DCU School of Electronic Engineering
Assignment Submission

Student Name(s):	**Alex Doherty**
Student Number(s):	**15508783**
Programme:	**M.Eng. in Electronic and Computer Engineering**
Project Title:	**EE513 Assignment 2**
Module code:	**EE513**
Lecturer:	**Derek Molloy**
Project Due Date:	**20/04/2020**

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying is a grave and serious offence in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion, or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references.

I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified.

This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. I have read and understood the referencing guidelines found

at http://www.library.dcu.ie/citing&refguide08.pdf and/or recommended in the assignment guidelines.

I understand that I may be required to discuss with the module lecturer/s the contents of this submission.

I/me/my incorporates we/us/our in the case of group work, which is signed by all of us.

Signed: Alex Doherty

Assignment 2

Establish an MQTT Framework Architecture

The MQTT broker Mosquitto was downloaded to the Debian virtual machine (VM). The server will be running on the VM and later in the assignment, the C++ programs which are run to read the sensor data on the raspberry pi (RPi) will connect to the server. The mosquito broker was initially tested on the VM, to ensure subscribing to topics and publishing to topics worked correctly on the server. The message which was published to the topic in figure 2, is displayed in figure one where a client is subscribed to the same topic.

```
alex@debian:~$ mosquitto_sub -v -t 'ee513/test'
ee513/test Hello World
```

Figure 1: Subscription to ee513/test topic

```
alex@debian:~$ mosquitto_pub -t 'ee513/test' -m 'Hello World'
alex@debian:~$
```

Figure 2: Publish message to ee513/test topic

Adapt the Server Configuration to Require a Username/Password

The server config file was edited with the following two lines added so that connecting to the server required a username and password.

```
password_file /etc/mosquitto/passwd
allow_anonymous false
```

The mosquito server is restarted to apply the changes to authentication. Figure 4 shows the client being refused from the server without credentials.

```
alex@debian:~$ sudo mosquitto_passwd -c passwd alex
Password:
Reenter password:
alex@debian:~$ more passwd
alex:$6$RCTAttgsjNlxuOt3$6hTLDz0LtuAV0Ffz0reAoDG/Sfyz/WIZKC2oMRK/PottQToF3lUi34c
21vmjrkn0zhfNsdKJ2W8d3BZtIFl0/g==
alex@debian:~$ sudo systemctl restart mosquitto
alex@debian:~$
```

Figure 3: Set username and password for server

```
alex@debian:~$ mosquitto_sub -v -t 'ee513/test'
ee513/test Hello World
Connection Refused: not authorised.
alex@debian:~$
```

Figure 4: Connection denied as authentication is required

In demonstrating that your framework works, use the -d option to display the full debug messages when using the mosquitto-client tools

Now that the server requires a username and password, figures 5 and 6 below display the use of the username and password to publish and subscribe to the ee513/test topic on the server. The -d option is used to display full debug messages.

```
alex@debian:~$ mosquitto_pub -d -u alex -P toilet11 -t 'ee513/test' -m 'Hello Wo
rld'
Client mosqpub|3064-debian sending CONNECT
Client mosqpub|3064-debian received CONNACK (0)
Client mosqpub|3064-debian sending PUBLISH (d0, q0, r0, m1, 'ee513/test', ... (1
1 bytes))
Client mosqpub|3064-debian sending DISCONNECT
alex@debian:~$
```

Figure 5: Credentials used to publish message with debugging messages displayed

```
alex@debian:~$ mosquitto_sub -d -u alex -P toilet11 -t 'ee513/test'
Client mosqsub|3063-debian sending CONNECT
Client mosqsub|3063-debian received CONNACK (0)
Client mosqsub|3063-debian sending SUBSCRIBE (Mid: 1, Topic: ee513/test, QoS: 0)
Client mosqsub|3063-debian received SUBACK
Subscribed (mid: 1): 0
Client mosqsub|3063-debian received PUBLISH (d0, q0, r0, m0, 'ee513/test', ... (
11 bytes))
Hello World
```

Figure 6: Credentials used to subscribe to topic with debugging messages displayed

Design and Develop an MQTT Publisher Sensor Application

It should read data from the sensor connected to your RPi/BBB and integrate time and CPU load/temperature from your RPi/BBB

The publish and subscribe C++ files are provided for this assignment. [1] These files interface with the ADXL345 board which connects to the RPi using the I2Cbus. The publish and subscribe C++ files are modified to use the IP address of the VM that the server is running on at port 1883. The authentication for the username and password is also changed in these files as shown below.

```
#define ADDRESS     "tcp://192.168.1.19:1883"
#define CLIENTID    "rpi2"
#define AUTHMETHOD  "alex"
#define AUTHTOKEN   "toilet11"
```

```
pi@raspberrypi:~/ass2 $ ./publish
Waiting for up to 10 seconds for publication of {"d":{"CPUTemp": 29.862000 }}
on topic ee513/CPUTemp for ClientID: rpi1
Message with token 1 delivered.
pi@raspberrypi:~/ass2 $
```

Figure 7: Publish the CPU temperature to ee513/CPUTemp topic

As can be seen in figure 8, the JSON payload which includes the CPU temperature is seen by the subscribed client when the message is published.

```
pi@raspberrypi:~/ass2 $ ./subscribe
Subscribing to topic ee513/CPUTemp
for client rpi2 using QoS1

Press Q<Enter> to quit

Message arrived
     topic: ee513/CPUTemp
   message: {"d":{"CPUTemp": 29.862000 }}
```

Figure 8: Subscribe to ee513/CPUTemp topic

The following code was then added to the publisher C++ file to retrieve the sensor data from the ADXL345 chip, which includes the acceleration along the x,y and z axes as well as the pitch and roll.

```
   unsigned int I2CBus = 1;
   unsigned int I2CAddress=0x53;
   exploringRPi::ADXL345 object(I2CBus,I2CAddress);
   object.readSensorState();
   short AccelerationX = object.getAccelerationX();
   short AccelerationY = object.getAccelerationY();
   short AccelerationZ = object.getAccelerationZ();
   float pitch = object.getPitch();
   float roll = object.getRoll();
```

The time from the raspberry pi was also integrated. To get the time in a format that did not result in an encoding issue when sending it in the JSON payload, the system clock was used in the following code.

```
system_clock::time_point today = system_clock::now();
   std::time_t tt;

   tt = system_clock::to_time_t ( today );
```

The JSON payload is edited to include the data retrieved from the ADXL345 sensor. This information can be read by the subscriber once the message is published to the ee513/ADXLdata topic.

```
   sprintf(str_payload, "{\"CPUTemp\": %f , \"X-Acceleration\": %d , \"Y-
Acceleration\": %d , \"Z-
Acceleration\": %d , \"Pitch\": %f , \"Roll\": %f , \"TimeDate\": %s }", getCP
UTemperature(), AccelerationX, AccelerationY, AccelerationZ, pitch, roll, ctim
e(&tt));
```

```
pi@raspberrypi:~/ass2 $ ./publish
today is: Mon Apr 20 12:06:41 2020
X acceleration: 4
Y acceleration: -11
Z acceleration: 256
Pitch: 0.894349
Roll: -2.46011
Waiting for up to 10 seconds for publication of {"CPUTemp": 29.323999 , "X-Acceleration": 4 , "Y-Acceleration": -11 , "Z-Acceleration": 256 , "Pitch": 0.8943
49 , "Roll": -2.460114 , "TimeDate": Mon Apr 20 12:06:41 2020
 }
on topic ee513/ADXLdata for ClientID: rpi1
Press Q<Enter> to quit
q
Message with token 1 delivered.
```

Figure 9: Publishing ADXL345 data

```
pi@raspberrypi:~/ass2 $ ./subscribe
Subscribing to topic ee513/ADXLdata
for client rpi2 using QoS1

Press Q<Enter> to quit

Message arrived
     topic: ee513/ADXLdata
   message: {"CPUTemp": 29.323999 , "X-Acceleration": 4 , "Y-Acceleration": -11 , "Z-Acceleration": 256 , "Pitch": 0.894349 , "Roll": -2.460114 , "TimeDate":
 Mon Apr 20 12:06:41 2020
 }
```

Figure 10: Subscriber to the ee513/ADXLdata topic

Demonstrate the use of a "last will" message under normal and abnormal disconnections.

A last will and testament message is used to notify subscribers of a topic that there has been a break in the connection. The broker will send the last will message to relay this message when a client has ungracefully disconnected.

The MQTT client library [2] was used to understand the functions and definitions which could be used to apply the last will message to the program.

The following code was used to add the last will message in the publish C++ file.

```
   MQTTClient_willOptions willOpts = MQTTClient_willOptions_initializer; //ini
tialiser for MQTT will options
   opts.will = &willOpts;

   const char* LWTMessage = "Last will and testament";
   const char* LWTTopic = "LastWill";

   opts.will->qos = QOS;
   opts.will->message = LWTMessage;
   opts.will->topicName = LWTTopic;
   opts.will->retained = 0;
```

The various elements in the will were set, such as the QOS remaining consistent with the rest of the program, the last will message being set, the topic for the last will being set as well as the retained message flag being set to zero.

```
pi@raspberrypi:~/ass2 $ ./publish
X acceleration: 12
Y acceleration: 5
Z acceleration: 257
Pitch: 2.67284
Roll: 1.11335
Waiting for up to 10 seconds for publication of {"d":{"CPUTemp": 31.476000 , "X Acceleration": 12 , "Y Acceleration": 5 , "Z Acceleration": 257 , "Pitch": 2.
672843 , "Roll": 1.113351 }}
on topic ee513/ADXLdata for ClientID: rpi1
Message with token 1 delivered.
```

Figure 11: Publisher for normal disconnection

```
pi@raspberrypi:~/as2 $ ./subscribe
Subscribing to topic ee513/ADXLdata
for client rpi2 using QoS1

Press Q<Enter> to quit

Message arrived
     topic: ee513/ADXLdata
   message: {"d":{"CPUTemp": 31.476000 , "X Acceleration": 12 , "Y Acceleration": 5 , "Z Acceleration": 257 , "Pitch": 2.672843 , "Roll": 1.113351 }}
```

Figure 12: Subscriber for normal disconnection

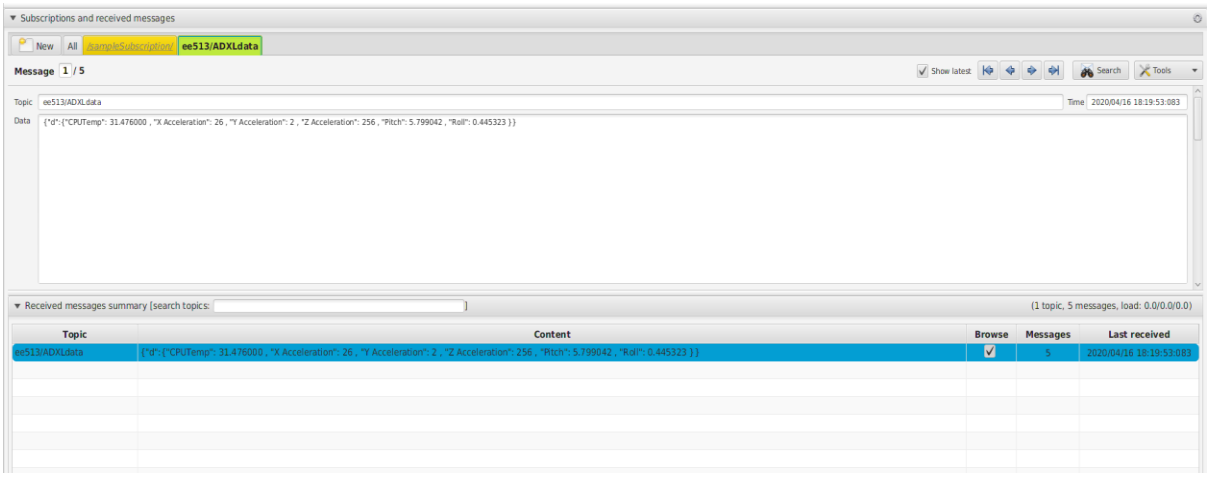

Figure 13: MQTT Spy displaying message for ADXLdata topic with normal graceful disconnection of subscriber

There is no last will message received under a normal disconnection, as can be seen in figures 11, 12 and 13. When the subscriber disconnects correctly with a graceful disconnection, the last will message is discarded by the broker.

For the abnormal disconnect, some of the code from the subscriber C++ file is reused.

```
    do {
        ch = getchar();
    } while(ch!='Q' && ch != 'q');
    MQTTClient_disconnect(client, 10000);
    MQTTClient_destroy(&client);
```

This code waits on the user to enter the character ‘q’ or ‘Q’. If the user does not enter either these characters within a time frame, an abnormal disconnect occurs. The last will message is displayed in MQTT spy under the LastWill topic that was created, for an abnormal disconnect. This can be seen in figure 14.

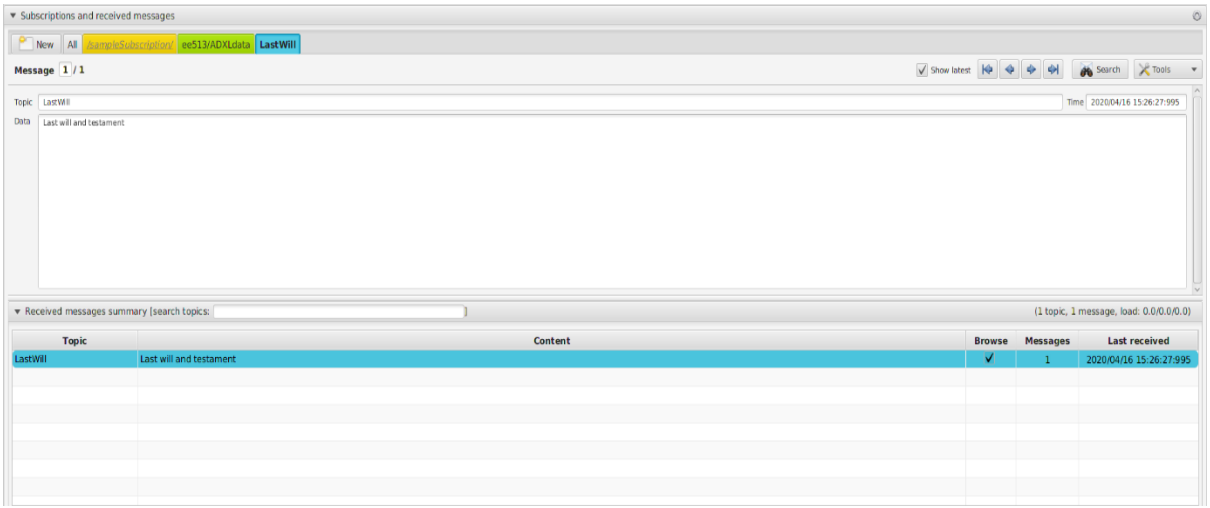

Figure 14: MQTT Spy displaying message for last will with abnormal disconnection of subscriber

Demonstrate the use of different MQTT QoS levels

There are three different QoS levels, which provide a varying quality of service.

QoS0 provides the least reliable service in delivering the message but it delivers it at a quick rate. This can be seen demonstrated in figure 15, where the delivery of 50 messages are all done within a second. There is no retransmission of the message and there are no acknowledgements sent by the receiver when the message is received.

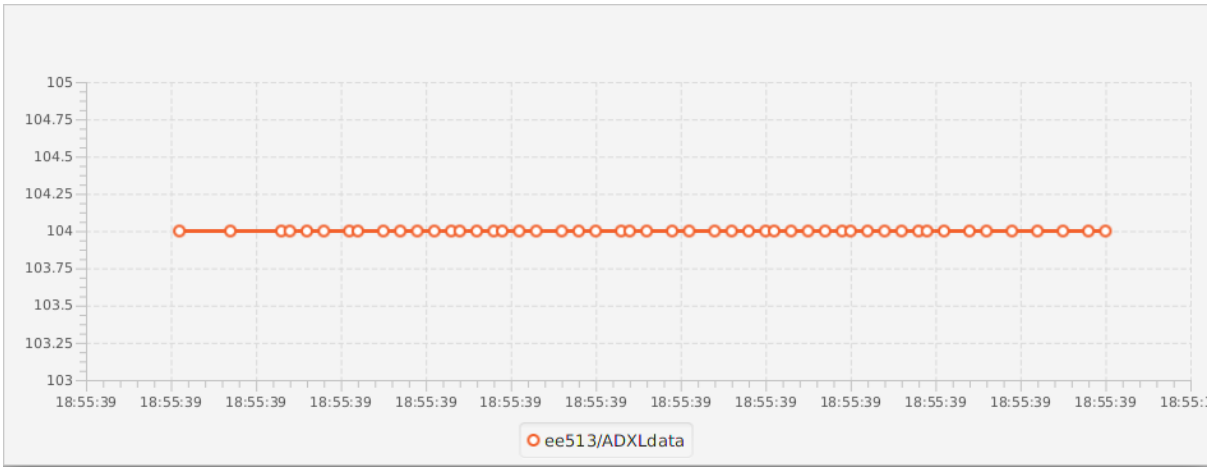

Figure 15: QOS0

QoS1 provides a more reliable service. It achieves this by guaranteeing that the message will be delivered at least once. The sender stores the message until it has received an acknowledgement from the receiver that it has received the message. There is a time limit for the sender to receive an acknowledgement from the receiver and if it does not receive one within this time frame, the message will be resent. Figure 16 displays the delivery of 50 messages using QoS1, which takes 5 seconds to transmit.

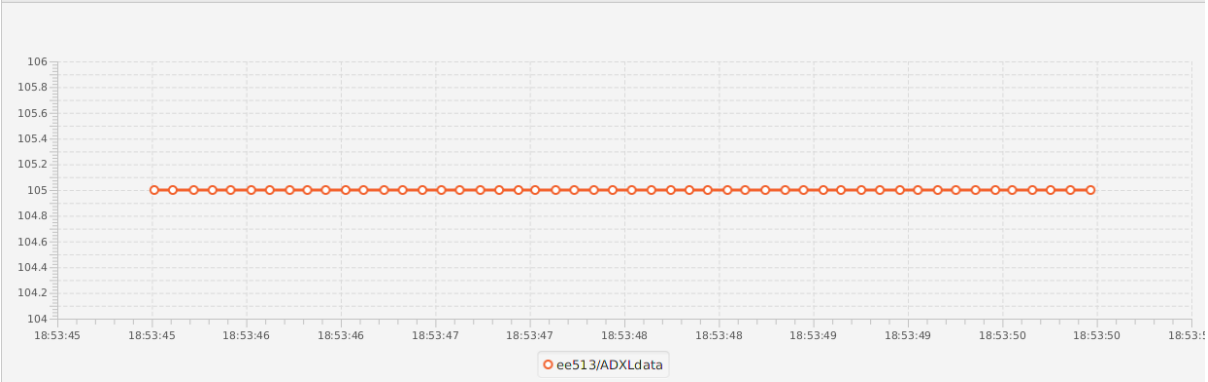

Figure 16: QOS1

Finally, QoS2 provides the most reliable service of the three levels. The messages are guaranteed to be received just once by the receiver, with no duplicates unlike QoS1. This uses at least two request/response flows between the sender and receiver of the message. This is displayed in figure 17 where there are no losses but a significant amount of time is taken to transmit the messages.

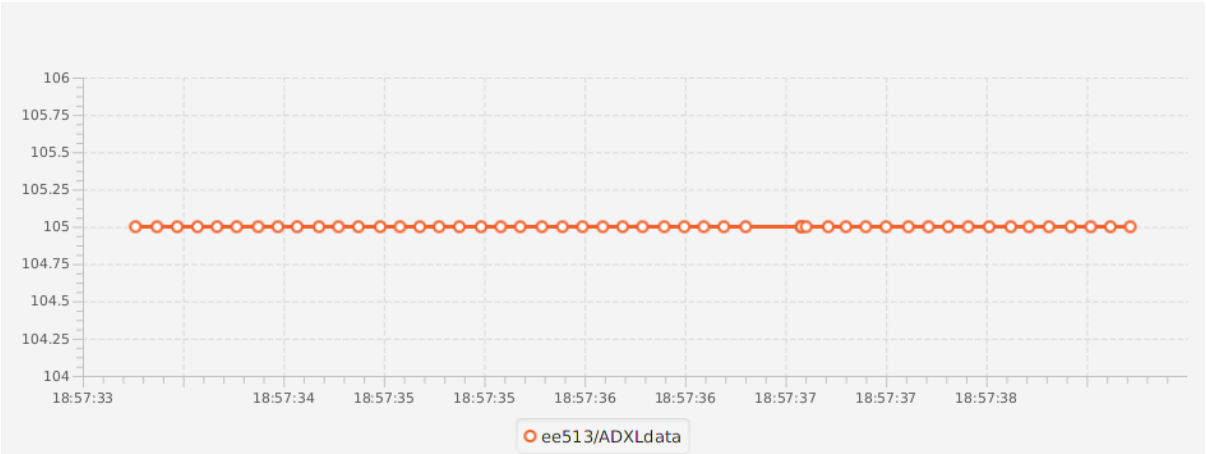

Figure 17: QOS2

Design and Develop Multiple MQTT Subscriber Actuator Applications

First Application

The first subscriber actuator application created is a simple concept. When the acceleration in the x or y direction is read as above 50 or the acceleration in the z direction is greater than 250, the LED is turned on for 10 seconds. To satisfy this condition, the JSON message payload had to be read in and set as variables. This was done using the jsoncpp library [3]. The following code was used in the msgarrvd function to read in the JSON payload, parse the JSON string, set the different values to variables and an if statement which called another function to turn the LED on.

```
payloadptr = (char*) message->payload;
    for(i=0; i<message->payloadlen; i++) {
        ch = *payloadptr;
        string1.append(1, ch);
        putchar(*payloadptr++);
    }
    putchar('\n');
    char * charString = (char*) message->payload;
    Json::Value val;
    Json::Reader reader1;
    bool parsingStatus = reader1.parse(charString, val);

    if(!parsingStatus){
        cout<<"Error parsing string"<<endl;
    }

    int AccelerationX=val["X-Acceleration"].asInt();
    int AccelerationY=val["Y-Acceleration"].asInt();
    int AccelerationZ=val["Z-Acceleration"].asInt();
    float pitch=val["Pitch"].asFloat();
    float roll=val["Roll"].asFloat();
    string sysTime=val["TimeDate"].asString();


    if(AccelerationX > 50 || AccelerationY > 50 || AccelerationZ > 250){
        cout<<"success"<<endl;
        flashLED10();

    }
```

The flashLED10 function is used to turn the LED on for ten seconds. The LED was connected to GPIO24 on the raspberry pi, as well as ground. The GPIO pins can be set as inputs or outputs. It is set as an output for this function by writing to a direction file on the filesystem. There is also a value file which can be used to write a value of ‘1’ or ‘0’ to the pin. The GPIO code provided from the exploringRPi repository was used to set the GPIO pin. An object of this class is made in the flashLED10 function to access its functions and set the pin as high for the LED to turn on for ten seconds. The usleep Linux command is used to keep the LED on for 10 seconds, before turning the LED off again.

```
void flashLED10(){
    GPIO LEDobj(24);
    LEDobj.setDirection(OUTPUT);
    LEDobj.setValue(HIGH);
    usleep(10000000); //wait for 10 seconds
    LEDobj.setValue(LOW);
}
```

When compiling the subscribe C++ file, an extra compiler flag is used for the jsoncpp library to be used, as well as the GPIO code needing to also now be compiled. The command is now as follows:

g++ subscribe.cpp GPIO.cpp -o subscribe -ljsoncpp -lpaho-mqtt3c

Second Application

The second actuator application used the pitch and roll data from the sensor. The following code is used to flash the LED ten times.

```
usleep(10000000); //waits 10 seconds

    if(pitch < 0 || roll < 0){
        cout<<"success2"<<endl;
        flashLED();
    }
```

This if statement follows the previous application. There is a wait of ten seconds after the first application and then the check for the second application is ran. If either of the pitch and roll are negative, the LED is flashed ten times.

```
void flashLED(){
    GPIO LEDobj(24);
    LEDobj.setDirection(OUTPUT);
    for(int i=0; i<11; i++){
        LEDobj.setValue(HIGH);
        usleep(1000000);
        LEDobj.setValue(LOW);
        usleep(1000000);
    }
}
```

The GPIO class is used, similarly to the first application. A for loop is used to repeatedly set the output of the GPIO24 pin to high and then low. This is done 10 times, which results in the LED flashing on and off.

Demonstrate the use of persistent connections

The use of persistent connections in MQTT can be used by the subscriber, so that the broker will store the client's information and topics they have subscribed to. This means that after the client has disconnected and then subsequently reconnects, the broker will send all the undelivered messages to the client that were published to that topic while they were disconnected. The broker can do this by saving the clients unique id. [4] Persistent connections can be applied to the C++ code by setting the clean session flag in the client connection options to '0'. This was demonstrated by a client subscribing to a topic and then disconnecting. The publisher then publishes a message to the topic, and when the client re-subscribed, it received

the message that was published when it was disconnected. This can be seen in figures 18 and 19.

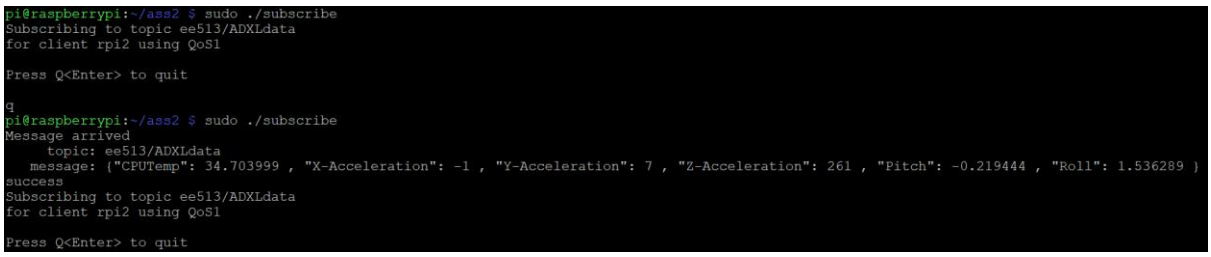

Figure 18: Client subscribing to ee513/ADXLdata topic and then disconnecting. It then reconnects to the broker and receives the message it missed while disconnected.

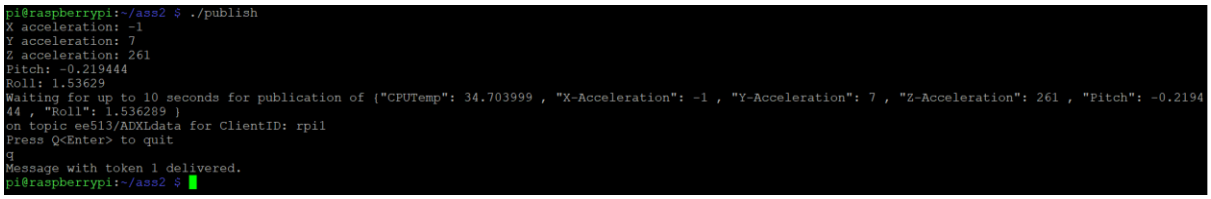

Figure 19: Publisher publishing a message to the ee513/ADXLdata topic while the subscriber is disconnected.

Design and Develop a QT MQTT Visualisation GUI Application

You should be able to manually choose the topic using the GUI

To manually choose the topic using the GUI, a QListWidget was added to the GUI using the design tab in Qt to edit the mainwindow.ui. The functionality of this list could be coded in the mainwindow.cpp file. Firstly, it was given values for the different items in the list. These items were the topics available from the broker. The following code was applied to the mainwindow function, so that it would set the values for the items as soon as the GUI window opened.

```
this->ui->listWidget->addItem("ee513/CPUTemp");

this->ui->listWidget->addItem("ee513/ADXLdata");

this->ui->listWidget->addItem("ee513/LastWill");
```

Following this, the items on the list needed to react to being clicked by triggering an event. This was done using the following code to send the item that was clicked to a function that was created to subscribe to the topic of the item that was clicked.

```
connect(ui->listWidget, SIGNAL(itemClicked(QListWidgetItem*)),
                this, SLOT(onListWidgetItemClicked(QListWidgetItem*)));
```

The onListWidgetItemClicked function acts similar to the on_ConnectButton_clicked function provided with the code. It reconnects to the broker before subscribing to the selected topic. The selected topic has to be extracted from the QListWidgetItem object, and is then converted from type QString to a const char *, which is the parameter type that the MQTTClient_subscribe function accepts. Figure 20 shows the GUI window, with the QListWidget on the right hand side of the GUI, and the box below the chart outputs text stating which topic is being subscribed to when selected.

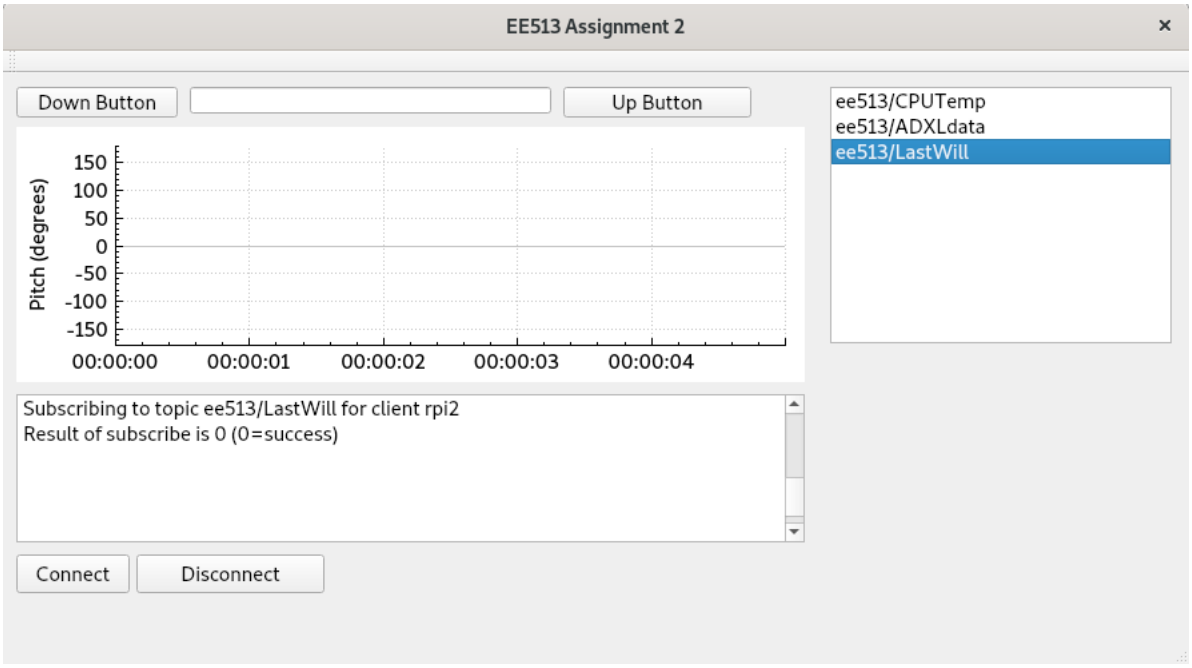

Figure 20: GUI window with the option to select the topic to subscribe to

The QT Application should parse the JSON payload

To parse the JSON payload, a parse function was used which would parse the payload. The following code demonstrates this process.

```
void MainWindow::parseJSON(QString payload){

    QJsonDocument d = QJsonDocument::fromJson(payload.toUtf8());

    QJsonObject obj = d.object();

    qDebug() << "Payload: " << payload << endl;


    this->AccelerationX = obj["X-Acceleration"].toInt();

    this->AccelerationY = (double)obj["Y-Acceleration"].toDouble();

    this->AccelerationZ = (double)obj["Z-Acceleration"].toDouble();

    this->pitch = (double)obj["Pitch"].toDouble();

    this->roll = (double)obj["Roll"].toDouble();

    this->CPUTemp = (double)obj["CPUTemp"].toDouble();
```

The different values in the JSON string are extracted into the variables in the program. This parseJSON function is called from the on_MQTTMessage method, so that the JSON payload is parsed whenever a message is received.

The QT Application should plot the sensor data

This section was to plot the various data sent through in a chart.

Following the parsing of the payload, the parseJSON function sets the count variable to the value for one of the sensor data information, depending on which is selected. This is done using a case method, as there is a switchNumber integer which stores a number that determines which of the sensor data has been selected in the GUI. After the parseJSON method is called in the on_MQTTmessage method, the update function is called. This will result in the new data being displayed on the chart.

Each time one of the different sensor data to be plotted is selected, the current graph is cleared and then replotted so that it does not contain data for something else. This is done using the following code.

```
ui->customPlot->clearGraphs();

   ui->customPlot->addGraph();

   ui->customPlot->replot();
```

The Y-axis is also adjusted appropriately for each of the different sensor data, using the following code when selected.

```
    this->ui->customPlot->yAxis->setRange(-80,80);
```

The range is set to an appropriate value, using the if statements which determine which sensor data has been selected. This is also where the switchNumber variable is set, which is used to set the value being plotted.

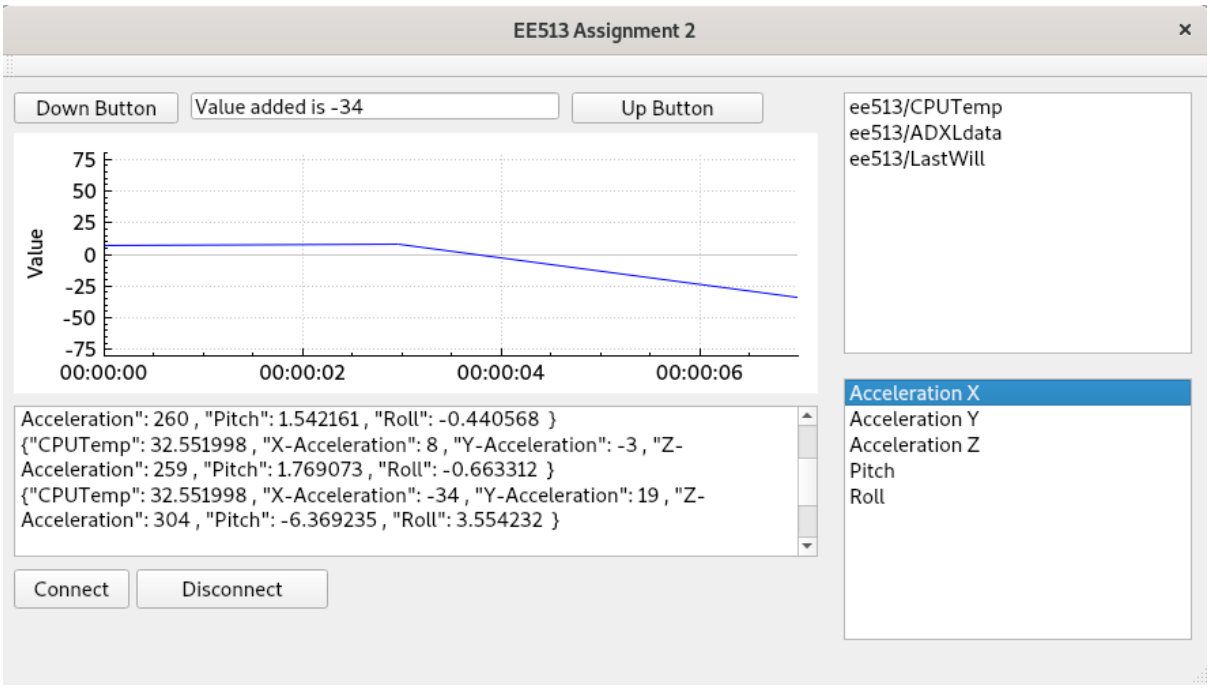

Figure 21: Plot of X-Acceleration

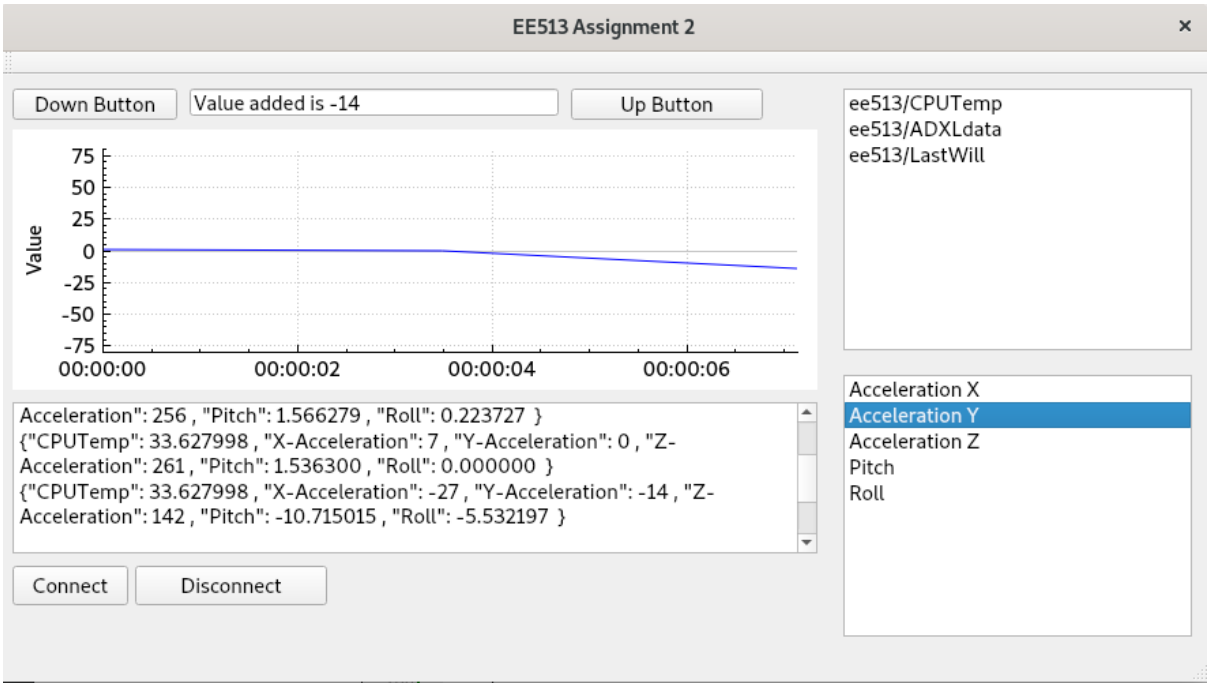

Figure 22: Plot of Y-Acceleration

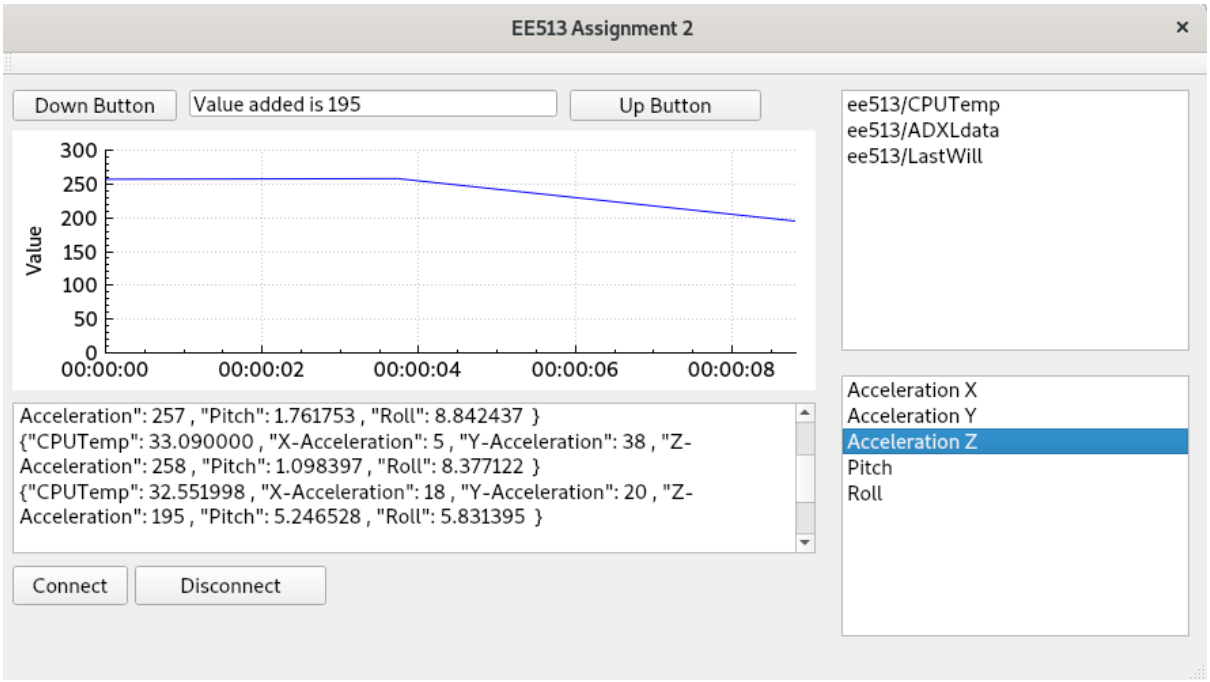

Figure 23: Plot of Z-Acceleration

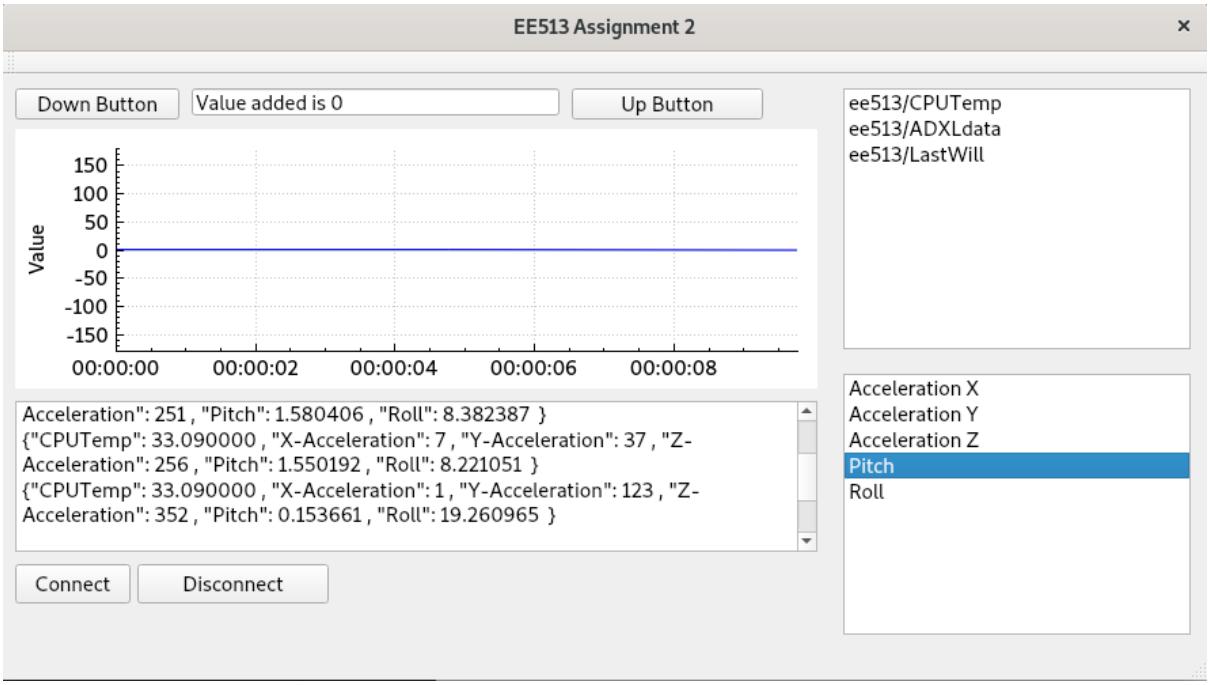

Figure 24: Plot of pitch

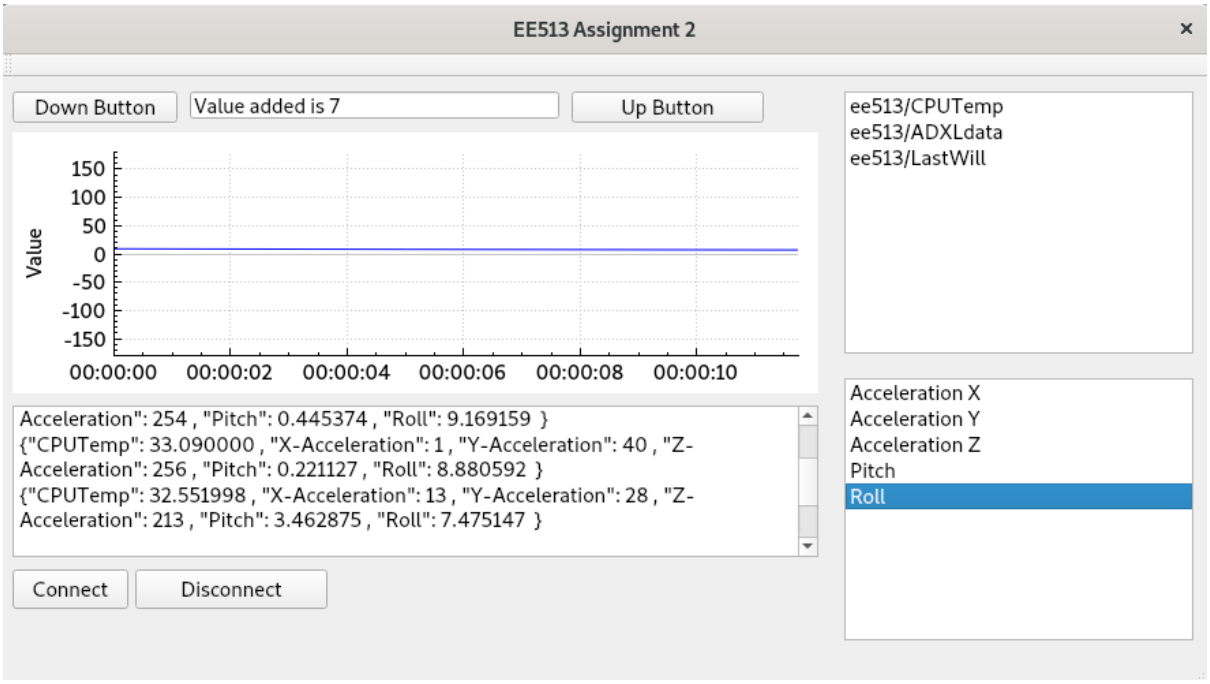

Figure 25: Plot of Roll

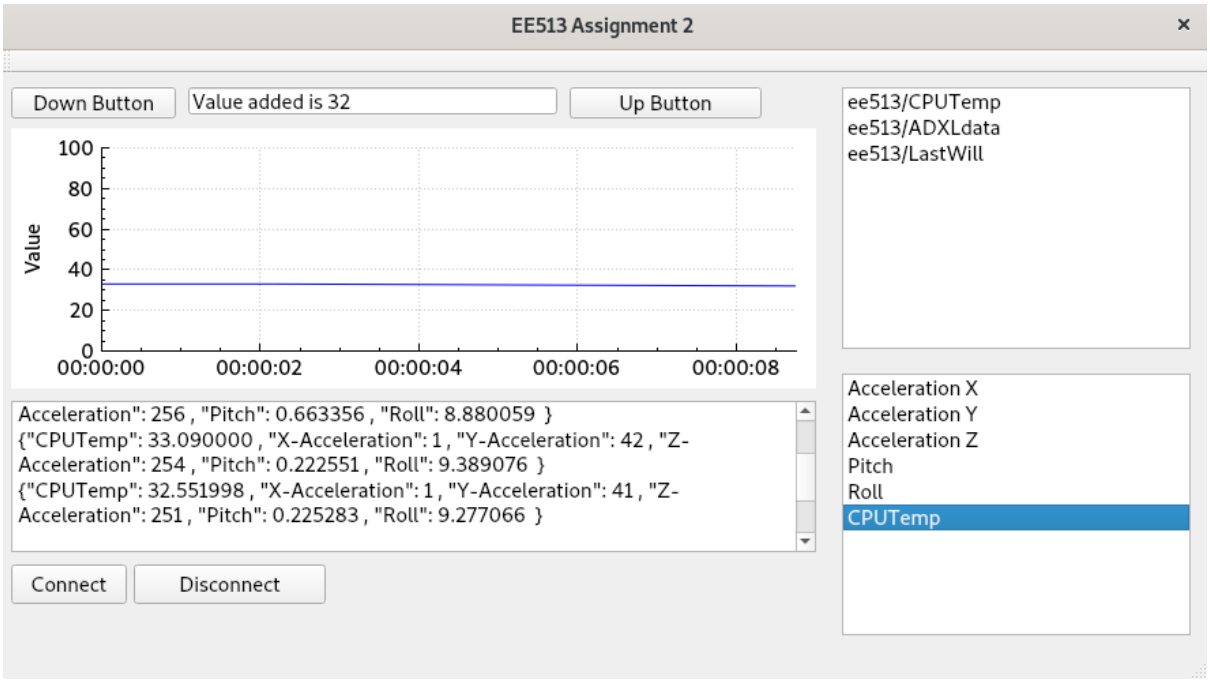

Figure 26: CPU temperature plot

Commit History

```
Date:   Mon Apr 20 23:46:50 2020 +0100

    qt section and clean up

commit a2a41582032f8b621dc8f381a0b2134890d869db
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Mon Apr 20 11:58:20 2020 +0100

    change subscriber applications and add time to payload

commit ea467a16465dfdf11d416320c505f15cae765980 (subscriber-actuator-apps)
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Fri Apr 17 20:44:29 2020 +0100

    develop multiple subscriber actuator applications

commit 502a22e7fe57d95b99643ed3f2f19c4229e96c64 (publisher-sensor-app)
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Fri Apr 17 13:29:24 2020 +0100

    design and develop mqtt publisher sensor app
:...skipping...
commit bda56d11b848e2531cd1d78aeb5a9195dfa903b4 (HEAD -> master)
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Mon Apr 20 23:46:50 2020 +0100

    qt section and clean up

commit a2a41582032f8b621dc8f381a0b2134890d869db
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Mon Apr 20 11:58:20 2020 +0100

    change subscriber applications and add time to payload

commit ea467a16465dfdf11d416320c505f15cae765980 (subscriber-actuator-apps)
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Fri Apr 17 20:44:29 2020 +0100

    develop multiple subscriber actuator applications

commit 502a22e7fe57d95b99643ed3f2f19c4229e96c64 (publisher-sensor-app)
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Fri Apr 17 13:29:24 2020 +0100

    design and develop mqtt publisher sensor app

commit 6784fb85010f0c9633019977f62809cd2c9f1977
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Wed Apr 8 14:02:04 2020 +0100
```

References

[1] D. Molloy, “ADXL345 Source Code,” Dublin, 2016.

[2] [Online]. Available: https://www.eclipse.org/paho/files/javadoc/org/eclipse/paho/client/mqttv3/MqttClient.html. [Accessed 2020 April 15].

[3] “jsoncpp,” [Online]. Available: https://github.com/open-source-parsers/jsoncpp. [Accessed 17 April 2020].

[4] M. Tolia, “MQTT Message Queuing & Persistent Session (With Example),” 13 November 2018. [Online]. Available: https://mntolia.com/mqtt-message-queuing-persistent-session-with-example/. [Accessed 17 April 2020].

[5] D. Molloy, *EE513 Assignment 1,* Dublin, 2020.

[6] Claudix, “How to read/write arbitrary bits in C/C++,” Stack Overflow, 5 August 2012. [Online]. Available: https://stackoverflow.com/questions/11815894/how-to-read-write-arbitrary-bits-in-c-c. [Accessed 6 March 2020].

[7] Maxim Integrated, “DS3231 Extremely Accurate I2C-Integrated RTC/TCXO/Crystal,” 2015. [Online]. Available: https://datasheets.maximintegrated.com/en/ds/DS3231.pdf. [Accessed 3 March 2020].

[8] J. Tranter, “How to Control GPIO Hardware from C or C++,” 14 August 2019. [Online]. Available: https://www.ics.com/blog/how-control-gpio-hardware-c-or-c. [Accessed 17 April 2020].