Laboratory 2: Connecting to a WiFi access point and communication via UDP

Introduction

This laboratory aims to equip you with knowledge on how to:

- Connect to a wireless access point (**AP**).
- Receive packets on the ESP8266 via User Datagram Protocol (UDP).
- Transmit packets to a remote machine via UDP.
- Transmit packets containing useful information (such as sensor values).
- Use UDP packets to control the state of a LED on the Aston IOT application trainer.

Equipment and resources

Hardware:

- ESP8266 Aston IoT hardware platform.
- USB A to micro B cable.
- A laptop, access point/hotspot and packetsender.

Software & documentation:

- Arduino Integrated Development Environment (IDE).

A reminder on saving and file name convention

You should keep a copy of every task that you complete on your **student network drive (H:), in the folder H:\EE3IOT\labs.** Sketches should be saved in the format of **surnameinitialLTX**, where **L** is the lab number and **X** is the task number.

For example, if Fred Smith has completed task 1 of Lab 1, the sketch should be saved as **smithfL1T1**.

Task 1: Connecting to an access point

Note that step by step instructions on how to use the Arduino IDE will no longer be given. If guidance is required, refer to laboratory 1.

C/C++ case sensitivity

Remember that C and C++ are both **case sensitive languages**. For example, the following code will generate an error:

```
int Myvariable;
myvariable = 5;
```

As myvariable does not exist within the function and compilers scope. Case sensitivity is also commonly problematic with C functions and class methods (functions associated with the class). For example, the `serial.begin(9,600);` will generate an error (it should be `Serial.begin(9600);`)

1. Create a new sketch in Arduino IDE and set up the sketch for the Aston IOT hardware platform.
2. Enter the code given below in Figure 1.
   - N.B. The Service Set Identifier (**SSID**) is an identifier used to differentiate Wireless Local Area Networks (**WLANs**) from each other and forms a part of the IEEE 802.11 specifications.
   - **const char** defines character storing variables. The square brackets after the variable's name (e.g. ssid**[]**) defines that the variable is an array. Normally, the length of the array would have to be specified within the square brackets. However, when initialising a char array with a string (e.g. "EE3IOT" or "EE4IOT"), the C compiler sets the length of the array automatically.

– The while loop in the setup function waits for a connection the access point to be established and prints a full-stop every 0.5 seconds whilst waiting for a connection to be made.

```
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>

const char ssid[]     = "EE3IOT";
const char password[] = "abadpassword";

void setup()
{
  Serial.begin(115200);                         // open a serial port at 115,200 baud
  Serial.print("Attempting to connect to ");    // Inform of us connection attempt
  Serial.print(ssid);                           // print the ssid over serial

  WiFi.begin(ssid, password);                   // attemp to connect to the access point SSID with the password

  while (WiFi.status() != WL_CONNECTED)         // whilst we are not connected
  {
    delay(500);                                 // wait for 0.5 seconds (500ms)
    Serial.print(".");                          // print a .
  }
  Serial.print("\n");                           // print a new line
  Serial.println("Successfully connected");      // let the user know a connection has been established

  // add more lines here to print the IP address etc

}

void loop()
{
   while(1);      // note that this introduces numerous problems, but we will look at ways of not
                  // not writing blocking code in week 3.
}
```

Remember to replace this with your own access point or hot spot.

Figure 1: Code to connect to an access point

3. Change the SSID as follows:
   – If you are in private accommodation, use your WiFi access point SSID and password (as you would normally use for your phone)
   – If you are in student accommodation or the university, use either a WiFi hotspot from your phone or if in the IOT lab, the access point details there. These boards will not work on Aston Connect etc. If you are using a hot spot, you will need to ensure you laptop/PC is also connected to this hotspot when using packetsender.
4. Download the sketch to the Aston IOT hardware platform.
5. Open the **Serial Monitor,** ensuring to select the correct Baud rate **(115,200)**.
   – Does the ESP8266 successfully connect to the access point? Note it may be worth to press the **RST** (reset) button on the **ESP8266 module** (top left of the Aston IOT hardware platform) to ensure that you are observing your sketch starting from the beginning.
   – Does the serial monitor do anything unexpected? The **while(1)** in the **loop() function** is an **infinite loop** with nothing in it. It will prevent the loop function from completing as the processor will continuously execute this loop, as a 1 is always true.
6. Your sketch will generate errors that can be observed with the serial monitor, with the module reporting a "**soft WDT reset**", followed by the connection process starting again. This is known as a Watch Dog Timer reset and is typically caused by blocking code, i.e. code that waits for an event to happen or stops other code from executing. This is a safety feature as the **WDT** will reset the microcontroller if the code hangs or crashes as it better to start the code again and hope for the best rather than doing nothing!

   Typically, the sketch must reset the timer periodically to prevent the WDT reset being activated, although this job is performed for us by the Arduino libraries. The **while(1)** loop prevents this, and as

of such the **WDT** resets the ESP8266 module. A flow-chart of the typical process is given in for reference Figure 2. This is important as any code created must not stop the loop function from completing within a reasonable time frame. This is the reason why in laboratory 1 the button debouncing code did not use a loop to wait for the button press and instead used an **if** to check the button state every time the loop function was executed.

Approaches to remove blocking code and to schedule tasks at regular intervals will be covered in laboratory three.

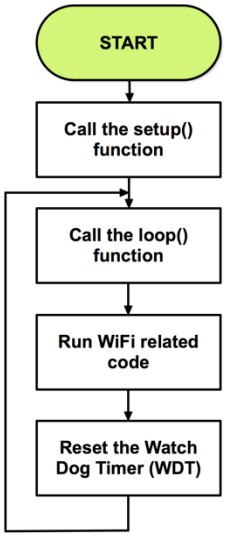

Figure 2: Simplified ESP8266 Arduino program flow

7. **Remove** the **while(1)** loop from the **loop()** function, leaving the function empty.
8. Download the sketch to the Aston IOT hardware platform and monitor the output from the ESP8266 on the **Serial Monitor**.
   - Confirm that the microcontroller no longer resets due to the WDT timer timing out.
9. Modify the code in the **setup()** function to print the:
   - **MAC** address, the function **WiFi.macAddress()** can be used to determine the MAC address.
   - **IP** address, the function **WiFi.localIP()** can be used to determine the IP address.
   - **Subnet mask**, the function **WiFi.subnetMask()** can be used to determine the subnet mask.
   - **Gateway**, the function **WiFi.gatewayIP()** can be used to determine the gateway IP address.
   - **Domain Name Server**, the function **WiFi.dnsIP()** can be used to determine the DNS address.
   - Hint: **Serial.print** and **Serial.println** can be used to print text to the Serial Monitor. Note that the result from a function (such as those above) can be printed by placing the function call where you normally place an argument such as text or a variable.
10. Upload the modified sketch to the board and verify that you obtain the expected addresses.
11. Make a copy of this sketch, saved as **basicwificonnection** the **setup()** function will form the basis of subsequent laboratories

Task 2: UDP initialisation

1. Create a new sketch, and copy the code from task 1 to use as a starting point.
2. Modify the code such that you include an **additional** library in the sketch, **WiFiUdp.h**
   - To include a new library use **#include**

- Library includes should be placed at the top of a sketch before any functions.

3. Include a **#define** macro that defines **SOCKET** as **8888**.
   - Our sketch will listen on **port 8888** for packets being transmitted from the packetsender.
4. Create a class instance of **WiFiUDP** as:

```
WiFiUDP UDP;
```

   - This should be performed some place **above** the **setup()** function.
5. Now that a class instance exists, we can initialise the library and start listening on a port to receive **UDP** packets. This will be achieved by calling the **WiFiUDP.begin()** method. Call the method **begin()** at the end of your **setup()** function.
   - The method accepts one argument and returns a 1 on the successful initialisation of the socket. Refer to https://www.arduino.cc/en/Reference/WiFiUDPBegin for further details.
   - Note that **WiFiUDP** should be replaced by the name of your class instance, i.e. **UDP.begin()**
6. Add code such that your sketch checks to see if the socket was successfully opened, and prints a message to the serial monitor indicating whether the initialisation was successful. This will require the use of a conditional flow control, such as an **IF**.

Task 3: UDP communication

Now that we have initialised the UDP socket and associated processes through the **WiFiUDP.begin()** method, the library can now be used to receive packets and transmit a packet on reception as a form of acknowledgement (remember that UDP does not guarantee delivery or the reception of packets in the correct order – such functionality must be implemented by the programmer or by using protocols which implement such functionality i.e. Transmission Control Protocol (**TCP**)).

1. Add the following code **below** the **#include** directives and **above** your **setup()** function.
   - This defines two **global** arrays of **chars** (8 bit values used to store characters).
   - A macro is used (**#define**) such that the length of the arrays can easily be changed throughout the entire sketch by changing one value.

```
#define BUFFERLEN 255
char inBUFFER[BUFFERLEN];     // buffer to store incoming packets
char outBUFFER[BUFFERLEN];    // a reply string to send back
```

Figure 3: Code to be entered after #includes and before setup()

2. Modify the sketch from Task by entering the code given below in Figure 4 into the loop function of your sketch.
   - **parsePacket must** be called first to evaluate the data received, check if it is a UDP packet and if so, how many bytes are contained within the packet. The **parsepacket** method returns the size of the packet as an **integer variable** which is stored in **packetsize**.
   - An **IF** flow control structure checks if any packets have been received. This is done by the placing **packetsize** within the IF's braces, i.e. **if (packetsize)**. Remember that a logical condition is true in the C language if the value is **different to 0**. Hence, when we do not receive a packet (packetsize = 0) the code within the **IF**'s code block is not executed. However, when **packetsize** is greater than 0, the conditional expression evaluates to **TRUE** and the code within the braces is executed.
   - The contents of the library's (commonly referred to as a **software stack**) parsed **UDP** buffer are copied into a buffer in our sketch via **read(destination, length)** method. Note that the first parameter determines the array of where the message will be copied to and length determines the number of bytes to be copied. This is to avoid a problem known as **buffer overflow**, as the length parameter will always be set to the length of the array or less.

- A **'\0'** or **null terminator** is inserted at the end of our local input buffer **inBUFFER[packetlength] = '\0';** . This is to ensure that only the current string data is printed. In C, functions read a string until a null terminator is found, as the null terminator is used to indicate the end of a string. If this is missing, the code will continue printing characters until a null terminator is found (typically inserted via the longest message sent).
- The number of bytes received, the source IP address and the source port are then printed over serial for debugging/information purposes.
- Finally, a UDP packet is sent back to the source IP address and port. This is achieved by calling the **beginPacket(destination IP, destination port)** method to set the **destination IP** and **port**.
- **strcpy(destination, source)** is then used to copy a message into our local output buffer, **outBUFFER**. The first argument is the destination array, and the second argument is the source, this could be another array or as our case here, a constant string literal. Note that the contents of destination array will be **replaced** with the source array, although this is not a problem here, it is worth bearing in mind!

```
void loop()
{
  int packetsize = 0;                 // a temporary variable to store the size of received packets
  packetsize = UDP.parsePacket();     // check for the presence of a UDP packet and return the size of it.

  if (packetsize)                     // if a packet is present, i.e. size greater than 0
  {
    UDP.read(inBUFFER,BUFFERLEN);     // read the contents of the UDP packet to inBUFFER, with a max length of BUFFERLEN
    inBUFFER[packetsize] = '\0';      // add a null terminator at the end of the buffer to ensure we always only read a correct length string
    Serial.print("Received ");        // print the number of characters (bytes) received.
    Serial.print(packetsize);
    Serial.println(" bytes");

    Serial.println("Contents:");      // print the contents of the packet
    Serial.println(inBUFFER);         // we do this by printing the inBUFFER as a string
    Serial.print("From IP ");         // print the IP of the remote connection (i.e. the PC that sent the message)
    Serial.println(UDP.remoteIP());   // simply print the result of the remoteIP method.
    Serial.print("From port ");       // print the the PORT that sent the packet - i.e. what port sent a packet to 8888
    Serial.println(UDP.remotePort()); // Simply print the result of the remotePort method

    // Start a UDP packet to send back to the source IP. This is achieved via sending to the remote IP and the remote port
    UDP.beginPacket(UDP.remoteIP(),UDP.remotePort());

    // copy a simple message into the output message array. This message could be replaced by something else.
    strcpy(outBUFFER, "This is a a reply from Dr Nock's ESP8266 firmware");
    // Fill the UDP buffer with the contents of our output message array
    UDP.write(outBUFFER );
    UDP.endPacket();                  // call end packet to send the packet out over the network

  }
}
```

Figure 4: Code to receive a UDP packet and reply to senders IP and port with a reply

3. Compile your sketch, debug any errors and upload the sketch to the Aston IOT hardware platform.
4. Open the **serial monitor** and make a note of the IP address that has been assigned to your ESP8266.
   - **N.B. This may change each time you connect dependent upon other connections.**
5. Configure packetsender (a free utility used for sending and receiving UDP and TCP packets) as :
   - **Name:** UDPTest followed by your username. (Note that this could be anything and is only a label if you wish to save the packet for later testing)
   - **ASCII:** Hello, this is **username** testing.
   - **Hex:** This can be left alone as it will be automatically populated from the ASCII string you enter above.
   - **Address:** The IP address that you noted from step 4.
   - **Port:** 8888 (we have set this in the code created)
   - **Resend Delay:** Leave at the default value.
   - Ensure **UDP** is selected (to the left of the send button).

6. Try sending a packet to your Aston IOT hardware platform. Does it respond as expected?

Task 4: Replying with useful data

This task will modify your sketch from **task 3** such that your sketch replies with actual useful data.

1. Refer to your sketch from **Laboratory 1 task 2**. Using the sketch from task 3 of this lab as a starting point, add code such that each time the ESP8266 receives a UPD packet, it samples the **LDR** circuit output with the **ADC** and stores the value in an integer variable.
2. Remove the **strcpy()** function call. Where the **strcpy()** function call was, call the **sprintf** C function to convert the ADC integer value into a string (char array) with accompanying text. The string should be stored in the **outBUFFER** array and should be formatted in the form of **username ADC value is x**, where X is the ADC value.
   - Information on the function can be obtained here (note that the example is targeted for a command line application, the Arduino environment does not use printf): http://www.cplusplus.com/reference/cstdio/sprintf/
3. Upload the sketch to the IOT hardware platform and verify it works as expected by checking with packetsender.
   - Upon sending a packet to your board, the board should respond with a string containing your username and ADC value.

Task 5: Remote control

This task will modify your sketch from **task 4** such that the state of the LED on your board can be controlled remotely.

1. In the l**oop()** function after the `inBUFFER[packetsize] = '\0';` line, add code that examines the contents of the **inBUFFER** array. If **inBUFFER** is equal to:
   - **"LEDON",** turn the LED connected to pin 15 **on.**
   - **"LEDOFF",** turn the LED **off**.
   - **strcmp()** is C function that compares if strings are equal and returns a 0 on equality being found. Refer to http://www.cplusplus.com/reference/cstring/strcmp/ for further details.
   - Do not forget to set the LED pin as an output, refer to lab 1 for guidance!
2. Upload the sketch to the IOT hardware platform and verify it works as expected by checking with packetsender. You should be able to control the state of the LED remotely.

