Chapter 8: Class Project

Objective

Time: 2 ¾ Hours

Fundamentals

Exercise(s)

Introduction

Basic

Your project is to build an IoT weather station. It will connect to the class AWS broker 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 analog coprocessor and publish values to your *thing*'s shadow when you press a button.

Advanced

Publish your device's IP address when a connection is established.

Toggle a weather alert and publish it when a button is pressed.

Use a timer so that weather data is published periodically (e.g. every 30 seconds) in addition to when a button is pressed.

Display the local weather (including alert state) and device information (thing name and IP address) on the OLED display.

Subscribe to the shadow topic for weather stations from other students in the class. Use the OLED display to display their information. Use CapSense buttons to scroll through local info or other weather station info (1 page for each thing that you are subscribed to).

Add a UART interface to allow a more featured user interface.

Add an introducer so that the device can be configured at initial run time to connect to any WiFi network.

Details and Hints

Basic

It is probably best to start with the publisher project (see chapter 7b). 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 complex than the publisher project so you should understand it first if you decide to go that way.

You will connect to the class MQTT broker:

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 broker for previous exercises, you can use the same certificate and key. If you did not use the class broker, 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/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 analog coprocessor. 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* will have five state variables to keep track of information that you will publish:

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 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 sprintf 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];

sprintf(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 Test interface on the AWS site to examine the messages that you are sending. For example, to see all shadow messages for the *thing* named ww101_00, you would subscribe to:

$aws/things/ww101_00/shadow/#

Messages that show up to the topic $aws/things/ww101_00/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 Shadow. You will then see the data that is published for your *thing* in real time.

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. Publish data to the Cloud
2. Read weather data from the PSoC
3. Update the OLED display
4. Monitor CapSense buttons
5. Perform the UART command interface functions (both input and output)
6. Subscribe to other *thing* shadows from the Cloud

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

Advanced

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 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.

Subscriptions

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* that you are subscribed to. For example, use CapSense button 0 to display the local weather station's data and CapSense button 1 to scroll through the data from the other *things*.

> 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.

Serial Terminal

Add a serial terminal interface 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 publish
h – Print humidity and publish
l - Print ambient light value and publish
A – Publish weather alert ON
a – Publish weather alert OFF
S – Turn subscriptions for other things ON
s – Turn subscriptions for other things OFF
l – Scan for all *things* with valid weather data (i.e. non-zero values) and print the list
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

Introducer

The shadow example exercise in chapter 7b 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 a STA. The configuration data is written to the DCT so that on subsequent power cycles it remembers which AP to connect to.