Chapter 8: Class Project

Time: 2 ¾ Hours

At the end of this chapter you will have created an IoT weather station. It will:

- Read local weather conditions
- Display weather data on the OLED display
- Post weather data to the Cloud
- Get weather data for other student's IoT devices

8.1 Introduction

8.1.1 Basic

Your project is to build an IoT weather station. It will connect to the class AWS account and will publish weather updates to a *thing* which will be assigned to you by the class leader. You will read the temperature, humidity, and ambient light values from the PSoC on the shield and will update the values on your *thing*'s shadow when you press a button.

8.1.2 Advanced

1. Send your device's IP address to the *thing's* shadow when a connection is established.
2. Toggle a weather alert and send it to the cloud when a button is pressed.
3. Use a timer so that weather data is sent to the cloud periodically (e.g. every 30 seconds) in addition to when a button is pressed.
4. Display the local weather (including the alert state) and device information (*thing* name and IP address) on the OLED display.
5. Get weather information from weather stations assigned to other students in the class. Use the OLED display to display their information. Use CapSense buttons to select either local info or other weather station info (1 page for each thing).
6. Add a Serial Terminal (UART) interface to allow a more featured user interface.
7. Add an introducer so that the device can be configured at initial run time to connect to any WiFi network.

8.2 Details and Hints

8.2.1 Basic

If you are using MQTT, it is probably best to start with the publisher project (see chapter 7D). You will edit the message so that it sends JSON messages to update the shadow instead of just alternately sending LIGHT OFF and LIGHT ON.

> Hint: If you plan to add the web based introducer, you may want to start with the shadow application instead. However, that application is much larger and more complex than the publisher project so you should understand it first if you decide to go that way.

The publisher project already publishes when a button is pressed so you can keep that functionality.

If you are using HTTP, the HTTP Bin example is a good starting point (see chapter 7C). You will use POST requests to send your data to the server.

You will connect to the class AWS IoT endpoint:

amk6m51qrxr2u.iot.us-east-1.amazonaws.com

Your *thing* name will be "ww101_<nn>" where <nn> will be a 2-digit number assigned to you. For example, ww101_01. The *things* have already been setup so you do not need to create it.

If you used the class AWS account for previous exercises, you can use the same certificate and key. If you did not use the class account, you can copy the client certificate (client.cer) and private key (privkey.cer) from the class material folder. They can be found in *WW101 Files/ClassCerts/AWS_Broker_Info*.

> Hint: After copying the files, you should run a "Clean" on the project. Otherwise, the project will not see the new files.

You will need to use I2C to read the weather information from the PSoC. See the I2C exercises in chapter 2. You should read the values on a regular basis (e.g. every 500ms).

> Hint: Don't forget to use __attribute__((packed)) if you have an I2C buffer that isn't all 32-bit values. See the I2C section of the peripherals chapter for details.

Your weather station *thing* has five state variables to keep track of information that you will be sending:

1. "temperature" (float)
2. "humidity" (float)
3. "light" (float)
4. "weatherAlert" (true or false)
5. "IPAddress" (ipv4 4dot syntax)

The starting (empty) shadow for your *thing* will look like the following. You will publish or POST JSON messages to the *thing* shadow to provide updates.

Shadow state:

```
{
  "reported": {
    "temperature": 0,
    "humidity": 0,
    "light": 0,
    "weatherAlert": false,
    "IPAddress": "0.0.0.0"
  }
}
```

You can use the snprintf function to create the JSON messages. Remember that spaces and carriage returns are not required. Also remember that quotation marks in the message must be escaped with a \ character. For example, to create a JSON message to send the temperature from the structure psoc_data.temperature, you could do something like this:

```
char json[100];

snprintf(json, sizeof(json), "{\"state\" : {\"reported\" : {\"temperature\":%.1f} } }",
psoc_data.temperature);
```

Hint: Make sure the array you use to hold the message is large enough. If it isn't you will get very unpredictable results.

Hint: When doing initial testing, use the MQTT Test Client on the AWS site to examine the messages that you are sending. For example, to see all shadow messages for the *thing* named ww101_01, you would subscribe to:

$aws/things/ww101_01/shadow/#

Messages that show up for the topic $aws/things/ww101_00/shadow/update are the messages that you are sending. You will also see messages that tell you whether the broker accepted or rejected your update.

Once you see that the broker is accepting your updates, go to your *thing* and click on the Shadow. You will then see the data that is published by your *thing* in real time.

If you are using MQTT, the publisher application that you start with contains several threads. To maintain modularity and reduce complexity it is HIGHLY RECOMMENED that you add additional functionality in new threads. For example, you may want separate threads to:

1. Get everything up and going in application_start including connecting to WiFi, connecting to the MQTT broker, and subscribing to topics for other things. This thread might exit when it is done.
2. Publish data to the Cloud.
3. Read weather data from the PSoC.
4. Update the OLED display.

5. Monitor CapSense buttons.
6. Perform the UART command interface functions (both input and output).

Remember that interaction between threads is controlled using semaphores, queues, and mutexes.

8.2.2 Advanced

1. IP Address

Once you connect to the AP and to the AWS site, send your device's local IP address (just once) so that it shows up in the shadow. Notice that it is a string in the format "N.N.N.N".

2. Weather Alert

Monitor a button press (not the one used for publishing) to toggle the weather alert status and send it to the cloud. Notice that the weather alert is a Boolean. The JSON message you send must have a value of either *true* or *false* with no quotes around it. For example:

```
snprintf(json, sizeof(json), "{\"state\" : {\"reported\" : {\"weatherAlert\":true} } }");
snprintf(json, sizeof(json), "{\"state\" : {\"reported\" : {\"weatherAlert\":false} } }");
```

3. Timer

Add a timer so that weather data is sent to the cloud every 30 seconds in addition to whenever you press the button. Note that the periodic update may happen while a button press update is going on or vice versa so you will want to use a queue.

4. Display

The OLED display for your *thing* should look something like this:

```
ww101_00 *ALERT*
198.51.100.149
Temp:     25.5
Humidity: 50.5
Light:    250
```

The "*ALERT*" after the *thing* name is used to indicate an active weather alert. It will be displayed only if the weather alert value is true.

You should only update the display if one of the values has changed. Hint: use a semaphore or a queue.

Hint: When you add the OLED display functionality, you may need a MUTEX around the I2C transactions to prevent conflicts between the PSoC analog coprocessor and display.

Hint: Allow a maximum of 12 pixels in height when writing each row of data to the OLED. This allows you to fit 5 lines of data on the screen.

5. Get Weather Data from the Cloud

If you are using MQTT, use the subscriber project as a reference. Some functions are common between the publisher and subscriber so you will not need to duplicate those.

It is easiest to just maintain a list of all the *things* that have been assigned for the class (i.e. ww101_01, ww101_02, etc.)

> Hint: there is a library of linked list functions in *utilities/linked_list* that you can use to maintain a local database of *thing* data.

Use CapSense buttons to display weather data for the other *things*. For example, use CapSense button 0 to display the local weather station's data, CapSense button 1 to go to the page for the previous *thing*, CapSense button 2 to go to the page for the next *thing*, and CapSense button 3 to increment the display 10 *things* at a time.

> Hint: Since the CapSense values are read from the PSoC using I2C, remember to use a MUTEX to prevent conflicts between different threads that use the I2C resource.

6. Serial Terminal

Add a serial terminal to implement a more complex user interface (see UART exercises in the peripherals chapter). For example, you could implement the following:

t – Print temperature and send to the cloud
h – Print humidity and send to the cloud
l - Print ambient light value and send to the cloud
A – Turn ON Weather Alert and send to the cloud
a – Turn OFF Weather Alert and send to the cloud
P – Turn ON printing of update messages from all things
p – Turn OFF printing of update messages from all things
x – Print the current known state of the data from all *things*
c – Clear the terminal and move the cursor to the upper left corner
? – Print the list of commands

Hint: Use VT100 escape codes to make a pretty screen:
http://ascii-table.com/ansi-escape-sequences-vt-100.php

7. Introducer

The shadow example exercise in chapter 7D shows an example of how to use a soft AP to serve a web page from the WICED device to use for device a configuration. Once configured, the device resets and connects to the specified device as an STA. The configuration data is written to the DCT so that on subsequent power cycles it remembers which AP to connect to.