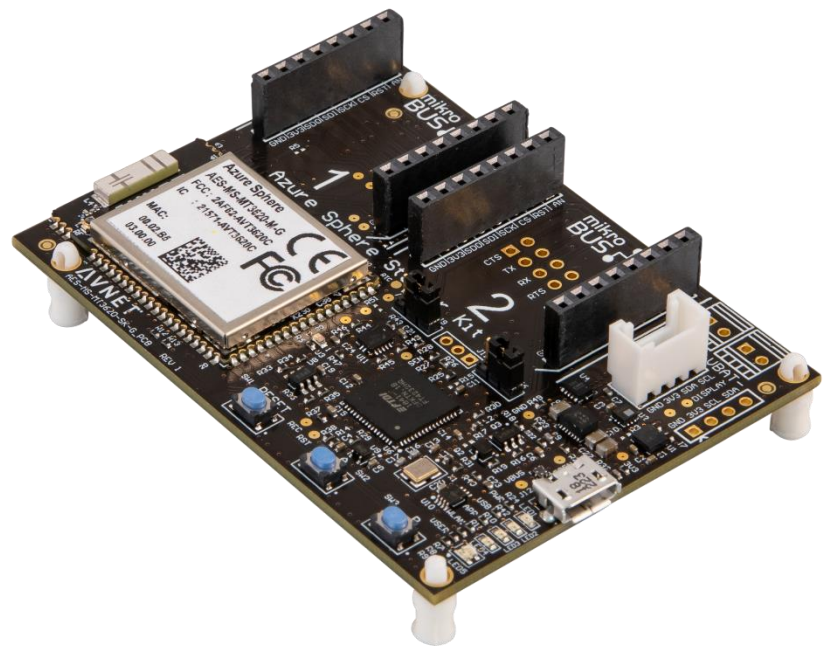


Avnet Technical Training Course

Azure Sphere: Digging Deeper Into Application Code Lab 3



Azure Sphere SDK:	19.11
Training Version:	v4
Date:	24 December 2019

Introduction

This Lab will document the pieces of code that implements a few basic IoT concepts. We'll review how to read a GPIO pin, how to send IoT telemetry to Azure, and how to manage device twin messages. There are many different ways to code these concepts, this lab will document how the example project implements the features. After reviewing the source code, we will follow up with a section on the application manifest file, and I'll touch on the hardware abstraction layer.

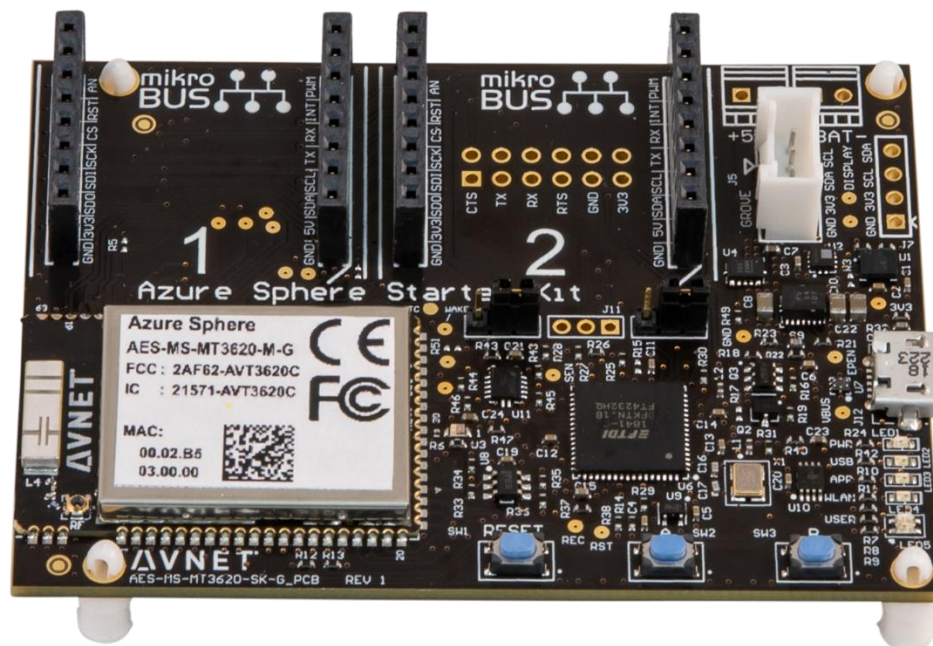
The telemetry and device twin code is currently not enabled by Lab-2 (the non-connected build configuration), but Lab-4 and Lab-5 will use these implementations and I'll assign a couple of code assignments related to the concepts described in this lab.

Avnet Azure Sphere Starter Kit Overview

The Avnet Azure Sphere Starter Kit from Avnet Electronics Marketing provides engineers with a complete system for prototyping and evaluating systems based on the MT3620 Azure Sphere device.

The Avnet Azure Sphere MT3620 Starter Kit supports rapid prototyping of highly secure, end-to-end IoT implementations using Microsoft's Azure Sphere. The small form-factor carrier board includes a production-ready MT3620 Sphere module with Wi-Fi connectivity, along with multiple expansion interfaces for easy integration of off-the-shelf sensors, displays, motors, relays, and more.

The Starter Kit includes Avnet's MT3620 Module. Having the module on the Starter Kit means that you can do all your development work for your IoT project on the Starter Kit and then easily migrate your Azure Sphere Application to your custom hardware design using Avnet's MT3620 Module.



Avnet Azure Sphere Starter Kit

Lab 3: Objectives

The objectives of Lab-3 are to dig into the sample project source code and understand how some basic IoT features are implemented.

- Understand how to configure and read a button press using a GPIO hardware signal
- Understand how to send IoT telemetry to Azure
- Understand how to process device twin messages from Azure
- Learn about the app_manifest.json file that's included in every Azure Sphere project
- Learn about the hardware abstraction layer

Lab-3 must be started after Labs 0-2 have been completed.

Requirements

Hardware

- A PC running Windows 10 Anniversary Update or later (Version 1607 or greater)
- An unused USB port on the PC
- An Avnet Azure Sphere Starter Kit
- A micro USB cable to connect the Starter Kit to your PC

Software

- Visual Studio 2019 Enterprise, Professional, or Community version 16.04 or later; or Visual Studio 2017 version 15.9 or later **installed**
- Azure Sphere SDK 19.11 or the current SDK release **installed**

GPIO

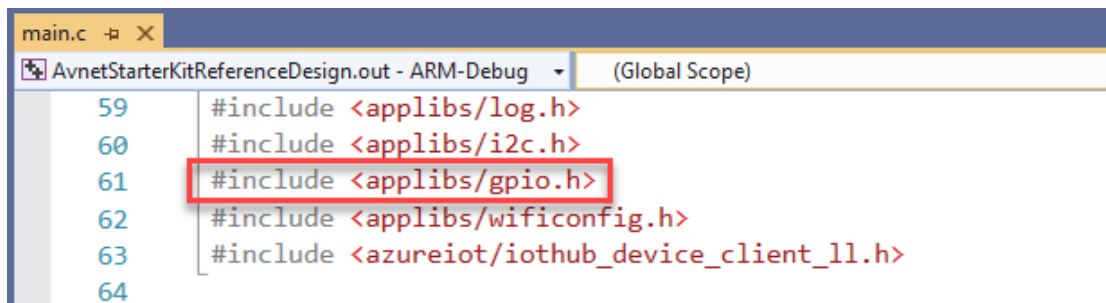
Working with hardware interfaces is common with IoT projects. Whether it's driving an LED, or reading an I2C sensor the Azure Sphere OS and OS services have the APIs to help you work with your hardware devices. In this section we'll identify the code required to read the General Purpose I/O (GPIO) signal in our Azure Sphere project that's connected to User Button A. Working with GPIO interfaces is pretty simple, we need to do 5 things . . .

1. Include the GPIO libraries in the project
2. Add the GPIO reference to the app_manifest.json file
3. Declare a file descriptor that we'll associate to our GPIO signal
4. Open the GPIO as an input and assign a file descriptor to work with the hardware
5. Read the GPIO level using the file descriptor as a reference to the hardware
6. Close the file descriptor

Let's identify the source code for each of these items . . .

Include the GPIO libraries in the project

To add the GPIO libraries to our project just include gpio.h header file



The screenshot shows a code editor window with a tab labeled 'main.c'. The editor displays the following code:

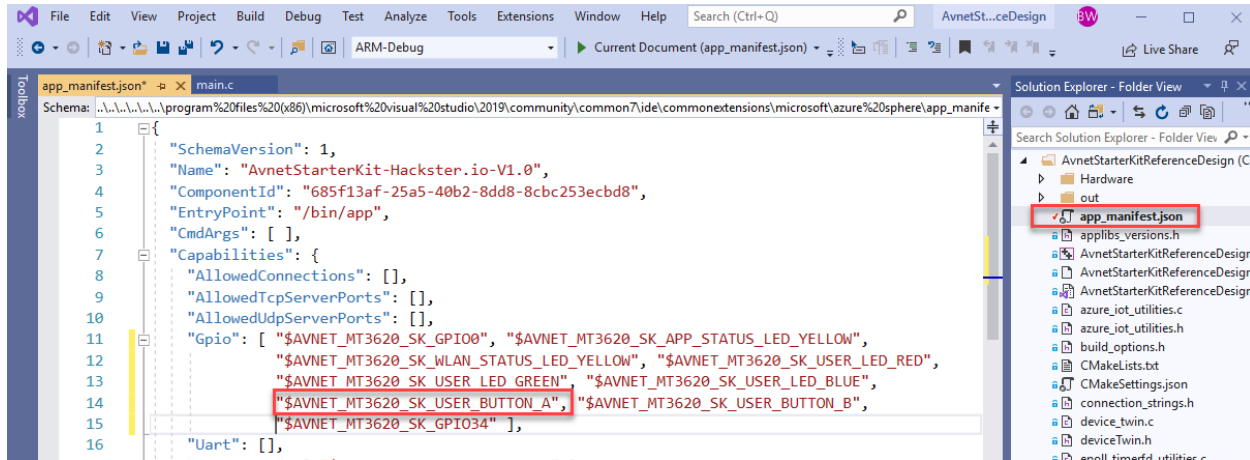
```
59 #include <applibs/log.h>
60 #include <applibs/i2c.h>
61 #include <applibs/gpio.h>
62 #include <applibs/wificonfig.h>
63 #include <azureiot/iothub_device_client_ll.h>
64
```

The line `#include <applibs/gpio.h>` is highlighted with a red rectangular box. The editor's status bar at the bottom indicates '(Global Scope)'.

Add the GPIO reference to the app_manifest.json file

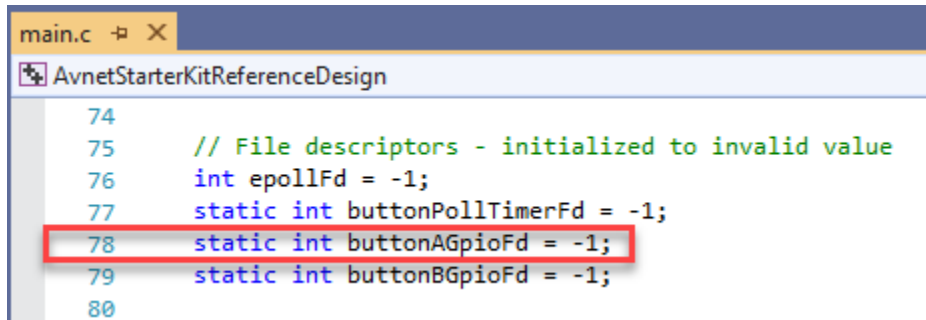
We need to explicitly grant the application permission for the application to use GPIO signal #12. We add the "\$AVNET_MT3620_SK_USER_BUTTON_A" to the app_manifest.json file in the Capabilities → Gpio section. At build time the Hardware Abstraction layer implementation will map "\$AVNET_MT3620_SK_USER_BUTTON_A" to the number 12. If we were to omit this step, and the application attempted to open GPIO 12, then the OS would not allow the application to open the GPIO when we attempt to open it as an input.

The app_manifest.json file is described on page 12 in this document. The Microsoft documentation on the app_manifest.json file can be reviewed [here](#).



Declare a file descriptor

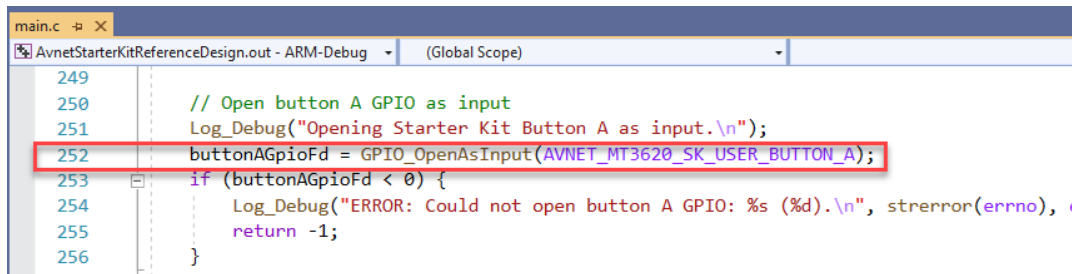
The system uses file descriptors to reference and operate on hardware interfaces. We declare a file descriptor `buttonAGpioFd` to allow the code to operate on the GPIO interface. You'll notice that all calls to initialize and read the GPIO pin use the file descriptor.



```
main.c  X
AvnetStarterKitReferenceDesign
74
75 // File descriptors - initialized to invalid value
76 int epollFd = -1;
77 static int buttonPollTimerFd = -1;
78 static int buttonAGpioFd = -1;
79 static int buttonBGpioFd = -1;
80
```

Open the GPIO as an input

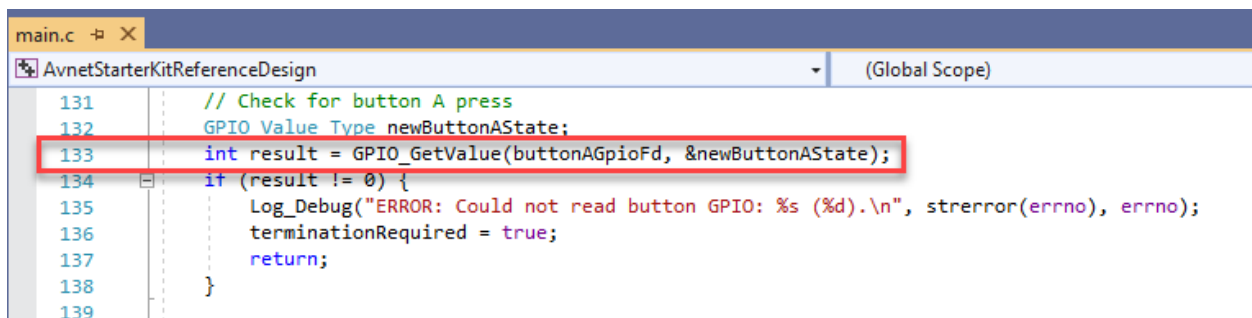
This line of code opens GPIO 12 (`AVNET_MT3620_SK_USER_BUTTON_A`) as an input. That will allow us to read the GPIO level.



```
main.c  X
AvnetStarterKitReferenceDesign.out - ARM-Debug (Global Scope)
249
250 // Open button A GPIO as input
251 Log_Debug("Opening Starter Kit Button A as input.\n");
252 buttonAGpioFd = GPIO_OpenAsInput(AVNET_MT3620_SK_USER_BUTTON_A);
253 if (buttonAGpioFd < 0) {
254     Log_Debug("ERROR: Could not open button A GPIO: %s (%d).\n", strerror(errno),
255     return -1;
256 }
```

Read the GPIO level

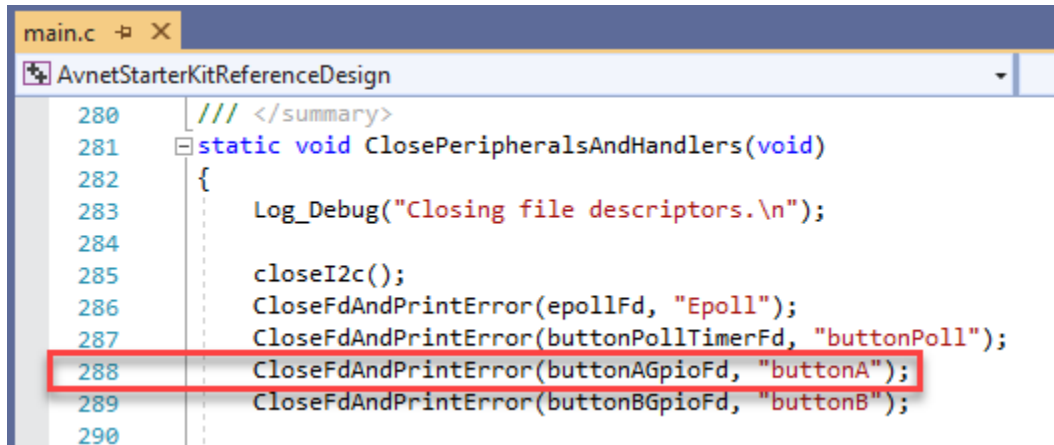
This is the code that actually reads the GPIO level.



```
main.c  X
AvnetStarterKitReferenceDesign (Global Scope)
131 // Check for button A press
132 GPIO_Value_Type newButtonAState;
133 int result = GPIO_GetValue(buttonAGpioFd, &newButtonAState);
134 if (result != 0) {
135     Log_Debug("ERROR: Could not read button GPIO: %s (%d).\n", strerror(errno), errno);
136     terminationRequired = true;
137     return;
138 }
139
```

Close the file descriptor

When the application exits, it calls the routine to clean up. We call the routine that will close the file descriptor.



```
main.c  X
AvnetStarterKitReferenceDesign
280  ///
```

That's all there is to reading a GPIO signal. I welcome you to review the source code to see the logic when the GPIO (newButtonAState) is read.

The easiest way I've found to look at an existing Azure Sphere project to understand how a hardware interface is implemented is to look at the file descriptor. If you can find all the code that uses some piece of hardware's file descriptor, you'll see all the necessary code for that interface.

Assignment: In Visual Studio trace the following file descriptors to see how each is used

- i2cFd
- epollFd;

Send IoT Telemetry to Azure

Sending telemetry is a basic function for any IoT project. Typically, IoT devices collect data, they may or may not pre-process the data, and then they send data in the form of telemetry to the cloud. Our project reads the on-board I2C sensors and sends that data to Azure. Sending telemetry data to Azure is pretty easy. There are 3 things that need to happen . . .

1. Establish and maintain a secure connection to an Azure IoT Hub or IoT Central application
2. Construct a JSON object that contains one or more {"key": value} pairs
3. Call the routine to send the data to our IoT Hub or IoT Central application

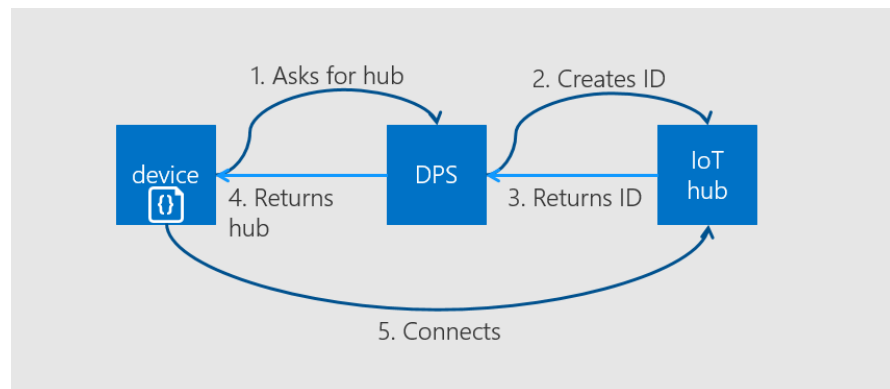
Establish a connection to Azure

I'm not really going to dive into this step as it could consume an entire course all by itself. I'll provide a brief description instead, here's the 50,000' overview.

The process to connect to an Azure IoT Hub or an IoT Central application is pretty much the same.

1. Provision the device

- a. The recommended method to provision IoT devices in Azure is to use an Azure service called a Device Provisioning Service (DPS). You can read all about DPS [here](#). Using DPS you can deploy a single software application build onto millions of devices. The first time each device connects to the internet, and then the global DPS server, they will all automatically be provisioned to one or more IoT Hubs and then connect to the IoT Hub that each device was provisioned to. This is a powerful IoT concept and is required to deploy IoT devices at scale. The diagram below shows the process.



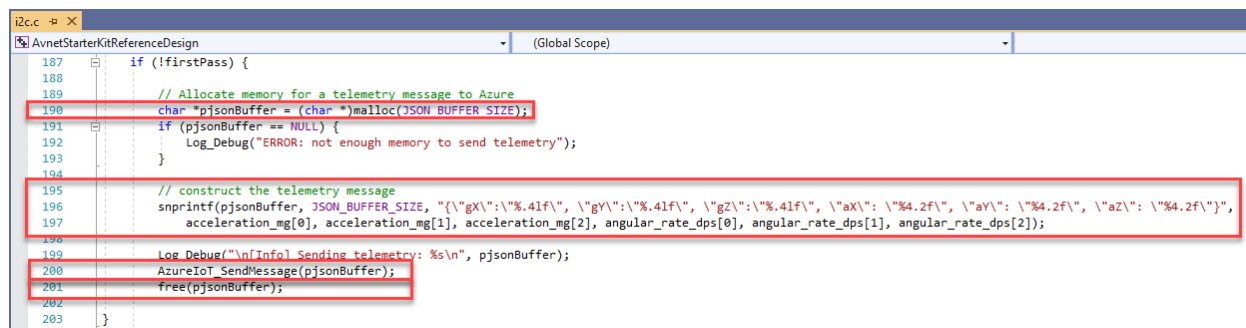
2. Establish and Maintain a secure connection

- a. There are Sphere OS services that establish and maintain a secure (TLS) connection to Azure. If you're really interested in the details search the example project for `iothubClientHandle`.

Construct a JSON object, and send the telemetry

Below you'll see the code to create the JSON telemetry message. There are basically four things to do . .

1. Allocate a buffer in memory to store the JSON object (see line 190)
2. Construct the JSON object in the new memory buffer (see line 196-197)
3. Send the JSON object to Azure (see line 200)
4. Free the memory (see line 201)



```
i2c.c
AvnetStarterKitReferenceDesign (Global Scope)
187 if (!firstPass) {
188
189 // Allocate memory for a telemetry message to Azure
190 char *pjsonBuffer = (char *)malloc(JSON_BUFFER_SIZE);
191 if (pjsonBuffer == NULL) {
192     Log_Debug("ERROR: not enough memory to send telemetry");
193 }
194
195 // construct the telemetry message
196 sprintf(pjsonBuffer, JSON_BUFFER_SIZE, "{\"gX\":%.4f\", \"gY\":%.4f\", \"gZ\":%.4f\", \"aX\": %.4.2f\", \"aY\": %.4.2f\", \"aZ\": %.4.2f\"}\",
197     acceleration_mg[0], acceleration_mg[1], acceleration_mg[2], angular_rate_dps[0], angular_rate_dps[1], angular_rate_dps[2]);
198
199 Log_Debug("\n\nInfo! Sending telemetry: %s\n", pjsonBuffer);
200 AzureIoT_SendMessage(pjsonBuffer);
201 free(pjsonBuffer);
202
203 }
```

One thing that I think is really cool about telemetry is that you can send any {"key": value} pair, or any valid JSON object, you want to Azure. You don't have to tell Azure anything about your data. As long as the data is valid JSON the IoT Hub will accept the data and store it for you to use. Of course if you want to access that data, some other Azure thing will need to know about your data so it can ingest it and do something meaningful with it, but the IoT Hub does not care as long as it's valid JSON.

Process Device Twin Messages from Azure

Device twins are another powerful IoT concept. You can read the Azure documentation on device twins [here](#).

"*Device twins* are JSON documents that store device state information including metadata, configurations, and conditions. Azure IoT Hub maintains a device twin for each device that you connect to IoT Hub."

Using device twins you can make changes in the cloud to a device twin's desired property and the IoT device will receive a message containing the new desired property. Your application then uses the information in the desired property to do something in your application. For example, toggle a GPIO signal to control an LED, change a property that defines how your application behaves, or anything that your creative mind can think up.

The Lab-3 lecture walks the student through the Device Twin Implementation in our example code. For the Lab, I'll show a different Device Twin implementation that is more straightforward than the table driven example project's implementation. The code below was taken from the Microsoft Azure Sphere GitHub Sample called AzureIoT. You can find the project [here](#).

To work with Device Twins we need four different things . . .

1. Define the JSON {"key": value} pair that we want to implement for our solution
2. Setup a callback routine that will be instantiated when the Azure IoT Hub sends a Device Twin update

3. Implement the callback routine, then add code to look for and do something with our specific {"key": value} pair data
4. Send a Device Twin reported properties message back to Azure with the new reported value of our property.

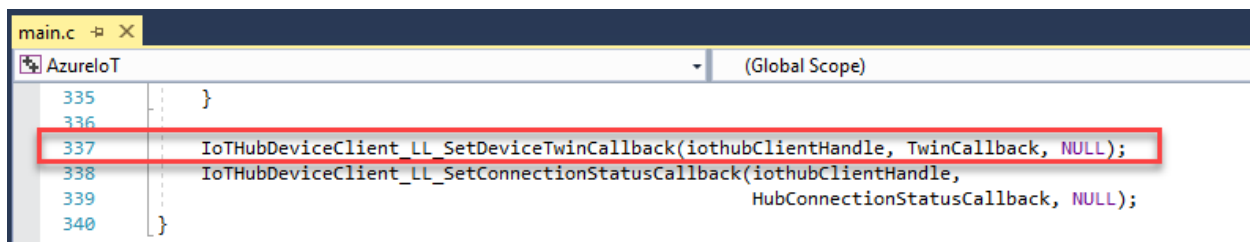
Define the JSON {"key": value}

The first thing we need to do is define our {"key": value} pair. For the AzureIoT example they implement a Device Twin called `statusLED`, the JSON is shown below, it's a Boolean entry.

```
{"statusLED": (true | false)}
```

Setup a Device Twin callback routine

Before we can receive a Device Twin update, we need to tell the Azure OS services how to inform the application when a new Device Twin message is received. In `main.c` on line #337 the application informs the OS services that the routine `TwinCallback`, will be used to process incoming Device Twin messages.



```
main.c -P X
AzureIoT (Global Scope)
335 }
336
337 IoTHubDeviceClient_LL_SetDeviceTwinCallback(iothubClientHandle, TwinCallback, NULL);
338 IoTHubDeviceClient_LL_SetConnectionStatusCallback(iothubClientHandle,
339 HubConnectionStatusCallback, NULL);
340 }
```

Implement the callback routine

The callback implementation is shown below. There are basically four sections identified in the graphic.

(1) Allocate memory to construct a null terminated desired properties JSON object

This section of code (lines 351 – 356) simply allocates memory that will be used to store the desired properties JSON object.

(2) Pull the desired properties out of the incoming payload

The code (lines 358 – 374) builds out a null terminated desired properties JSON object in the allocated buffer and then sets up a pointer to the `desiredProperties` JSON object.

(3) Process the `statusLED` device twin

Lines 376 – 383 handle the device twin property. Coming into this code we have a null terminated JSON object called `desiredProperties`.

- Line 377 we pull out the `statusLED` object, if it exists.
- Line 378 we check to see we found the `statusLED` object
- Line 379 pulls the value piece of our {"key": value} pair and stores it into our variable `statusLedOn`
- Line 380 – 381, changes the GPIO signal for the LED based on the new value
- Line 382 calls the routine to report our reported property to Azure

(4) Cleanup

The last section of code lines 386 – 388, cleans up the `rootProperties` pointer and frees the buffer created at the top of the routine.

```

main.c  [X]
AzureIoT  (Global Scope)

347  ///

```

1

2

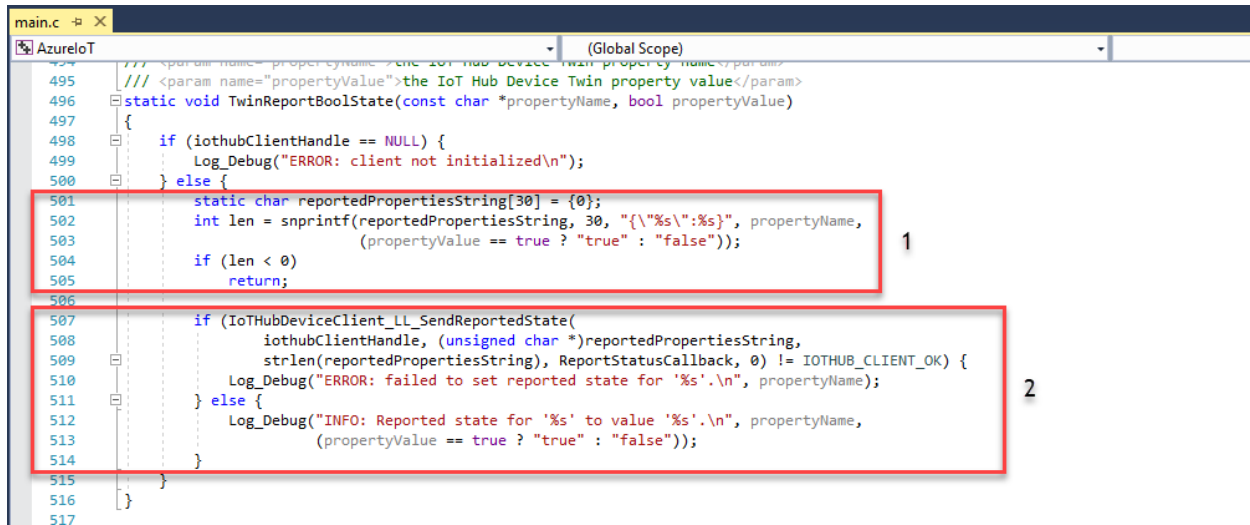
3

4

Send a Device Twin reported properties message back to Azure

The last thing we need to do is to send Azure a message with our new {"key": value} reported property. This implementation created a routine called `TwinReportBoolState()` that does this work for any boolean key: value pair.

- Lines 501 – 505 create the JSON {"key": value} pair string and use the passed in `propertyName` (key) and the bool `propertyValue` (value). The code uses the `snprintf()` routine to construct the object.
- Lines 507 – 514 send the object to Azure and checks to make sure the message was accepted by the `IoHubDeviceClient_LL_SendReportedState()` routine.



```
main.c - X
AzureIoT
(Global Scope)
495  /// <param name="propertyName">the IoT Hub Device Twin property name</param>
496  /// <param name="propertyValue">the IoT Hub Device Twin property value</param>
497  static void TwinReportBoolState(const char *propertyName, bool propertyValue)
498  {
499      if (iothubClientHandle == NULL) {
500          Log_Debug("ERROR: client not initialized\n");
501      } else {
502          static char reportedPropertiesString[30] = {0};
503          int len = snprintf(reportedPropertiesString, 30, "{\"%s\":%s}", propertyName,
504                          (propertyValue == true ? "true" : "false"));
505          if (len < 0)
506              return;
507          if (IoHubDeviceClient_LL_SendReportedState(
508              iothubClientHandle, (unsigned char *)reportedPropertiesString,
509              strlen(reportedPropertiesString), ReportStatusCallback, 0) != IOTHUB_CLIENT_OK) {
510              Log_Debug("ERROR: failed to set reported state for '%s'.\n", propertyName);
511          } else {
512              Log_Debug("INFO: Reported state for '%s' to value '%s'.\n", propertyName,
513                      (propertyValue == true ? "true" : "false"));
514          }
515      }
516  }
517
```

This section reviewed the device twin implementation from the AzureIoT example project on Microsoft's GitHub Azure Sphere project. This implementation is different from the example application we've been working with. I wanted to share both methods for variety.

app_manifest.json

The Microsoft documentation on this Azure Sphere feature is very well written, and I don't think I can add anything to this very important discussion. The documentation is [here](#), it's a short document that should be reviewed by every Azure Sphere developer. **The text below is taken from the Microsoft documentation.**

"Every Azure Sphere application must have an `app_manifest.json` file. The application manifest describes the resources, also called application capabilities, which an application requires when it executes.

Applications must opt-in to use capabilities by listing each required resource in the Capabilities section of the application manifest; no capabilities are enabled by default. If an application requests a capability that is not listed, the request fails. If the application manifest file contains errors, sideloading the application fails.

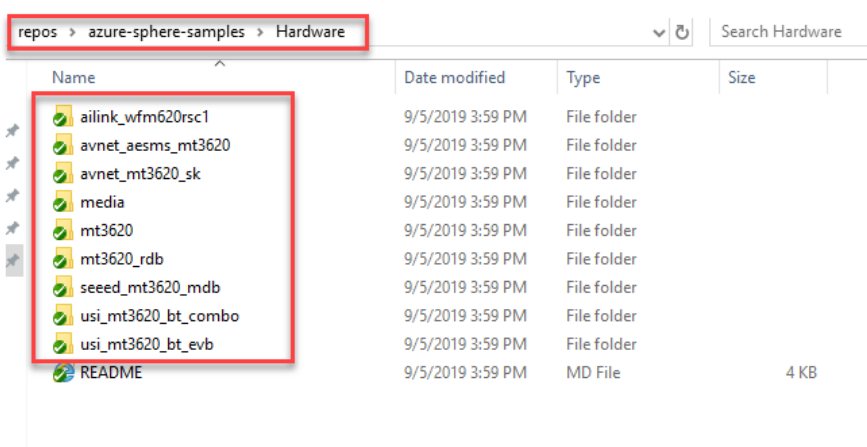
Each application's manifest must be stored as `app_manifest.json` in the root directory of the application folder on your PC. The Azure Sphere templates automatically create a default application manifest when you create an application based on a template. You must edit the default manifest to list the capabilities that your application requires. When the application is sideloaded or deployed to the device, the Azure Sphere runtime reads the application manifest to ascertain which capabilities the application is allowed to use. Attempts to access resources that are not listed in the manifest will result in API errors such as `EPERM` (permission denied)."

Hardware Abstraction Layer

Microsoft has implemented a hardware abstraction layer so that the GitHub samples can run on multiple hardware platforms. All the Microsoft Azure Sphere GitHub examples use this abstraction layer. This is a great feature because there are now multiple Azure Sphere platforms (8 as of this writing) and each of these platforms may expose features of the underlying Azure Sphere device in different ways.

For example the Seeed development kits and the Avnet Starter Kit all use pushbuttons and user LEDs. These devices use GPIO signals. Microsoft needed an implementation so that a single sample application could correctly reference the GPIO signals for the hardware platform that the application was built for, even if different hardware platforms used different signals for a common function like driving a LED.

To implement the abstraction layer, they have included a “Hardware” folder in the azure-sphere-samples directory structure. Under the “Hardware” folder, there are folders for each of the 8 currently supported platforms.



Name	Date modified	Type	Size
ailink_wfm620rsc1	9/5/2019 3:59 PM	File folder	
avnet_aesms_mt3620	9/5/2019 3:59 PM	File folder	
avnet_mt3620_sk	9/5/2019 3:59 PM	File folder	
media	9/5/2019 3:59 PM	File folder	
mt3620	9/5/2019 3:59 PM	File folder	
mt3620_rdb	9/5/2019 3:59 PM	File folder	
seeed_mt3620_mdb	9/5/2019 3:59 PM	File folder	
usi_mt3620_bt_combo	9/5/2019 3:59 PM	File folder	
usi_mt3620_bt_evb	9/5/2019 3:59 PM	File folder	
README	9/5/2019 3:59 PM	MD File	4 KB

Each folder contains a JSON file and a header file that maps the board-specific, or module-specific features to the underlying Azure Sphere MCU. The example projects use identifiers when referring to hardware signals which are mapped to the specific peripherals on the target hardware platform.

For example the graphic below was taken from the AzureIoT example project. Each of these identifiers will resolve to an integer that represents the GPIO signal that the sample buttons and sample LED are physically wired to on the hardware platform. Pretty cool!



```
1  {
2    "SchemaVersion": 1,
3    "Name": "AzureIoT",
4    "ComponentId": "819255ff-8640-41fd-aea7-f85d34c491d5",
5    "EntryPoint": "/bin/app",
6    "CmdArgs": [],
7    "Capabilities": {
8      "AllowedConnections": [ "global.azure-devices-provisioning.net" ],
9      "Gpio": [ "$SAMPLE_BUTTON_1", "$SAMPLE_BUTTON_2", "$SAMPLE_LED" ],
10     "DeviceAuthentication": "00000000-0000-0000-0000-000000000000"
11   },
12   "ApplicationType": "Default"
13 }
```

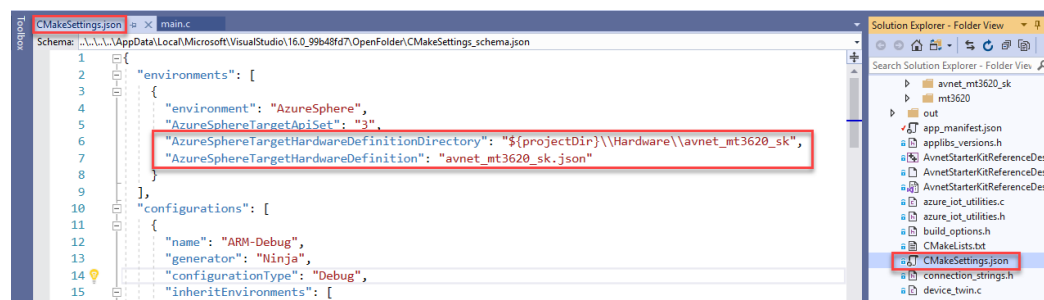
Here’s a code snippet where the AzureIoT example opens the GPIOs for the buttons. Note that we don’t pass in the GPIO integer, but an identifier that references the actual GPIO signal, an integer. In this way

different hardware platforms can use different GPIO signals for the buttons, but the sample application will work for all hardware platforms as long as the hardware definitions are correctly defined and the correct hardware platform is defined for the build.

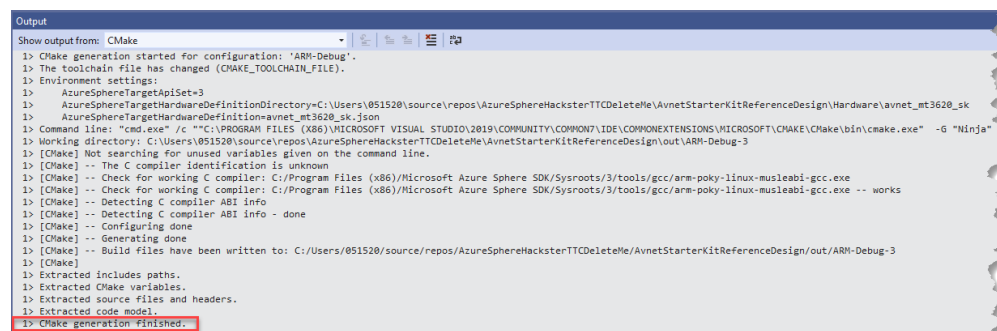
```
main.c | app_manifest.json
(AzureIoT) | (Global Scope)

208 | }
209 |
210 | // Open button A GPIO as input
211 | Log_Debug("Opening SAMPLE_BUTTON_1 as input\n");
212 | sendMessageButtonGpioFd = GPIO_OpenAsInput(SAMPLE_BUTTON_1);
213 | if (sendMessageButtonGpioFd < 0) {
214 |     Log_Debug("ERROR: Could not open button A: %s (%d).\n", strerror(errno), errno);
215 |     return -1;
216 | }
217 |
218 | // Open button B GPIO as input
219 | Log_Debug("Opening SAMPLE_BUTTON_2 as input\n");
220 | sendOrientationButtonGpioFd = GPIO_OpenAsInput(SAMPLE_BUTTON_2);
221 | if (sendOrientationButtonGpioFd < 0) {
222 |     Log_Debug("ERROR: Could not open button B: %s (%d).\n", strerror(errno), errno);
223 |     return -1;
224 | }
225 | }
```

So how do we tell Visual Studio which hardware platform we're using? From Visual Studio we open the CMakeSettings.json file and change the "AzureSphereTargetHardwareDefinitionDirectory" and "AzureSphereTargetHardwareDefinition" properties to reference your hardware directory and the target *.json file. The graphic below is from our example project.



When you modify and save this file, the CMAKE subsystem will automatically re-generate the configuration. If there is a problem with the path or filename that you added, you'll see errors. A successful process will look like the graphic below.



That's all there is to it. Microsoft has documented this feature and how to extend the sample hardware definitions. This documentation can be found [here](#).

Note that the default platform for all the Microsoft GitHub examples is the Saeed development kit. So if you're using the Avnet Starter Kit, you'll need to update this property each time you open a sample project from GitHub for the first time.

Wrap Up

In this Lab we learned a little more about the code in the example project

- How to configure and read a button press using a GPIO signal
- How to send IoT telemetry to Azure
- How to process device twin messages from Azure
- Learned about the app_manifest.json file
- Learned about the hardware abstraction layer and how to use it

Revision History

Date	Version	Revision
1 July 19	01	Preliminary release
9 July 19	02	Minor updates based on document reviews
5 September 19	03	Added section discussing how to use the hardware abstraction layer released in 19.07
24 December 19	04	Updated for CMAKE changes