CHAPTER 9 MQTT

Message Queueing Telemetry Transport (MQTT) is a publish-subscribe network protocol, that includes communication between ESP32 microcontrollers on different Wi-Fi networks. MQTT was developed in 1999 by Andy Stanford-Clark and Arlen Nipper for connecting oil pipeline telemetry systems. The MQTT broker enables data transfer between devices on different Wi-Fi networks 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 by an MQTT client to the MQTT broker is termed *publish* and *subscrib*e is the term to access information through the MQTT broker. There are several MQTT brokers with the Blynk (blynk.io/developers), Arduino IoT Cloud (cloud.arduino.cc) and Adafruit IO (io.adafruit.com) MQTT brokers used in the Chapter. The ESP32 microcontroller is an MQTT client communicating with the MQTT broker. An MQTT dashboard is accessible from anywhere in the world to provide remote access to information from sensors or for remote control of devices connected to an ESP32 module. MQTT is illustrated with the ESP32 microcontroller transmitting air quality measurements or energy usage readings, provided by a smart meter, to the MQTT broker for display on the MQTT broker dashboard.

CO2 and TVOC

Carbon dioxide and total volatile organic compounds are indicators of air quality. Equivalent carbon dioxide (eCO2) and equivalent total volatile organic compounds (eTVOC) are measured by the CCS811 module. The measurable eCO2 and eTVOC ranges are 400 to 33kppm and 0 to 29kppb, respectively. Suggested normal values for eCO2 and eTVOC levels are <500ppm and <50ppb, respectively, with >1500ppm and >1000ppb for poor air ventilation. The Adafruit CCS811 module (not shown) measures eCO2 and eTVOC from 400 to 8192ppm and from 0 to 1187ppb, respectively. The *Adafruit CCS811* library is for the Adafruit CCS811 module. The CCS811 HDC1080 module measures eCO2, eTVOC, temperature and relative humidity, as the module includes an HDC1080 sensor in addition to the CCS811 sensor. The *CCS811* library by Maarten Pennings is recommended and is downloaded from github.com/maarten-pennings/CCS811.

The CSS811 module requires a voltage of 1.8 to 3.6V and communicates with I2C (Inter-Integrated Circuit). The default I2C address of *0x5A* is changed to *0x5B* by setting the *ADD* pin to *HIGH*. The CSS811 module is reset by setting the *RST* pin to *LOW*. Connections between a CCS811 module and an ESP32 DEVKIT DOIT module are given in Table 9-1.

The CSS811 module operates in constant power mode with sampling every second or in low-power mode with sampling at 10s or 60s intervals. Constant power mode is initiated with the `start(CCS811_MODE_1SEC)` instruction and the *WAK* pin is connected to GND. The instruction `start(CCS811_MODE_10SEC)` or `start(CCS811_MODE_60SEC)` initiates low-power mode and the *WAK* pin is connected to a GPIO pin, such as GPIO 19 (see Figure 9-1). The `CSS811 css811(19)` instruction identifies GPIO 19 as attached to the module *WAK* pin. If the *WAK* pin is connected to GND, then the instruction is `CSS811 css811(-1)`. The interval between start-up and the first reading is 4, 33 or 200s for a sampling interval of 1, 10 or 60s, respectively.

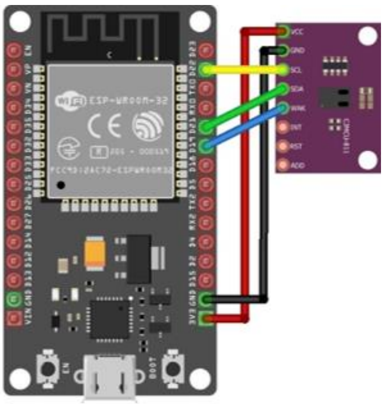

Figure 9-1 *TVOC and CO_2 measurement*

Table 9-1 *TVOC and CO_2 measurement*

CCS811 module	ESP32
VCC	3V3
GND	GND
SCL	GPIO 22
SDA	GPIO 21
WAK	GPIO 19

Equivalent carbon dioxide (eCO2) and total volatile organic compounds (eTVOC) are measured at 10s intervals in Listing 9-1. Given a valid data reading, *eCO2* and *eTVOC* values and the interval between readings are displayed on the Serial Monitor and the state of an activity indicator LED, which is the built-in LED on GPIO 2, is alternated. Each second between readings, the no data error status triggers the display, on the Serial Monitor, of a dot as a state indicator. With the *CCS811* library, if an error is detected, an error message consisting of a letter series indicates the error source. Details of the *errstat* letter series are included in the *ccs811.cpp* file of the *CCS811* library. The *CC2811* library references the *Wire* library, so the `#include <Wire.h>` instruction is not required.

Listing 9-1 *TVOC and CO2 measurement*

```
#include <ccs811.h>                              // include CCS811 and
#include <Wire.h>                                // Wire libraries
CCS811 ccs811(19);                               // nWAKE connected to GPIO 19
uint16_t CO2, TVOC, errstat, rawdata;
unsigned long last, diff;
int LEDpin = 2, LED = 0;                         // LED as activity indicator

void setup()
{
  Serial.begin(115200);                          // Serial Monitor baud rate
  pinMode(LEDpin, OUTPUT);
  digitalWrite(LEDpin, LED);                     // turn off LED
  Wire.begin();                                  // initialise I2C
  ccs811.begin();                                // initialise CCS811
  ccs811.start(CCS811_MODE_10SEC);               // set reading interval at 10s
}

void loop()
{
  ccs811.read(&CO2, &TVOC, &errstat, &rawdata); // read CCS811 sensor data
  if(errstat == CCS811_ERRSTAT_OK)               // given valid readings
  {
    diff = millis() - last;
    last = millis();                             // interval since last reading
    LED = 1-LED;                                 // turn on or off indicator LED
    digitalWrite(LEDpin, LED);                   // display readings
    Serial.printf("\n int %d CO2 %dppm TVOC %dppb \n", diff, CO2, TVOC);
  }                                              // print dot between readings
  else if(errstat == CCS811_ERRSTAT_OK_NODATA) Serial.print(".");
  else if(errstat & CCS811_ERRSTAT_I2CFAIL) Serial.println("I2C error");
  else
  {                                              // display error message
    Serial.print("errstat = "); Serial.print(errstat,HEX);
    Serial.print(" = "); Serial.println(ccs811.errstat_str(errstat));
  }
  delay(1000);                                   //arbitrary delay of 1s to display
}                                                //    dot between readings
```

MQTT brokers

Equivalent carbon dioxide (eCO2) and total volatile organic compounds (eTVOC) readings are forwarded by the ESP32 microcontroller to the MQTT broker. The eCo2 and eTVOC readings, which are taken every 60s, and a countdown indicator, which is updated every 5s, are displayed on the MQTT broker dashboard (see Figure 9-2). An LED, connected to the ESP32 module, is controlled through the MQTT broker dashboard.

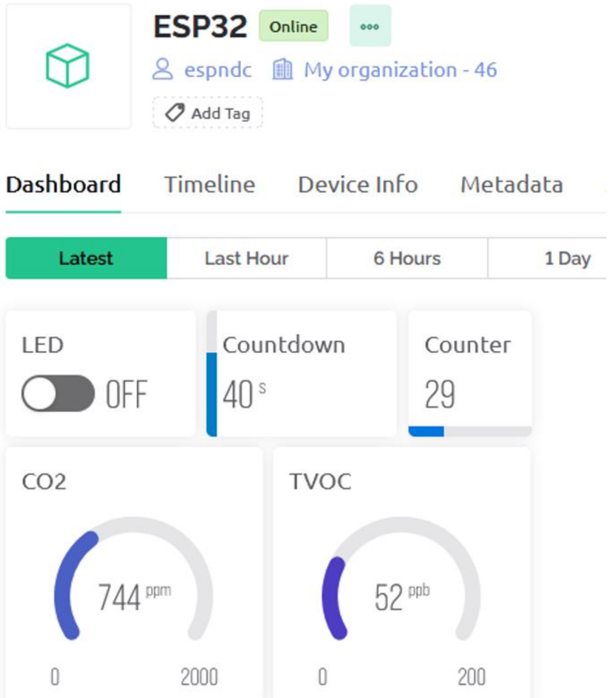

Figure 9-2 *TVOC and CO2 measurement and Blynk MQTT broker*

The Blynk, Arduino IoT Cloud and Adafruit IO MQTT brokers provide a dashboard to display information from sensors connected to an ESP32 microcontroller. Information from sensors is displayed numerically, or as a dial or graphically, with binary variables displayed as *ON/OFF*. A device, such as a relay, is turned on or off from the MQTT broker dashboard, which provides both local and remote access to a device. Listing 9-1 is extended to display information on the MQTT broker dashboard for the Blynk (see Listing 9-2) and Arduino IoT Cloud (see Listings 9-4 and 9-5) MQTT brokers.

The Adafruit IO MQTT broker is illustrated with a dashboard controlled event confirmed by a change in a dashboard gauge value, an updated dashboard indicator widget and by sending an email notification of the event (see Listing 9-6). The Adafruit IO has a map facility (see Listing 9-7).

Blynk MQTT Broker

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 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 ESP32 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 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 MQTT broker is managed by a Blynk *timer* function, rather than by including delays in the sketch, as delays will block an 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 *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("ESP32 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 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 9-2) with the `Blynk.virtualWrite(virtual pin, variable)` instruction.

Blynk generates templates (see Figure 9-3) for allocation of *Datastreams* and *Web Dashboard* layouts to more than one device, such as an 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.

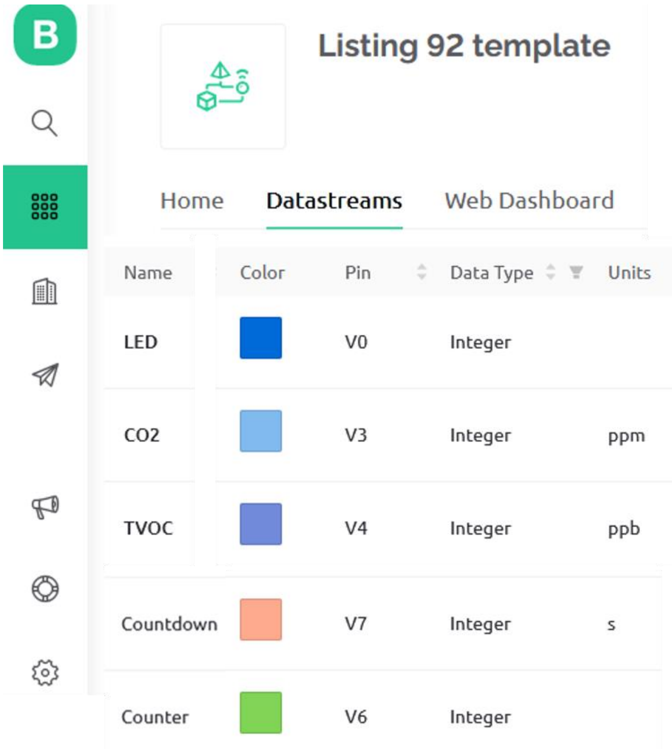

Figure 9-3 *Blynk template*

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 *ESP32* 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 test sketch.

To turn on and off the built-in LED on the ESP32 module, when the Blynk dashboard button is clicked, the following instructions are included in test sketch:

```
int LEDpin = 2;
pinMode(LEDpin, OUTPUT);                    // included in the void setup function
```

In the *BLYNK_WRITE(V0)* function, include the `digitalWrite(LEDpin, value)` instruction and then compile and load the test sketch to the ESP32 microcontroller. When the button on the Blynk dashboard is clicked, the built-in LED on the ESP32 module is turned on or off and the switch label value is updated. The *uptime* value is incremented every second.

A widget is added to a template by clicking the "*keypad*" logo, clicking on the required template, clicking the *Edit* button, at the top right of the screen, and selecting the *Web Dashboard* option. Drag a widget from the *Widget* box onto the dashboard and move the cursor over the widget to display the "*cog*" icon, which is the *Settings* option. Click the *Save* button after defining the settings and click the *Save and Apply* button at the top right of the screen. Click the "*magnifying glass*" logo and the *Device* to display the updated dashboard.

Details of the virtual pins are listed and edited on the template *Datastreams* option. Click the *Edit* button at the top right of the screen and click on the required *Virtual Pin Datastream* to define the virtual pin, the data type and the unit of measurement from the drop-down lists, and the minimum and maximum values. Click the *Save* button and then the *Save and Apply* button at the top right of the screen.

The *Blynk IoT* app is downloaded from the *Google Play Store* and after logging on to your account, the *ESP32* device is displayed. Dashboard widgets, such as buttons and charts, are added to the dashboard by clicking on the "*spanner*" logo, clicking on the "*plus*" logo and selecting the required widget. A widget is positioned on the dashboard and sized by pressing on the widget, with the *Design* option selected to format with the widget.

The sketch in Listing 9-2 is based on Listing 9-1 and only the additional instructions are commented. When the LED state is changed by the MQTT broker, the ESP32 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. 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 CCS811 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 ESP32 microcontroller from receiving input by the MQTT broker. The `BLYNK_CONNECTED` function is called when an ESP32 microcontroller connects to the Blynk MQTT broker. The *WiFi* and *WiFiClient* libraries are referenced by the *BlynkSimpleEsp32* and *WiFi* libraries, respectively, and are not explicitly referenced in the sketch.

Listing 9-2 *TVOC and CO2 measurement and Blynk MQTT broker*

```
#define BLYNK_TEMPLATE_ID   "xxxx"          // change xxxx to Template details
#define BLYNK_TEMPLATE_NAME "Listing 92 template"
#define BLYNK_AUTH_TOKEN    "xxxx"          // change xxxx to Authentication token
#define BLYNK_PRINT Serial                              // avoids Serial print noise
#include <Wire.h>
#include <ccs811.h>;
#include <BlynkSimpleEsp32.h>                           // Blynk MQTT library
```

```
#include <ssid_password.h>                          // file with logon details

CCS811 ccs811(19);
uint16_t CO2, TVOC, errstat, rawdata;
int LEDpin = 2;
int LEDMQTTpin = 4;                                 // LED controlled with MQTT
int count = 0, countDown = 200;
unsigned long LEDtime = 0, countTime;

BlynkTimer timer;                                   // declare Blynk timer
void timerEvent()                                   // function called by Blynk timer
{
  ccs811.read(&CO2, &TVOC, &errstat, &rawdata);
  if(errstat == CCS811_ERRSTAT_OK)
  {
    count++;                                        //increment reading counter
    countDown = 60;
    countTime = millis();                           // set countdown time
    Blynk.virtualWrite(V3, CO2);                    // send data to MQTT broker
    Blynk.virtualWrite(V4, TVOC);
    Blynk.virtualWrite(V6, count);
    Blynk.virtualWrite(V7, countDown);
    Serial.printf("count %d \n", count);
    LEDtime = millis();
    digitalWrite(LEDpin, HIGH);                     // turn on indicator LED
  }                                                 // turn off LED after 100ms
  if(millis() - LEDtime > 100) digitalWrite(LEDpin, LOW);
  if(millis() - countTime > 5000)                   // countdown interval 5s
  {
    countTime = millis();                           // update countdown time
    countDown = countDown - 5;                      // update countdown
    Blynk.virtualWrite(V7, countDown);              // send to Blynk MQTT broker
  }
}

BLYNK_WRITE(V0)                                     // Blynk virtual pin 0
{
  digitalWrite(LEDMQTTpin, param.asInt());          // turn on or off LED
}

BLYNK_CONNECTED()                             // function called when ESP32 connected
{
  Serial.println("ESP32 connected to Blynk");
}

void setup()
{
  Serial.begin(115200);                       // initiate Blynk MQTT broker
  Blynk.begin(BLYNK_AUTH_TOKEN, ssid, password, "blynk.cloud", 80);
  timer.setInterval(1000, timerEvent);        // timer event called every second
  pinMode(LEDpin, OUTPUT);
  digitalWrite(LEDpin, LOW);
  pinMode(LEDMQTTpin, OUTPUT);                // define LED pin as output
  digitalWrite(LEDMQTTpin, LOW);              // turn off LED
  Wire.begin();
  ccs811.begin();
```

```
  ccs811.start(CCS811_MODE_60SEC);          // set reading interval at 60s
}

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

Email with Blynk MQTT broker

An email and a notification to the *Blynk IoT* app are sent by a Blynk event, such as when a sensor variable exceeds a threshold (see Figure 9-4). 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 "*CO2 above 1500: 1603*" is included in the email text with the `Blynk.logEvent("alert", "CO2 above 1500: " + String(CO2))` instruction for a CO2 value of 1603ppm and an event called *alert*.

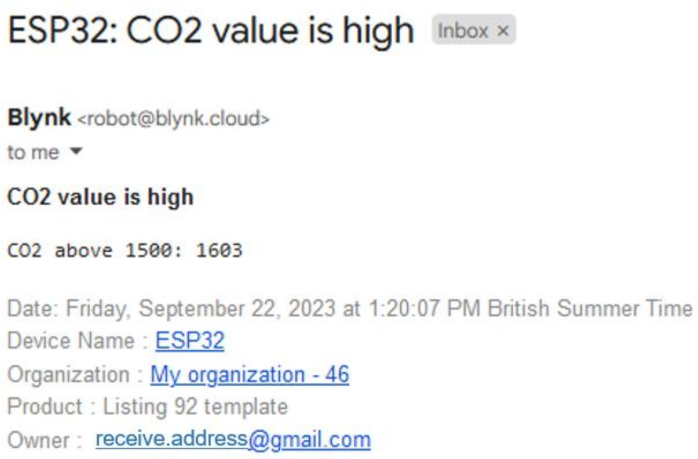

Figure 9-4 *Blynk email*

A Blynk event is defined in a template *Events* option (see Figure 9-5). 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.

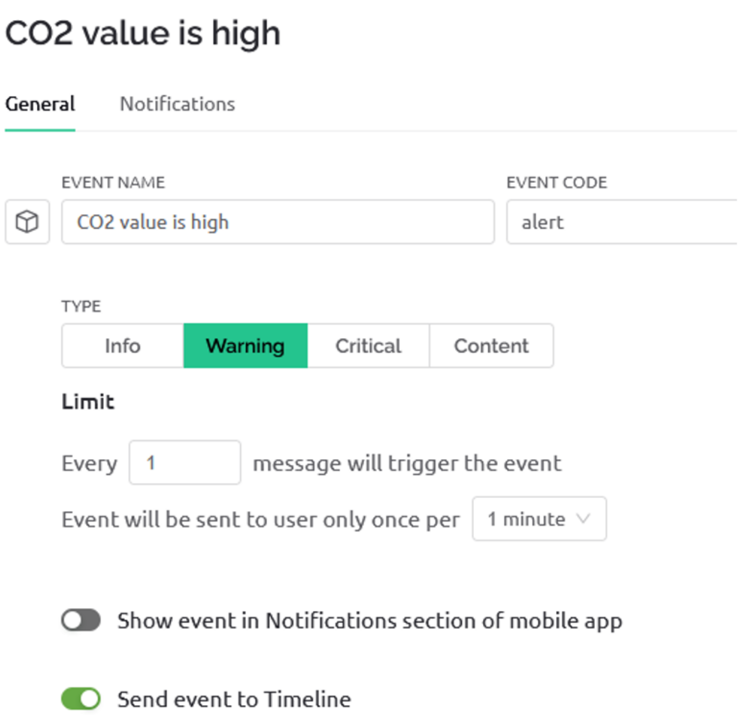

***Figure* 9-5** *Blynk event*

Listing 9-3 includes the additional instructions to Listing 9-2 for triggering a Blynk event, called *alert*, when the *CO2* value exceeds a threshold of 1500ppm. Every second, the *CO2* variable is compared to the threshold and when the threshold is exceeded, the Blynk event is triggered to send an email. To avoid emails being repeatedly sent, the Blynk event is only triggered when the *CO2* value exceeds the threshold and the *flag* value is zero, with the *flag* variable set to one when the CO2 value initially exceeds the threshold. The *flag* variable is reset to zero only when the *CO2* value is below the threshold. The *flag* and *lag* variables are defined as an integer equal to zero and as an unsigned long at the start of the sketch.

***Listing* 9-3** *Event triggered by high CO2*

```
if(millis() - lag > 1000)                        // at one second intervals
{
  if(CO2 > 1500 && flag == 0)                    // when CO2 level > threshold
  {                                              //   and email not sent already
    digitalWrite(LEDMQTTpin, HIGH);              // turn on LED and virtual LED
    Blynk.virtualWrite(V0, 1);                   // call Blynk event called alert
    Blynk.logEvent("alert", "CO2 above 1500: " + String(CO2));
    flag = 1;                                    // flag indicates event triggered
  }
  else if(CO2 < 1500 && flag == 1)               // when CO2 < threshold
```

```
  {                                        //   after sending email
    flag = 0;                              // reset event flag
  }
  lag = millis();                          // update time counter
}
```

Arduino IoT Cloud

Details, documentation and examples sketches of the Arduino IoT (Internet of Things) Cloud MQTT broker are available at cloud.arduino.cc and docs.arduino.cc/arduino-cloud, respectively. The *ArduinoIoTCloud* library is available in the Arduino IDE. When the *ArduinoIoTCloud* library is installed, there is the option to install several other libraries, but with an ESP32 microcontroller the required libraries are: *ArduinoIoTCloud*, *Arduino_ConnectionHandler*, *ArduinoMqttClient*, *Arduino_ESP32_OTA* and *Arduino_DebugUtils*.

The Arduino IoT Cloud maps sketch variables to the dashboard with the *addProperty* instruction, which defines the read/write status of a variable and the timing of variable updates. For example, the `addProperty(sensor, READ, 5*SECONDS, NULL)` instruction updates the Arduino IoT Cloud dashboard with the sketch *sensor* value at 5s intervals, while the `addProperty(LEDstate, READWRITE, ON_CHANGE, LEDchange)` instruction calls the *LEDchange* function in the sketch when the dashboard *LEDstate* variable is changed. Variables passed between the Arduino IoT Cloud dashboard and the ESP32 microcontroller are defined in the sketch. Variables specific to the Arduino IoT Cloud dashboard and to associated devices, such as an ESP32 microcontroller, are defined in the Arduino IoT Cloud (see Figure 9-6).

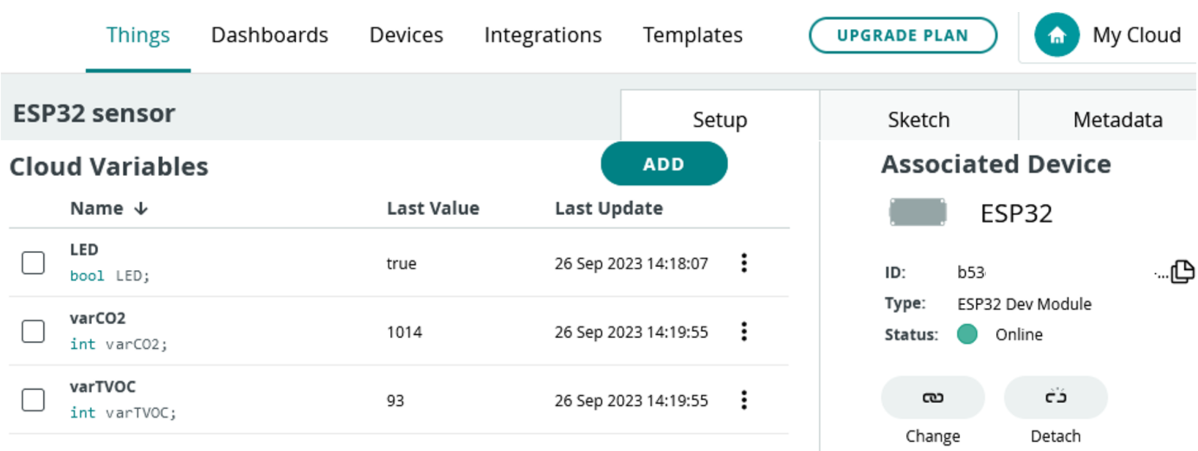

Figure 9-6 *Arduino IoT Cloud Thing*

From cloud.arduino.cc, click the "*keypad*" logo at the top right of the screen. select *IoT Cloud* and create an account or login to your account. After logging on, click *CREATE THING* button and change *Untitled* to the name of the project, such as *ESP32 CO2 sensor*. To add variables specific to the Arduino IoT Cloud dashboard, click the *ADD VARIABLE* button, add a variable name and select the variable type from the *Basic types* option, consisting of *Boolean*, *Character String*, *Float Point Number* and *Integer Number* (see Figure 9-7). NOTE: ensure that the variable *Name* and the *Declaration* name are identical. For *Variable Permission*, select *Read &Write* for an *OUTPUT* variable, such as an LED state, or select *Read Only* for an *INPUT* variable, such as a sensor reading. For *Variable Update Policy*, select *On change* to update the variable on Arduino IoT Cloud dashboard when the variable changes state or value, such as when a dashboard button is clicked. In contrast, when a variable is uploaded to the Arduino IoT Cloud dashboard at set intervals, select *Periodically* and then enter the number of seconds between updates, such as for a sensor reading. Finally, select *ADD VARIABLE*. The properties of a variable are edited, by clicking the three dots to the right of the variable name (see Figure 9-6).

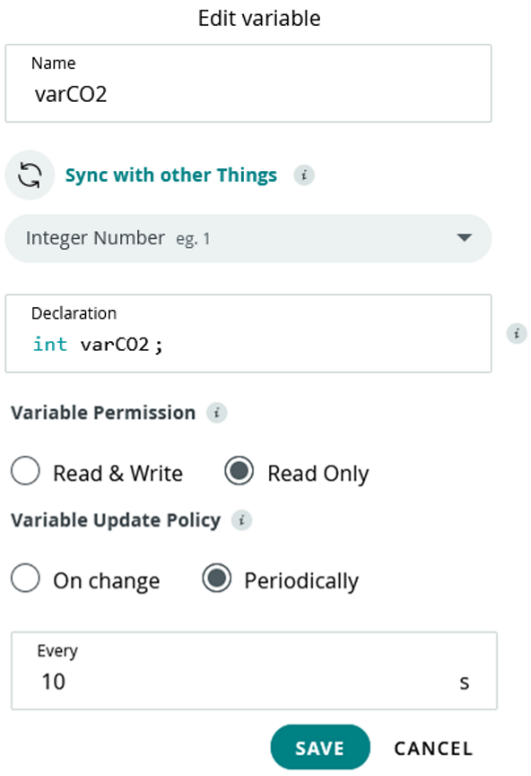

Figure 9-7 *Arduino IoT Cloud variable*

A device, such as an ESP32 microcontroller, is associated with an Arduino IoT Cloud dashboard after clicking the *Select Device* button under the *Associated Device* option. Select *SET UP NEW DEVICE*, click the *Third party device* button, click the *ESP32* button and select the model, such as *ESP32 Dev Module*. Click the *CONTINUE* button and give the device a name, such as *ESP32*, and click the *Next* button. A Device ID and a Secret Key for the ESP32 microcontroller are automatically generated by the Arduino IoT Cloud. Click on the text *download the PDF* to save a PDF document with the Device ID and a Secret Key or copy the details. Tick the box beside *I saved my device ID and Secret Key* and click *CONTINUE*

The Arduino IoT Cloud dashboard (see Figure 9-8) is built by selecting the *Dashboards* option, at the top of the screen, and clicking the *BUILD DASHBOARD* button. Change the *Untitled* name of the dashboard, such as to *CO2 and TVOC sensor*, and click the *ADD* button. From the left-side menu, click the *Switch* icon and update the name of the switch. The switch is linked to a variable by clicking the *Link variable* button, choosing from the displayed variables, such as *LED Boolean*, and clicking the *LINK VARIABLE* and *DONE* buttons. To display a variable value, select a *Value* or a *Gauge* widget and add minimum and maximum values in the *Min* and *Max* boxes. The position and size of widgets on the Arduino IoT Cloud dashboard, for the Desktop or the Mobile layout are changed by clicking the "*pencil on paper*" logo, the *Desktop* or *Mobile* logos and then the "*cross arrow*" logo at the top of the screen. After updating the layout, click the *DONE* button.

The *Arduino IoT Cloud Remote* app is downloaded from the *Google Play Store* and after logging on to your account, the dashboard is displayed.

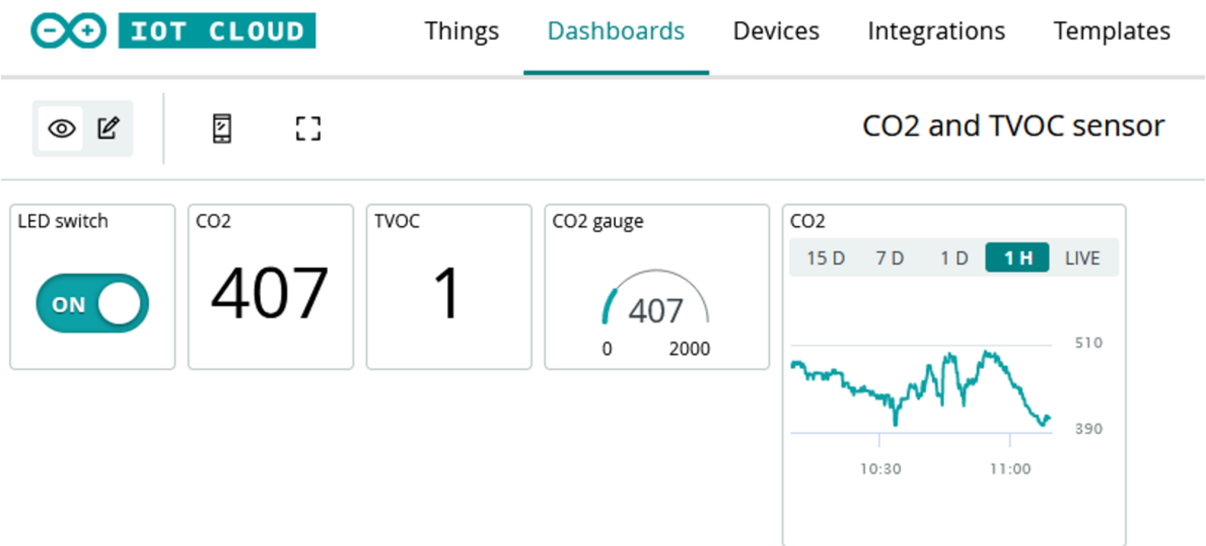

Figure 9-8 *Arduino IoT dashboard*

Listing 9-4 defines the device, such as an ESP32 microcontroller, associated with the Arduino IoT Cloud dashboard and the *initProperties* function defines the Arduino IoT Cloud dashboard variables, the read/write status of each variable and the timing of variable updates. The `WiFiConnectionHandler` instruction connects the device, which is the ESP32 microcontroller, to the WLAN and to the Arduino IoT Cloud. Instructions in Listing 9-4 are included in the *properties.h* tab with the main sketch shown in Listing 9-5.

Listing 9-4 *Arduino IoT Cloud MQTT broker details*

```
#include <ArduinoIoTCloud.h>                        // Arduino IoT Cloud libraries
#include <Arduino_ConnectionHandler.h>              // ESP32 device name and key
const char DEVICE_LOGIN_NAME[] = "xxxx";            // change xxxx to Device login
const char DEVICE_KEY[] = "xxxx";                   // change xxxx to Device Key
#include <ssid_password.h>                          // file with logon details

void LEDchange();                                   // forward declaration of function

bool LED = 0;                                       // Arduino dashboard variables
int varCO2, varTVOC;

void initProperties()                               // details of device name & key
{                                                   //   and dashboard variable
  ArduinoCloud.setBoardId(DEVICE_LOGIN_NAME);
  ArduinoCloud.setSecretDeviceKey(DEVICE_KEY);
  ArduinoCloud.addProperty(LED, READWRITE, ON_CHANGE, LEDchange);
  ArduinoCloud.addProperty(varCO2, READ, 10 * SECONDS, NULL);
  ArduinoCloud.addProperty(varTVOC, READ, 10 * SECONDS, NULL);
}
                                                    // connection to your Wi-Fi router
WiFiConnectionHandler connHandl(ssid, password);
```

Listing 9-5 is broadly similar to Listing 9-1 with the additional instructions to Listing 9-1 commented. When the LED state is changed on the Arduino IoT Cloud dashboard, the ESP32 microcontroller receives the updated state and the *LEDchange* function is called by the `addProperty(LED, READWRITE, ON_CHANGE, LEDchange)` instruction. The *CO2* values are sent to the Arduino IoT Cloud dashboard with the `addProperty(CO2, READ, 10*SECONDS, NULL)` instruction, with the time interval included as a parameter and similarly for the *TVOC* values. Note that the *DEVICE_LOGIN_NAME* and *DEVICE_KEY* are equal to the *Device ID* and *Secret Key*, as generated by the Arduino IoT Cloud. The `printDebugInfo(N)` instruction displays information on the state of the network, the Arduino IoT Cloud connection and errors, with a default level of 0 (only errors) and maximum of 4.

The *CO2change* function detects when the *CO2* value initially exceeds the threshold and turns on the LED to alert that the *CO2* value is high. The LED is turned off remotely through the Arduino IoT Cloud dashboard. When the *CO2* value decreases to below the threshold, the *flag* variable is reset to enable detection of a subsequent high *CO2* value. Without the *flag* variable, turning the LED off through the Arduino IoT Cloud dashboard would not be possible, while the *CO2* value exceeds the threshold.

Listing 9-5 *TOC and CO2 measurement and Arduino IoT Cloud MQTT broker*

```
#include "properties.h"                     // device and dashboard details
#include <Wire.h>
#include <ccs811.h>;
CCS811 ccs811(19);
uint16_t CO2, TVOC, errstat, rawdata;
int LEDMQTTpin = 4;                         // LED controlled with MQTT
int flag = 0;                               // flag to indicate high CO2

void setup()
{
  Serial.begin(115200);
  initProperties();                         // Arduino IoT Cloud properties
  ArduinoCloud.begin(connHandl);            // connect to Arduino IoT Cloud
  pinMode(LEDMQTTpin, OUTPUT);              // define LED pin as output
  digitalWrite(LEDMQTTpin, LOW);            // turn off LED
  Wire.begin();
  ccs811.begin();
  ccs811.start(CCS811_MODE_10SEC);
  setDebugMessageLevel(4);                  // status of Arduino IoT Cloud
  ArduinoCloud.printDebugInfo();            //   connection
}

void loop()
{
  ArduinoCloud.update();                    // Arduino IoT Cloud update
  ccs811.read(&CO2, &TVOC, &errstat, &rawdata);
  if(errstat == CCS811_ERRSTAT_OK)
  {
    varCO2 = CO2;
    varTVOC = TVOC;
    Serial.printf("CO2 %d TVOC %d \n", varCO2, varTVOC);
    CO2change();                            // call CO2change function
  }
}

void CO2change()                     // function to set LED state
{
  if(CO2 > 1000 && flag == 0)        // CO2 initially exceeds threshold
  {
    flag = 1;
    Serial.println("trigger = 1");
    LED = 1;                         // only changes switch state on dashboard
    LEDchange();                     // required to change LED state
```

```
  }
  else if(CO2 < 1000 && flag == 1)          // CO2 now less than threshold
  {
    flag = 0;
    Serial.println("trigger = 0");
  }
}

void LEDchange()                            // function to turn on or off LED
{                                           // LED state set by MQTT broker
  digitalWrite(LEDMQTTpin, LED);
  if(LED == 1) Serial.println("LED ON");
  else Serial.println("LED OFF");
}
```

Email and Arduino IoT Cloud

One option to link the Arduino IoT Cloud with an email function is to utilise the IFTTT (if this, then that) internet service to initiate a web request when an event occurs, such as an Arduino IoT Cloud dashboard variable exceeding a threshold. The IFTTT service is linked to the Webhooks function (ifttt.com/maker_webhooks), which receives and actions the web request to transmit an email. However, the IFTTT service is triggered to make a web request when any Arduino IoT Cloud dashboard variable is updated, such as when the `ArduinoCloud.update()` instruction occurs, rather than when a specific variable is updated.

Adafruit IO

Details and documentation for the Adafruit IO (Internet Of things) MQTT broker are available at io.adafruit.com and learn.adafruit.com/search?q="adafruit io". The *Adafruit_IO_Arduino* library contains a comprehensive set of example sketches and with the library available in the Arduino IDE. When the *Adafruit_IO_Arduino* library is installed, there is the option to install several other libraries, but with an ESP32 microcontroller, only the additional *ArduinoHttpClient* and *Adafruit_MQTT_Library* libraries are required.

The Adafruit IO MQTT broker consists of data feeds between the MQTT broker and an ESP32 microcontroller with an Adafruit IO dashboard displaying information from the data feeds. In the Adafruit IO, a data feed is defined by a data feed name with a data feed key automatically generated by the Arduino IO to consist of the lower-case alphanumeric characters *a* to *z* and *0* to *9* (see learn.adafruit.com/naming-things-in-adafruit-io). For example, a data feed for the *DATA* variable has an Adafruit IO data feed name and key of *DATAfeed* and *datafeed*, respectively. In a sketch, a data feed is defined by the `AdafruitIO_Feed`

`*DATAvariable = adaIO.feed("DATAfeed")` instruction containing the data feed reference, *DATAvariable*, and the data feed name, *DATAfeed*, which can differ from the data feed reference. One option is to define the data feed reference and the data feed name relative to data transmission to or from an ESP32 microcontroller and the Adafruit IO. With the *DATA* example, the data feed is from the ESP32 microcontroller with a data feed reference of *DATAout*, while relative to the Adafruit IO, the data feed name is defined as *DATAin*., resulting in the `AdafruitIO_Feed *DATAout = adaIO.feed("DATAin")` instruction.

Data is sent by an ESP32 microcontroller to the Adafruit IO with the `save()` function, as in the example of `DATAout->save(DATA)` instruction. The `save()` instruction should be limited to no more than 30 per minute.

An Adafruit IO data feed is accessed directly with the `get()` instruction, rather than waiting for a data feed from the Adafruit IO. To directly access a data feed, the data feed key, rather than the data feed name, is included in the `AdafruitIO_Feed *DATAin = adaIO.feed("dataout")` instruction. In the *setup* function of a sketch, the `get()` instruction provides the dashboard value or state of a data feed when an ESP32 microcontroller restarts, as in the `DATAin->get()` instruction.

Figure 9-9 illustrates the data feed, *redLEDout*, sent from the Adafruit IO with a data feed key of *redlenout*. In contrast, the data feed, *LDRin*, received by the Adafruit IO has a data feed key of *ldrin*.

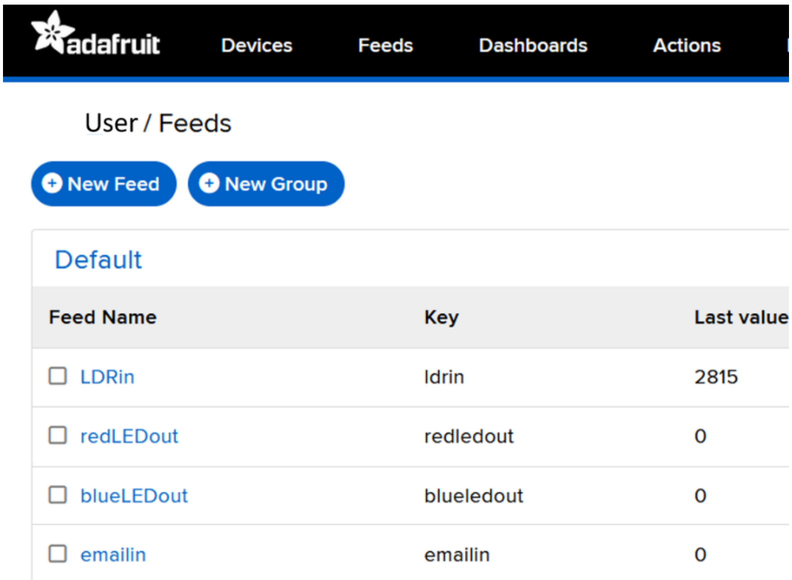

Figure 9-9 *Adafruit IO data feeds*

Following an action on the Adafruit IO dashboard, such as clicking a button to control an LED state, data is sent by the Adafruit IO to an ESP32 microcontroller, with the data feed defined in the sketch by the `AdafruitIO_Feed *LEDin = adaIO.feed("LEDout")` instruction. When the data feed is received by the ESP32 microcontroller, the `LEDin->onMessage(LEDchange)` instruction calls the *LEDchange* function, defined by the `void LEDchange(AdafruitIO_Data *data)` instruction to update the LED state with the `digitalWrite(LEDpin, data->toInt())` instruction.

Integer, binary, real or string variables are received from the Adafruit IO with the `data->toInt(),data->toPinLevel(),data->value(),data->toString()` instructions. A data feed name is obtained with the `data->feedName()` instruction.

Adafruit IO data feeds and dashboard are accessed on io.adafruit.com by creating an Adafruit IO account or logging in to your account. Click on the *Dashboards* option, click on the *New Dashboard* box and enter a name for the dashboard and click *Create*. To add a dashboard widget or block, click on the Dashboard Settings "*cog*" logo at the top-right of the dashboard and select *Create New Block*. Select an existing data feed or enter a new data feed name and click the *Create* button that appears directly to the right of the text box. In the *Connect a Feed* screen, select the required data feed and click *Next Step*. In the *Block settings* screen, details of the displayed block values, text and icons are entered.

Blocks on the Adafruit IO are positioned and re-sized by clicking on the Dashboard Settings "*cog*" logo and selecting the *Edit Layout* option. To change block characteristics, click the "*cog*" logo of the block and select the *Edit Block* option. Details of block characteristics are displayed by selecting the *Block info* option.

Details of a data feed are obtained by selecting the required feed from the *Feeds* option at the top of the io.adafruit.com webpage, where the *Feed History* and the *Download All Data* options are accessed.

The number of data feeds and dashboards and any live feeds are displayed by clicking on *Account* at the top-right of the io.adafruit.com webpage and selecting the *Adafruit IO Profile* option.

The Adafruit IO dashboard is illustrated with a notification example (see Figure 9-10). Two Adafuit IO dashboard buttons control the states of a red LED and a blue LED connected to an ESP32 microcontroller. A light dependent resistor, *LDR*, is also connected to the ESP32 microcontroller in the vicinity of the blue LED, with the LDR reading increasing when the blue LED is turned on. The Adafruit IO displays the LDR reading, with an Adafruit

IO gauge block, to confirm when an LED is turned on or off. In the example, the Adafruit IO gauge block changes colour with LDR data feed values above or below the high or low level warning levels of 3000 and 1000, respectively.

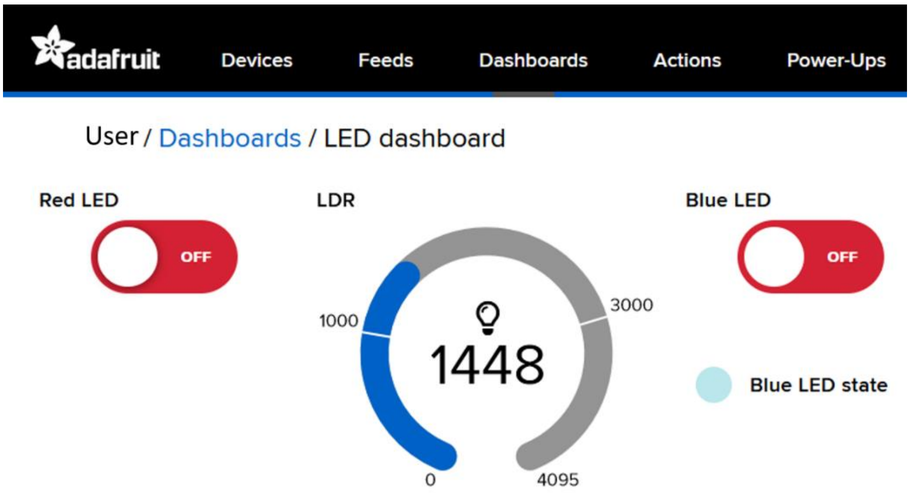

Figure 9-10 *Adafruit IO dashboard*

When the blue LED is turned on or off, the Adafruit IO *emailin* data feed is updated with values 1 or 0. The *emailin* data feed is mapped to an Adafruit IO indicator block, *Blue LED state* , which changes colour depending on the value (see Figure 9-11), and to an IFTTT trigger (see next section) to send an email when the blue LED is turned on.

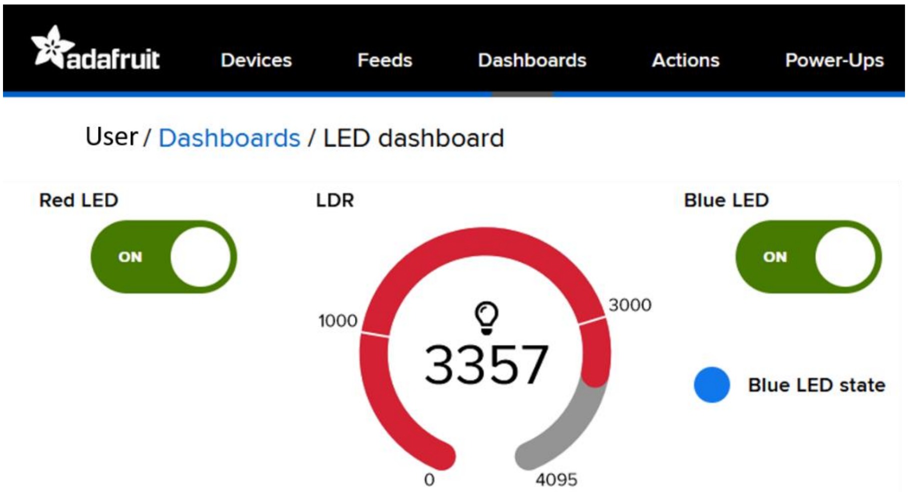

Figure 9-11 *Adafruit IO dashboard notifications*

A sketch utilising data feeds to and from the Adafruit IO requires an API username, *IO_USERNAME*, and key, *IO_KEY*, which are obtained by clicking the yellow "*key*" logo at the top right of the io.adafruit.com screen. Only the *AdafruitIO_WiFi* library is explicitly declared in a sketch, as the library references the *wifi/AdafruitIO_ESP32* library, which calls the *Adafruit_MQTT* and *AdafruitIO* libraries, with the latter calling the component libraries, such as the *ArduinoHttpClient* library.

In Listing 9-6, the data feeds from the Adafruit IO to the ESP32 microcontroller for the red and blue LEDs are defined by the `AdafruitIO_Feed *LEDin = adaIO.feed("ledout")` instruction format, with the data feed reference, *LEDin*, and the data feed key, *ledout*, indicating a data feed input to the ESP32 microcontroller and a data feed output from the Arduino IO, respectively. When a data feed input is received by the ESP32 microcontroller, the `LEDin->onMessage(LED_fn)` instruction calls the *LED_fn* function, which is mapped to the data feed reference, with the function defined by the `void LED_fn(AdafruitIO_Data *data)` instruction.

In contrast, when a variable is an output from the ESP32 microcontroller and an input to the Arduino IO, the Adafruit IO data feed is defined by the `AdafruitIO_Feed *DATAout = adaIO.feed("DATAin")` instruction.

Listing 9-6 *Adafruit IO*

```
#include "AdafruitIO_WiFi.h"                             // include library
#define IO_USERNAME "xxxx"                               // change xxxx to your
#define IO_KEY      "xxxx"                               // IO_USERNAME and IO_KEY
#include <ssid_password.h>
AdafruitIO_WiFi adaIO(IO_USERNAME, IO_KEY, ssid, password);
AdafruitIO_Feed *redLEDin = adaIO.feed("redledout");   // data feed key defined
AdafruitIO_Feed *blueLEDin = adaIO.feed("blueledout"); //   for ->get() command
AdafruitIO_Feed *LDRout = adaIO.feed("LDRin");
AdafruitIO_Feed *emailout = adaIO.feed("emailin");

int redLEDpin = 26;                                      // LED and LDR GPIO pins
int blueLEDpin = 27;
int LDR, LDRpin = 35;
unsigned long lag = 0;

void setup()
{
  Serial.begin(115200);                                  // Serial Monitor baud rate
  pinMode(redLEDpin, OUTPUT);                            // define pins as OUTPUT
  pinMode(blueLEDpin, OUTPUT);                           //   or INPUT
  pinMode(LDRpin, INPUT);
  adaIO.connect();                                       // connect to io.adafruit.com
  redLEDin->onMessage(redLED_fn);                        // call function when message
  blueLEDin->onMessage(blueLED_fn);                      // received from Adafruit IO
  while(adaIO.status() < AIO_CONNECTED)                  // wait for connection
  {                                                      //   to Adafruit IO
    Serial.print(".");
    delay(500);
  }
  Serial.println();
```

```
  Serial.println(adaIO.statusText());         // "Adafruit IO connected"
  redLEDin->get();                            // get dashboard states
  blueLEDin->get();                           // of red and blue LEDs
}

void redLED_fn(AdafruitIO_Data *data)         // function called when LED
{                                             //   state changed on dashboard
  Serial.print("red LED state ");
  if(data->toPinLevel() == HIGH) Serial.println("HIGH");
  else Serial.println("LOW");
  digitalWrite(redLEDpin, data->toPinLevel());   // turn on or off LED
}

void blueLED_fn(AdafruitIO_Data *data)
{
  Serial.print("blue LED state ");
  if(data->toPinLevel() == HIGH) Serial.println("HIGH");
  else Serial.println("LOW");
  digitalWrite(blueLEDpin, data->toPinLevel());
  int LED = data->toInt();                    // return data feed to Adafruit IO
  emailout->save(LED);                        //   to trigger IFTTT email
}

void loop()
{
  adaIO.run();                    // maintain client connection to io.adafruit.com
  if(millis() - lag > 10000)                  // after 10s intervals
  {
    LDR = analogRead(LDRpin);                 // update LDR reading
    Serial.printf("LDRout %d \n", LDR);
    LDRout->save(LDR);                        // send data feed to Adafruit IO
    lag = millis();
  }
}
```

Email with Adafruit IO

The sending of an email by the Adafruit IO is triggered when a condition is satisfied, such as when a sensor variable exceeds a threshold (see Figure 9-12). Accounts on the Adafruit IO and the IFTTT (IF This, Then That) internet service are connected with an IFTTT Applet created to send an email. Note that the Adafruit IO data feed name and corresponding feed key must be identical, as IFTTT monitors changes in the specified Adafruit IO data feed.

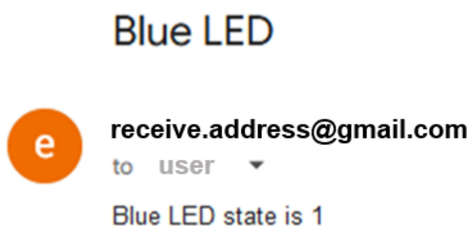

Figure 9-12 *Adafruit IO email*

Open ifttt.com and login or open an account on the IFTTT internet service. Click the *Create* button at the top of the screen. In the *If This* box, click *Add*, select the *Adafruit* service and select the *Monitor a feed on Adafruit IO* option, which will trigger an action when a condition is satisfied. [When IFTTT is accessed for the first time, click the *Connect* button, which loads the Adafruit IO webpage, and click the *Authorize* button, which connects your Adafruit IO and IFTTT accounts.] On the IFTTT *Monitor a feed on Adafruit IO* screen, update the *Feed* option to the required Adafruit IO data feed, the *Relationship* and *Value* options to the required trigger values, such as ">" and "0", and then click *Create trigger*.

In the *Then That* box, click *Add*, select the *Gmail* service and select the *Send yourself an email* option, which sends an email when the trigger is activated. [When IFTTT is accessed for the first time, click the *Connect* button, which loads the Gmail webpage, and after logging into your Gmail account, click the *Allow* button, which connects your Gmail address and IFTTT accounts.] On the IFTTT *Send an email* screen, update the *Subject* and *Body* fields with the email header and content, and then click *Create action*. Finally, click the *Continue* button, which displays the IFTTT Applet Title, and click the *Finish* button.

To delete an IFTT applet on ifttt.com/p/username/applets/private, click the Applet to be deleted and click *Delete* to remove the Applet from your IFTTT account.

When a condition on the Adafruit IO is satisfied, such as a received data feed exceeding a threshold or an Adafruit IO button state changing, then the IFTTT internet service should send an email. For example, when the Adafruit IO *emailin* data feed exceeds 0, an email is sent by the IFTTT internet service (see Figure 9-13). When an email is sent, the lag time between Adafruit IO triggering an email and receiving the email alert is at least 75s, in practice, although the Adafruit IO website indicates a lag time of up to 15 minutes.

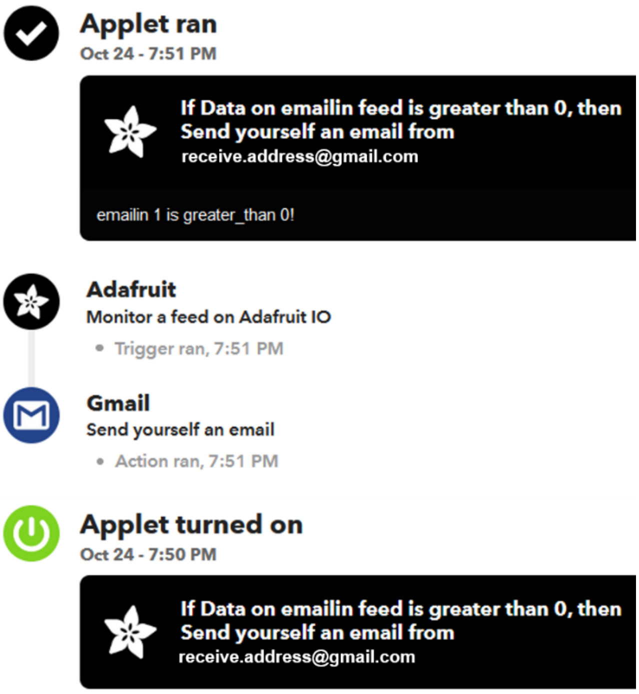

Figure 9-13 *Adafruit IO and IFTTT email trigger*

Map facility with Adafruit IO

The Adafruit IO includes the facility to display an *OpenStreepMap* map covering the location defined by a latitude and longitude pair. Figure 9-14 illustrates an Adafruit IO dashboard with a *Map* block and a *Stream* block, with the latter displaying location data feed inputs to the Adafruit IO.

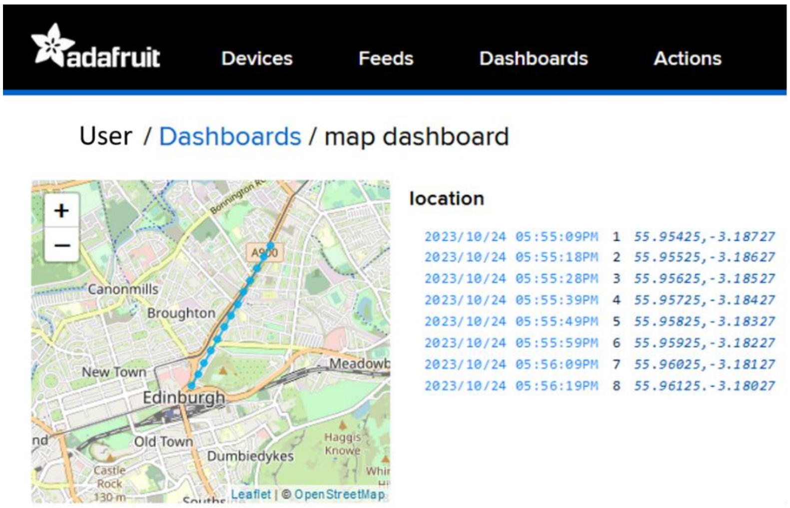

Figure 9-14 *Adafruit IO map*

The sketch in Listing 9-7 sends a data feed with updated values for longitude and latitude to the Adafruit IO. The data feed format is an integer variable and three double variables, as in the `location->save(count, lat, lon, elev)` instruction.

Listing 9-7 *Adafruit IO map*

```
#include "AdafruitIO_WiFi.h"                    // include library
#define IO_USERNAME "xxxx"                      // change xxxx to your
#define IO_KEY      "xxxx"                      // IO_USERNAME and IO_KEY
#include <ssid_password.h>
AdafruitIO_WiFi adaIO(IO_USERNAME, IO_KEY, ssid, password);
AdafruitIO_Feed *location = adaIO.feed("location");

unsigned long lag = 0;
int count = 0;
double lat = 55.953251;                         // initial latitude
double lon = -3.188267;                         //   and longitude
double elev = 0;                                // m above sea level

void setup()
{
  Serial.begin(115200);                         // Serial Monitor baud rate
  adaIO.connect();                              // connect to io.adafruit.com
  while(adaIO.status() < AIO_CONNECTED)         // wait for connection
  {                                             //   to Adafruit IO
    Serial.print(".");
    delay(500);
  }
  Serial.println();
  Serial.println(adaIO.statusText());           // "Adafruit IO connected"
}

void loop()
{
  adaIO.run();                        // maintain client connection to io.adafruit.com
  if(millis() - lag > 10000)                    // after 10s intervals
  {
    count++;                                    // increment counter,
    lat = lat + 0.001;                          //   latitude
    lon = lon + 0.001;                          //   and longitude
    location->save(count, lat, lon, elev);      // data feed to Adafruit IO
    lag = millis();
  }
}
```

Blynk, Arduino IoT Cloud and Adafruit IO MQTT brokers

In the Arduino IoT Cloud, the webpage and the *Arduino IoT Cloud Remote* app dashboards are generated simultaneously, with the position and size of widgets on the *Arduino IoT Cloud*

Remote app dashboard managed through the Arduino IoT Cloud website. In contrast, the Blynk web dashboard and the *Blynk IoT* app dashboard are built separately in the Blynk webpage and in *Blynk IoT* app, respectively. Widget colour is selectable in the *Blynk IoT* app. The Adafruit IO does not have an associated app. Blynk and Adafruit IO enable transmission of an email, with a notification to the *Blynk IoT* app, when a specific Blynk and Adafruit IO dashboard variable exceeds a threshold. The Blynk email function is easier to set up and more reliable than with the Adafruit IO. Historic data is downloadable from the Arduino IoT Cloud and Adafruit IO dashboards.

MQTT and smart meter

An ESP32 module is connected to a smart meter and provides energy usage readings to the MQTT broker for display on the MQTT broker dashboard. The ESP32 module is battery powered and the battery voltage is also transmitted to the MQTT broker to alert the user when to replace or charge the battery. Power consumption is measured at one minute intervals, with the ESP32 microcontroller in deep sleep mode between readings, to save battery power. The deep sleep function enables the ESP32 microcontroller to wake up, measure and transmit energy usage, power consumption and battery voltage to the MQTT broker and then return to deep sleep mode.

Figure 9-15 illustrates power consumption measured every minute over a 90 minute period. A central heating boiler operated until 7:45AM, a 2kW kettle was switched on at 7:40AM and again at 8:15AM with an 8kW shower used from 8:05AM. The baseline power usage was 200W.

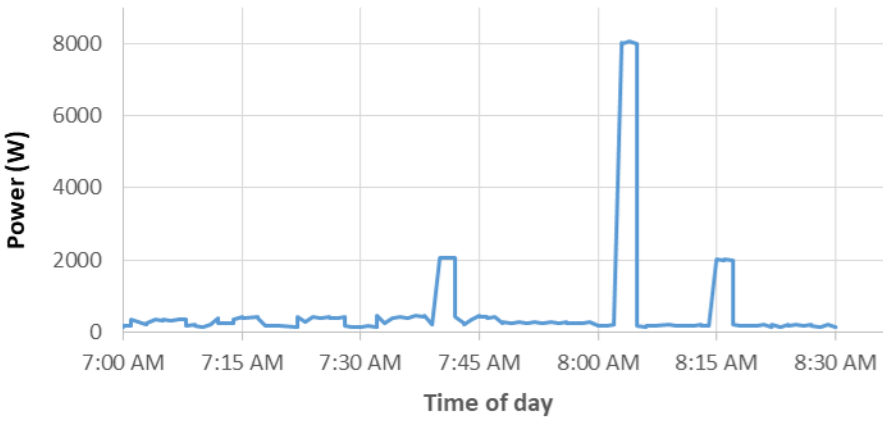

Figure 9-15 *MQTT dashboard of power consumption*

Energy usage measurement

Current usage is measured with the SCT013 current transformer, which is a current clamp meter. When a conductor (cable) supplying a load is clamped, the conductor is effectively the primary winding of a current transformer. The wire coil around the SCT013 current transformer core is the secondary winding. The alternating current in the conductor produces an alternating magnetic field in the SCT013 core, that induces an alternating current in the secondary winding, which is converted to a voltage and quantified by the ESP32 microcontroller analog to digital converter (ADC).

The SCT013 current transformer outputs a maximum current of 50mA given a load maximum current usage of 100A, as the SCT013 current transformer includes 2000 coil turns. The current in the secondary winding is the current in the primary winding divided by the number of coil turns on the secondary winding. The SCT013 output current, I_{OUT}, is determined from the measured voltage across a burden resistor of known value (see Figure 9-16). The SCT013 current transformer output signal is combined with an offset voltage, as explained in the next paragraphs, equal to half of the voltage divider supply voltage, *Vsply*. A burden resistor of $\frac{Vsply/2}{\sqrt{2}\times IOUT}\Omega = \frac{1.65}{\sqrt{2}\times 0.05}\Omega$ or 23.3Ω is required and a 22Ω burden resistor is sufficient. The scalar of $\sqrt{2}$ converts the RMS (Root Mean Square) SCT013 current transformer output, I_{OUT}, to the peak current of an AC (Alternating Current) signal. Further details on measuring current and apparent power with the SCT013 current transformer are available at learn.openenergymonitor.org/electricity-monitoring/ct-sensors/introduction.

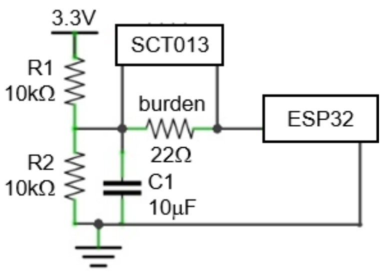

Figure 9-16 *SCT013 current transformer connections*

The SCT013 output current, I_{OUT}, is alternating current (AC) and the measured voltage across the burden resistor follows a sinusoidal wave or sine curve (see Figure 9-17). The SCT013 output current corresponding to a load current of L amps is $L\times\sqrt{2}\times 0.05/100$ A, given the SCT013 maximum current of 50mA with a load maximum current of 100A. When measured with an oscilloscope, the peak-to-peak voltage across the burden resistor, *R*, is $2 \times I_{OUT} \times$R volts, with the factor of two reflecting the positive and negative AC signal of the SCT013 current transformer. For example, a load current of 20A RMS corresponds to an SCT013 output current of 14.1mA = $20 \times \sqrt{2} \times 0.05/100$ A and a peak-to-peak voltage, measured with an oscilloscope, across the 22Ω burden resistor of 622mV = 2×14.1mA×22Ω.

The peak-to-peak voltage is measured with the analog to digital convertor (ADC) of the ESP32 microcontroller. The ADC only measures positive voltages and a direct current (DC) offset voltage is combined with the SCT013 AC output, which is centred on zero. The offset voltage is provided by a voltage divider, formed by resistors *R1* and *R2* (see Figure 9-16). The combined AC and DC signal has minimum and maximum voltages of $Vsply/2 \pm I_{OUT} \times$R volts. For example, a load current of 20A RMS corresponds to minimum and maximum voltages of 1339 and 1961mV = 3.3V/2 ± 311mV and a peak-to-peak voltage of 622mV. Without the offset voltage, the maximum and minimum voltages of the sinusoidal signal, as measured by the ESP32 microcontroller ADC, are 311 and 0, respectively (see Figure 9-17). If an offset DC voltage is not added to the sinusoidal AC voltage and the ADC measured voltage is just doubled, then clipping of the AC signal will result in an underestimate of the peak to peak voltage.

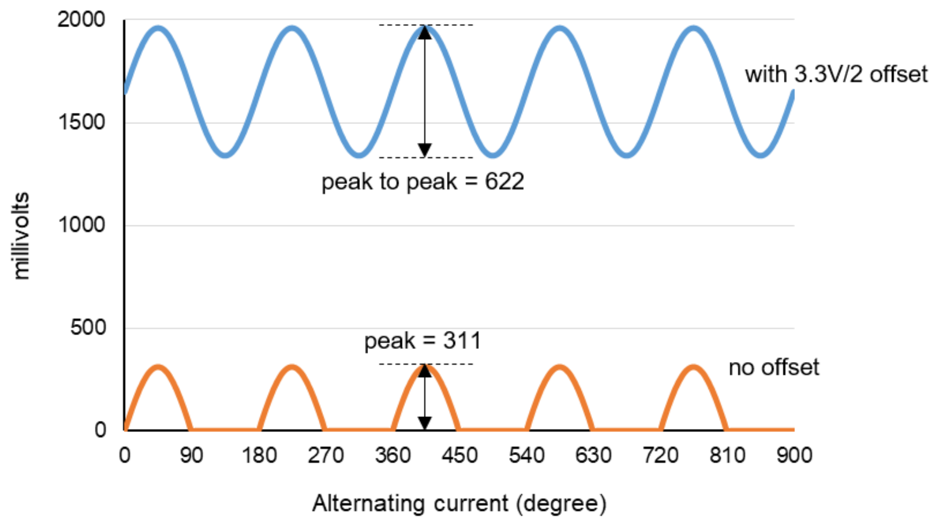

Figure 9-17 *SCT013 current transformer sensor readings*

The load usage current simplifies to $P2P/\sqrt{2}R$ RMS amps, given the peak-to-peak voltage, *P2P* volts, of the combined AC and DC signal across the burden resistor, *R*. From the earlier example, a peak-to-peak voltage of 622mV and a 22Ω burden resistor correspond to a load usage of 20A RMS.

For the Chapter, the current usage of a hairdryer at the high setting was estimated with the SCT013 current transformer. The peak-to-peak voltage, measured with the ESP32 microcontroller ADC, of 280mV equated to a current of 9.0A RMS, which was consistent with the current of 8.52A RMS, when measured with a multimeter. At low current loads, the voltage measured, on an oscilloscope, across the SCT013 burden resistor formed a clipped sinusoidal wave, which negatively biased the estimated current usage.

Wi-Fi, MQTT and smart meter

Monitoring energy usage with data forwarded to the MQTT broker, to display on the MQTT broker dashboard, requires an ESP32 microcontroller with a low current requirement, when the ESP32 microcontroller is battery powered. An ESP32 DEVKIT DOIT module requires 51mA when active, between 58 and 130mA when accessing a Wi-Fi network and 12mA in deep sleep mode. The corresponding current requirements of a TTGO T-Display V1.1 module are 38mA, 45 to 113mA and 325µA. A battery will power a TTGO T-Display V1.1 module longer than a ESP32 DEVKIT DOIT module, primarily due to the lower deep sleep current requirement of the TTGO T-Display V1.1 module.

Connections between a TTGO T-Display V1.1 module, an SCT013 current transformer, a burden resistor and a voltage divider are shown in Figure 9-18 and listed in Table 9-2.

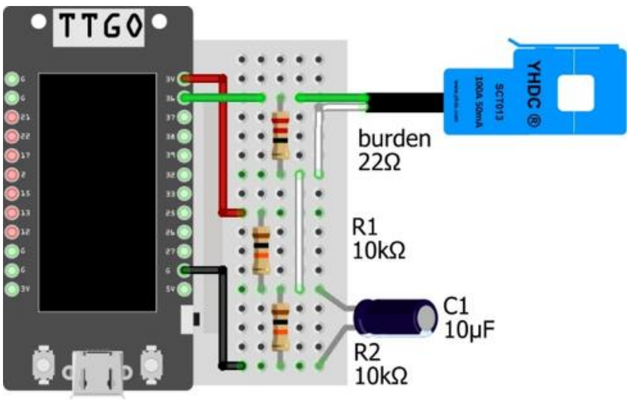

Figure 9-18 *ESP32 with SCT013 current transformer sensor*

Table 9-2 *SCT013 connections*

Component	Connect to	SCT103 wire colour
SCT013 signal	ESP32 GPIO 36	Red, but green in Figure 9-18
SCT013 signal	22Ω burden resistor	
SCT013	22Ω burden resistor	White
SCT013	Voltage divider mid-point	
Voltage divider 10kΩ resistors	ESP32 3V3 and GND	
10μF capacitor positive	Voltage divider mid-point	
10μF capacitor negative	GND	

The sketch for an SCT013 current transformer, acting as a smart meter, connected to an ESP32 module with energy usage information sent to the Arduino IoT Cloud, which is the MQTT broker, for display of information on the Arduino IoT Cloud dashboard, is given in Listing 9-8. Power consumption, battery voltage, ESP32 microcontroller re-connection time to the Arduino IoT Cloud and a counter are transmitted to the MQTT broker at one minute intervals. Connection to the Arduino IoT Cloud with the `WiFiConnectionHandler connHandle` instruction fails on the first attempt and a four second delay is automatically implemented prior to a further connection attempt. Inclusion of the *WiFi* library connection instructions prior to the `WiFiConnectionHandler connHandle` instruction reduces connection time to the MQTT broker.

After an ESP32 microcontroller connects to the Arduino IoT Cloud, the `ArduinoCloud.connected()` parameter changes from zero to one and the `ArduinoCloud.addCallback(ArduinoIoTCloudEvent::CONNECT, onConnect)` instruction calls the *onConnect* function. In Listing 9-8, the *onConnect* function determines the elapsed time for re-connection to the Arduino IoT Cloud and when the *connected* parameter changes from zero to one, energy usage variables are determined and updated on the Arduino IoT Cloud, then the ESP32 microcontroller is moved to deep sleep mode. The 5s delay, prior to `esp_deep_sleep_start()` instruction, allows time for updating the Arduino IoT Cloud dahsboard, as without a sufficient time delay, the Arduino IoT Cloud dashboard is not updated. The `ArduinoCloud.printDebugInfo()` instruction provides confirmation of data transmission with the *TimeServiceClass::setTimeZoneData offset: 3600 dst_unitl Unix epoch time* message. In practice, a value of 45000 = `45E6`, equating to 45s, for the `esp_sleep_enable_timer_wakeup(N)` instruction enabled an update to the Arduino IoT Cloud dashboard at one minute intervals.

Espressif notes that the *esp_deep_sleep* function does not thoroughly shut down connections with the Wi-Fi and Bluetooth communication protocols (see docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/sleep_modes.html). Wi-Fi or Bluetooth connections are closed with the `WiFi.disconnect(true)` and `WiFi.mode(WIFI_OFF)` instructions or the `btStop()` instruction before implementing the `esp_deep_sleep_start()` instruction.

The ESP32 microcontroller RTC (Real Time Clock) is active in deep-sleep mode and an incrementing counter is stored in the RTC memory at each energy reading. A counter to be stored in RTC memory is defined with the `RTC_DATA_ATTR int counter` instruction.

In Listing 9-8, the *taskFunction* obtains the SCT013 current transformer voltage across the burden resistor and also the battery voltage. When the TTGO T-Display V1.1 is battery powered, the ADC enable port on GPIO 14 must be set *HIGH* to activate the ADC on GPIO 34. In the *getCurrent* function, the initial readings from the SCT013 current transformer, which may be noisy, are discarded. A total of 500 voltage readings took 51.6ms to complete, ensuring that the minimum and maximum voltages were detected as readings were taken over 2.5 cycles, given the 50Hz frequency of the AC signal and cycle length of 20ms.

Listing 9-8 *Smart meter and MQTT*

```
#include "properties.h"                    // device and dashboard details
int battPin = 34, enabPin = 14, SCTpin = 36;
float RMS, mA, current;
int mV, volt, minVolt, maxVolt;
int Nread = 500;                           // number of current readings
unsigned long lag, micro = 45E6;           // shorthand 45×10^6 for 45secs
RTC_DATA_ATTR int count = 0;               // store count in RTC memory
int flag = 0;

void setup()
{
  Serial.begin(115200);
  pinMode(enabPin, OUTPUT);
  digitalWrite(enabPin, HIGH);             // pin set HIGH to active ADC
  lag = millis();
  initProperties();                        // Arduino IoT Cloud properties
  WiFi.mode(WIFI_AP_STA);                  // access point and station mode
  WiFi.begin(ssid, password);              // initialize and connect to Wi-Fi
  while (WiFi.status() != WL_CONNECTED) delay(500);
  ArduinoCloud.begin(connHandle);          // connect to Arduino IoT Cloud
  ArduinoCloud.addCallback(ArduinoIoTCloudEvent::CONNECT, onConnect);
  setDebugMessageLevel(4);                 // status of Arduino IoT Cloud
  ArduinoCloud.printDebugInfo();           //   connection
  esp_sleep_enable_timer_wakeup(micro);    // delay of 45s → restart at 60s
```

```
}

void onConnect()
{
  cnectTime = millis() - lag;                    // connect time to MQTT broker
  Serial.print("connect status ");Serial.println(ArduinoCloud.connected());
}

void taskFunction()
{
  battery = analogReadMilliVolts(battPin)*2.0/1000.0;  // battery voltage
  count++;
  countN = count;
  getCurrent();                                  // call function to measure current
}

void getCurrent()                          // function to calculate current usage
{
  maxVolt = 0;
  minVolt = 5000;
  for (int i=0; i<100; i++) analogRead(SCTpin); // ignore initial readings
  for (int i=0; i<Nread; i++)
  {
    volt = analogReadMilliVolts(SCTpin);         // SCT013 output voltage
    if(volt > maxVolt) maxVolt = volt;           // update maximum and
    if(volt < minVolt) minVolt = volt;           //   minimum voltages
  }
  mV = maxVolt - minVolt;                        // peak to peak mV
  mA = 0.5*mV/22.0;                              // mV with burden resistor to mA
  current = mA*100.0/50.0;                       // mA to current usage in amps
  RMS = current/sqrt(2.0);                       // RMS current usage
  power = 230.0 * RMS;                           // convert current to power
}

void loop()
{
  ArduinoCloud.update();                         // Arduino IoT Cloud update
  if(ArduinoCloud.connected() == 1 && flag == 0)
  {
    taskFunction();                        // call taskFunction once connected
    flag = 1;                              // flag to only set lag value once
    lag = millis();
  }                                        // time to pass data to Arduino IoT Cloud
  if((millis() - lag) > 5000 && flag == 1)
  {
    WiFi.disconnect();                     // disconnect Wi-Fi
    WiFi.mode(WIFI_OFF);
    esp_deep_sleep_start();                // ESP32 in deep sleep mode
  }
}
```

Listing 9-9 defines the device, which is an ESP32 microcontroller, associated with the Arduino IoT Cloud dashboard and the *initProperties* function defines the Arduino IoT Cloud dashboard variables and that variables are updated `ON_CHANGE`, rather than after a set time, as in Listing 9-4.

Listing 9-9 *Arduino IoT Cloud MQTT broker details with ON_CHANGE variables*

```
#include <ArduinoIoTCloud.h>                    // Arduino IoT Cloud libraries
#include <Arduino_ConnectionHandler.h>          // ESP32 device name and key
const char DEVICE_LOGIN_NAME[] = "xxxx";        // change xxxx to Device login
const char DEVICE_KEY[] = "xxxx";               // change xxxx to Device Key
#include <ssid_password.h>                      // file with logon details

void onConnect();                               // forward declaration of function

float battery;                                  // Arduino dashboard variables
int power, cnectTime, countN;

void initProperties()                           // details of device name & key
{                                               //   and dashboard variables
  ArduinoCloud.setBoardId(DEVICE_LOGIN_NAME);
  ArduinoCloud.setSecretDeviceKey(DEVICE_KEY);
  ArduinoCloud.addProperty(battery, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(power, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(cnectTime, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(countN, READ, ON_CHANGE, NULL);
}
                                                // connection to your Wi-Fi router
WiFiConnectionHandler connHandle(ssid, password);
```

A TTGO T-Display V1.1 module was powered by an 18650 Li-Ion battery mounted in an 18650 battery V3 micro USB charging shield. After 5199 wake-measure-transmit-deep sleep cycles, each lasting 68s, the battery voltage dropped to 2.4V and the battery stopped powering the microcontroller. There was a gradual decrease in battery voltage over 90 hours, followed by a rapid decline (see Figure 9-19).

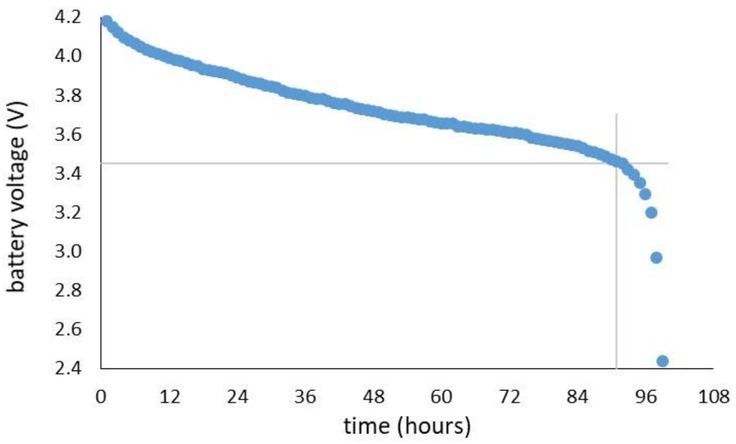

Figure 9-19 *18650 battery discharge curve*

A practical smart meter must be powered on a single battery for longer than four days. Extending the one minute interval between SCT013 current transformer readings will increase the time powered by the battery, but will decrease the resolution of daily cumulative energy usage. Several high capacity 18650 batteries combined in a 18650 battery V3 micro USB charging shield or in an 18650 battery case is one solution.

ESP-NOW, MQTT and smart meter

In Listing 9-8, a battery powered ESP32 microcontroller transmits SCT013 current transformer data to the MQTT broker with Wi-Fi communication (see Figure 9-20a). Connection to the Arduino IoT Cloud MQTT broker and dashboard updates required 12.8s (see Table 9-3). In an alternative arrangement, a battery powered ESP32 microcontroller transmits data, with the ESP-NOW protocol, to a second ESP32 microcontroller, which is not battery powered and has a permanent connection to the MQTT broker (see Figure 9-20b). The second ESP32 microcontroller forwards the data to the MQTT broker. An ESP32 microcontroller requires only 364µs to establish ESP-NOW communication with the recipient ESP32 microcontroller and just 227µs to transmit data (see Table 9-3), which is faster than allocating a time interval to update the Arduino IoT Cloud. The advantage of two ESP32 microcontrollers and the combination of ESP-NOW and Wi-Fi communication is the lower connection and data transmission times with the ESP-NOW protocol, which enable the battery powered ESP32 microcontroller to operate for a longer period.

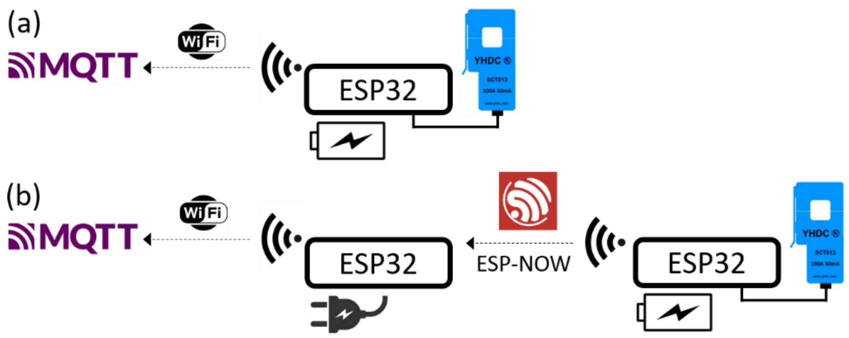

Figure 9-20 *ESP-NOW, smart meter and MQTT*

For ESP-NOW communication between two ESP32 microcontrollers, when one microcontroller is connected to a router for Wi-Fi communication, the ESP-NOW communication channel must be the same as the Wi-Fi communication channel. Wi-Fi routers automatically switch to the least congested channel and the available networks must be scanned to identify the communication channel of the specified router. Figure 9-21 illustrates a router switching between communication channels over a 30 hour time period.

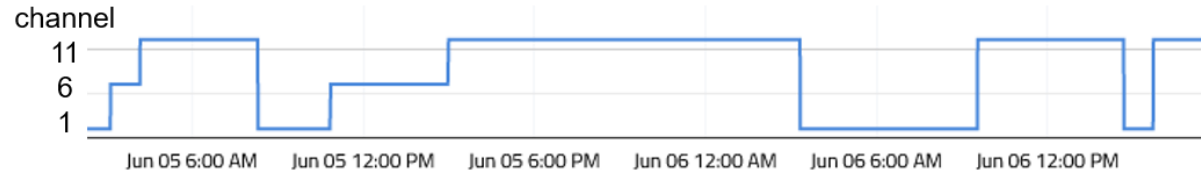

Figure 9-21 *Router switching communication channels*

The sketch (see Listing 9-10) for a battery-powered ESP32 microcontroller, transmitting data with the ESP-NOW protocol to a second ESP32 microcontroller, is based on Listing 9-8. The *WiFi* and *esp-wifi* libraries are required to scan available networks and to both define the communication channel and manage deep sleep mode of an ESP32 microcontroller. The sketch scans available networks to determine the router communication channel, as used by the second ESP32 microcontroller, which is defined as the ESP-NOW receiver, with MAC (Media Access Control) address defined in the *receiveMAC* array. The MAC address of an ESP32 microcontroller is obtained with the *WiFi* library `WiFi.macAddress()` instruction and is also displayed by the Arduino IDE when a sketch is compiled and loaded. The ESP32 microcontroller MAC address in Listing 9-10 must be replaced by the MAC address of your ESP32 microcontroller. The `esp_now_peer_info_t receiver = {}` instruction is required to prevent the "*E (66) ESPNOW: Peer interface is invalid*" error message.

The router SSID is compared to available network SSIDs with the C++ *strcmp* function. The function compares two strings character by character, and returns the index of the first character that differs, with a zero return indicating that strings are identical. Alternatively, the router SSID and a network SSID are compared with the `if(SSIDstr == WiFi.SSID(i).c_str())` instruction, with the router SSID, *SSIDstr*, defined as a string rather than a character array.

In Listing 9-10, the instructions for ESP-NOW communication replace the instructions in Listing 9-8 to connect with and transmit data to an MQTT broker. The *taskFunction* is called to obtain the SCT013 current transformer data, the battery voltage and router communication channel, which is contained in a structure, *payload*, for transmission. Following data transmission, there is a 500ms delay for receipt of the transmission callback, before the ESP32 microcontroller is moved to deep sleep mode with the instructions:

`esp_sleep_enable_timer_wakeup(micro)` and `esp_deep_sleep_start()`.

Listing 9-10 *ESP-NOW transmitting ESP32*

```
#include <WiFi.h>                                    // include ESP-NOW and Wi-Fi
#include <esp_wifi.h>                                //  libraries
#include <esp_now.h>                                 // receiving ESP32 MAC address
uint8_t receiveMAC[] = {0x94, 0xB9, 0x7E, 0xD2, 0x20, 0xEC};
char ssid[] = "XXXX";                                // replace with your router SSID
typedef struct                                       // structure for data
{
  int mV;                                            //  SCT013 voltage
  float battery;                                     //  battery voltage
  int channelPL;                                     //  router Wi-Fi channel
  int countPL;                                       //  data counter
  int rep;                                           // repeated transmissions
} dataStruct;
dataStruct payload;
int battPin = 34, enabPin = 14, SCTpin = 37;         // battery pin when USB powered
int chk, scan, channel = 0;
int Nread = 500, volt, minVolt, maxVolt;             // number of current readings
RTC_DATA_ATTR int count = 0;                         // store count in RTC memory
unsigned long micro = 60E6;                          // shorthand 60×106 for 60secs

void setup()
{
  pinMode(enabPin, OUTPUT);
  digitalWrite(enabPin, HIGH);                       // pin set HIGH to active ADC
  WiFi.mode(WIFI_STA);                               // ESP32 in station mode
  scan = WiFi.scanNetworks();                        // number of found Wi-Fi devices
  for (int i=0; i<scan; i++)
  {
    if(!strcmp(ssid, WiFi.SSID(i).c_str()))          // compare to router SSID
    {
      channel = WiFi.channel(i);                     // router Wi-Fi channel
      i = scan;                                      // exit the "for" loop
    }
  }
  esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);
  esp_now_init();                                    // initialise ESP-NOW
  esp_now_peer_info_t receiver = {};                 // establish ESP-NOW receiver
  memcpy(receiver.peer_addr, receiveMAC, 6);
  receiver.channel = channel;                        // ESP-NOW receiver channel
  receiver.encrypt = false;
  esp_now_add_peer(&receiver);                       // add ESP-NOW receiver
  esp_now_register_send_cb(sendData);                // sending data callback function
  esp_sleep_enable_timer_wakeup(micro);              // restart after fixed time
  taskFunction();                                    // call task function
  esp_now_send(receiveMAC, (uint8_t *) &payload, sizeof(payload));
  delay(500);                                  // interval for callback before deep sleep
  esp_deep_sleep_start();                            // ESP32 in deep sleep mode
```

```
}

void taskFunction()                              // manage data collection
{
  getCurrent();                                  // call function to measure current
  payload.battery = analogReadMilliVolts(battPin)*2.0/1000.0;
  payload.channelPL = channel;                   // router Wi-Fi channel
  payload.countPL = count++;                     // incremented counter
  payload.rep = 0;                               // transmission repeats
}

void getCurrent()                          // function to calculate current usage
{
  maxVolt = 0;                                   // minimum and maximum values
  minVolt = 5000;                                // ignore initial readings
  for (int i=0; i<100; i++) analogReadMilliVolts(SCTpin);
  for (int i=0; i<Nread; i++)
  {
    volt = analogReadMilliVolts(SCTpin);         // SCT013 output voltage
    if(volt > maxVolt) maxVolt = volt;           // update maximum and
    if(volt < minVolt) minVolt = volt;           //   minimum voltages
  }
  payload.mV = maxVolt - minVolt;                // peak to peak mV
}

void sendData(const uint8_t * mac, esp_now_send_status_t chkS)
{                                                // function to count transmissions
  if(chkS != 0)                                  // transmission not received
  {
    payload.rep++;                               // increment transmission number
    esp_now_send(receiveMAC, (uint8_t *) & payload, sizeof(payload));
  }                                              // re-transmit data
}

void loop()                                      // nothing in loop function
{}
```

A TTGO T-Display V1.1 module is powered through the USB connector of an 18650 battery V3 micro USB charging shield (see Figure 9-22). The 18650 Li-Ion battery is directly connected to GPIO 34 and to GND for measurement of the battery voltage.

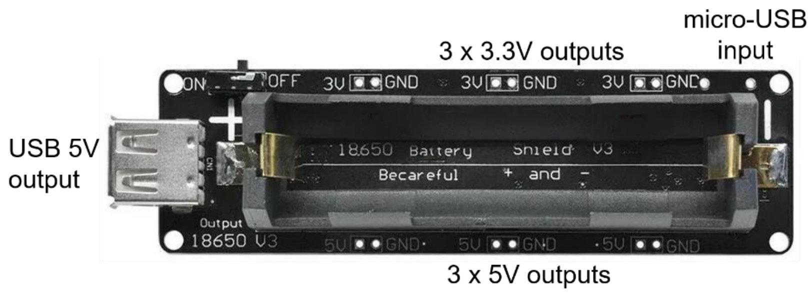

Figure 9-22 *18650 battery V3 micro USB charging shield*

The sketch for an ESP32 microcontroller, which receives data transmitted with the ESP-NOW protocol and then transmits, with Wi-Fi communication, data to the MQTT broker is given in Listing 9-11. Data is received in a structure, defined in Listing 9-10 for the transmitting ESP32 microcontroller, with the processed data then transmitted to the MQTT broker in the *receiveData* function.

Information is displayed on a TTGO T-Display V1.1 module LCD screen by the *display* function, with text colours defined in the *TFT_eSPI.h* file of the *TFT_eSPI* library (see Chapter 15 *Libraries*). For consistency of letter and digit sizes, numbers are converted to strings and displayed with the *drawString* instruction rather than with the *drawNumber* instruction. Labels are displayed once on a TTGO T-Display V1.1 module LCD screen by the *layout* function, rather than re-displaying the labels when data values are updated.

In Wi-Fi station mode, defined by the `WiFi.mode(WIFI_AP_STA)` or `WiFi.mode(WIFI_STA)` instructions, an ESP32 microcontroller periodically enters power saving mode and sets Wi-Fi communication to standby. Consequently, data transmissions are not received by the ESP32 microcontroller, when the ESP32 microcontroller is in power saving mode. Power saving is prevented with the `WiFi.setSleep(WIFI_PS_NONE)` instruction, which increases the ESP32 microcontroller power requirement, but Wi-Fi communication with the ESP32 microcontroller is not interrupted.

The *time* library is required to determine daily power usage, based on the time of data reception. Methods to obtain the current time are described in Chapter 15 *Libraries*. Daily power usage, determined from the SCT013 current transformer data, is constantly incremented until the time, *hhmm*, of data receipt is less than the previous time, *hhmmOld*, indicating the start of a new day. For example, if data is received every five minutes, then reception times of *23:58* and *00:03*, which equate to *hhmmOld* and *hhmm* values of 1438 and 3, respectively, identify the start of a new day.

Listing 9-11 *ESP-NOW receiving ESP32*

```
#include "properties.h"                 // device and dashboard details
#include <TFT_eSPI.h>                   // include TFT_eSPI library
TFT_eSPI tft = TFT_eSPI();              // associate tft with library
#include <WiFi.h>                       // include Wi-Fi and ESP-NOW
#include <esp_now.h>                    //    libraries
typedef struct                          // structure for data
{
  int mV;                               //   SCT013 voltage
  float battery;                        //   battery voltage
  int channelPL;                        //   router Wi-Fi channel
```

```
  int countPL;                                          //   data counter
  int rep;                                              //  repeated transmissions
} dataStruct;
dataStruct payload;
float mA, current, RMS, lag;
unsigned long last = 0;
#include <time.h>                                       // include time library
int GMT = 0, daylight = 3600;                          // GMT and daylight saving offset
int hh, mm, ss, hhmm, hhmmOld;                          //    in seconds
struct tm timeData;

void setup()
{
  WiFi.setSleep(WIFI_PS_NONE);                          // prevent Wi-Fi sleep mode
  WiFi.mode(WIFI_AP_STA);                               // access point and station mode
  WiFi.begin(ssid, password);                           // initialize and connect to Wi-Fi
  while (WiFi.status() != WL_CONNECTED) delay(500);
  esp_now_init();                                       // initialise ESP-NOW
  esp_now_register_recv_cb(receiveData);                // receiving data callback function
  initProperties();
  ArduinoCloud.begin(connHandle);                       // connect to Arduino IoT Cloud
  setDebugMessageLevel(4);                   // status of Arduino IoT Cloud connection
  ArduinoCloud.printDebugInfo();
  configTime(GMT, daylight, "uk.pool.ntp.org");               // NTP pool
  while (!getLocalTime(&timeData)) delay(500);          // wait for connection to NTP
  hhmmOld = 60*timeData.tm_hour + timeData.tm_min;            // set current hhmm value
  layout();                                             // function for LCD display labels
}
                                                        // function to receive data
void receiveData(const uint8_t * mac, const uint8_t * data, int len)
{
  memcpy(&payload, data, sizeof(payload));              // copy data to payload structure
  mA = 0.5*payload.mV/22.0;                             // convert SCT013 voltage
  current = mA*100.0/50.0;                              //    to current
  RMS = current/sqrt(2.0);                              // convert to RMS current
  power = 230.0 * RMS;                                  // convert to power (Watt)
  lag = (millis() - last)/1000.0;                       // interval between receiving
  last = millis();                                      //    ESP-NOW transmissions
  getLocalTime(&timeData);                              // update current time
  hh = timeData.tm_hour;
  mm = timeData.tm_min;
  ss = timeData.tm_sec;
  hhmm = 60*hh + mm;                                    // update hhmm value
  if(hhmm < hhmmOld) kWh = power/(3600.0*1000);         // reset power for new day
  else kWh = kWh + power*lag/(3600.0*1000);             // increment power
  hhmmOld = hhmm;
  display();                                  // function to display data on LCD screen
}

void layout()                                           // function for LCD display labels
{
  tft.init();                                           // initialise LCD screen
  tft.setRotation(3);                                   // landscape with USB on left
```

```
  tft.setTextSize(1);
  tft.fillScreen(TFT_BLACK);                  // colours from TFT_eSPI.h
  tft.setTextColor(TFT_GREEN,TFT_BLACK);
  tft.drawString("power", 0, 0, 4);           // labels for power and kWh
  tft.drawString("kWh", 0, 35, 4);
  tft.setTextColor(TFT_YELLOW,TFT_BLACK);
  tft.drawString("battery", 0, 70, 4);        // labels for battery and count
  tft.drawString("count", 0, 105, 4);
}

void display()                          // function to display data on LCD screen
{
  tft.fillRect(100, 0, 140, 135, TFT_BLACK);
  tft.setTextColor(TFT_GREEN,TFT_BLACK);
  String txt = String(power);                 // convert variable to string
  tft.drawString(txt, 100, 0, 4);             // display current and
  txt = String(kWh);                          //    cumulative power
  tft.drawString(txt, 100, 35, 4);
  tft.setTextColor(TFT_YELLOW,TFT_BLACK);
  txt = String(payload.battery);              // display battery voltage
  tft.drawString(txt, 100, 70, 4);
  txt = String(payload.countPL);              // display data counter
  tft.drawString(txt, 100, 105, 4);
  tft.setTextColor(TFT_RED,TFT_BLACK);
  tft.drawString("time", 190, 0, 4);          // display data reception time
  if(hh > 9) txt = String(hh); else txt = "0"+String(hh);
  tft.drawString(txt, 200, 35, 4);
  if(mm > 9) txt = String(mm); else txt = "0"+String(mm);
  tft.drawString(txt, 200, 70, 4);
  if(ss > 9) txt = String(ss); else txt = "0"+String(ss);
  tft.drawString(txt, 200, 105, 4);
}

void loop()
{
  ArduinoCloud.update();                      // Arduino IoT Cloud update
}
```

Listing 9-12 defines the device, which is an ESP32 microcontroller, associated with the Arduino IoT Cloud dashboard and the *initProperties* function defines the Arduino IoT Cloud dashboard variables. Listing 9-12 is included in the *properties.h* tab.

Listing 9-12 *Arduino IoT Cloud MQTT broker details for smart meter*

```
#include <ArduinoIoTCloud.h>                  // Arduino IoT Cloud libraries
#include <Arduino_ConnectionHandler.h>        // ESP32 device name and key
const char DEVICE_LOGIN_NAME[] = "xxxx";      // change xxxx to Device login
const char DEVICE_KEY[] = "xxxx";             // change xxxx to Device Key
#include <ssid_password.h>                    // file with logon details

float kWh = 0, varBattery;                    // Arduino dashboard variables
int power, countN, channel;

void initProperties()                         // details of device name & key
{                                             //   and dashboard variables
```

```
  ArduinoCloud.setBoardId(DEVICE_LOGIN_NAME);
  ArduinoCloud.setSecretDeviceKey(DEVICE_KEY);
  ArduinoCloud.addProperty(varBattery, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(power, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(countN, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(kWh, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(channel, READ, ON_CHANGE, NULL);
}
                                        // connection to your Wi-Fi router
WiFiConnectionHandler connHandle(ssid, password);
```

Wi-Fi or Wi-Fi and ESP-NOW

The cycle times of two scenarios to transmit energy usage data to the MQTT broker are shown in Table 9-3, with the Arduino IoT Cloud as the MQTT broker. The first scenario consists of a battery-powered ESP32 microcontroller transmitting data, by Wi-Fi communication through a router, to the MQTT broker. The second scenario consists of a battery-powered ESP32 microcontroller transmitting data, with the ESP-NOW protocol, to a non-battery powered ESP32 microcontroller, which forwards the data, by Wi-Fi communication through a router, to the MQTT broker (see Figure 9-20). The main difference in transmission cycle times of the two scenarios was the time required to connect to the Wi-Fi router and to the MQTT broker, which resulted in a fourfold increase in the cycle time.

Table 9-3 *Transmission cycle times*

Task	One ESP32 Wi-Fi only	Two ESP32s ESP-NOW and Wi-Fi
Scan available Wi-Fi or ESP-NOW networks	2079 ms	2823 ms
Connect to Wi-Fi	4566 ms	
Configure device for Arduino IoT Cloud	706 ms	
Connect to Arduino IoT Cloud	3323 ms	
Connect to receiving ESP32 with ESP-NOW		<1 ms
Collect data	52 ms	52 ms
Arduino IoT Cloud updates	2047 ms	
Transmit data		<1 ms
Time to receive callback		500 ms
Total time	12.8 s	3.4 s

The current requirements during the transmission cycle, for a battery-powered ESP32 microcontroller, in the two scenarios are shown in Figure 9-23. The ESP32 microcontroller communicating directly with the MQTT broker is shown in Figure 9-20a and the ESP32 microcontroller transmitting data, with the ESP-NOW protocol, to a second ESP32 microcontroller is shown in Figure 9-20b. In the first scenario, after scanning available Wi-Fi networks and connecting to the required network, a connection is made with the MQTT broker with sensor data updated on the MQTT broker dashboard over a period of 12.8s. In contrast, scanning available ESP-NOW devices and connecting to the required device requires less than 3s. The area under the current graph, measured in milliamp-seconds (mAs), is inversely related to the length of time that a battery, of capacity *N* mAh, can supply power to a transmitting ESP32 microcontroller. The ESP32 microcontroller, which transmits data directly to the MQTT broker, required four times the charge of the ESP32 microcontroller that scanned available networks to determine the router communication channel before transmitting data with the ESP-NOW protocol to a second ESP32 microcontroller. For the scenario with two ESP32 microcontrollers, the ESP32 microcontroller receiving data by ESP-NOW communication and then transmitting data to the MQTT broker required a constant current of 160mA.

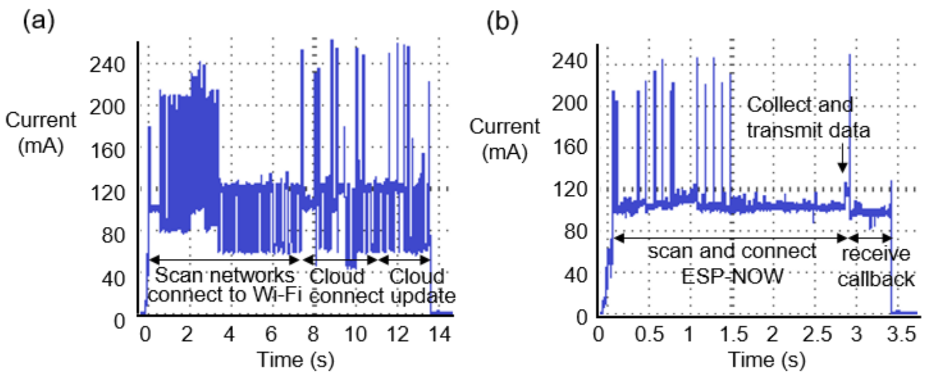

Figure 9-23 *Current requirements with Wi-Fi or with ESP-NOW and Wi-Fi*