CHAPTER 8

Updating a webpage

Devices connect to a wireless local area network (WLAN) with Wi-Fi communication (see Chapter 7 Wireless local area network). A router connected to an Internet Service Provider (ISP), through a telephone line using DSL (Digital Subscriber Line) technology, provides access to the internet (see Figure 8-1). The internet is formed by interconnected computer networks using the Internet Protocol suite, consisting of the Transmission Control Protocol and the Internet Protocol (TCP/IP), to globally link devices. The World Wide Web (WWW) identifies information resources by URLs (Uniform Resource Locator). The client or web browser, such as *Google Chrome* or *Mozilla Firefox*, sends an HTTP (HyperText Transfer Protocol) request to the server hosting device to retrieve information at the web address defined by the URL. The server responds to the client HTTP request and the client displays the requested webpage information on the Android tablet or mobile phone. The client must request webpage information from the server, as the server cannot impose updates on the client. The exception is WebSocket, as discussed in Chapter 9.

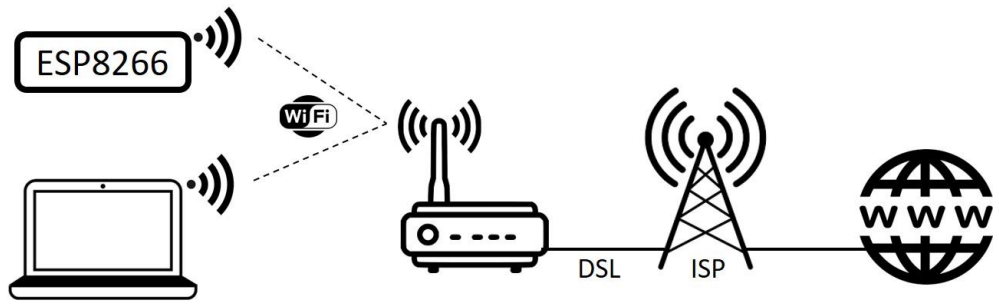

Figure 8-1. *Client, internet service provider and WWW*

In the Chapter, an ESP8266 or an ESP32 microcontroller is the server. The *Webserver* library instructions for the ESP8266 microcontroller are:

```
#include <ESP8266WebServer.h>
ESP8266WebServer server
```

and instructions for the ESP32 microcontroller are:

```
#include <WebServer.h>
WebServer server(80);                    // requires a port number
```

To demonstrate the process of a client requesting information from a server, the sketch in Listing 8-1 displays the BMP280 temperature reading, a counter and the state of an LED (see Figure 8-2). The webpage is automatically refreshed and the reloading time is determined by the webpage HTML code `<meta http-equiv='refresh' content='N'>`, with a refresh every *N* seconds. A function, *BMP*, updates the BMP280 temperature reading, increments the counter, changes the LED state and the server then sends the updated webpage HTML code to the client. The timing of the *BMP* function is controlled by the *Ticker* library, with the instruction `timer.attach(T, BMP)`, and the *BMP* function is called every *T* seconds. If the webpage refresh interval, *N*, is substantially less than the *BMP* function call interval, *T*, then the client will wait (*T*-*N*) seconds before the server sends the updated HTML code. In Listing 8-1, the *BMP* function call interval equals *N+1* seconds, but not *N* seconds as the server and client are not synchronised.

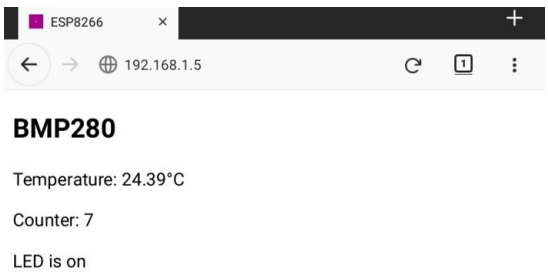

Figure 8-2. *BMP280 and LED counter webpage*

Figure 8-3 shows the BMP280 temperature sensor and LED with ESP8266 and ESP32 development boards. Connections are given in Table 8-1. The *Adafruit_BMP280* library is available within the Arduino IDE.

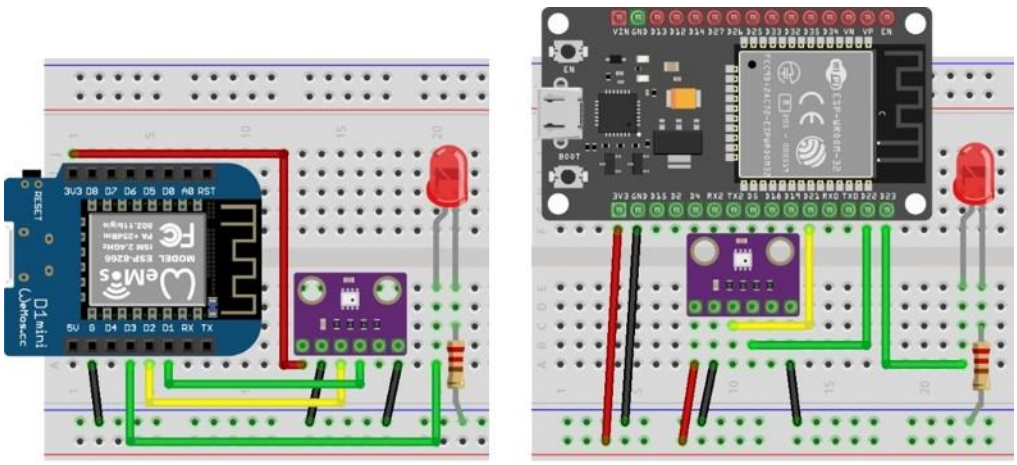

Figure 8-3. *BMP280 and LED with LOLIN(WeMos) D1 mini and ESP32 DEVKIT DOIT*

Table 8-1. *BMP280 and LED with ESP8266 and ESP32 development boards*

Component	ESP8266 connections		ESP32 connections	
BMP280 VCC	3V3		3V3	
BMP280 GND	GND		GND	
BMP280 SDA	D2		GPIO 21	
BMP280 SCK	D1		GPIO 22	
BMP280 SD0	GND		GND	
LED long leg	D3		GPIO 23	
LED short leg	220Ω resistor	GND	220Ω resistor	GND

In the *setup* function of Listing 8-1, the Wi-Fi connection is established, the server IP address is displayed on the Serial Monitor, the WLAN webpage is mapped to the *webcode* function and the timing of the *BMP* function is defined. The *ESP8266WiFi* library is referenced by the *ESP8266WebServer* library, so does not need to be explicitly included in the sketch. Similarly, for the ESP32 microcontroller with the *WebServer* and *WiFi* libraries. The *webcode* function returns a string, *page*, containing the webpage HTML code with updated values of the temperature, counter and LED state. There are no conditional statements in the HTML code, as in Chapter 7 (Wireless local area network), so the webpage URL is mapped to the *webcode* function directly. A line-by-line build-up of the HTML code incorporates the variables: *temp*, *counter* and *LED*, which are not constant, so HTML code cannot be included as a *string literal*. Variable values in HTML code are enclosed in single quotes, as in the instruction: `"<meta http-equiv='refresh' content='9'>"` with the HTML code enclosed in double quotes to indicate a string.

Listing 8-1. *HTTP request with BMP280 and LED*

```
#include <ESP8266WebServer.h>                  // include ESP8266WebServer lib
ESP8266WebServer server;                       // associate server with library
char ssid[] = "xxxx";                          // change xxxx to Wi-Fi SSID
char password[] = "xxxx";                      // change xxxx to Wi-Fi password
#include <Adafruit_Sensor.h>                   // include Unified Sensor
#include <Adafruit_BMP280.h>                   //         and BMP280 libraries
Adafruit_BMP280 bmp;                           // associate bmp with BMP280
int BMPaddress = 0x76;                         // I2C address of BMP280
#include <Ticker.h>                            // include Ticker library
Ticker timer;                                  // associate timer with Ticker lib
int lag = 10;                                  // set timer interval at 10s
int LEDpin = D3;                               // LED pin on D3
String LED = "off";                            // initial LED state
int count = 0;
String temp, counter;
```

```
void setup()
{
  Serial.begin(115200);                          // define Serial Monitor baud rate
  WiFi.begin(ssid, password);                    // initialise Wi-Fi
  while (WiFi.status() != WL_CONNECTED) delay(500);    // wait for Wi-Fi connect
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());                // display server IP address
  server.begin();
  server.on("/", webcode);                       // map URL to function
  bmp.begin(BMPaddress);                         // initialise BMP280
  timer.attach(lag, BMP);                        // BMP called every lag seconds
  pinMode(LEDpin, OUTPUT);
  digitalWrite(LEDpin, LOW);                     // turn off LED
}

void BMP()                                       // function to get readings
{
  temp = String(bmp.readTemperature());          // update BMP280 reading
  counter = String(count++);                     // increment counter
  digitalWrite(LEDpin, !digitalRead(LEDpin));    // turn on or off the LED
  if(LED == "on") LED = "off";                   // update LED state
  else LED = "on";
  server.send (200, "text/html", webcode());     // send response to client
}

String webcode()                                 // return HTML code
{
  String page;
  page = "<!DOCTYPE html><html><head>";
  page += "<meta http-equiv='refresh' content='9'>";        // refresh every 9s
  page += "<title>ESP8266</title></head>";
  page += "<body>";
  page += "<h2>BMP280</h2>";
  page += "<p>Temperature: " + temp + " " + "&degC</p>";   // display temp
  page += "<p>Counter: " + counter + "</p>";               // counter
  page += "<p>LED is " + LED + "<p>";                      // LED state
  page += "</body></html>";
  return page;
}

void loop()
{
  server.handleClient();
}
```

The client HTTP request results in the server sending the HTML code for the whole webpage with the whole webpage then reloaded. The webpage in Listing 8-1 is for example purposes only, but if the webpage contained more information and images, then the time to reload the whole webpage would be important.

Listing 8-1 is for an ESP8266 microcontroller. The only changes to the sketch for an ESP32 microcontroller are including the *WebServer* library, rather than the

ESP8266WebServer library, and defining the LED pin. An alternative to microcontroller specific sketch instructions is to use the compiler directive equivalent of an *if..then..else* group for conditional compilation of the sketch. Instructions for both the ESP8266 and ESP32 microcontrollers are included in the sketch. If the microcontroller is not an ESP8266 or ESP32, then an error message is displayed in the Arduino IDE. Further details are included in Chapter 21 (Microcontrollers). For example, Listing 8-2 contains the instructions to include at the start of Listing 8-1.

Listing 8-2 *Pin definitions for ESP8266 and ESP32 development boards*

```
#ifdef ESP32
  #include <WebServer.h>                          // include ESP32 library
  WebServer server (80);                          //       and define LED pin
  int LEDpin = 23;
#elif ESP8266
  #include <ESP8266WebServer.h>                   // include ESP8266 library
  ESP8266WebServer server;                        //       and define LED pin
  int LEDpin = D3;
#else                                             // Arduino IDE error message
  #error "ESP8266 or ESP32 microcontroller only"
#endif
```

XML HTTP requests, JavaScript and AJAX

Chapter 7 (Wireless local area network) describes updating a specific variable on a webpage with an XML HTTP request, rather than having to reload the whole webpage. Converting the sketch in Listing 8-1 from HTML code to AJAX code does not require the *Ticker* library, so the following instructions are deleted:

```
#include <Ticker.h>
Ticker timer
int lag = 10
timer.attach(lag, BMP)
```

as timing of webpage updates is managed by the AJAX code. The *BMP* function in Listing 8-1 is no longer required and is split into three functions, *tempFunct*, *countFunct* and *LEDfunct*, to update the BMP280 temperature reading, increment the counter and change the LED state. Each function sends updated information for one variable to the client. The *base* function replaces the *webcode* function, in Listing 8-1, and sends the default webpage HTML code to the client when the webpage is initially loaded. URLs are mapped to the four functions with the instructions:

```
server.on("/", base);
server.on("/tempUrl", tempFunct);
server.on("/countUrl", countFunct);
server.on("/LEDurl", LEDfunct);
```

The sketch in Listing 8-3 lists the functions to source data and instruct the server to send the information to the client. Note that the parameters *"text/html"* and *page* are included in the *base* function for the server to return HTML code to the client, while the *tempFunct*, *countFunct* and *LEDfunct* functions include *"text/plain"* and the variable name.

Listing 8-3. *XML HTTP requests for BMP280 temperature, counter and LED state*

```
void base()                                   // function to load default webpage
{                                             // and send HTML code to client
  server.send (200, "text/html", page);
}

void tempFunct()                              // function to get temperature reading
{                                             // and send value to client
  temp = String(bmp.readTemperature());
  server.send (200, "text/plain", temp);      // send plain text not HTML code
}

void countFunct()                             // function to increment counter
{                                             // and send value to client
  counter = String(count++);
  server.send (200, "text/plain", counter);
}

void LEDfunct()                               // function to update LED
{                                             // and send LED state to client
  digitalWrite(LEDpin, !digitalRead(LEDpin));
  if(LED == "on") LED = "off";
  else LED = "on";
  server.send (200, "text/plain", LED);
}
```

Listing 8-4 contains the AJAX code for the webpage and XML HTTP requests, defined as a *string literal*. The AJAX code is contained in the *buildpage.h* tab to separate the AJAX code from the main sketch with the instruction `#include "buildpage.h"`. The additional tab is created in the Arduino IDE by selecting the triangle below the *Serial Monitor* button, on the right side of the IDE, and choosing *New Tab* from the drop-down menu. The *New Tab* is titled *buildpage.h*. Note that the *loop* function only includes the instruction `server.handleClient()`.

The `<body>` section contains the webpage HTML code and the variables *tempId*, *countId* and *LEDid* correspond to the XML HTTP requests. The JavaScript instruction `setInterval(function, time)` controls the time interval, in milliseconds, between the XML HTTP requests, which is five seconds for the *reload* function to obtain the temperature reading and one second for the counter and LED state. JavaScript scripts, bracketed by

`<script>...</script>`, are positioned prior to the HTML `</body>` code to improve webpage display speed.

The whole webpage is no longer reloaded when the temperature or LED state are updated, as only specific variables are renewed. On the web browser, the *webpage loading* indicator, located beside the webpage title, is now absent.

Listing 8-4. *AJAX request with BMP280 and LED*

```
char page[] PROGMEM = R"(
<!DOCTYPE html><html>
<head><title>ESP8266</title></head>
<body>
<h2>BMP280</h2>
<p>Temperature: <span id = 'tempId'>0</span>&degC</p>
<p>Counter: <span id = 'countId'>0</span></p>
<p>LED is <span id = 'LEDid'> </span><p>

<script>
setInterval(reload, 5000);                  // time in milliseconds
function reload()                           // reload function called every 5s
{                                           // to get tempId from tempUrl
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function()
  {
    if(this.readyState == 4 && this.status == 200)
    document.getElementById('tempId').innerHTML = this.responseText;
  };
  xhr.open('GET', 'tempUrl', true);
  xhr.send();
}

setInterval(LEDreload, 1000);
function LEDreload()                        // LEDreload function called every 1s
{                                           // to get countId from countUrl
  var xhr = new XMLHttpRequest();           // and LEDid from LEDurl
  xhr.onreadystatechange = function()
  {
    if(this.readyState == 4 && this.status == 200)
    document.getElementById('countId').innerHTML = this.responseText;
  };
  xhr.open('GET', 'countUrl', true);
  xhr.send();

  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function()
  {
    if(this.readyState == 4 && this.status == 200)
    document.getElementById('LEDid').innerHTML = this.responseText;
  };
  xhr.open('GET', 'LEDurl', true);
  xhr.send();
}
</script>
</body></html>
)";
```

Each variable displayed on the webpage is associated with an XML HTTP request, the URL associated with the variable and a function to obtain the variable value. For consistency and to make the sketch more readily interpretable, a variable named *X* is referenced as *Xid* in the webpage HTML code and XML HTTP request, the URL is referenced as *Xurl* in both the *setup* function of the main sketch and the XML HTTP request, with the function sourcing the variable value referenced as *Xfunct* in the main sketch.

JSON

In Listing 8-3, each variable was associated with a specific function attached to a specific URL and in Listing 8-4, each variable was updated by a separate XML HTTP request. For data collected simultaneously, it is more efficient to have one XML HTTP request for all variables and one function, attached to one URL, to source the information. JSON (JavaScript Object Notation) combines several variable values as text, which is sent by the server to the client. The client parses the JSON text to the component variables for updating the webpage. JSON text consists of variable name and value pairs, each in double quotes and separated by a colon, with variable name andvalue pairs separated by a comma and the JSON text contained in curly brackets, `{}`. An example JSON text with three variable name and value pairs is `{"device": "LED", "state": "off", "pin": "15"}`.

In Listing 8-4, the counter and LED state are updated simultaneously, so two XML HTTP requests, two URLs and two functions are combined. The combined URL, "*/countLEDurl*", and combined function, *countLEDfunct*, are defined by the instruction: `server.on("/countLEDurl", countLEDfunct)` in the *setup* function of the main sketch. The *countLEDfunct* function in the main sketch consists of the following instructions:

```
void countLEDfunct()
{
  count++;                                          // increment count
  digitalWrite(LEDpin, !digitalRead(LEDpin));       // turn on or off the LED
  if(LED == "on") LED = "off";                      // update LED state
  else LED = "on";
  JsonConvert(count, LED);                          // convert to JSON text
  server.send (200, "text/json", json);             // send JSON text to client
}
```

The string *json* is defined in the main sketch, with the instruction `String json`. Note the `server.send()` instruction indicates that JSON text is being sent. The *JsonConvert* function, to combine the integer *count* and string *LED* into JSON text, consists of the instructions:

```
String JsonConvert(int val1, String val2)
{
  json  = "{\"var1\": \"" + String(val1) + "\",";
```

```
  json += " \"var2\": \"" + val2          + "\"}";
  return json;
}
```

which produces the JSON text of `{"var1": "123", "var2": "off"}` when the counter, *val1*, is equal to *123* and the LED state, *val2*, is *off*. The character pair, `\"`, which are underlined to emphasise that the characters are paired, is interpreted as a double quote character and not as an end of a string indicator. The `\` character is termed the *backslash escape* character.

The HTML code to display the counter and LED state is changed to:

```
<p>Counter: <span id = 'var1'>0</span></p>
<p>LED is <span id = 'var2'> </span><p>
```

which references the JSON names of *var1* and *var2*. The JavaScript code to process text sent by the server:

```
document.getElementById('Xid').innerHTML = this.responseText
```

is changed to

```
var obj = JSON.parse(this.responseText);
document.getElementById('var1').innerHTML = obj.var1
document.getElementById('var2').innerHTML = obj.var2
```

which parses the JSON text into the two name and value pairs for the counter, *var1*, and LED state, *var2*.

Listing 8-5 contains the updated AJAX code for the webpage and combines the two XML HTTP requests and the two URLs for the counter and LED state in Listing 8-4 and parses the JSON text, with the updates highlighted in bold.

Listing 8-5. *AJAX code with JSON parsing*

```
char page[] PROGMEM = R"(
<!DOCTYPE html><html>
<head><title>ESP8266</title></head>
<body>
<h2>BMP280</h2>
<p>Temperature: <span id = tempId>0</span>&degC</p>
<p>Counter: <span id = 'var1'>0</span></p>
<p>LED is <span id = 'var2'> </span><p>

<script>
setInterval(reload, 5000);                          // time in milliseconds
function reload()                                   // update the temperature every 5s
{
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function()
  {
    if(this.readyState == 4 && this.status == 200)
    {document.getElementById('tempId').innerHTML = this.responseText;}
  };
```

```
  xhr.open('GET', '/tempUrl', true);
  xhr.send();
}

setInterval(countLEDreload, 1000);
function countLEDreload()                          // update the counter and
{                                                  // LED state every second
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function()
  {
    if(this.readyState == 4 && this.status == 200)
    {                                                          // parse JSON text
      var obj = JSON.parse(this.responseText);
      document.getElementById('var1').innerHTML = obj.var1;
      document.getElementById('var2').innerHTML = obj.var2;
    }
  };
  xhr.open('GET', '/countLEDurl', true);
  xhr.send();
}
</script>
</body></html>
)";
```

Accessing WWW data

Displaying, on a webpage, data supplied by an ESP8266 or ESP32 microcontroller does not require access to the internet, as a WLAN to connect the client with the server can be established by the microcontroller, acting as the server, as discussed in Chapter 7 (Wireless local area network). To demonstrate displaying data accessed from the World Wide Web, date and time information is accessed from the websites www.calendardate.com/todays.htm and 24timezones.com/Edinburgh/time, with temperature and humidity for Edinburgh, Scotland accessed from the website www.metoffice.gov.uk.

Information is obtained with an API (Application Programming Interface) key, to retrieve data using HTTP requests, which is issued by *ThingSpeak* (www.thingspeak.com). Under the *ThingSpeak Apps* menu, the *ThingHTTP* option generates an API key to access a specific item on a given webpage. For example, an API key to source the current time from the 24timezones.com/Edinburgh/time website is generated by right-clicking on the displayed time and selecting *Inspect Element (Q)*. On the displayed *Web Console*, click the three dots and select *Dock to Right*. The HTML code corresponding to the selected time is highlighted in blue and moving the cursor over the HTML code generates a box surrounding the selected item on the webpage. Right-click the highlighted HTML code and select *Copy* and *XPath*. On the webpage thingspeak.com/apps/thinghttp, select *New ThingHTTP*, enter a name for the API, the URL of the webpage containing the data, for example 24timezones.com/Edinburgh/time, and

in the *Parse String* box, paste the copied *XPath* and click *Save ThingHTTP*. An API key is generated by *ThingSpeak* to access the required information, which is tested by loading a webpage with URL api.thingspeak.com/apps/thinghttp/send_request?api_key=*API key*.
If the *ThingSpeak* API key returns the message "*Error parsing document, try a different parse string*", then an alternative data source is required on the webpage or a linked webpage.

Information obtained with a *ThingSpeak* API key requires parsing. For example, date and time information is generated as * Today's Date is Monday June 15, 2020 * and *6:05:34 PM, Monday 15, June 2020*, respectively, while temperature and humidity are provided in HTML format as * 15° * and *90%*, respectively. The *date* substring is generated from the text following the text *is*. The *time* substring is text prior to the comma. Both the *temperature* and *humidity* substrings are bracketed by the = and > characters. The *toInt* and *toFloat* functions extract an integer and a real number, respectively, from a string, provided the first character of the string is a digit.

The sketch in Listing 8-6 accesses the current date, time, temperature and humidity with *ThingSpeak* API keys and parses the information into a JSON string for inclusion in the server response to the client HTTP request. Note that the webpage updates at 30 second intervals, so the initial values are all zero.

Listing 8-6 *Parsing data accessed with ThingSpeak API keys*

```
#include <ESP8266WebServer.h>                  // include Webserver library
ESP8266WebServer server;                       // associate server with library
WiFiClient client;                             // associate client with WiFi library
#include "buildpage.h"                         // webpage AJAX code
char ssid[] = "xxxx";                          // change xxxx to your Wi-Fi SSID
char password[] = "xxxx";                      // change xxxx to your Wi-Fi password
char APItime[] = "xxxx";
char APIdate[] = "xxxx";                       // change xxxx to ThingSpeak API key
char APItemp[] = "xxxx";
char APIhumid[] = "xxxx";
char url[] = "/apps/thinghttp/send_request?api_key=";
char host[] = "api.thingspeak.com";
int indexS, indexF, chk, humid;
float temp;
String data, ndata, text, json, mdy, tim;

void setup()
{
  Serial.begin(115200);                        // define Serial Monitor baud rate
  WiFi.begin(ssid, password);                  // connect and initialise Wi-Fi
  while (WiFi.status() != WL_CONNECTED) delay(500);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());              // display server IP address
```

```
  server.begin();                                   // initialise server
  server.on("/", base);                             // load default webpage
  server.on("/API", APIfunct);
}

void APIfunct()
{
  getData(APIdate, "date");                         // call function to access date,
  getData(APItime, "time");                         //      time,
  getData(APItemp, "temp");                         //      temperature,
  getData(APIhumid, "humid");                       //      and humidity
  JsonConvert(mdy, tim, temp, humid);               // convert to JSON text
  server.send (200, "text/json", json);
}

String JsonConvert(String val1, String val2, float val3, int val4)
{
  json  = "{\"var1\": \"" + val1         + "\",";      // start with {
  json += " \"var2\": \"" + val2         + "\",";      // end with comma
  json += " \"var3\": \"" + String(val3) + "\",";
  json += " \"var4\": \"" + String(val4) + "\"}";      // end with }
  return json;
}

void getData(String APIkey, String text)          // function to access data
{
  for (int i=0; i<5; i++)                           // with up to five attempts
  {
    getVal(APIkey, text);                           // call function to get information
    if(chk > 0) i = 5;                              // data accessed successfully
  }
}

void getVal(String APIkey, String text)           // function to access information
{
  chk = 0;
  Serial.print("sourcing ");Serial.println(text);
  client.connect(host, 80);
  client.println(String("GET ") + url + APIkey);
  client.println(String("Host: ") + host);
  client.println("User-Agent: ESP8266/0.1");
  client.println("Connection: close");
  client.println();
  client.flush();
  delay(100);
  while(client.connected())                         // while connected to ThingSpeak
  {
    if(client.available())                          // if data is available
    {
      data = client.readStringUntil('\n');          // read data till end of line
      Serial.println(data);
      if(text == "humid")                           // parse humidity data
      {
        indexS = data.lastIndexOf("=");             // position of last "=" in string
        indexF = data.indexOf("%");
        ndata = data.substring(indexS+2, indexF-2);
```

```
        humid = ndata.toInt();
        chk = data.length();
      }
      else if(text == "temp")                    // parse temperature data
      {
        indexS = data.indexOf("=");              // position of first "=" in string
        ndata = data.substring(indexS+2);
        temp = ndata.toFloat();
        chk = data.length();
      }
      else if(text == "date")                    // date: day month dd, yyyy
      {
        indexS = data.indexOf("is");
        mdy = data.substring(indexS+2);
        chk = data.length();
      }
      else if(text == "time")                    // time: hh:mm:ss AM or PM,
      {
        indexF = data.indexOf(",");
        tim = data.substring(0, indexF);
        chk = data.length();
      }
      client.stop();                         // close connection after data collected
      delay(100);
    }
  }
}

void base()                                  // function to return HTML code
{
  server.send (200, "text/html", page);
}

void loop()
{
  server.handleClient();                     // handle HTTP requests
}
```

The parsed information is displayed on a webpage at 30 second intervals (see Listing 8-7), for example purposes only as weather doesn't generally change that fast. Different time intervals are selected for the date, time and weather parameters by defining separate *reload* functions with appropriate time intervals.

Listing 8-7 *AJAX code with JSON parsing*

```
char page[] PROGMEM = R"(
<!DOCTYPE html><html>
<head><title>ESP8266</title></head>
<body>
<h2>BMP280</h2>
<p>Date: <span id = 'var1'>00 000 0000</span></p>
<p>Time: <span id = 'var2'>00:00:00</span></p>
<p>Temp is <span id = 'var3'>0</span>&degC<p>
<p>Humidity is <span id = 'var4'>0</span>%<p>

<script>
```

```
setInterval(APIreload, 30000);                    // time in milliseconds
function APIreload()
{
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function()
  {
    if(this.readyState == 4 && this.status == 200)
    {
      var obj = JSON.parse(this.responseText);
      document.getElementById('var1').innerHTML = obj.var1;
      document.getElementById('var2').innerHTML = obj.var2;
      document.getElementById('var3').innerHTML = obj.var3;
      document.getElementById('var4').innerHTML = obj.var4;
    }
  };
  xhr.open('GET', 'API', true);
  xhr.send();
}
</script>
</body></html>
)";
```

MQTT broker

Communication between devices on different Wi-Fi networks requires a different solution than communication between devices within a Wi-Fi network. The MQTT (Message Queuing Telemetry Transport) protocol enables communication between devices and an MQTT broker to pass information between one device and the MQTT broker and between the MQTT broker and a second device, with the two devices on different Wi-Fi networks. The MQTT broker enables data transfer between devices without breaching firewall safeguards. When a device on one Wi-Fi network requests information from a second device on another network, the information is allowed through the network firewall, as the request came from the Wi-Fi network. Provision of information to the MQTT broker is termed *publish* and *subscribe* is the term to access information from the MQTT broker. There are several MQTT brokers and the Blynk MQTT broker is used in the Chapter.

Blynk (see blynk.io/developers) provides a dashboard to display information from devices connected to an ESP8266 or ESP32 microcontroller (see Figure 8–4). The Blynk dashboard is visible locally or remotely with the *Blynk IoT* app. Information from devices is displayed numerically or graphically, with binary variables displayed as *ON/OFF*. A device is turned on or off from the Blynk dashboard, which provides both local and remote access to a device.

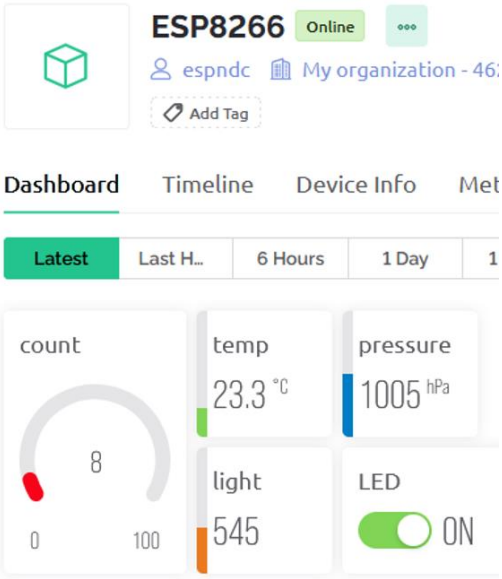

Figure 8-4 *Cayenne dashboard and Cayenne app*

Details, documentation and examples sketches of the Blynk MQTT broker are available at blynk.io/developers, docs.blynk.io/en and examples.blynk.cc, respectively. The *Blynk* library is available in the Arduino IDE or from github.com/blynkkk/blynk-library. Communication between an ESP8266 or an ESP32 microcontroller and the Blynk MQTT broker is through virtual pins, which are numbered *V0*, *V1*, *V2* etc. Data is received from the Blynk MQTT broker with the `BLYNK_WRITE` function. For example, when a switch, connected to virtual pin 0 on the Blynk dashboard, is turned on or off, an LED connected to the ESP8266 microcontroller is automatically turned on or off with the instructions:

```
BLYNK_WRITE(V0)                  // function called when virtual pin 0 state changed
{
  LEDstate = param.asInt();                  // set pin state to a variable
  digitalWrite(LEDpin, LEDstate);            // turn on or off the LED immediately
}
```

Integer, real or string variables are received from the Blynk MQTT broker with the `param.asInt()`, `param.asFloat()` or `pararm.asStr()` instructions.

Data is sent to the Blynk MQTT broker for display on the Blynk dashboard with the `Blynk.virtualWrite(virtual pin, variable)` instruction. For example, the `Blynk.virtualWrite(V3, light)` instruction transmits the *light* variable on virtual pin *V3*. The `Blynk.virtualWrite` instructions should be limited to no more than 60 per minute. The timing of sending information to the Blynk MQTT broker is managed by a Blynk *timer* function, rather than by including delays in the sketch, as delays will block an ESP8266 or ESP32 microcontroller from receiving data from the MQTT broker. For example, the

timerEvent function is triggered every 2s to update a variable on the Blynk dashboard with the instructions:

```
BlynkTimer timer;                              // define Blynk timer and
timer.setInterval(2000, timerEvent);           //   call timerEvent function every 2s
timer.run();                                   // include in loop function
void timerEvent()
{
  light = analogRead(LDRpin);                        // update light variable
  Blynk.virtualWrite(V3, light);                     //   and send to MQTT broker
}
```

The Blynk dashboard is accessed from blynk.io/developers and click *Start Free* to create a Blynk account or from blynk.cloud/dashboard/login. From the webpage left-side menu, select the *Quickstart* option by clicking on the "*lifebelt*" logo. In the hardware and connectivity type options, select *ESP8266* or *ESP32* and *WiFi*, select the *Arduino* IDE option and copy the displayed test sketch into the Arduino IDE.

In the test sketch, update the instructions:

```
char ssid[] = "";
char pass[] = "";
```

to the name and password of your WLAN router. Replace the contents of the *BLYNK_CONNECTED* function to `Serial.println("ESP8266 connected to Blynk")`. Comment out the `Blynk.begin(BLYNK_AUTH_TOKEN, ssid, pass)` instruction and uncomment the `Blynk.begin(BLYNK_AUTH_TOKEN, ssid, pass, "blynk.cloud", 80)` instruction. Compile and load the test sketch to an ESP8266 or ESP32 microcontroller.

Return to the Blynk console webpage and click the *Go To Device* button. Blynk creates a *Quickstart Device* mapped to the *Quickstart Template*, which includes a *Datastream* of three virtual pins and a *Web Dashboard* of three widgets: a button control, a switch label and an uptime value, which are mapped to the virtual pins. The virtual pins are mapped to variables in the sketch (see Listing 8-8) with the `Blynk.virtualWrite(virtual pin, variable)` instruction.

Blynk generates templates (see Figure 8-5) for allocation of *Datastreams* and *Web Dashboard* layouts to more than one device, such as an ESP8266 or ESP32 microcontroller, rather than each device being allocated a specific datastream and dashboard. Blynk templates are accessed by clicking the "*keypad*" logo. Blynk devices are accessed by clicking the "*magnifying glass*" logo, with details of the device template available in the *Device Info* option.

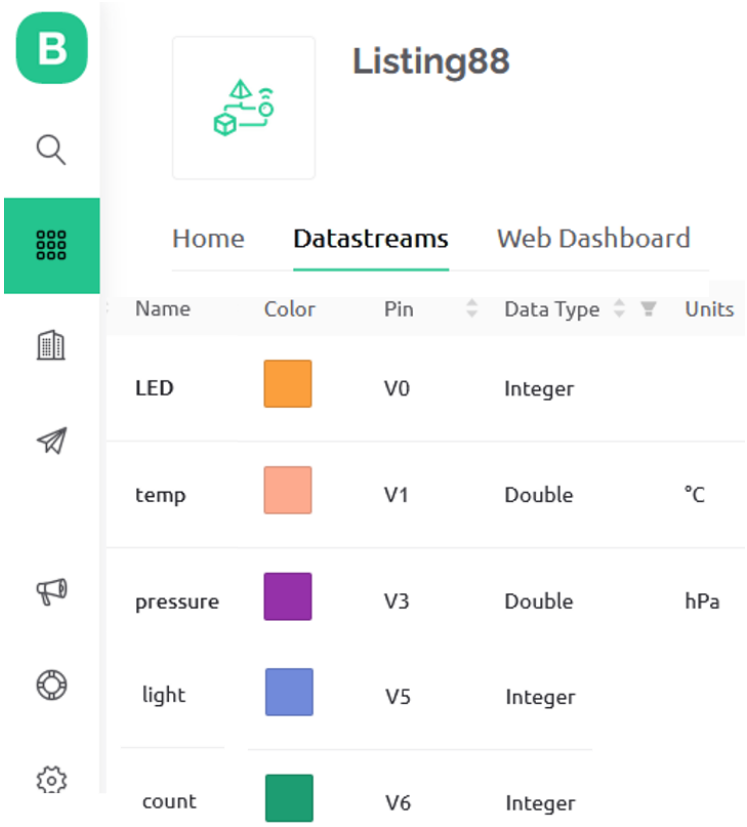

Figure 8-5 *Cayenne variables and devices*

A new template is created by copying the *Quickstart Template*. With the *Quickstart Template* open, click the "*three dots*", next to the *Edit* button at the top right of the screen, click the *Duplicate* option, click the *Configure template* option, rename the template to "*test sketch template"* and click the *Save* button. Click the *Add first Device* option and enter a device name, such as *ESP8266* and click the *Create* button. Copy the updated *BLYNK_TEMPLATE_ID, BLYNK_TEMPLATE_NAME*. and *BLYNK_AUTH_TOKEN* information. Paste the updated template information into the sketch (see Listing 8-8).

Listing 8-8 transmits to the Blynk dashboard or app (see Figure 8-4) temperature and pressure measurements from a BMP280 sensor, ambient light using a light dependent resistor and a counter. When the LED state is changed by the Blynk dashboard, the ESP8266 microcontroller receives the updated state, on Blynk virtual pin 0, with the `BLYNK_WRITE(V0)` and `param.asInt()` instructions. Values are sent to the Blynk MQTT broker with the `Blynk.virtualWrite` instruction with the virtual pin and variable as parameters. In the series of `Blynk.virtualWrite` instructions, virtual pin *V2* is not used to avoid confusion with GPIO 2 for the *flashPin* variable. The Blynk timer calls the *timerEvent* function at one second intervals, as defined with the `timer.setInterval(1000, timerEvent)` instruction. Instructions to periodically read the BMP280 sensor and send data to the Blynk MQTT

broker are included in the *timerEvent* function, rather than in the *loop* function. A `delay` instruction is not included in the sketch, as a delay would block an ESP8266 microcontroller from receiving input by the MQTT broker. The `BLYNK_CONNECTED` function is called when an ESP8266 or ESP32 microcontroller connects to the Blynk MQTT broker.

The *ESP8266WiFi* library is referenced by the *BlynkSimpleEsp8266* library and is not explicitly referenced in the sketch. For an ESP32 microcontroller, the `#include <BlynkSimpleEsp32.h>` instruction replaces the `#include <BlynkSimpleEsp8266.h>` instruction.

Listing 8-8 *Blynk, ESP8266 with LED, LDR and BMP820 sensor*

```
#define BLYNK_TEMPLATE_ID   "xxxx"          // change xxxx to Template details
#define BLYNK_TEMPLATE_NAME "Listing88"
#define BLYNK_AUTH_TOKEN    "xxxx"          // change xxxx to Authentication token
#define BLYNK_PRINT Serial                  // avoids Serial print noise
#include <BlynkSimpleEsp8266.h>             // Blynk MQTT library
#include <ssid_password.h>                  // file with Wi-Fi logon details
#include <Adafruit_Sensor.h>                // include Adafruit_Sensor library
#include <Adafruit_BMP280.h>                // include Adafruit_BMP280 library
Adafruit_BMP280 bmp;                        // associate bmp with BMP280 library
int LEDpin = D3;                            // LED pin
int LDRpin = A0;                            // light dependent resistor pin
int flashPin = D4;                          // flashing LED pin
int count = 0;
int interval = 10000;                       // 10s interval between MQTT messages
float temp, pressure;
int light;

BlynkTimer timer;                           // declare Blynk timer
void timerEvent()                           // function called by Blynk timer
{
  temp = bmp.readTemperature();             // BMP280 temperature
  pressure = bmp.readPressure()/100.0;      //  and pressure
  light = analogRead(LDRpin);               // ambient light intensity
  light = constrain(light, 0, 1023);        // constrain light reading
  count++;                                  // increment counter
  if(count>99) count = 0;
  digitalWrite(flashPin, LOW);              // turn on then off flashing LED
  delay(10);
  digitalWrite(flashPin, HIGH);             // send readings to Blynk on virtual pins
  Blynk.virtualWrite(V1, temp);             // temperature
  Blynk.virtualWrite(V3, pressure);         // pressure
  Blynk.virtualWrite(V5, light);            // luminosity
  Blynk.virtualWrite(V6, count);            // counter
}

BLYNK_WRITE(V0)                             // Blynk virtual pin 0
{
```

```
  digitalWrite(LEDpin, param.asInt());      // turn on or off LED
}

BLYNK_CONNECTED()                           // function called when ESP8266 connected
{
  Serial.println("ESP8266 connected to Blynk");
}

void setup()
{
  Serial.begin(115200);                                 // initiate Blynk MQTT broker
  Blynk.begin(BLYNK_AUTH_TOKEN, ssidEXT, password, "blynk.cloud", 80);
  timer.setInterval(interval, timerEvent);              // timer event called every 10s
  bmp.begin(0x76);                                      // initiate bmp with I2C address
  pinMode(LEDpin, OUTPUT);                              // define LED pins as output
  digitalWrite(LEDpin, LOW);
  pinMode(flashPin, OUTPUT);
}

void loop()
{
  Blynk.run();                                          // maintain connection to Blynk
  timer.run();                                          // Blynk timer
}
```

The Blynk dashboard is used to mimic an alarm system, which is triggered by a high light-intensity reading on a light dependent resistor, such as when a door is opened. If the light intensity increases above a threshold and the alarm on the Blynk dashboard is set to *ON*, then an email is sent to notify that the event has occurred (see Figure 8-6) and the Blynk dashboard LED is turned on (see Figure 8-7).

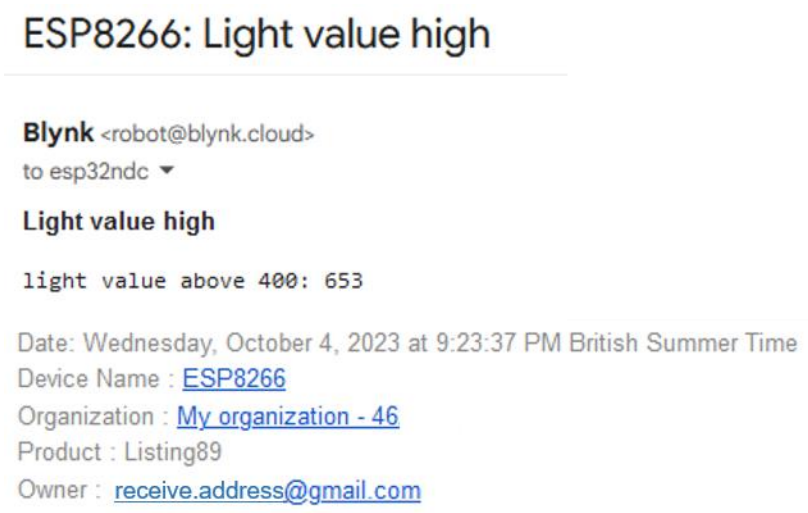

Figure 8-6 *Blynk email*

If the alarm setting on the Blynk dashboard is set to *ON*, then the light intensity reading is sent to the Blynk MQTT broker on virtual pin 1, but with a value of zero if the

alarm is set to *OFF*. Alarm and LED switch widgets on virtual pins 3 and 0 on the Blynk dashboard turn on or off the alarm and provide an LED indicator when the alarm is triggered. Attached to the ESP8266 or ESP32 module are a blue (*alarmPin*) and a red (*LEDpin*) LED to indicate the alarm state and if the alarm is triggered. The two LEDs are turned on or off with the *BLYNK_WRITE(3)* and *BLYNK_WRITE(0)* functions.

Figure 8-7 shows the Blynk dashboard with the alarm set to *ON* and an initial light intensity reading of 144 (see Figure 8-7a), which is below the threshold to trigger the alarm and turn on the indicator LED widget. When the light intensity rises above the threshold of 400, the alarm is triggered and the indicator LED widget is turned on (see Figure 8-7b).

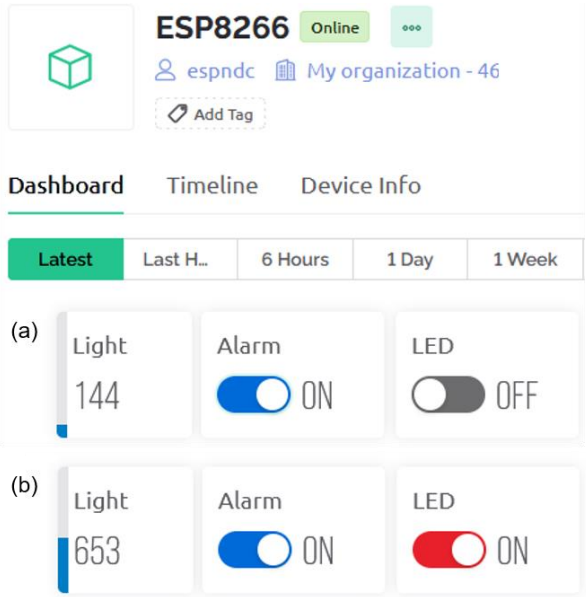

***Figure 8-7** Alarm, LED and light intensity*

Email notification is triggered by a Blynk event. The instruction to initiate a Blynk event called *event* is `Blynk.logEvent("event", "text")`, where *text* is the text to be included in the email. For example, the text "*light value above 400: 653*" is included in the email text with the `Blynk.logEvent("alert", "light value above 400: " + String(light))` instruction for a *light* value of 653 and an event called *alert*.

A Blynk event is defined in a template *Events* option (see Figure 8-8). Click the "*keypad*" logo to select the required template, click the *Events* option and the *Add New Event* button. In the *General* tab, enter the text for the email header in the *EVENT NAME* box and enter the event name in the *EVENT CODE* box. For example, if the *alert* event is defined in the `Blynk.logEvent("alert", "text")` instruction, then the *EVENT CODE* box is set to *alert*. Set the *TYPE* level to *Warning* and in the *Event will be sent to user only once per box*,

select *1 minute* from the drop-down list. Check the *Send event to Timeline* option. In the *Notifications* tab, check *Enable notifications* and in the *E_MAIL TO* box, select *Device Owner* from the drop-down list. Click the *Create* and the *Save And Apply* buttons. Return to the Blynk dashboard, by clicking on the "*magnifying glass*" logo and select the required device.

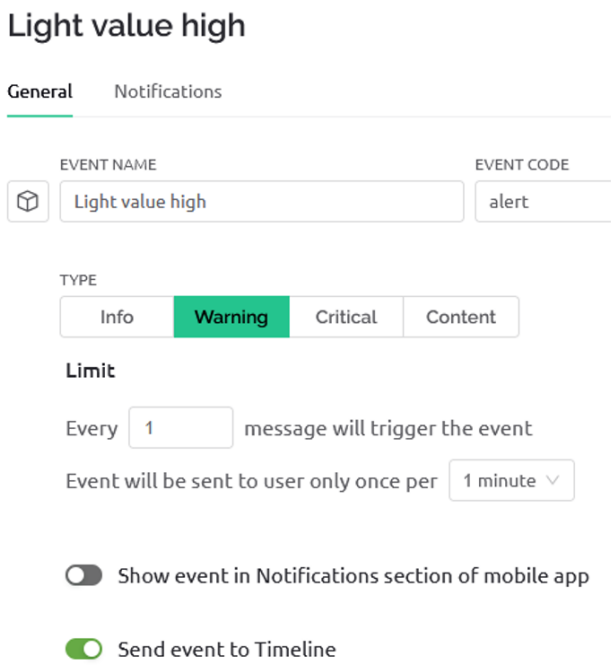

Figure 8-8 *Blynk event*

In Listing 8-9, when the light intensity increases above a threshold of 400, with the alarm widget on the Blynk dashboard set to *ON*, the Blynk event is triggered to send the email notification. The alarm and indicator LEDs attached to the ESP8266 or ESP32 module are turned on or off depending on the values of the alarm and LED widgets on the Blynk dashboard. The ESP8266 or ESP32 module built-in LED is flashed every two seconds to indicate that the microcontroller is powered on.

Listings 8-8 and 8-9 include the *BlynkSimpleEsp8266* library for an ESP8266 microcontroller. The *BlynkSimpleEsp32* library is required for an ESP32 microcontroller, which is included in the *Blynk* library. No changes, other than pin numbers for attached sensors (see Table 8-2), are required for sketches with an ESP32 microcontroller.

Listing 8-9 *Alarm, LED and light intensity*

```
#define BLYNK_TEMPLATE_ID   "xxxx"          // change xxxx to Template details
#define BLYNK_TEMPLATE_NAME "Listing89"
#define BLYNK_AUTH_TOKEN    "xxxx"          // change xxxx to Authentication token
```

```
#define BLYNK_PRINT Serial                          // avoids Serial print noise
#include <BlynkSimpleEsp8266.h>                     // Blynk MQTT library
#include <ssid_password.h>                          // file with Wi-Fi logon details
int LEDpin = D3;
int alarmPin = D5;                                  // define LED, alarm and LDR pins
int LDRpin = A0;
int flashPin = D4;                                  // flashing LED
int reading, alarm, alert;
unsigned long lag = 0;
int flag;

BlynkTimer timer;                                   // declare Blynk timer
void timerEvent()                                   // function called by Blynk timer
{
  reading = analogRead(LDRpin);                     // send light reading to Blynk
  if (alarm == 1) Blynk.virtualWrite(V1, reading);     // when alarm is ON
  else Blynk.virtualWrite(V1, 0);
  digitalWrite(flashPin, LOW);
  delay(10);                                        // flash LED to indicate power on
  digitalWrite(flashPin, HIGH);

  if(alarm == 1 && reading > 400 && flag == 0)      // alarm is set to ON and
  {                                                 // light above threshold
    Blynk.virtualWrite(V0, 1);                      // update Blynk dashboard
    Blynk.logEvent("alert", "light value above 400: " + String(reading));
    digitalWrite(LEDpin, alarm);                    // turn on indicator LED
    flag = 1;                                       // flag to prevent repeat triggering
  }
  else if (reading < 400 && flag == 1) flag = 0;
}

BLYNK_WRITE(0)                                      // Blynk virtual pin 0
{
  alert = param.asInt();                            // get alarm triggered status
  digitalWrite(LEDpin, alert);                      // update alarm triggered LED
}
BLYNK_WRITE(3)                                      // Blynk virtual pin 3
{
  alarm = param.asInt();                            // get alarm set state
  digitalWrite(alarmPin, alarm);                    // update alarm set indicator LED
}

BLYNK_CONNECTED()                                // function called when ESP8266 connected
{
  Serial.println("ESP8266 connected to Blynk");
}

void setup()
{
  Serial.begin(115200);                             // initiate Blynk MQTT broker
  Blynk.begin(BLYNK_AUTH_TOKEN, ssidEXT, password, "blynk.cloud", 80);
  timer.setInterval(2000, timerEvent);              // timer event called every 2s
  pinMode(LEDpin, OUTPUT);                          // define LED and alarm pins
  pinMode(alarmPin, OUTPUT);                        // as output
  pinMode(flashPin, OUTPUT);
```

```
  alarm = 0;                                    // set alarm to OFF
}

void loop()
{
  Blynk.run();                                  // maintain connection to Blynk
  timer.run();                                  // Blynk timer
}
```

Figure 8-9 shows the schematic with connections in Table 8-2.

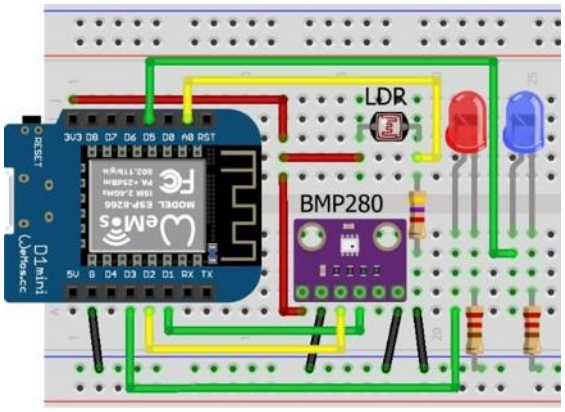

Figure 8-9 *Alarm, LED and light intensity with LOLIN (WeMos) D1 mini development board*

Table 8-2 *Alarm, LED and light intensity*

Component	ESP8266 connections		ESP32 connections	
BMP280 VCC	3V3		3V3	
BMP280 GND	GND		GND	
BMP280 SDA	D2		GPIO 21	
BMP280 SCK	D1		GPIO 22	
BMP280 SD0	GND		GND	
LDR left	3V3		3V3	
LDR right	4.7kΩ resistor	GND	4.7kΩ resistor	GND
LDR right	A0		GPIO 36	
LED long legs	D3, D5		GPIO 23	
LED short legs	220Ω resistor	GND	220Ω resistor	GND

Parsing text

Parsing text is often required, such as data from the Serial buffer or for data uploaded following an HTTP request. The instruction `Serial.read()` reads the next available character in the Serial buffer, while the instruction `Serial.readStringUntil('\n')` reads all the data in the Serial buffer until end of line character. The content of the Serial buffer is

stored as a string, `str`, with the instruction `str = Serial.readString()` and parsed into an integer or a real number with the instruction `Serial.parseInt()` or `Serial.parseFloat()`. For example, if the Serial buffer contains the text *abc25* or *abc3.14*, then parsing returns the integer *25* or the real number *3.14*, respectively. Parsing ignores the initial non-digit characters, other than a decimal point or a minus sign, and stops when a non-digit character is read after the last digit character. Parsing instructions are repeated to extract more than one number from the Serial buffer. For example, if the Serial buffer contains *abc-25de3.14*, then the integer *-25* and the real number *3.14* are returned with the instructions:

```
if(Serial.available()>0)
{
  x = Serial.parseInt();
  y = Serial.parseFloat();
}
```

The *parseInt* and *parseFloat* functions are blocking functions, which prevent the microcontroller from processing other instructions until parsing is completed.

Strings are parsed to integers or real numbers with the instructions `toInt()` or `toFloat()`, respectively, provided the first character of the string is a digit, a decimal point or a minus sign. For example, if a string *str* equals *-25abc* or *3.14abc*, then the instructions `str.toInt()` or `str.toFloat()` return the integer *-25* or the real number *3.14*, respectively. Parsing stops when a non-digit character is read after a digit character or a decimal point. Note that a real number is stored with a total number of six or seven digits, so converting the string *2.7182818284* to a real number results in the value of *2.718282*.

A string that does not start with a digit character is parsed by extracting a substring that does start with a digit character, a decimal point or a minus sign. For a string `str`, the instruction `str.substring(x, y)` creates a substring between positions *x* (inclusive) and *y* (exclusive). For example, comma positions of the string `str = "abc,def,gh"` are 4 and 8, so the instruction `str.substring(4+1, 8)` creates the substring *def* with characters 5, 6 and 7 of the string. If the end parameter is omitted, as in the instruction `str.substring(x)`, then the substring extends to the end of the string. For example, the instruction `str.substring(5)` creates the substring *def,gh*.

The instructions `str.indexOf("x")` and `str.indexOf("x", y)` locate the position of the substring *x* within the string, by searching from the first to the last character or from position *y* to the last character. Similarly, the instructions `str.lastIndexOf("x")` and `str.lastIndexOf("x", y)` locate the position of the substring *x* within the string by

searching from the last to the first character or from position *y* to the first character. The *indexOf* or *lastIndexOf* functions are combined with the *substring* function to create a specific substring of a string, when there are several substring delimiters in a string.

For example, the decimal humidity value contained in the string, `str = <span class="humidity" data-value="68.1">68%</span>` is bracketed by an = character and a > character, but there are two of each character in the string. The *toFloat* function cannot extract the humidity value, as the first character of the string is not a digit, so a substring is created to extract the decimal value, but not the integer value. The instructions `indexS = str.lastIndexOf("=")` and `indexF = str.indexOf("%")` locate positions of the last = character and the *%* character, with the substring defined by the instruction `str.substring(indexS+2, indexF-2)` containing the characters *68.1">*, from which the decimal value is extracted with the *toFloat* function.

The instruction `str.length()` determines the length of the string, which does not include the null terminating character, *\n*, and the string length is used to ensure that a substring position does not exceed the length of the string. To check if a string, *str*, starts or ends with a substring, *abc*, the instructions `str.startsWith("abc")` and `str.endsWith("abc")` return a value one if true or value zero if false.

Console log

When debugging a sketch or checking the progress of a sketch, information on a variable is displayed on the Serial Monitor with the `Serial.print("text")` instruction. The equivalent to displaying information on the Serial Monitor for a web browser is the console log, when using JavaScript code, to display the value of a variable or text, such as "*button pressed*". The console log is accessed in the browser by clicking the *F12* keyboard button, selecting *Console* and *Logs* (see Figure 8-10). To define the character set for the console log, the instruction `<meta charset='UTF-8'>` must be included in the `<head>` section of the webpage HTML code. The instruction `console.log(variable)` displays information on the console log, with *variable* equalling a variable, some text or the server response to the client. For example, in Listing 8-5, the instruction `console.log(this.responseText)` is included after `document.getElementById('tempId').innerHTML = this.responseText` and the instructions:

```
console.log("LED updated");
console.log(this.responseText);
console.log(obj.var1);
var value = document.getElementById('var2').innerHTML;
console.log(value);
```

are included after the instruction:

`document.getElementById('var2').innerHTML = obj.var2`. To generate repeated temperature readings, the *reload* function interval was reduced to one second and the *countLEDreload* function interval for the counter and LED was increased to five seconds. The console log produced the output in Figure 8-10, showing the five temperature readings, the text "*LED updated*", the JSON text received by the client, the counter and the LED state defined as a new variable, *value*.

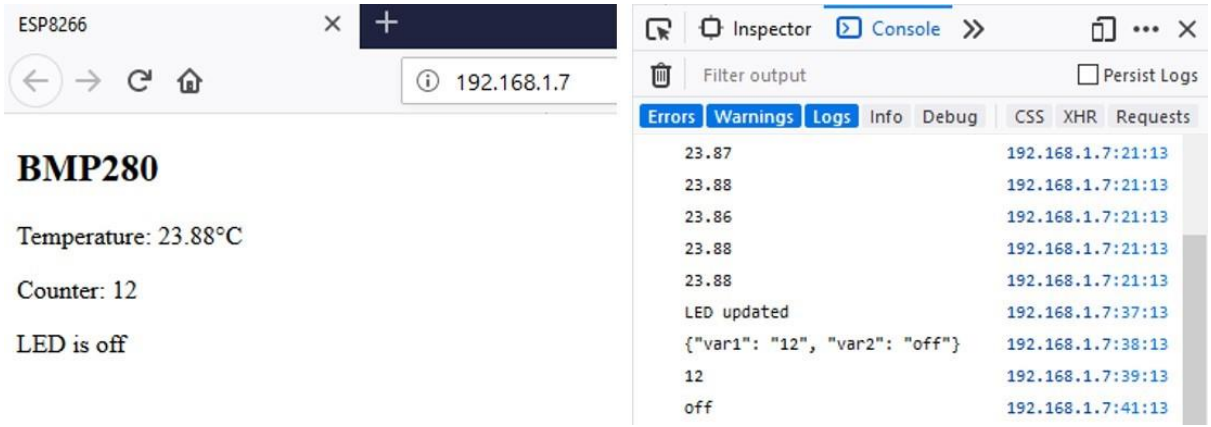

Figure 8-10. *Console log*

Wi-Fi connection

The ESP8266 and ESP32 libraries are automatically installed when the *esp8266 by ESP8266 Community* and *esp32* by *Expressif Systems* are installed in the Arduino IDE Board Manager. Library versions 2.7.4 and 1.0.4 were used in the Chapter. If the error message "*Downloading http://downloads.arduino.cc/packages/packages_index.json*" is displayed in the Arduino IDE Boards Manager, then delete all *.tmp* files in *user>AppData>Local>Arduino15* folder and restart the Arduino IDE.

The Wi-Fi connection between the ESP8266 or ESP32 microcontroller and the WLAN router is tested using the *ping* command given the ESP8266 or ESP32 IP address. Either right-click the *Windows logo* at the bottom left side of the screen and select *Run* or press the *Windows* key and *R* key, simultaneously, to open the Command window. Type *cmd* and press *OK*. In the *C:\\WINDOWS\system32\cmd.exe* window, the *Command prompt* window, type *ping* followed by the ESP8266 or ESP32 IP address, as shown in Figure 8-11. The *ping* command sends small data packets to the ESP8266 or ESP32 IP address, that are transmitted back to the sender. In the example, four data packets were sent and received,

which indicated that the Data Link, the Wi-Fi connection and the Internet Protocol were functioning correctly.

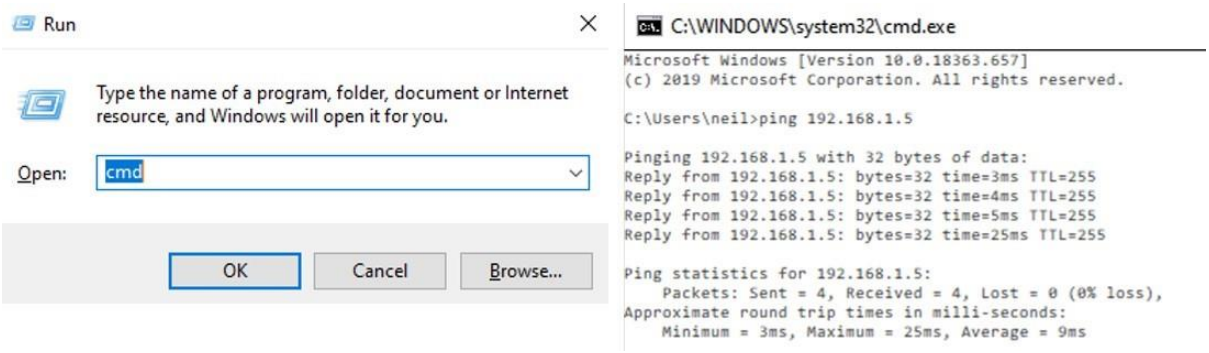

Figure 8-11. *Wi-Fi connection test*

Access information file

Access to a Wi-Fi network, ThingSpeak data (see Listing 8-6) or an MQTT broker (see Listing 8-8) requires a password, SSID, API key or MQTT broker keys. Instead of storing the access information in the sketch with instructions, such as `char mqttpass[] = "abcdef"`, the information is stored in a library, that is referenced by the sketch. A text file with the extension *.h* is created to hold the access information, with the file placed in the Arduino IDE libraries folder. To determine the location of the Arduino IDE libraries folder, select *File>Preferences* in the Arduino IDE and the libraries folder is shown in the *Sketchbook location*, for example : *C:\Users\user\Documents\Arduino*. The access information file is referenced by the sketch with the instruction `#include <access_info.h>`. The example access information file, in Listing 8-10, includes the access keys for Wi-Fi, ThingSpeak and Cayenne.

Listing 8-10 *Access information*

```
char ssid[] = "PhoneNetwork12";                  // Wi-Fi access
char password[] = "difficu1t";
char APItime[] = "efth1234";
char APIdate[] = "mhtd5678";                     // ThingSpeak API keys
char APItemp[] = "plmf4567";
char APIhumid[] = "thkl6789";
#define BLYNK_TEMPLATE_ID   " XYZ-567"           // Blynk access
#define BLYNK_TEMPLATE_NAME "Listing 88 template"
#define BLYNK_AUTH_TOKEN    " GHJ-876"
```

Summary

A webpage displayed the temperature reading from a BMP280 module, a counter and the state of an LED, with the ESP8266 or ESP32 microcontroller functioning as the server. The webpage HTML code was built line-by-line as a string and the whole webpage was reloaded to display the updated information. AJAX code, consisting of XML HTTP requests and JavaScript instructions, incorporated as a *string literal*, enabled updating of specific variables on the webpage, rather than reloading the whole webpage. Data on several variables was combined as JSON text in the server response to the client HTTP request and options for parsing text were described. Information from the World Wide Web was accessed with API keys and displayed on a webpage using both AJAX and JSON code. Access to an MQTT broker allowed uploading of sensor data to a webpage, with a sensor value above a threshold triggering an email notification of the event. The console log verified data sent by the server and received by the client.

Components List

- ESP8266 microcontroller: LOLIN (WeMos) D1 mini or NodeMCU board
- ESP32 microcontroller: ESP32 DEVKIT DOIT or NodeMCU board
- BMP280 module
- LED: 2×
- Resistor: 2× 220Ω