = serial.Serial('/dev/ttyACM0', 9600)
message = ser.readline()
```

The key pieces of information the arduino needed to transmit were:

- Temperature data
- The current player's ID
- The current player's score

To transmit these pieces of information the following format was chosen:



In the above example, the pi is told they are receiving a temperature value that is 5 characters long, the temperature is 20.35, that a player is active, the players ID is 1234 & their level is 7.

The method for transmitting the data is called within another method called readData(). This method is currently unfinished & only fulfills the role of calling the transmitData() method.
The initial goal was that after the player entered their ID the Arduino would transmit the players ID to the Raspberry Pi, & the raspberry pi would transmit back whether the player had an existing ID in the database, & if they did the raspberry pi would transmit their level back so that the player could resume their game.

The raspberry pi reads this into a string & breaks the string into parts & stores them as variables using substrings.

As mentioned earlier loops are used often in the arduino code & the device waits often for user input. In order to ensure that data is set at **regular intervals** arduino's proto threading is used.

Protothreading is when the Arduino executes a block of code on a regular interval, & uses checks inside the programs loop to check if it is time for the code block to fire again.
To use this the TimedAction library is included, and a TimedAction object is created & assigned a method to execute & an interval to execute it at.

```
// Timed actions are used here so that whilst we wait for user input,
// we still update the Cloud with our sensor data
TimedAction updateThread = TimedAction(3000, readData);
/** updateThread.check(); Needs to be in EVERY LOOP  **/
```

Inside each of the loops in our code the objects method check() is called, which checks if it is time for the thread to fire again.

Transmission of data to the database

Once the python script has the received data stored in variables a connection to the MySQL database is opened and a series of commands are executed to ensure that the tables that the data will be inserted to exist & contain the appropriate fields.

```
#DB connection variables
sys.stdout.write("-----OUTPUT BEGIN-----\n" )
host = 'localhost'
user = 'root'
password = 'root'
db = 'assignment2'
dbConn = MySQLdb.connect(host, user, password) or die("Could Not Connect To Db")

#Check that all our tables exist, and set them up if they dont
with dbConn:
    cursor = dbConn.cursor()
    cursor.execute("CREATE DATABASE IF NOT EXISTS assignment2;")
    cursor.execute("USE assignment2;")
    cursor.execute("CREATE TABLE IF NOT EXISTS temperatureDB(id int(7) NOT NULL AUTO_INCREMENT, Temp varchar(7) NOT NULL, primary key (id));")
    cursor.execute("CREATE TABLE IF NOT EXISTS TempLimit(temp_id int(5) NOT NULL AUTO_INCREMENT, TempLimit int(2), primary key (temp_id));")
    cursor.execute("CREATE TABLE IF NOT EXISTS FanStatus(id int(1) NOT NULL, FanOnOrOff varchar(3), primary key (id));")
    cursor.execute("CREATE TABLE IF NOT EXISTS players(player_id char(4) NOT NULL, current_level char(1), constraint player_pk primary key (player_id));")
    cursor.execute("INSERT IGNORE INTO FanStatus (id, FanOnOrOff) VALUES ('1', 'Off');")
```

Once the tables are created the python script checks to see if the current player already has an ID stored in our database, and if they do the script updates their score rather than creating a second entry for a single player.

```
cursor.execute("INSERT INTO temperatureDB (Temp) VALUES ('%s')" % (temperature))
rows = cursor.execute("SELECT * FROM players WHERE player_id LIKE '%s'" % (playerId))

if rows == 0:
#{
    sys.stdout.write("no rows found")
    cursor.execute("INSERT INTO players (player_id, current_level) VALUES ('%s', '%s');" % (playerId, level))
    dbConn.commit()
#}
else:
#{
    cursor.execute("UPDATE players SET current_level = '%s' WHERE player_Id= '%s';" % (level, playerId))
    dbConn.commit()
#}
```

Displaying the player data on the webpage

To display the player data on the page MemoryGame.php we simply use PHP to connect to the local database, we have a separate file (DataBaseCon.inc) which has the database details needed to connect, we then simply connect to the database with a mysqli_connect query and run a while loop to go through each row of the database, each time the loop is run it will print out the PLAYER ID and the CURRENT LEVEL. We have appropriate error checking/error messages if this process fails.