Sheffield Hallam University
Department of Engineering
BEng (Hons) Computer Systems Engineering
BEng (Hons) Electrical and Electronic Engineering

Activity ID		Activity Title			Laboratory Room No.	Level
Lab 104		IoT applications			4301 / 4302	6
Term	**Duration [hrs]**	**Group Size**	**Max Total Students**	**Date of approval/review**	**Lead Academic**	
1	6	2	25	09-21	Alex Shenfield	

Equipment (per student/group)

Number	Item
1	STM32F7 discovery board lab kit
1	SparkFun ESP8266 WiFi shield (with modified firmware)

Learning Outcomes

	Learning Outcome
2	Demonstrate an understanding of the various tools, technologies and protocols used in the development of embedded systems with network functionality
3	Design, implement and test embedded networked devices

Internet-of-Things applications using the STM32F7 discovery board

Introduction

Computing is about more than the PC on your desktop! Embedded devices are everywhere – from wireless telecommunications infrastructure points to electronic point of sale terminals. One definition of an embedded system is:

> "An embedded system is a computer system designed to perform one or a few dedicated functions often with real-time computing constraints."
>
> (http://en.wikipedia.org/wiki/Embedded_system)

In the laboratory sessions for this module you are going to be introduced to the STM32F7 discovery board – a powerful ARM Cortex M7 based microcontroller platform capable of prototyping advanced embedded systems designs. The STM32F7 discovery board includes advanced functionality such as Ethernet connectivity, UART over the USB connection, an LCD screen, and a micro-SD slot. Appendix A shows the various pins that are broken out from the STM32F7 discovery board (onto the Arduino form factor header).

This laboratory session will focus on creating Internet-of-Things enabled applications capable of communicating with remote servers (and other embedded devices) over the Internet using the popular MQTT protocol.

Bibliography

There are no essential bibliographic resources for this laboratory session aside from this tutorial sheet. However the following websites and tutorials may be of help (especially if you haven't done much electronics previously or your digital logic and/or programming is a bit rusty):

- http://www.cs.indiana.edu/~geobrown/book.pdf[1]
- https://visualgdb.com/tutorials/arm/stm32/
- http://www.keil.com/appnotes/files/apnt_280.pdf
- https://developer.mbed.org/platforms/ST-Discovery-F746NG/

1 Note: this book is for a slightly different board – however, much of the material is relevant to the STM32F7 discovery

Methodology

Check that you have all the necessary equipment (see Figure 1)!

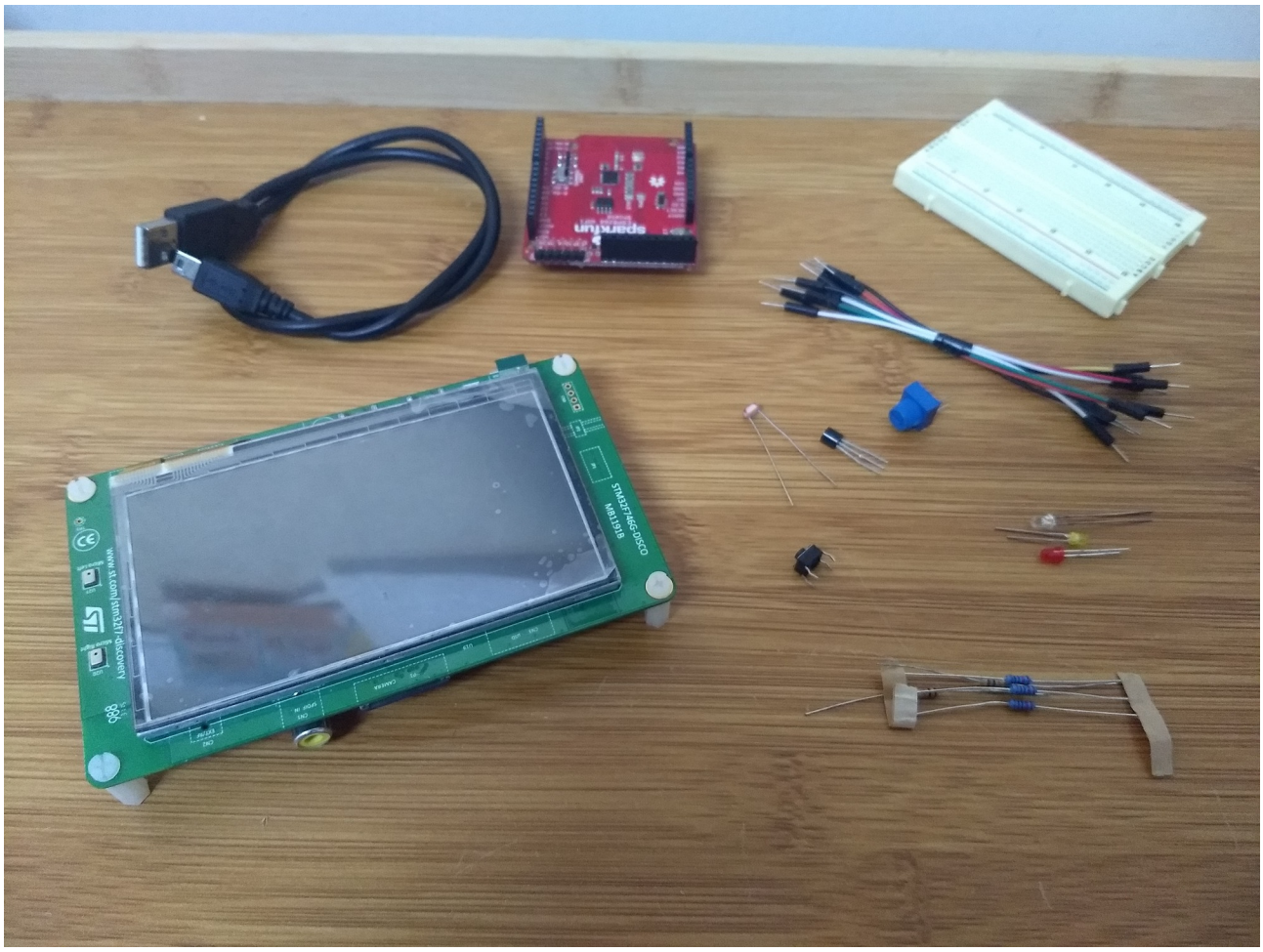


Figure 1 – The necessary equipment for this lab

Background

As we will discuss in the lectures, the commonly used HTTP protocol has several drawbacks when used for communication in Internet-of-Things applications. These drawbacks end up resulting in inefficiencies and increased power consumption – major limiting factors in lightweight embedded and mobile devices. For these reasons, many Internet-of-Things devices (particularly those with strict resource constraints such as small micro-controllers) avoid HTTP and use alternative protocols such as CoAP and MQTT.

The official MQTT 3.1.1 specification[2] says:

> "MQTT is a Client Server publish/subscribe messaging transport protocol. It is light weight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium."

One of the key benefits in comparison to HTTP is that MQTT uses 10s of bytes in protocol headers, whereas HTTP can use 100s – 1000s of bytes for headers. Stanford-Clark and Nipper (1999) specified the following goals for MQTT:

- Simple to implement
- Provide a Quality of Service Data Delivery
- Lightweight and Bandwidth Efficient
- Data Agnostic
- Continuous Session Awareness

Another benefit of MQTT is the **publish – subscribe** model which reduces network traffic by allowing a server to push updates to subscribed clients when there is new information of interest available. Figure 2 illustrates this interaction.

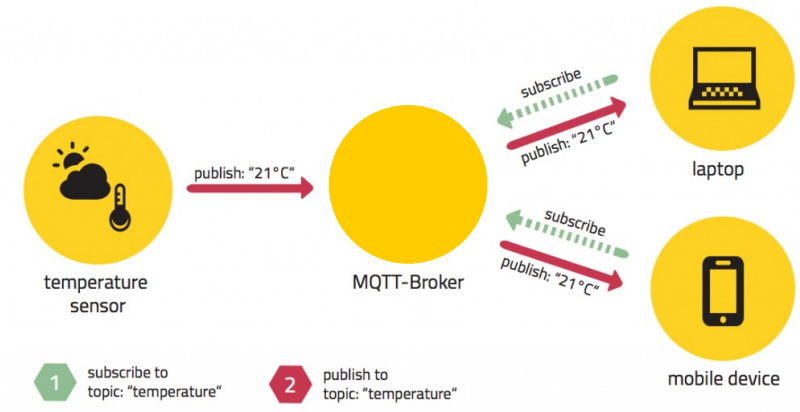


Figure 2 – MQTT's publish – subscribe model

2 http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

In this set of lab exercises, we are going to use the STM32F7 discovery board kit and the SparkFun ESP8266 WiFi shield (with upgraded firmware) as our embedded Internet-of-Things device. Whilst you can install the shield directly on the Arduino headers (on the bottom of the board) as shown in Figure 3, I have increasingly found that this causes network connection problems (see note 3, below). A better alternative is often to make external connections to the SparkFun board using jumper wires (as described in Appendix A).

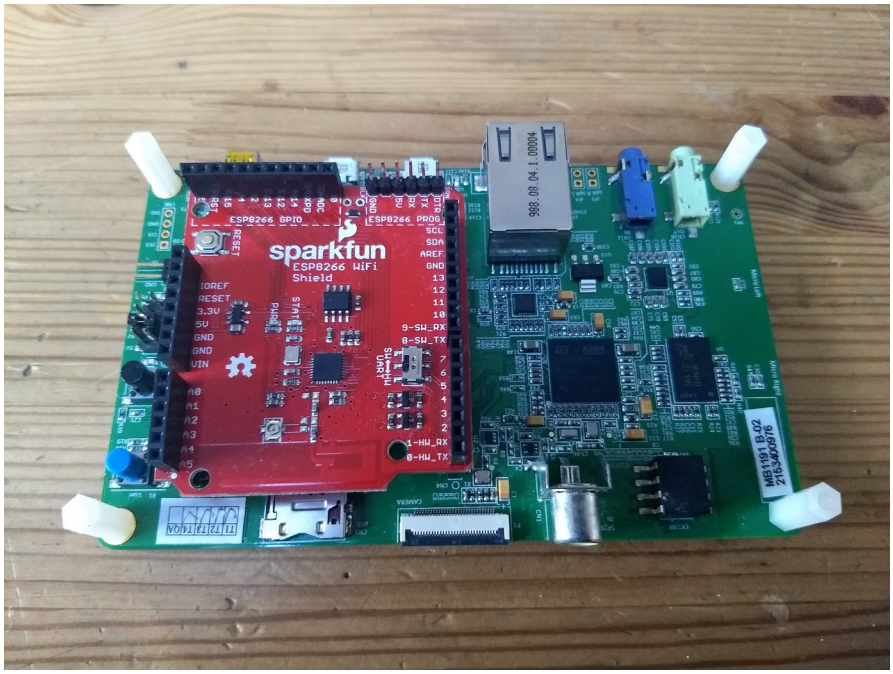


Figure 3 – The SparkFun ESP8266 shield installed on the STM32F7 discovery board kit

There are a few things to note with respect to the SparkFun shield:

1. The UART switch of the board should be set to “HW” (see Figure 4).

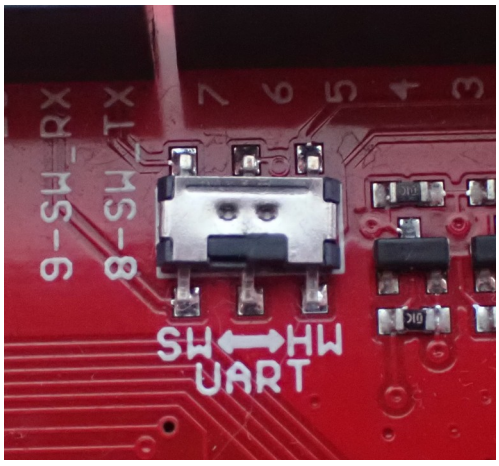


Figure 4 – The SparkFun shield UART switch

2. The WiFi shield only works with the 2.4GHz spectrum. This means your WiFi access point must have the 2.4GHz channels enabled (most modern APs support the use of dual-spectrums). If this is an issue you can often use Windows 10 connection sharing to share the network connection over WiFi with the SparkFun shield.

3. If you struggle to get a stable network connection (i.e. the WiFi frequently struggles to connect or drops out when communicating[3]) you can often improve this by removing the WiFi shield and attaching it externally with wires (see Appendix A).

In the applications we will develop in this set of lab exercises, we will be using the Eclipse Paho project's test broker – mqtt.eclipse.org – which is a public test MQTT broker that supports both encrypted and unencrypted traffic, multiple levels of quality of service, and supports the MQTT 3.1.1 protocol.

Note that this is a completely public broker, so anyone can see the messages you are publishing. **Do not use this for anything important!** Many other hosted services exist (some with limited free offerings) – for example:

- https://www.cloudmqtt.com/
- https://cloud.emqx.io/
- https://io.adafruit.com/

3 This can often be seen by WiFi connection errors, SSL certificate errors, or failures to subscribe or publish MQTT messages (e.g. the message "`return code from MQTT publish is -1`").

Task 1

In this task we are going to use the STM32F7 discovery board kit and the SparkFun ESP8266 WiFi shield (with upgraded firmware) to build a simple application capable of posting a series of status messages to a remote MQTT broker and receiving status updates back.

Figure 5 shows the application flow for this task. We will use a similar application flow for the other tasks in this lab session.

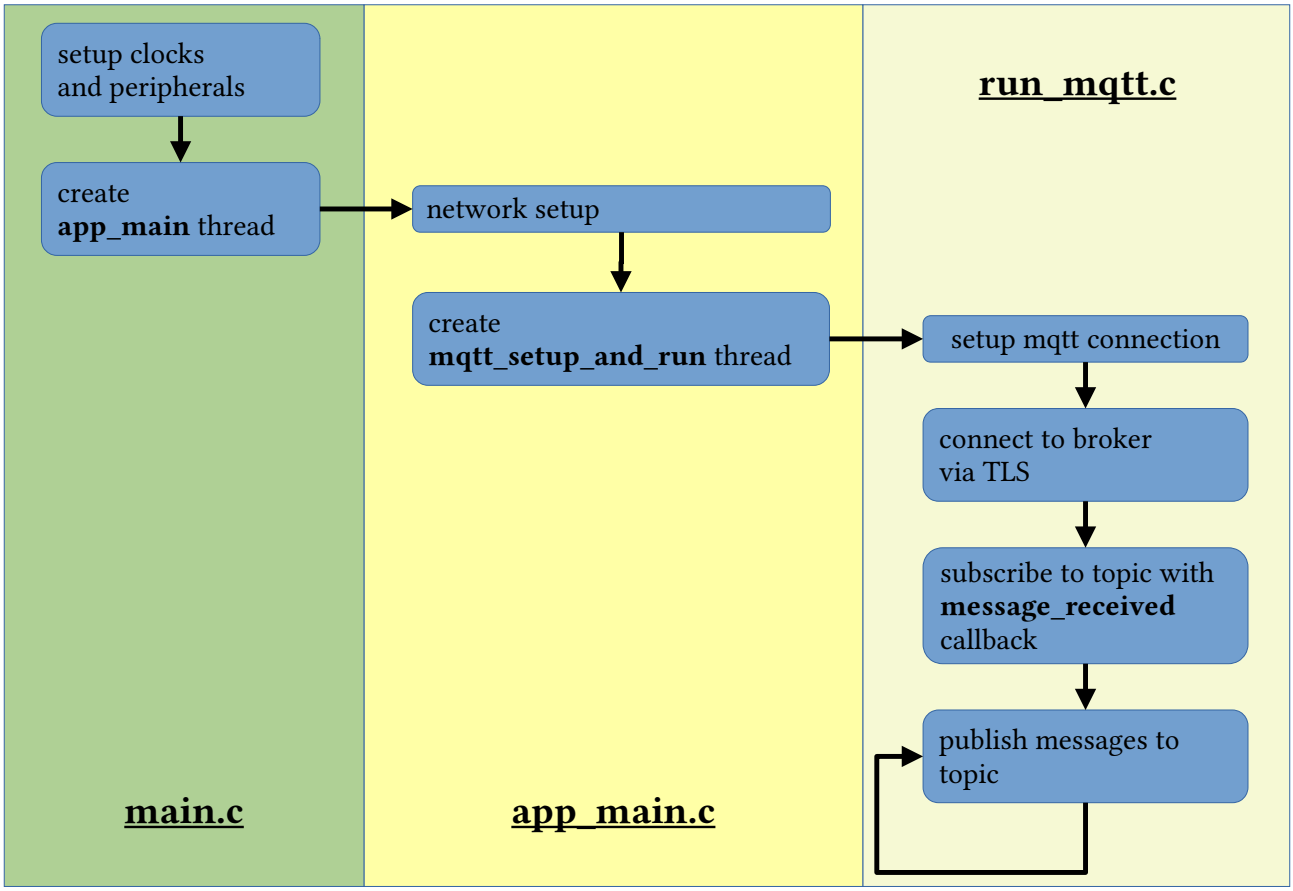


Figure 5 – MQTT application message flow

As you can see from Figure 5, most of the work is done in the **`mqtt_setup_and_run`** thread. Our **`main`** function just sets up the system clock and other peripherals before creating the **`app_main`** thread and passing control over to the real-time operating system (much as in the exercise in lab 103). The app_main thread initialises the WiFi network connection (using the **`socket_startup()`** function from **`socket_startup.c`**) and – if that is successful – then creates the **`mqtt_setup_and_run`** thread which does the rest of the work.

Figure 6 shows the project structure for this task.

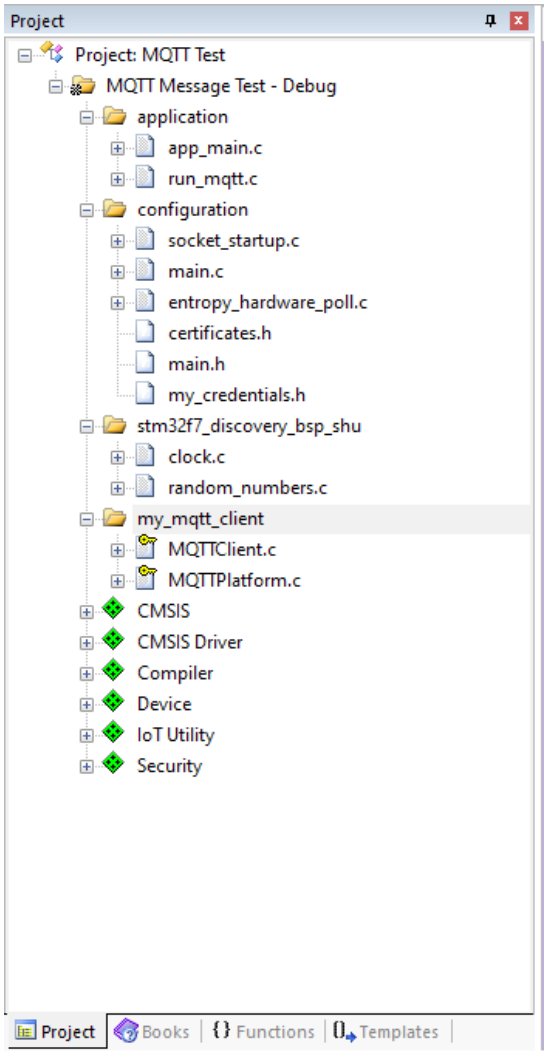


Figure 6 – The MQTT message test project structure

There's some important points to note here:

1. We are using a TLS secured connection to our MQTT broker, so we need to provide the root CA certificate (in **`certificates.h`**) when we establish a connection.
2. We have provided some simple code (**`entropy_hardware_poll.c`**) to provide a hardware source of entropy when creating the TLS secured connection – this is much more secure than using a software source of entropy[4].
3. We are using a customised version of the Eclipse Paho MQTT client (in the **`my_mqtt_client`** group) to get around some conflicts of definitions[5].
4. We are using the mbed TLS library (under "Security") to handle this secured connection.

4 See https://hackaday.com/2017/11/02/what-is-entropy-and-how-do-i-get-more-of-it/ for more information about entropy and why it matters.

5 See https://github.com/eclipse/paho.mqtt.embedded-c/pull/214 for discussion as to why changes are needed.

I like to separate out confidential stuff like WiFi network credentials and MQTT username and passwords out into a separate header file – in this case called **`my_credentials.h`** – as it helps to avoid committing them to public repositories on GitHub. A template for this **`my_credentials.h`** file is provided as part of this project and shown in code listing 1, below.

Code listing 1:

```
/*
* my_credentials.h
*
* this is the header file containing my wifi credentials and mqtt accound
* details (if needed)
*
* author:    Dr. Alex Shenfield
* date:      17/11/2020
* purpose:   55-604481 embedded computer networks - lab 104
*/

// define to prevent recursive inclusion
#ifndef __CREDS_H
#define __CREDS_H

// wifi credentials
#define SSID            "Your SSID"
#define PASSWORD        "Your password"
#define SECURITY_TYPE   ARM_WIFI_SECURITY_WPA2

//
// NOTE - make sure this clientID is unique. i recommend using your student
// id number
//

// mqtt credentials
#define mqtt_clientid  "Your client ID "
#define mqtt_username  "Your user name "
#define mqtt_password  ""

#endif
```

You will have to edit this to set your own WiFi credentials! We are using an open, public broker in these examples, so you don't need to put in a password (or even a unique username)[6]. However, you must make sure that the **`mqtt_clientid`** is unique! Multiple simultaneous connections from the same client ID often cause problems!

6 However, if you ever use something like adafruit.io, ubidots, or cloud MQTT you will need to provide the username and password they issue to you.

You also need to make sure that the CMSIS-Driver WiFi Driver is configured correctly to use the appropriate USART and USART settings to communicate with the SparkFun shield. These should be USART #6 at a baud-rate of 115200bps. The WiFi thread should also have an above normal priority (so it doesn't get interrupted when receiving data).

Figure 7 shows the configuration wizard for the WiFi driver with these settings.

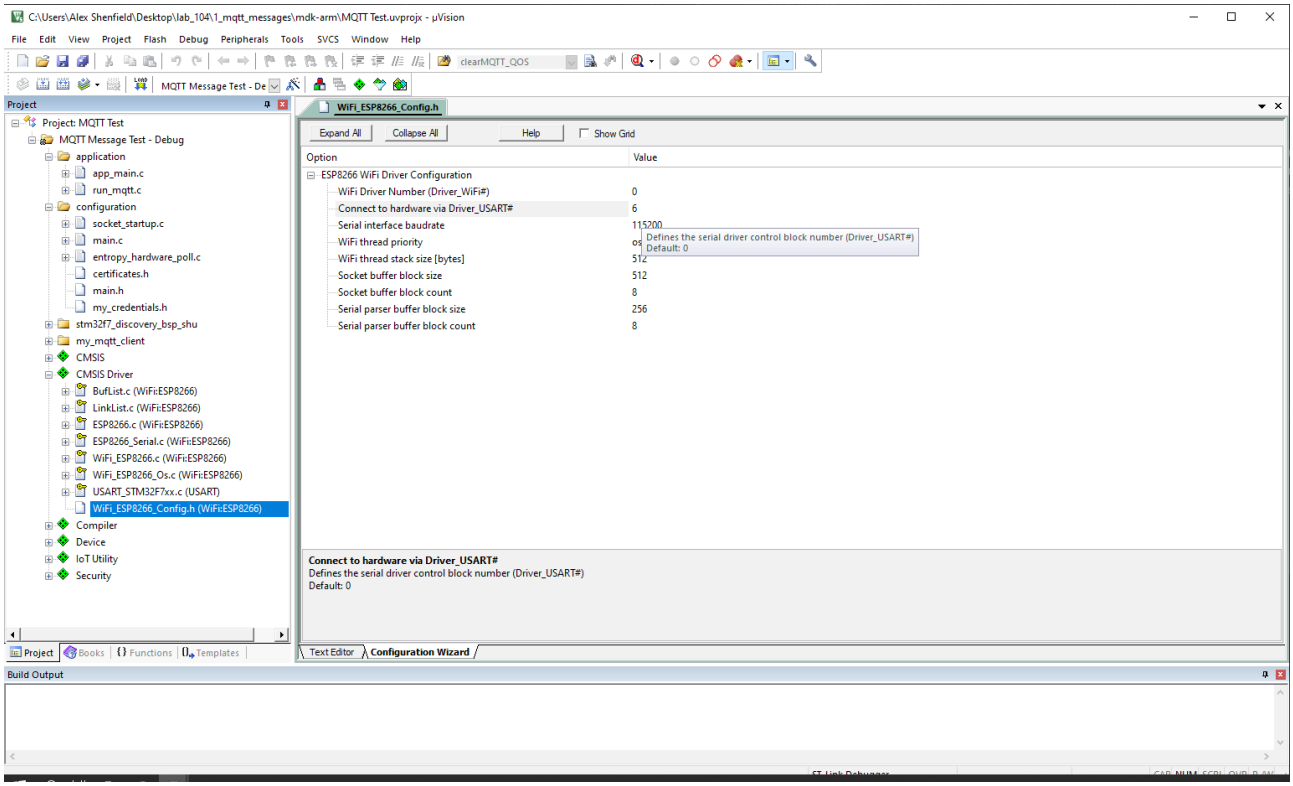


Figure 7 – Configuring the WiFi driver for the ESP8266 SparkFun WiFi shield

Code listing 2 (below) shows the first part of the **`run_mqtt.c`** file. This shows the code for setting up a message received callback that will be triggered every time we get a message on a specific topic. We have just created a simple callback in this task that just writes the message we have received and the complete message topic we have received it on[7].

Code listing 2:

```
/*
* run_mqtt.c
*
* this file contains the logic for connecting to an mqtt broker over tls
* and sending it some messages
*
* author:    Dr. Alex Shenfield
* date:      17/11/2020
* purpose:   55-604481 embedded computer networks - lab 104
*/

// include the c string handling functions
#include <string.h>

// include the paho mqtt client
#include "MQTTClient.h"

// include the root ca certificate (and client certificate and key - but we're
// not using those ...)
#include "certificates.h"

// include my mqtt credentials
#include "my_credentials.h"

// define the mqtt server
#define SERVER_NAME     "mqtt.eclipse.org"
#define SERVER_PORT     8883

// MQTT CALLBACK

// set up our message received callback (just dump the message to the screen)
void message_received(MessageData* data)
{
  printf("message arrived on topic %.*s: %.*s\n",
         data->topicName->lenstring.len,
         data->topicName->lenstring.data,
         data->message->payloadlen,
         (char *)data->message->payload);
}
```

Code listing 3 (on the next page) shows the setup code to establish a secured network connection to the MQTT broker and then make a connection between the MQTT client and the broker. We will reuse this code a lot in the tasks in this lab :)

7 Note the use of the "**`%.*s`**" format string here – this allows us to pass in some length information about the strings to be printed. This is important, as the MessageData object fields are not reset between calls to this callback and therefore can contain information from previous messages we have received! More discussion can be found here: https://stackoverflow.com/questions/7899119/what-does-s-mean-in-printf

Code listing 3:

```
// set up the mqtt connection as a thread
void mqtt_setup_and_run_task(void *argument)
{
  // SETUP MQTT CONNECTION STUFF ...

  // initialise structures for the mqtt client and for the mqtt network
  // connection
  MQTTClient client;
  Network network;

  // establish a secured network connection to the broker

  // set up the tls certificate
  TLScert tlscert = {(char *)CA_Cert, NULL, NULL};

  // initialise the network connection structure
  NetworkInit(&network);

  // try to connect to the mqtt broker using tls and - if it fails - print out
  // the reason why
  int rc = 0;
  if((rc = NetworkConnectTLS(&network, SERVER_NAME, SERVER_PORT, &tlscert)) != 0)
  {
    printf("network connection failed - "
           "return code from network connect is %d\n", rc);
  }

  // connect the client to the broker

  // initialise the mqtt client structure with buffers for the transmit and
  // receive data
  unsigned char sendbuf[128];
  unsigned char readbuf[128];
  MQTTClientInit(&client, &network, 30000, sendbuf, sizeof(sendbuf),
                 readbuf, sizeof(readbuf));

  // initialise the mqtt connection data structure
  MQTTPacket_connectData connectData = MQTTPacket_connectData_initializer;
  connectData.MQTTVersion = 4;
  connectData.clientID.cstring = (char*)mqtt_clientid;
  connectData.username.cstring = (char*)mqtt_username;
  connectData.password.cstring = (char*)mqtt_password;

  // start mqtt client as task (what this does is to make sure that the mqtt
  // processing functionality is protected by a mutex so it can only run
  // sequentially)
  if((rc = MQTTStartTask(&client)) != 1)
  {
    printf("return code from start tasks is %d\n", rc);
  }

  // connect ...
  if((rc = MQTTConnect(&client, &connectData)) != 0)
  {
    printf("return code from MQTT connect is %d\n", rc);
  }
  else
  {
    printf("MQTT Connected\n");
  }
```

We then subscribe to a topic (which means that the MQTT broker will forward on all messages pertaining to that topic) and attach the **`message_received`** callback that we saw earlier (in code listing 2). More details as to how the publish-subscribe model in MQTT works will be discussed in future lectures.

Code listing 4 (below) shows this subscription process. In this example, I am subscribing to all topics after:

`ashenfield/sheffield-hallam/iot-mqtt/`

using the '**`#`**' wildcard character[8]. When you specify your topics, make sure you choose something unique – remember, this is a public broker, so if you choose something generic like "**`house/temperature/#`**" there's a good chance you will see someone-else's data!

Note that we are also specifying a quality of service level (QOS2) for this communication between the broker and the MQTT client. QoS 2 is the highest quality of service level available and means that messages should be received once and only once. Again, this will be discussed in more depth in the lectures.

Code listing 4:

```
// SETUP MQTT MESSAGE STUFF ...

// manage mqtt topic subscriptions

// subscribe to the everything in the root ('#') of my test topic (we will
// publish messages to a subtopic of this root so this subscription means we
// can see what we're sending). we also need to register the callback that
// will be triggered everytime a message on that topic is received (here this
// is "message_received" from earlier) and set the QoS level (here we are
// using level 2 - but you need to check what your broker supports ...)
char topic[] = "ashenfield/sheffield-hallam/iot-mqtt/#";
if((rc = MQTTSubscribe(&client, topic, QOS2, message_received)) != 0)
{
  printf("return code from MQTT subscribe is %d\n", rc);
}
```

8 For more information about MQTT topics – see: https://mosquitto.org/man/mqtt-7.html

Finally, we are going to publish some messages to the:

ashenfield/sheffield-hallam/iot-mqtt/status-message

topic.

You can see from code listing 5 that we do this by writing our data into the payload part of the message and then calling **MQTTPublish** with the message and the topic we want to publish the data to. This example publishes 10 messages and then disconnects from the MQTT broker.

You can see we are also specifying a quality of service level (QOS1) for this communication between the MQTT client and the broker. We are using QoS 1 here, as QoS 2 is more time consuming (which isn't a problem for the MQTT broker, but might be for the MQTT client). Again, this will be discussed in more depth in the lectures.

Code listing 5:

```
  // publish messages

  // send 10 messages to our broker
  for(int count = 0; count < 10; count++)
  {
    // initialise a character array for the message payload
    char payload[30];

    // create the message (writing the data into the payload) and set up the
    // publishing settings
    MQTTMessage message;
    message.qos = QOS1;
    message.retained = 0;
    message.payload = payload;
    sprintf(payload, "message number %d", count);
    message.payloadlen = strlen(payload);

    // publish the message (printing the error if something went wrong)
    char topic[] = "ashenfield/sheffield-hallam/iot-mqtt/status-message";
    if((rc = MQTTPublish(&client, topic, &message)) != 0)
    {
      printf("return code from MQTT publish is %d\n", rc);
    }

    // delay for 1 second before sending the next message
    osDelay(1000);
  }

  // disconnect from the network
  NetworkDisconnect(&network);
  printf("MQTT disconnected\n");
}
```

Figure 8 shows the output in the debugging window of uVision. You can see here both the mbed TLS debugging messages (highlighted with the red box) as well as the messages received back from our MQTT broker via the **`message_received`** callback (highlighted with the blue box).

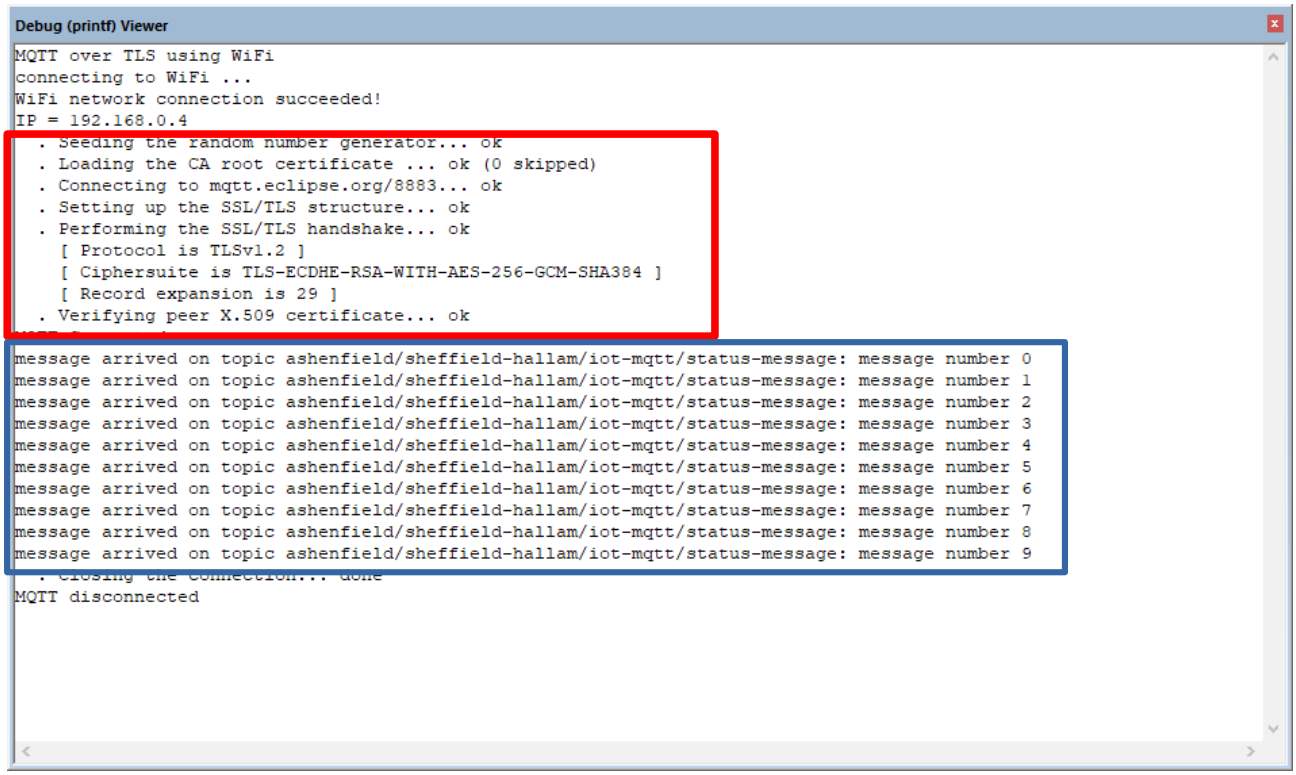


Figure 8 – MQTT message test debugging window output

Compile and run this project.

Task 2

In this task we are going to use the STM32F7 discovery board kit and the SparkFun ESP8266 WiFi shield to build an IoT data node capable of posting a set of sensor data readings to a remote MQTT broker. We are also going to introduce the use of the MQTT Explorer application to view (and graph!) this sensor data!

The basic structure of this project is very similar to the previous example – but we will introduce a new thread to do our data acquisition (in the same way as we did for task 3 in lab 103). Figure 9 shows this application architecture.

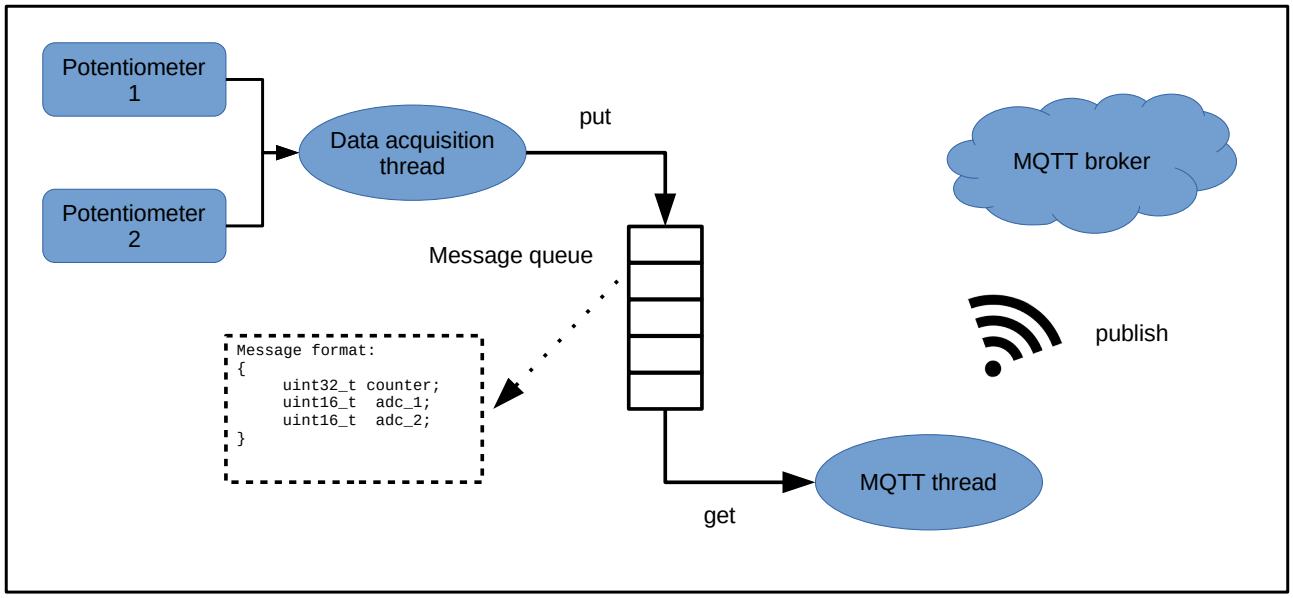


Figure 9 – MQTT data node application block diagram

We then have to build our sensor circuit. The circuit diagram for this application is shown in Figure 10 – it is simply two potentiometers and an LED connected to the STM32F7 discovery board. Note the use of the 3.3V supply to the circuit. This ensures that the potentiometers wont saturate the ADCs on the STM32F7 discovery board (as the maximum voltage the ADCs can read is 3.3V).

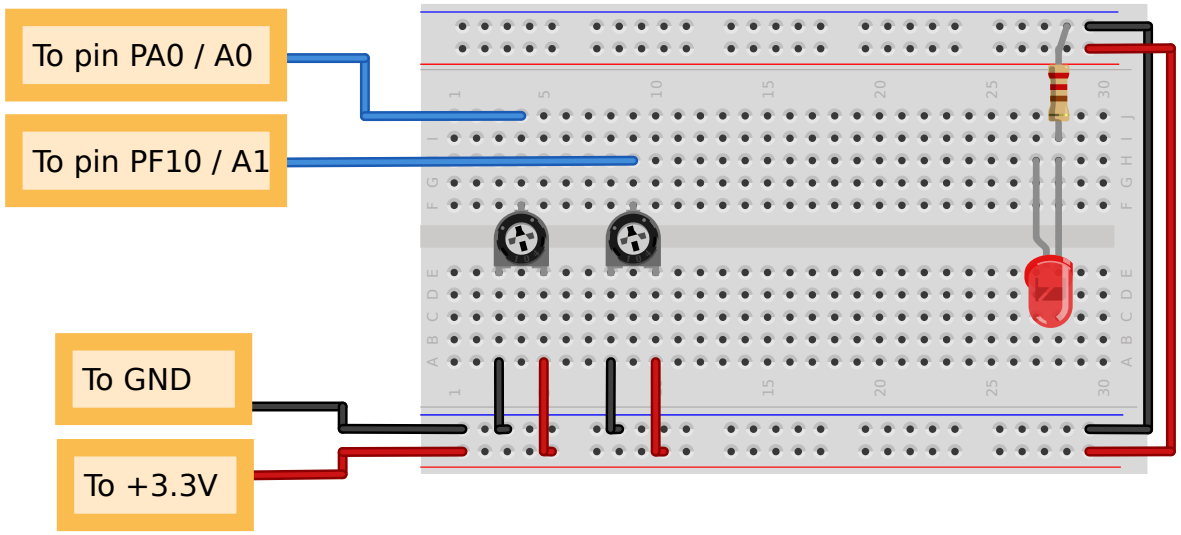


Figure 10 – The IoT data node circuit diagram

Whilst writing this lab task, I had a lot of network connectivity problems (both with the WiFi not connecting properly and the MQTT message publishing periodically failing)! So I have now resorted to making external connections to the SparkFun shield (as in Appendix A). My completed system is shown in Figure 11, below.

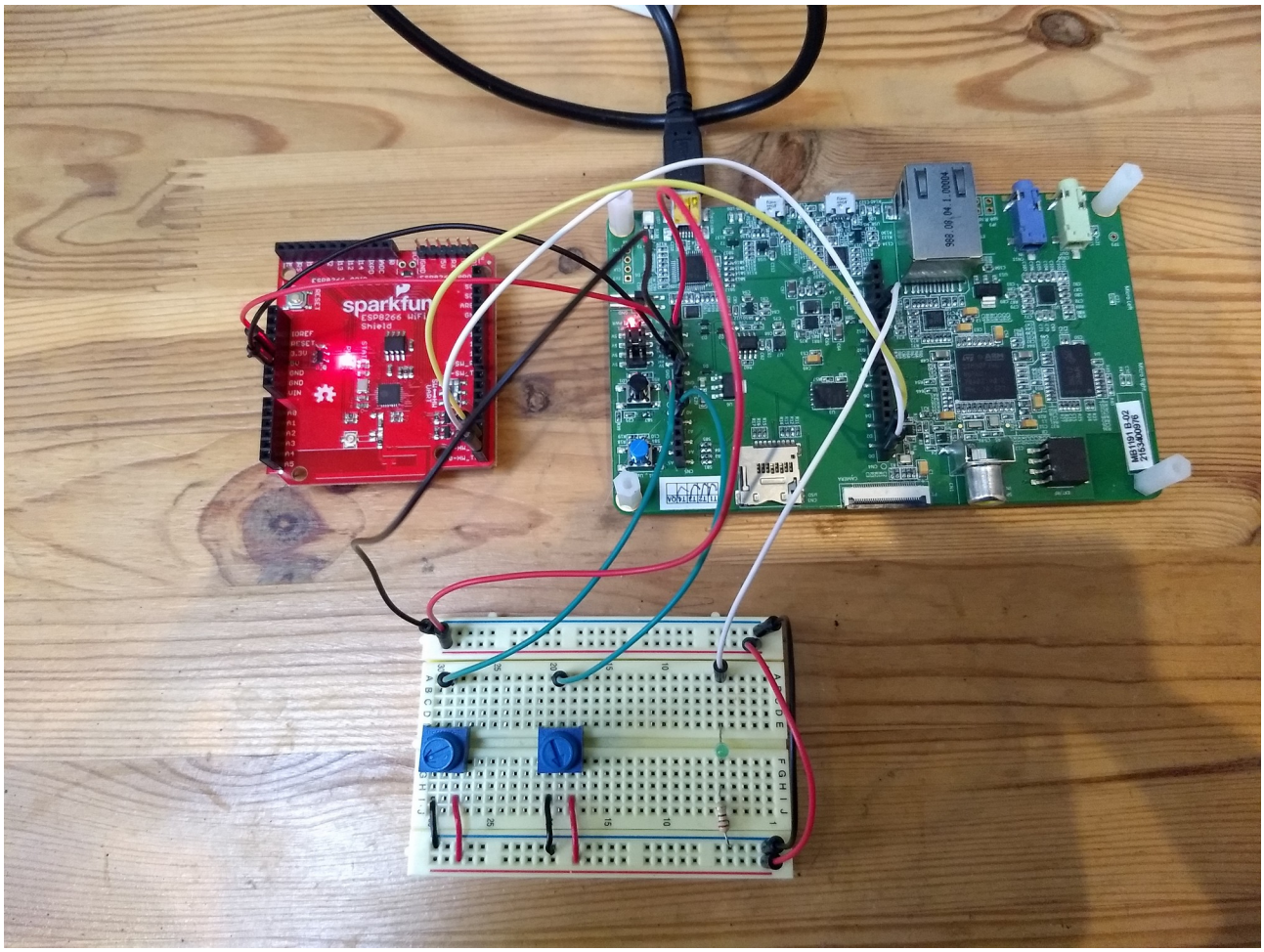


Figure 11 – My completed IoT data node

I have found this to be much more reliable!

In the code for this task we will modify the MQTT thread to run forever (rather than just publishing 10 messages and exiting) and wait for input from a message queue. The thread then formats the data it receives from the message queue and publishes it to the MQTT broker. However, this time it wont disconnect and close the connection to the MQTT broker (as it runs forever).

Code listing 6 shows the modified section of the MQTT thread.

Code listing 6:

```
...

  // publish messages

  // initialise our message object and the message priority
  message_t msg;
  uint8_t   priority;

  // run our main mqtt publish loop
  while(1)
  {
    // get a message from the message queue
    osStatus_t status = osMessageQueueGet(m_messages, &msg, &priority,
                                          osWaitForever);

    // check the message status
    if(status == osOK)
    {
      // initialise a character array for the message payload
      char payload[64];

      // create the basic message and set up the publishing settings
      MQTTMessage message;
      message.qos = QOS1;
      message.retained = 0;
      message.payload = payload;

      // send the message count (as a string based status message)

      // write the data into the payload and send it
      memset(payload, 0, sizeof(payload));
      sprintf(payload, "message number : %d", msg.counter);
      message.payloadlen = strlen(payload);

      // publish the message (printing the error if something went wrong)
      char cnt_topic[] = "ashenfield/sheffield-hallam/iot-mqtt/counter";
      if((rc = MQTTPublish(&client, cnt_topic, &message)) != 0)
      {
        printf("return code from MQTT publish is %d\n", rc);
      }
```

```
      // send the first sensor value

      // write the data into the payload and send it
      memset(payload, 0, sizeof(payload));
      sprintf(payload, "%04d", msg.adc_1);
      message.payloadlen = strlen(payload);

      // publish the message (printing the error if something went wrong)
      char adc_1_topic[] = "ashenfield/sheffield-hallam/iot-mqtt/adc_1";
      if((rc = MQTTPublish(&client, adc_1_topic, &message)) != 0)
      {
        printf("return code from MQTT publish is %d\n", rc);
      }

      // send the second sensor value

      // write the data into the payload and send it
      memset(payload, 0, sizeof(payload));
      sprintf(payload, "%04d", msg.adc_2);
      message.payloadlen = strlen(payload);

      // publish the message (printing the error if something went wrong)
      char adc_2_topic[] = "ashenfield/sheffield-hallam/iot-mqtt/adc_2";
      if((rc = MQTTPublish(&client, adc_2_topic, &message)) != 0)
      {
        printf("return code from MQTT publish is %d\n", rc);
      }

    }
  }

...
```

Code listing 7 shows the **`rtos_objects.h`** header file for our application. This is where we specify the type definition for our **`message_t`** data structure and the id for the message queue. This data structure is what we are going to use to store our data in and what we are going to stuff into the message queue to exchange this data between threads.

Code listing 7:

```
/*
* rtos_objects.h
*
* this is the header file containing the rtos objects for the rtos mail queue
* application
*
* author:    Dr. Alex Shenfield
* date:      22/10/2020
* purpose:   55-604481 embedded computer networks - lab 104
*/

// define to prevent recursive inclusion
#ifndef __RTOS_OBJ_H
#define __RTOS_OBJ_H

// include the header file for basic data types and the cmsis-rtos2 api
#include <stdint.h>
#include "cmsis_os2.h"

// create the objects to use in our rtos applications to pass data

// my message queue
extern osMessageQueueId_t m_messages;

// message queue data structure
typedef struct
{
  uint32_t  counter;
  uint16_t  adc_1;
  uint16_t  adc_2;
}
message_t;

#endif // RTOS_OBJ_H
```

Code listing 8 shows our data acquisition thread. This **`data_acquisition`** function reads our sensor data, fills the **`message_t`** data structure, and then puts it into the message queue. This is then picked up by our MQTT thread (see code listing 6).

Code listing 8:

```
/*
 * data_acquisition_thread.c
 *
 * this is a thread that periodically acquires some data and puts it in a
 * mail queue
 *
 * author:    Dr. Alex Shenfield
 * date:      22/11/2020
 * purpose:   55-604481 embedded computer networks - lab 104
 */

// include the c standard io library
#include <stdio.h>

// include cmsis_os for the rtos api
#include "cmsis_os2.h"

// include my rtos objects
#include "rtos_objects.h"

// include the shu bsp libraries for the stm32f7 discovery board
#include "pinmappings.h"
#include "gpio.h"
#include "adc.h"

// HARDWARE DEFINES

// led is on PI 1 (this is the inbuilt led)
gpio_pin_t led1 = {PI_1, GPIOI, GPIO_PIN_1};

// sensors
gpio_pin_t pot_1 = {PA_0,  GPIOA, GPIO_PIN_0};
gpio_pin_t pot_2 = {PF_10, GPIOF, GPIO_PIN_10};
```

```
// ACTUAL THREAD

// data acquisition thread - read the adcs and stuff the data in a message
// queue
void data_acquisition(void const *argument)
{
  // print a status message
  printf("data acquisition thread up and running ...\r\n");

  // note - generally this (using a shared resource from multiple threads) is a
  // really bad idea due to race conditions etc. - however, we can (probably)
  // get away with it here as we are only doing it once

  // set up the gpio for the led and initialise the random number generator
  init_gpio(led1, OUTPUT);

  // set up the adc for the temperature and light sensor
  init_adc(pot_1);
  init_adc(pot_2);

  // set up our counter and initialise a message container
  uint32_t i = 0;
  message_t msg;

  // infinite loop acquiring our data (one set of samples per 10 seconds)
  // we also toggle the led so we can see what is going on ...
  while(1)
  {
    // read the sensors
    uint16_t adc_val_1 = read_adc(pot_1);
    uint16_t adc_val_2 = read_adc(pot_2);

    // toggle led
    toggle_gpio(led1);

    // put our data in the message object
    i++;
    msg.counter     = i;
    msg.adc_1       = adc_val_1;
    msg.adc_2       = adc_val_2;

    // put the data in the message queue and wait for ten seconds
    osMessageQueuePut(m_messages, &msg, osPriorityNormal, osWaitForever);
    osDelay(10000);
  }
}
```

Compile this project and load it on to the board!

When you run this example in the debugger, you should see some output in the debug (printf) viewer similar to Figure 12.

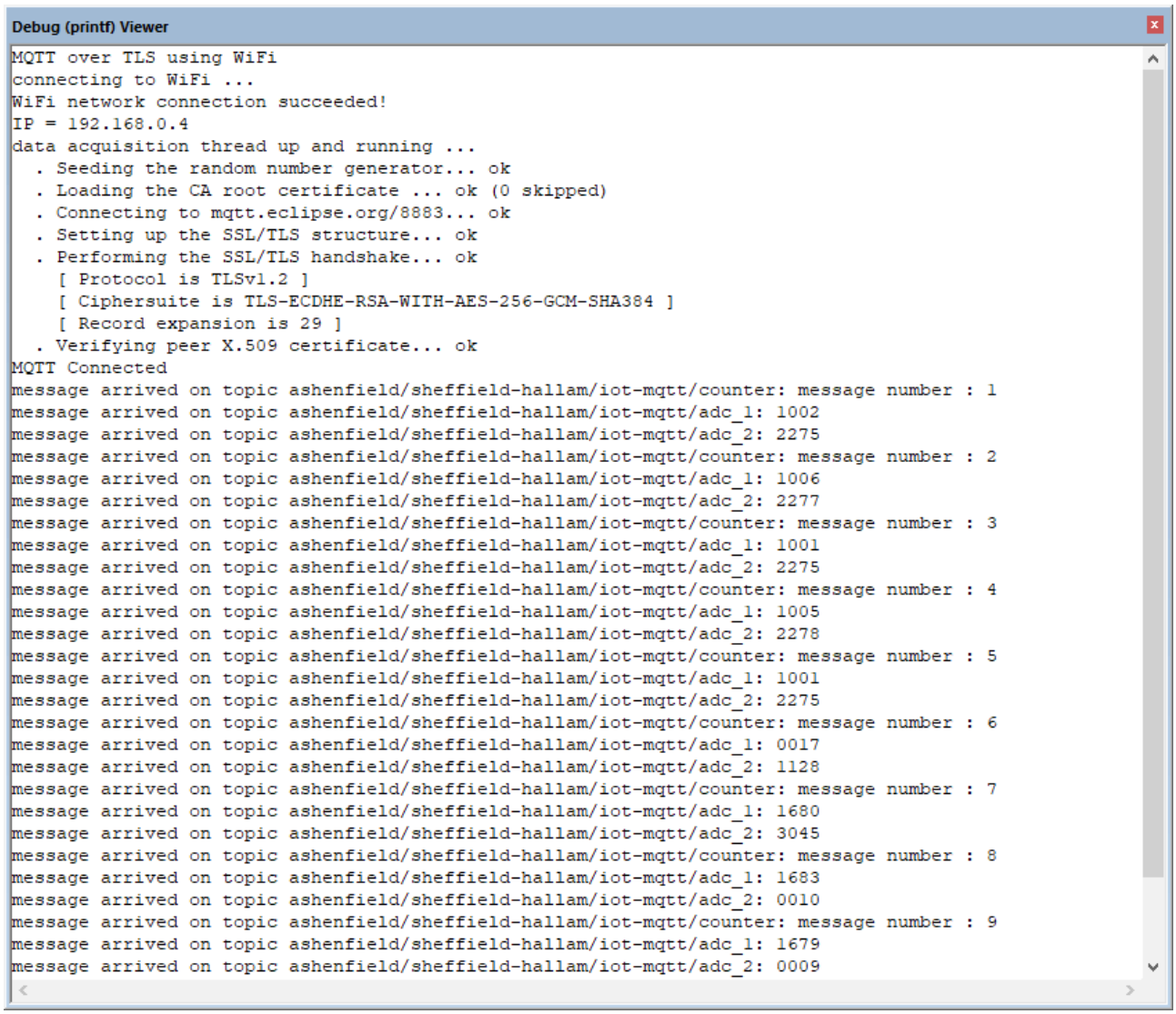


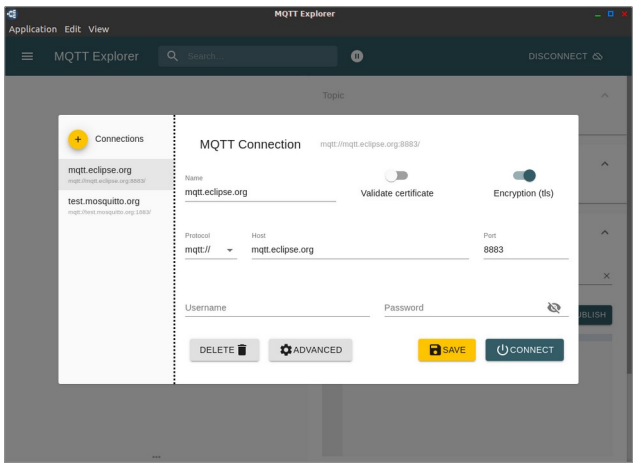
Figure 12 – The Debug (printf) Viewer output

You can see here that we are publishing a sequence of messages every time new data becomes available – the message number, the adc 1 reading, and the adc 2 reading.

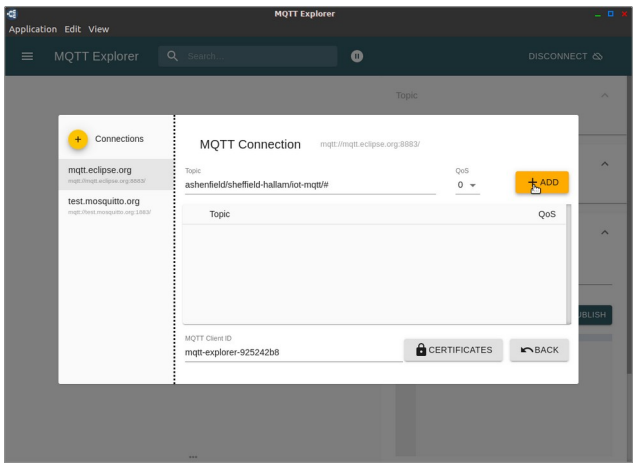
As well as receiving messages on the STM32F7 discovery board with the **`message_received`** callback (as shown in Figure 12, above), we can also use an external MQTT application to interact with the MQTT broker (both sending and receiving messages on specified topics). My favourite PC application for this is MQTT explorer as it allows you to create graphs from numeric data!

MQTT Explorer is available from: http://mqtt-explorer.com/.

Figure 13 shows the MQTT connection screens for MQTT explorer. In Figure 13a I am connecting to the **`mqtt.eclipse.org`** broker on port 8883 (which is the secure port), and in Figure 13b I am subscribing to the "**`ashenfield/sheffield-hallam/iot-mqtt/#`**" topic.



a) Connecting to the **`mqtt.eclipse.org`** test broker



b) Subscribing to the root topic of my feed

Figure 13 – MQTT explorer connection set up

Figure 14 shows the data coming in on the subscribed topic (and subtopics) from the STM32F7 discovery board. In this window you can see both the current messages (on the left hand side) and the message history (on the bottom right). You can also see that we have made graphs of the two sensor readings – you do this by clicking the graph icon on the topic history panel (see Figure 15).

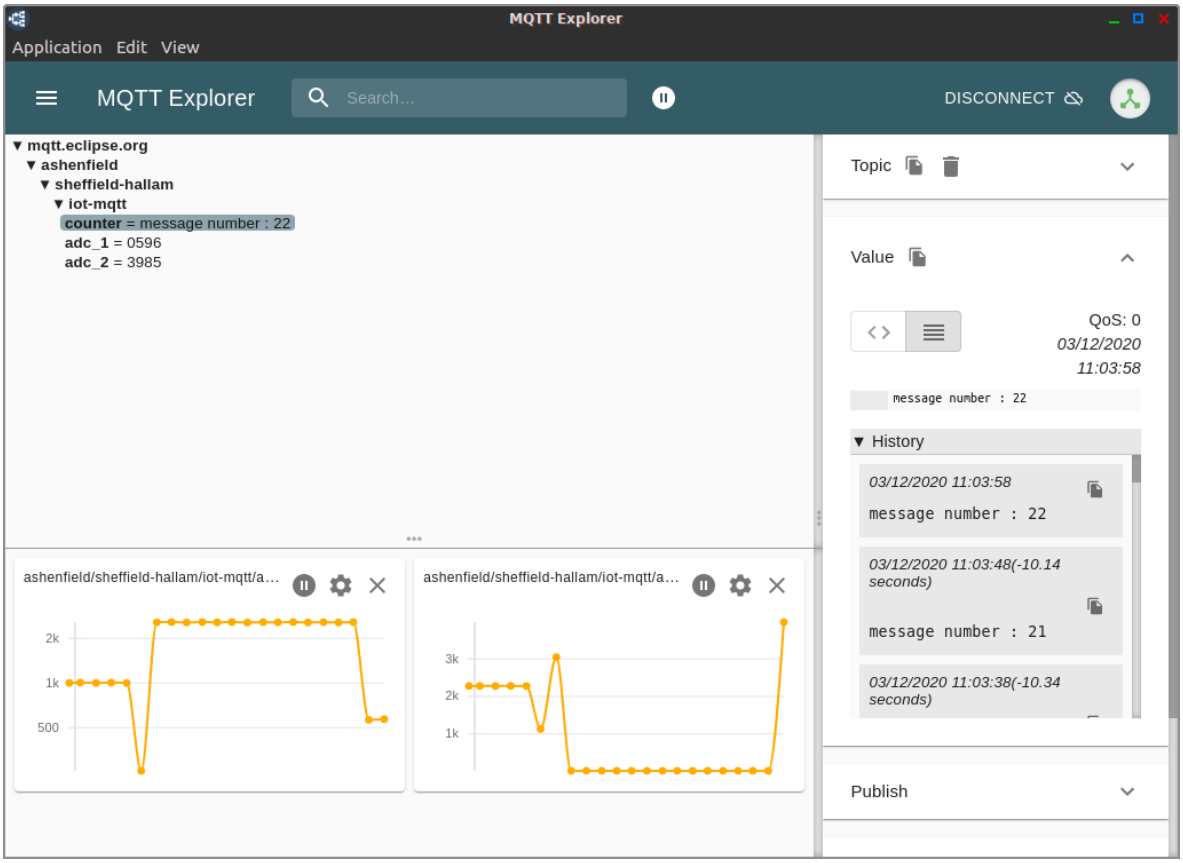


Figure 14 – Data published from the STM32F7 discovery board

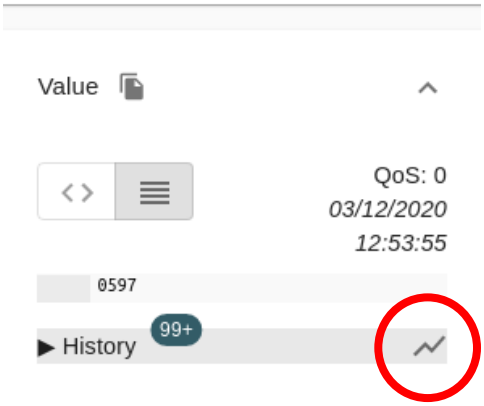


Figure 15 – Graphing the sensor data

Exercises

Now you are going to make some modifications to the program to add functionality:

1. Explore the other options of MQTT explorer – including publishing your own data to a subtopic of your root feed. What happens?

2. Replace the two potentiometers with a light sensor and a temperature sensor and change the topics they publish to to reflect this.

3. Add a temperature monitoring routine in your code that turns on and off another LED according to a temperature set point.

Task 3

In this task we are going to design and build an Internet-of-Things enabled light switch. The key requirement of this system is that the light bulb must be able to be controlled both remotely (by sending MQTT messages to the system) and locally (by pressing the attached push button).

The system architecture for this application is shown in Figure 16. Button debouncing is taken care of by a periodic timer callback that then signals the MQTT thread to publish a message to the MQTT broker reporting the desired state of the light (i.e. whether it should be on or off).

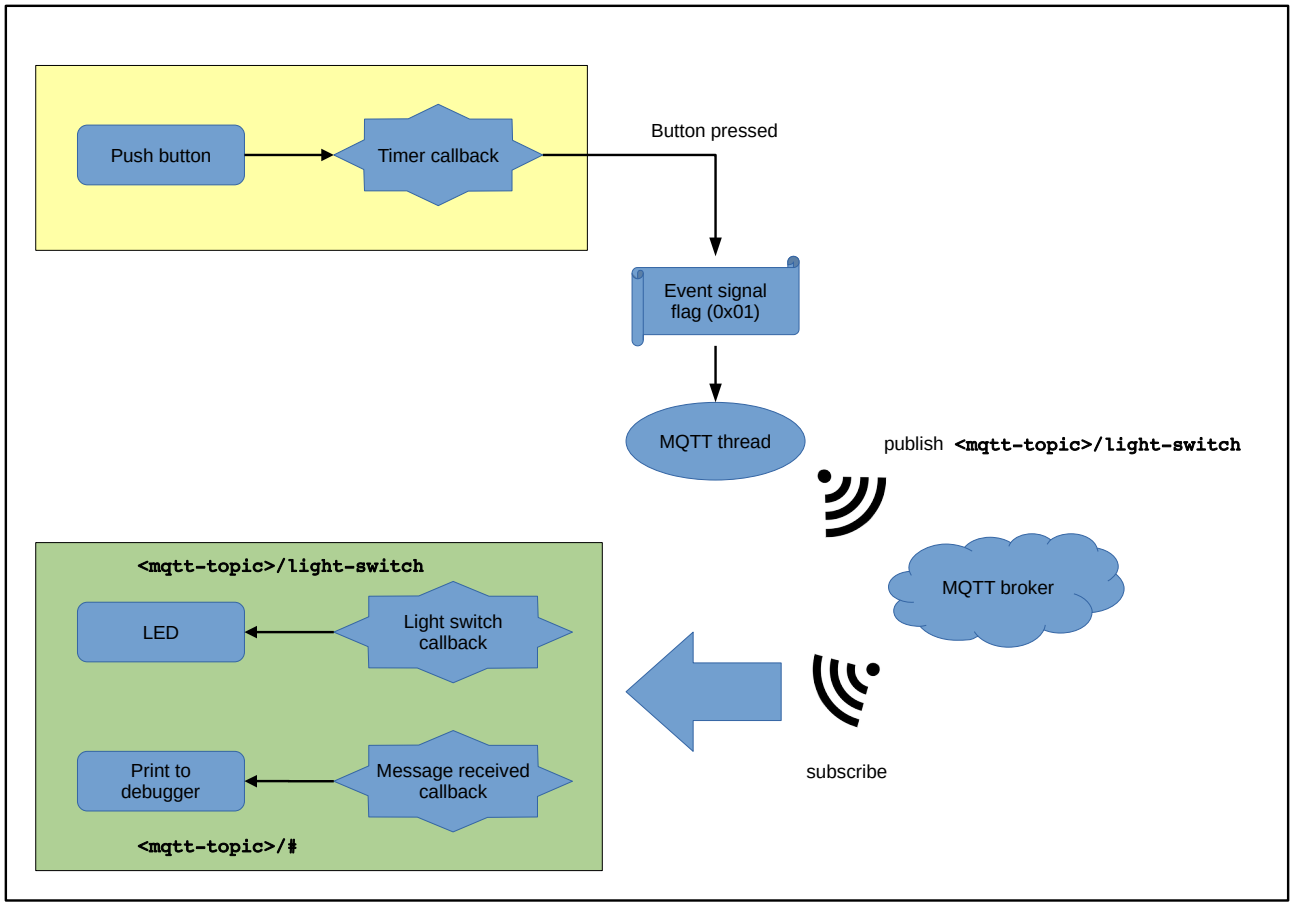


Figure 16 – The IoT light switch architecture

This use of timers in button debouncing was briefly covered in lecture 5 of the RTOS topic (and some simple example code was provided to go along with the lecture material). We will build on this code in this example to provide robust button debouncing[9] for real-time operating system based applications.

Code listing 9 shows the **`app_main.c`** file, where we create all our RTOS objects (e.g. the button debouncing timer callback, the event flags. and the application thread(s)).

Code listing 9:

```
/*
* app_main.c
*
* this is where we create the main application thread(s) and kick everything
* off
*
* author:    Dr. Alex Shenfield
* date:      17/11/2020
* purpose:   55-604481 embedded computer networks - lab 104
*/

// include functions from the c standard libraries
#include <stdint.h>
#include <stdio.h>

// include cmsis-rtos version 2
#include "cmsis_os2.h"

// include my rtos objects
#include "rtos_objects.h"

// include my gpio library
#include "pinmappings.h"
#include "gpio.h"

// DEFINES

// network initialisation function prototype is defined elsewhere
extern int32_t socket_startup(void);

// push button
gpio_pin_t pb1 = {PI_3, GPIOI, GPIO_PIN_3};

// RTOS DEFINES

// thread(s)

// mqtt thread (with increased stack size)
osThreadId_t mqtt_thread;
extern void mqtt_run_task(void *argument);
static const osThreadAttr_t mqtt_thread_attr =
{
  .name = "mqtt_thread",
  .priority = osPriorityNormal,
  .stack_size = 8192U,
};
```

9 A really good discussion of button debouncing can be found here: https://hackaday.com/2015/12/09/embed-with-elliot-debounce-your-noisy-buttons-part-i/ and here: https://hackaday.com/2015/12/10/embed-with-elliot-debounce-your-noisy-buttons-part-ii/

```
// timer

// declare a timer callback and a timer
void test_for_button_press(void * parameters);
osTimerId_t button;
static const osTimerAttr_t pb_timer =
{
  .name = "button_debouncing_timer",
};

// event flag

// declare button pressed event flag
osEventFlagsId_t button_flag;
static const osEventFlagsAttr_t button_flag_attr =
{
  .name = "button_event",
};

// MAIN THREAD

// provide an increased stack size to the main thread
static const osThreadAttr_t app_main_attr =
{
  .stack_size = 8192U
};

// main thread
static void app_main(void *argument)
{
  // print a startup message
  printf("MQTT over TLS using WiFi\r\n");

  // set up the network connection
  int32_t status = socket_startup();

  // initialise the pushbutton and create the button pressed event flag
  init_gpio(pb1, INPUT);
  button_flag = osEventFlagsNew(&button_flag_attr);

  // create the timer object for button debouncing and start it
  button = osTimerNew(&test_for_button_press, osTimerPeriodic, NULL, &pb_timer);
  osTimerStart(button, 5);

  // if the network connected ok then create the mqtt thread to send and
  // receive packets
  if(status == 0)
  {
    mqtt_thread = osThreadNew(mqtt_run_task, NULL, &mqtt_thread_attr);
  }
}
```

```
// BUTTON DEBOUNCING

// button debouncing using pattern matching (implemented as a timer callback)
void test_for_button_press(void * parameters)
{
  // 8 bits of button history
  static uint8_t button_history = 0xFF;

  // every time this timer callback is called we shift the button history
  // across and update the state
  button_history = button_history << 1;
  uint8_t val = read_gpio(pb1);
  button_history = button_history | val;

  // use some simple pattern matching to see if the button has been pressed
  // and released - if so, reset the button history and set the event flag ...
  if((button_history & 0xC7) == 0x07)
  {
    // reset button history
    button_history = 0xFF;

    // signal the button has been pressed
    osEventFlagsSet(button_flag, 0x01);
  }
}

// THREAD INITIALISATION

// initialise the application (by creating the main thread)
void app_initialize(void)
{
  osThreadNew(app_main, NULL, &app_main_attr);
}
```

So once the **app_main** thread has created the event flag, periodic timer object (and associated callback), and MQTT thread – it then lets the MQTT thread handle the rest of the application logic. Code listing 10 shows the MQTT thread (called **mqtt_run_task** in our application).

The MQTT set up code is very similar to that in the previous examples; however, in this application, we are declaring an additional MQTT subscription callback and attaching it to the

<mqtt-topic>/light-switch

subtopic (see green highlighted code). Whenever a message is received on this topic, this callback will process the message payload to see if the light bulb should be turned on or off.

Our main MQTT loop is also slightly different. In this case, we are listening out for the event flag denoting the button has been pressed (which is set in our timer callback) and, if it has been, we will send a message to the

<mqtt-topic>/light-switch

subtopic to update the desired state of the LED (i.e. if it was on, it should now be off and vice versa).

Code listing 10:

```
/*
* run_mqtt.c
*
* this file contains the logic for connecting to an mqtt broker over tls
* and sending / receiving messages (and taking action based on that
* information).
*
* author:    Dr. Alex Shenfield
* date:      17/11/2020
* purpose:   55-604481 embedded computer networks - lab 104
*/

// include the c string handling functions
#include <string.h>

// include the paho mqtt client
#include "MQTTClient.h"

// include the root ca certificate (and client certificate and key - but we're
// not using those ...)
#include "certificates.h"

// include our rtos objects
#include "rtos_objects.h"

// include my mqtt credentials
#include "my_credentials.h"

// include the shu gpio support
#include "pinmappings.h"
#include "gpio.h"

// define the mqtt server
#define SERVER_NAME     "mqtt.eclipse.org"
#define SERVER_PORT     8883

// MQTT CALLBACKS

// set up our message received callback (just dump the message to the screen)
void message_received_cb(MessageData* data)
{
  printf("message arrived on topic %.*s: %.*s\n",
         data->topicName->lenstring.len,
         data->topicName->lenstring.data,
         data->message->payloadlen,
         (char *)data->message->payload);
}
```

```
// my light switch callback

// light bulb
gpio_pin_t led = {PB_14, GPIOB, GPIO_PIN_14};

// set up light switch topic callback
void light_switch_cb(MessageData* data)
{
  // get the message from the message data object
  char message_text[128];
  sprintf(message_text, "%.*s", data->message->payloadlen, (char *)data->message->payload);

  // parse the message payload
  if(strcmp(message_text, "ON") == 0)
  {
    printf("turning on led\n");
    write_gpio(led, HIGH);
  }
  else if(strcmp(message_text, "OFF") == 0)
  {
    printf("turning off led\n");
    write_gpio(led, LOW);
  }
}

// MQTT THREAD

// set up the mqtt connection as a thread
void mqtt_run_task(void *argument)
{
  // initialise structures for the mqtt client and for the mqtt network
  // connection
  MQTTClient client;
  Network network;

  // initialise the gpio for the light bulb
  init_gpio(led, OUTPUT);
  write_gpio(led, LOW);

  // establish a secured network connection to the broker

  // set up the tls certificate
  TLScert tlscert = {(char *)CA_Cert, NULL, NULL};

  // initialise the network connection structure
  NetworkInit(&network);

  // try to connect to the mqtt broker using tls and - if it fails - print out
  // the reason why
  int rc = 0;
  if((rc = NetworkConnectTLS(&network, SERVER_NAME, SERVER_PORT, &tlscert)) != 0)
  {
    printf("network connection failed - "
           "return code from network connect is %d\n", rc);
  }
```

```
// connect the client to the broker

// initialise the mqtt client structure with buffers for the transmit and
// receive data
unsigned char sendbuf[128];
unsigned char readbuf[128];
MQTTClientInit(&client, &network, 30000, sendbuf, sizeof(sendbuf),
               readbuf, sizeof(readbuf));

// initialise the mqtt connection data structure
MQTTPacket_connectData connectData = MQTTPacket_connectData_initializer;
connectData.MQTTVersion = 4;
connectData.clientID.cstring = (char*)mqtt_clientid;
connectData.username.cstring = (char*)mqtt_username;
connectData.password.cstring = (char*)mqtt_password;

// start mqtt client as task (what this does is to make sure that the mqtt
// processing functionality is protected by a mutex so it can only run
// sequentially - i assume this means we can have subscriptions and
// publishing in multiple threads ... but we'll have to pass the client into
// those threads when they are started)
if((rc = MQTTStartTask(&client)) != 1)
{
  printf("return code from start tasks is %d\n", rc);
}

// connect ...
if((rc = MQTTConnect(&client, &connectData)) != 0)
{
  printf("return code from MQTT connect is %d\n", rc);
}
else
{
  printf("MQTT Connected\n");
}

// manage mqtt topic subscriptions

// subscribe to the everything in the root ('#') of my test topic (we will
// publish messages to a subtopic of this root so this subscription means we
// can see what we're sending). we also need to register the callback that
// will be triggered everytime a message on that topic is received (here this
// is "message_received" from earlier) and set the QoS level (here we are
// using level 2 - but you need to check what your broker supports ...)
char general_topics[] = "ashenfield/sheffield-hallam/iot-mqtt/#";
if((rc = MQTTSubscribe(&client, general_topics, QOS2, message_received_cb)) != 0)
{
  printf("return code from MQTT subscribe is %d\n", rc);
}

// subscribe to a light switch topic
char light_switch_topic[] = "ashenfield/sheffield-hallam/iot-mqtt/light-switch";
if((rc = MQTTSubscribe(&client, light_switch_topic, QOS2, light_switch_cb)) != 0)
{
  printf("return code from MQTT subscribe is %d\n", rc);
}
```

```
  // publish messages

  // run our main mqtt publish loop
  while(1)
  {
    // button check

    // check for button press (returning immediately)
    uint32_t flag = osEventFlagsWait(button_flag, 0x01, osFlagsWaitAny, 0);
    if(flag == 0x01)
    {
      // initialise a character array for the message payload
      char payload[64];

      // create the basic message and set up the publishing settings
      MQTTMessage message;
      message.qos = QOS1;
      message.retained = 0;
      message.payload = payload;

      // send the light switch message

      // write the data into the payload and send it
      memset(payload, 0, sizeof(payload));
      if(read_gpio(led))
      {
        sprintf(payload, "OFF");
        message.payloadlen = strlen(payload);
      }
      else
      {
        sprintf(payload, "ON");
        message.payloadlen = strlen(payload);
      }

      // publish the message (printing the error if something went wrong)
      if((rc = MQTTPublish(&client, light_switch_topic, &message)) != 0)
      {
        printf("return code from MQTT publish is %d\n", rc);
      }
    }
  }
}
```

My completed system looks like Figure 17.

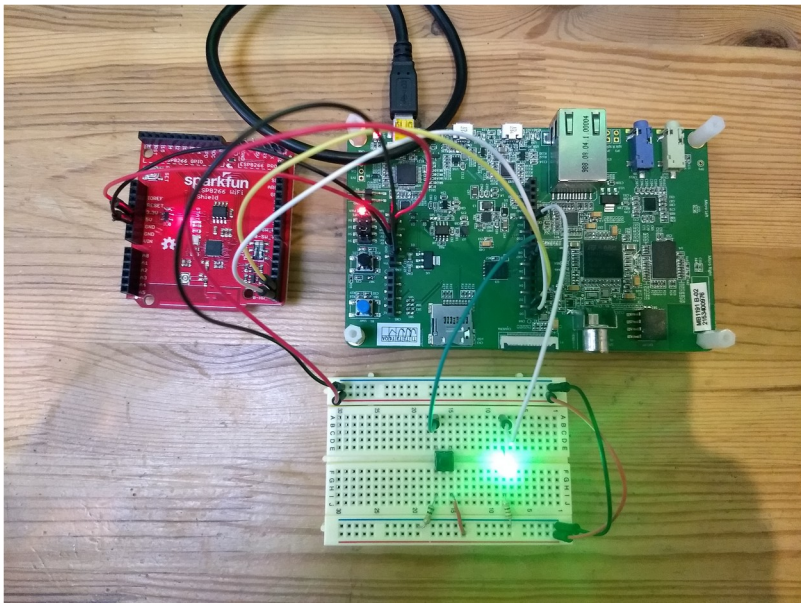


Figure 17 – Completed IoT light switch

Figure 18 shows the debug (printf) viewer log after we have pushed the button a few times.

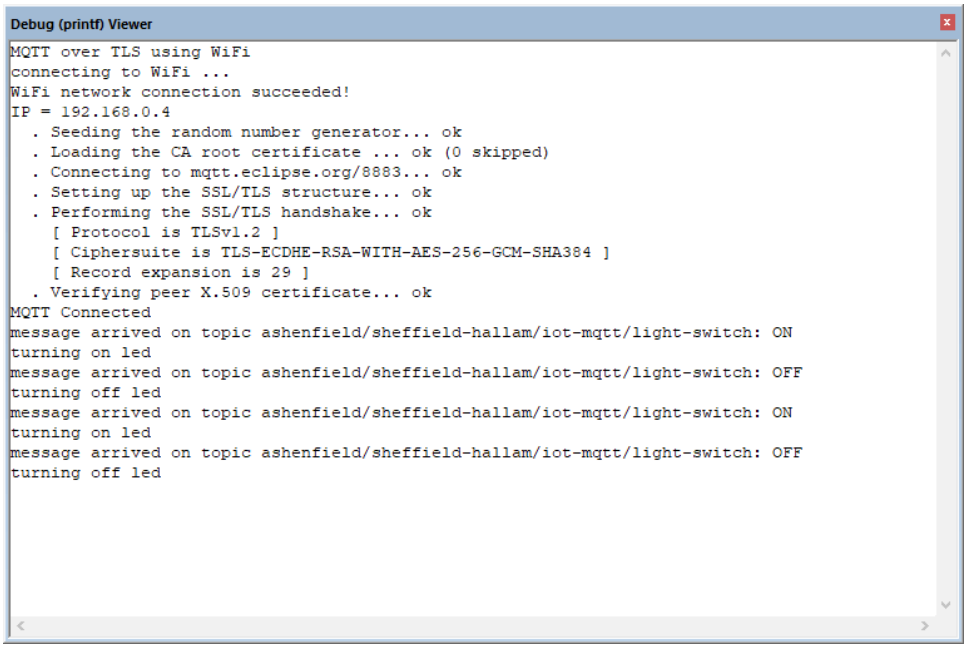


Figure 18 – The debug (printf) viewer log of our system

We can also trigger our light bulb by sending MQTT messages to the IoT broker (which are then passed back to the **`light_switch_cb`** callback function). To do this we will use MQTT explorer (as in the previous task). Figure 19 shows this process.

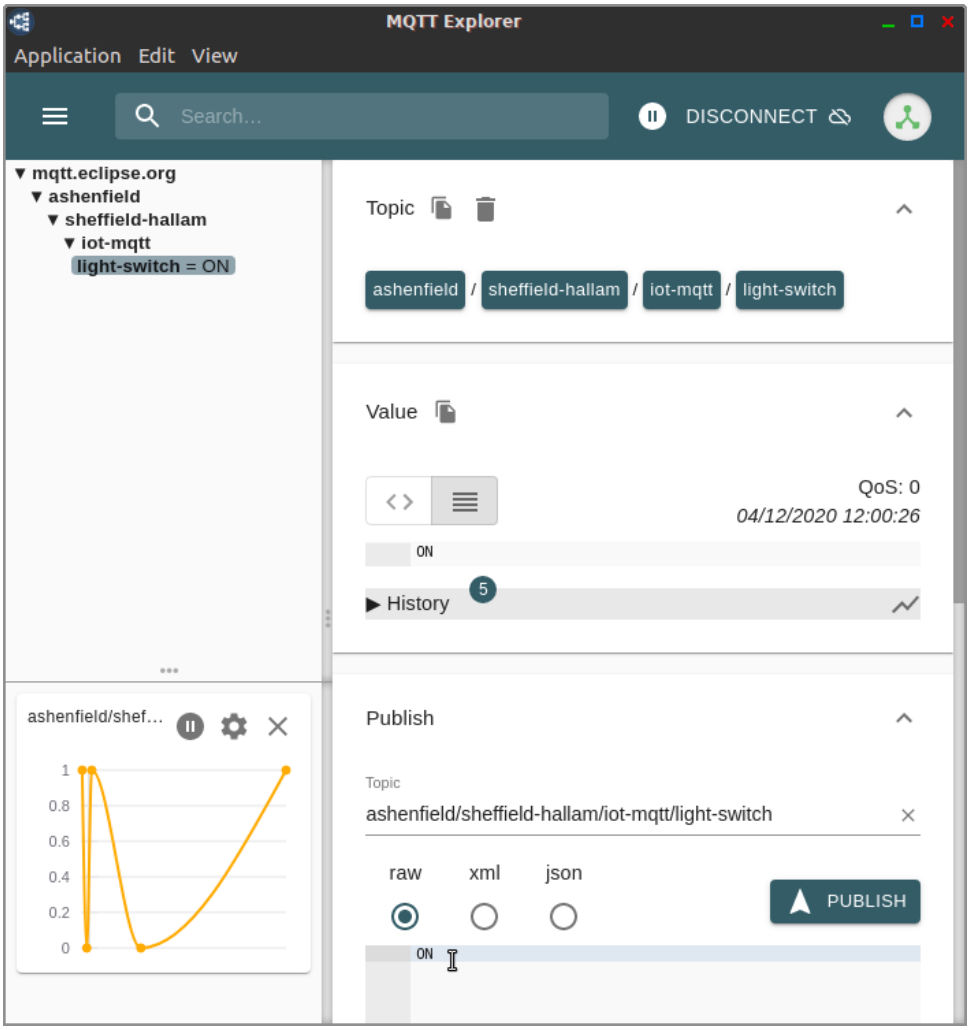


Figure 19 – Sending ON / OFF messages to our IoT light bulb from MQTT explorer

Exercises

Now you are going to make some modifications to the program to add functionality:

1. Incorporate an additional thread to read a light sensor, pass the reading over to the MQTT thread, and publish additional messages (to a separate topic) representing the current ambient light level.

2. Add the ability to turn an automatic monitoring routine on or off via MQTT messages (e.g. from MQTT explorer). This automatic monitoring routine should turn the LED on when the light level goes below a certain set point and off when it goes above the set point.

3. Add the ability to change the lighting set point via MQTT messages.

4. The are several applications for building MQTT dashboards for mobile devices (I am currently using IoTMQTTPanel for Android) – investigate using these to both display data from and send data to your embedded application. Figure 20 shows my simple dashboard to control the light switch application.

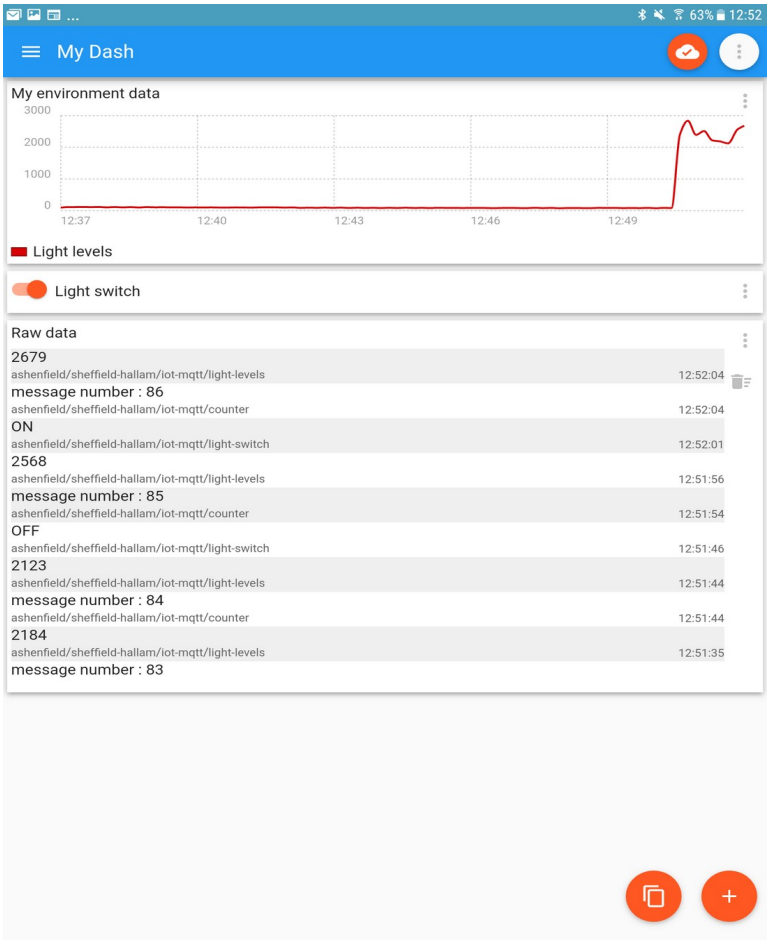
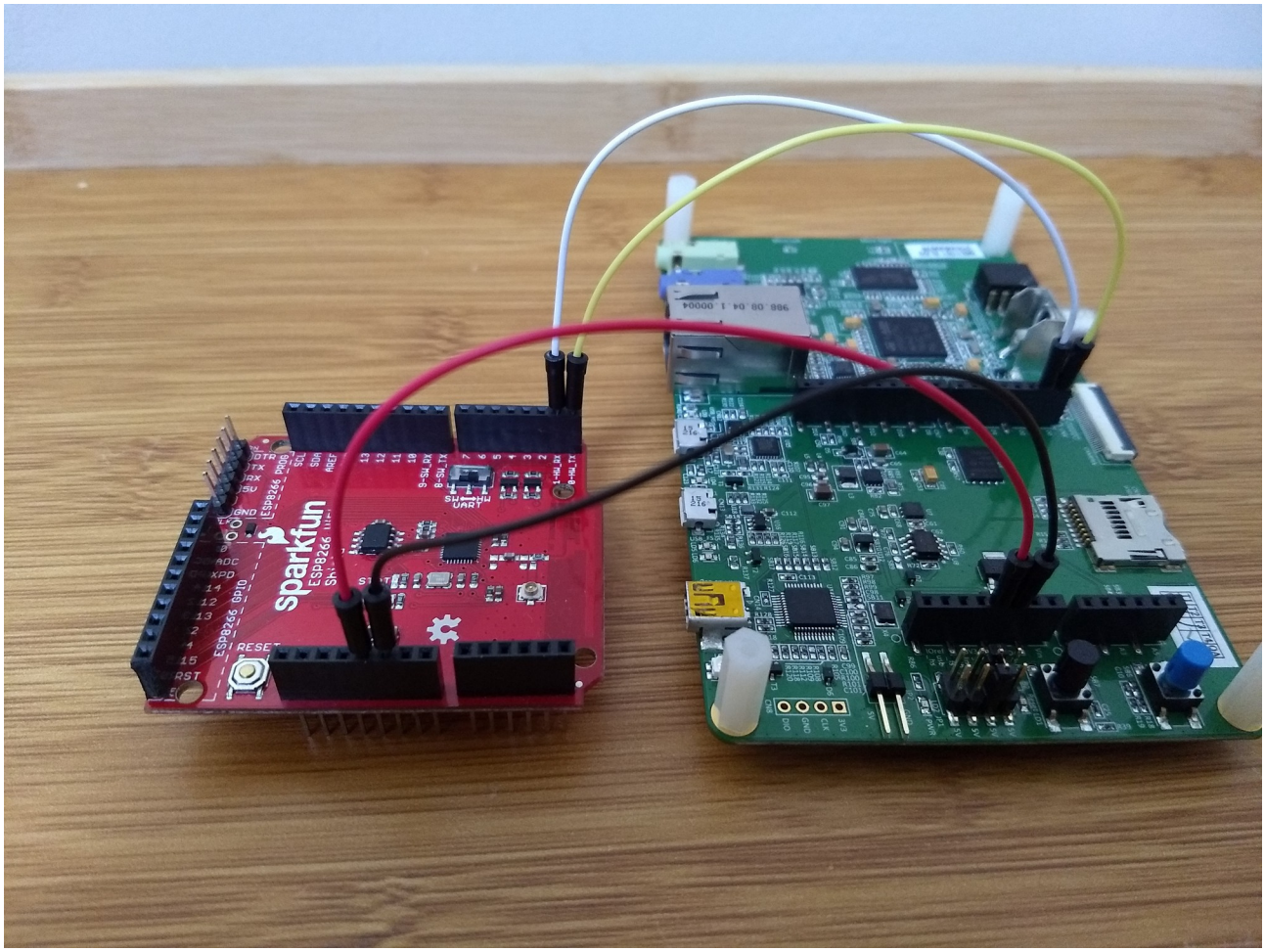


Figure 20 – IoT light switch dashboard

Appendix A – Wiring up the SparkFun WiFi shield

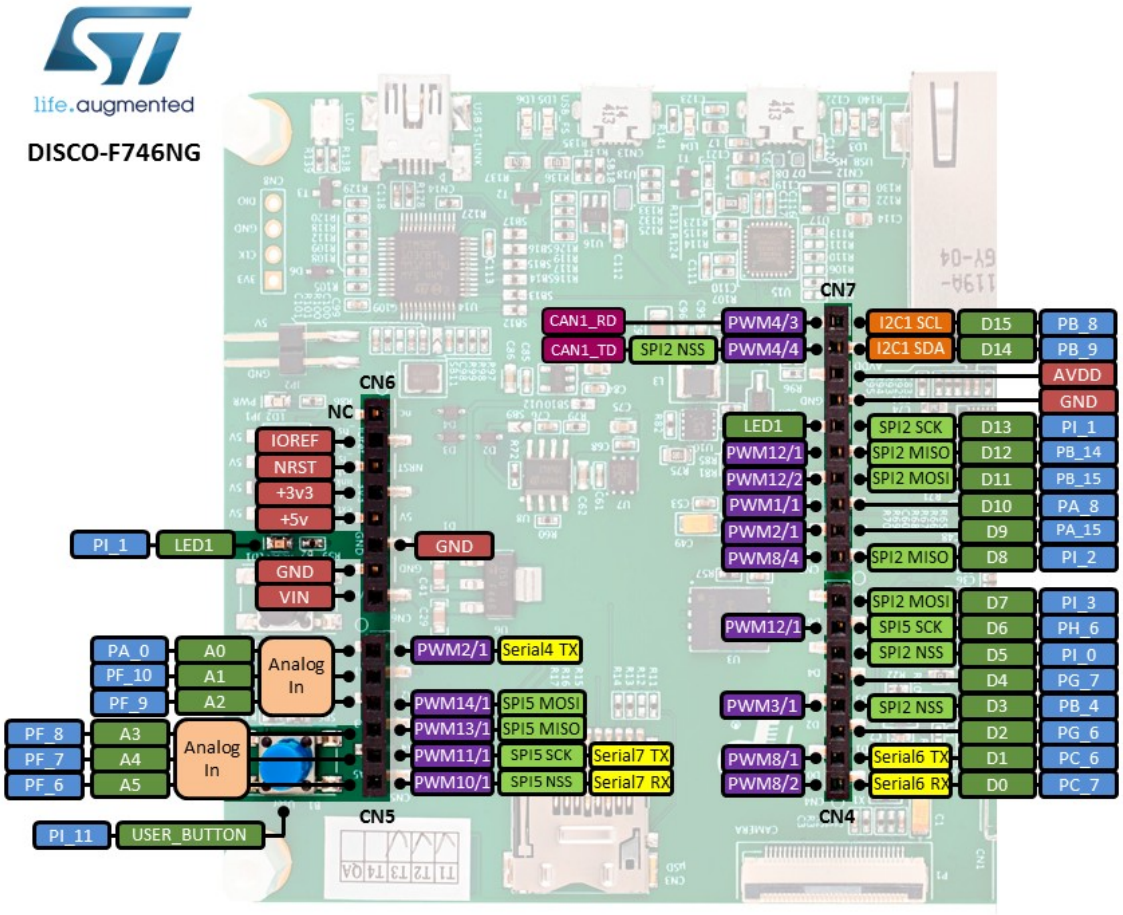


Wiring up the SparkFun WiFi shield with external connections

Note – the connections here are:

D0	→	0-HW_TX
D1	→	1-HW_RX
5V	→	5V
GND	→	GND

Appendix B – The STM32F7 discovery board schematic



STM32F7 discovery board pin outs

Check list

Feedback