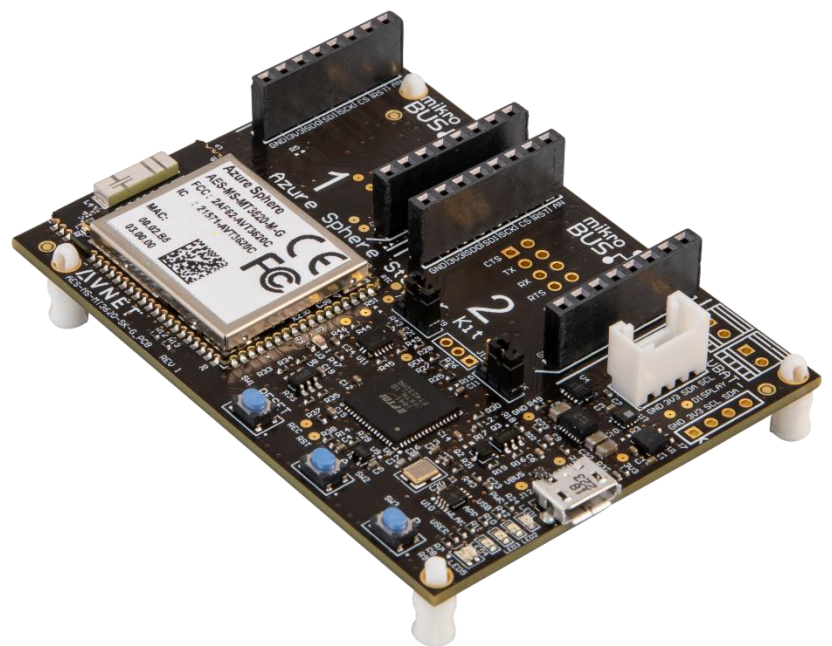


Avnet Technical Training Course

Azure Sphere: Develop a POC application from Scratch



| | |
|-------------------|--------------|
| Azure Sphere SDK: | 21.2 |
| Training Version: | v2 |
| Date: | 9 April 2021 |

Introduction

This lab directs the student through the process of creating an Azure Sphere Proof of Concept (POC) application using example applications published by Microsoft.

The POC is intended to connect a Guardian 100 to some downstream device over a UART. The downstream device implements some UART driven command response protocol. The POC will send commands, receive responses, parse them for telemetry data and then send that data to Azure.

The lab utilizes a Raspberry Pi device connected to the Guardian 100 over a USB serial port. There is a Raspberry Pi image that is already preconfigured to behave as the downstream device.

Contents

| | |
|---|----|
| Introduction | 2 |
| Lab: Objectives | 5 |
| Requirements..... | 5 |
| Hardware | 5 |
| Software | 5 |
| Other | 5 |
| Prerequisites | 6 |
| Clone the MSFT Azure Sphere Sample Repository | 6 |
| Load the Azure IoT example | 6 |
| Modify the application to make it unique for your target device/application | 6 |
| Change the Target HW to select the Avnet Azure Sphere Starter Kit | 8 |
| Connect the application to an IoT Hub..... | 8 |
| Update the app_manifest.json file with your Azure Resource details | 8 |
| Run the application and confirm functionality | 9 |
| Status Checkpoint | 9 |
| Load the UART_HighLevelApp example as a reference | 10 |
| Select the Avnet Azure Sphere Starter Kit Hardware definition..... | 10 |
| Identify all the UART specific code | 10 |
| Build/run/validate the example using a loopback connection | 12 |
| Port the UART functionality into the Azure IoT project | 14 |
| Build/run/validate the application using a loopback connection..... | 14 |
| Review Changes | 14 |
| Clean up the project | 14 |
| Remove unneeded code | 14 |
| Modify Application to request data every 10 seconds | 15 |
| Review Changes | 16 |
| Status Checkpoint | 16 |
| Port the project from the Avnet Starter Kit to the Avnet Guardian 100 | 16 |
| Update the Target HW Definitions | 16 |
| Update Remaining HW Definitions to reference the G100 hardware | 16 |
| Review Changes | 17 |
| Status Checkpoint | 17 |
| Bring up the Raspberry Pi surrogate device | 18 |
| Pi Details | 18 |
| Connect devices and power on..... | 19 |
| Prepare the G100 for development..... | 20 |

| | |
|--|----|
| Modify the app to send commands to the Pi | 20 |
| Modify the app to receive UART responses and output the responses..... | 21 |
| Modify the app to send responses to Azure as telemetry..... | 22 |
| Review Changes | 22 |
| Status Checkpoint | 22 |
| Deploy the POC application OTA..... | 23 |
| Review other improvements | 23 |
| Show connection status on LEDs | 23 |
| Review Changes | 24 |
| Create periodic events to request/send IP address updates | 24 |
| Review Changes | 24 |
| Control Send Interval with Device Twin | 24 |
| Review Changes | 24 |
| Reset or Shutdown the Pi device using Direct Methods | 24 |
| Review Changes | 24 |
| Add support to send data to Azure Storage | 25 |
| Review Changes | 27 |
| Update the OTA Deployment | 28 |
| Wrap Up | 29 |
| Appendix | 30 |
| Details on the Raspberry Pi Implementation | 30 |
| pi image..... | 30 |
| pi scripts | 30 |
| Pi script startup | 30 |
| Raspberry Pi Information | 30 |
| Revision History | 31 |

Lab: Objectives

- Walk through the entire process to develop a new application on development hardware and then transition to a production or custom hardware platform.
- Create a POC application that can be extended to any equipment with a UART command/response interface
- Show the student how to leverage existing example applications to create a custom POC application

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
- Avnet Guardian 100
 - USB A to B cable to connect Guardian 100 to Raspberry Pi
- Raspberry Pi (for downstream equipment)
 - SD card for Pi with python application running
 - Power supply for the Pi

Software

- Visual Studio 2019 version 16.4 or later (Enterprise, Professional, or Community version)
- Azure Sphere SDK 21.2 or the current SDK release

Other

- Access to an Azure Subscription
- The student should already have Azure Sphere training and understand basics like . . .
 - How to use Visual Studio to develop and debug Azure Sphere applications
 - How to open an Azure Sphere project in Visual Studio
 - How to run an Azure Sphere application from Visual Studio

Prerequisites

- Azure resources configured and validated
 - Azure IoT Hub configured
 - Azure DPS
 - Connecting to IoT Hub from an Azure Sphere device has been validated

Clone the MSFT Azure Sphere Sample Repository

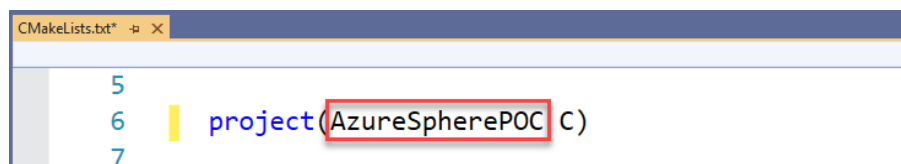
- Open Windows Powershell: **Start → Windows Powershell → Windows Powershell**
- Create the directory where you want your repo to exist on your local drive and move to that directory
 - **cd <path for new repo>**
 - **mkdir <repo folder name>**
 - **cd <repo folder name>**
- Clone your private repo
 - **git clone** <https://github.com/Azure/azure-sphere-samples>
 - Your repo is cloned onto your local drive

Load the Azure IoT example

- Open the Azure IoT example
 - Open Visual Studio 2019
 - Open the CMakeLists.txt project from the new repo
 - <your repo directory>/azure-sphere-samples/Samples/AzureIoT/CMakeLists.txt
 - The project is loaded and opens in the Visual Studio application

Modify the application to make it unique for your target device/application

- Open CMakeLists.txt
 - Edit the project name to reflect your target application



- Open app_manifest.json
 - Update the Name field to match the name you used in the CMakeLists.txt file
 - Generate a new GUID for the ComponentId

- Tools → Create GUID
- Copy the GUID and replace the GUID in your app_manifest.json file on line #4
- Modify the pasted GUID to conform to the expected JSON formatting

```

app_manifest.json
Schema: ..\..\..\..\..\program%20files%20(x86)\microsoft%20visual%20studio\2019\community\common7\ide\commonextensions\microsoft\azure%20sphere\app
1 {
2   "SchemaVersion": 1,
3   "Name": "AzureSpherePOC",
4   "ComponentId": "CF67D732-8BAC-4D36-89AD-4E9344FBBB1F",
5   "EntryPoint": "/bin/app",

```

- Open main.c
 - On ~line 492 update the static device twin properties to reflect your new product

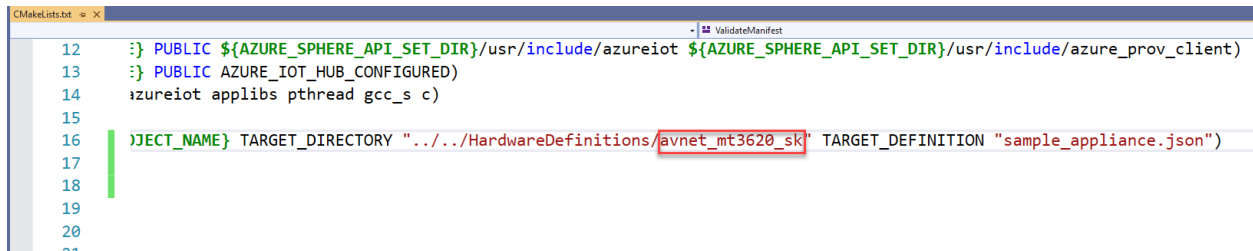
```

main.c
AzureSpherePOC.out - ARM-Debug (Global Scope) SetupAzureClient(void)
483 static void ConnectionStatusCallback(IOTHUB_CLIENT_CONNECTION_STATUS result,
484                                       IOTHUB_CLIENT_CONNECTION_STATUS_REASON reason,
485                                       void *userContextCallback)
486 {
487     iothubAuthenticated = (result == IOTHUB_CLIENT_CONNECTION_AUTHENTICATED);
488     Log_Debug("Azure IoT connection status: %s\n", GetReasonString(reason));
489     if (iothubAuthenticated) {
490         // Send static device twin properties when connection is established
491         TwinReportState(
492             "{\"manufacturer\": \"Avnet\", \"model\": \"Azure Sphere POC Device\"}");
493     }
494 }

```

Change the Target HW to select the Avnet Azure Sphere Starter Kit

- Open CMakeLists.txt
- Change ~line #16 to select the Avnet Azure Sphere Starter Kit
 - Change “/mt3620_rdb” to “avnet_mt3620_sk”

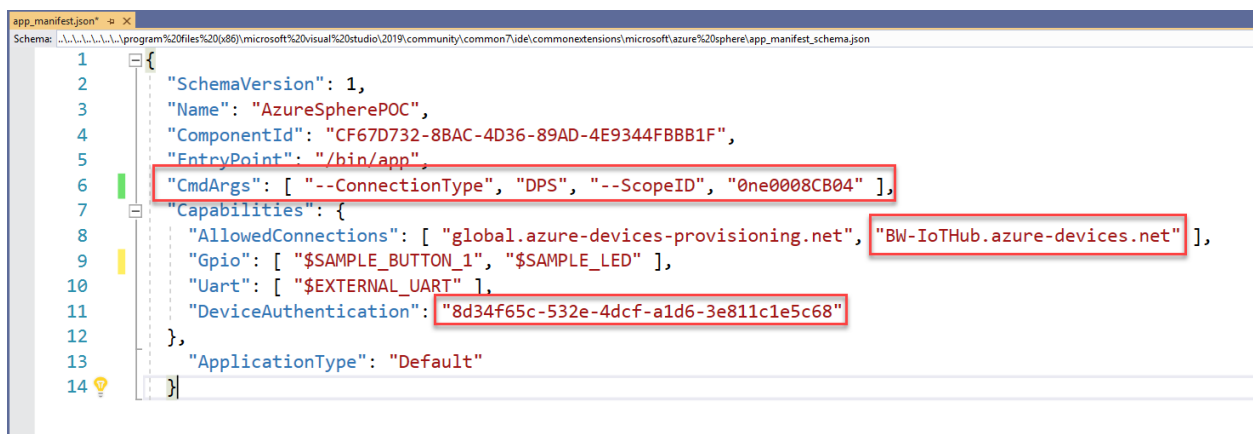


```
CMakeLists.txt
12  PUBLIC ${AZURE_SPHERE_API_SET_DIR}/usr/include/azureiot ${AZURE_SPHERE_API_SET_DIR}/usr/include/azure_prov_client)
13  PUBLIC AZURE_IOT_HUB_CONFIGURED)
14  azureiot applibs pthread gcc_s c)
15
16  )JECT_NAME} TARGET_DIRECTORY ".../HardwareDefinitions/avnet_mt3620_sk" TARGET_DEFINITION "sample_appliance.json")
17
18
19
20
21
```

Connect the application to an IoT Hub

Update the app_manifest.json file with your Azure Resource details

- Open app_manifest.json
- Using the Azure resource details from Lab4 or Lab5, copy the following data into your application manifest file
 - IoT Hub connection type and DPS Scope ID → “CmdArgs” entry
 - Add “--ConnectionType”, “DPS”,
 - Make sure the CmdArgs are all listed separately and enclosed in “” quotes
 - [“--ConnectionType”, “DPS”, “--ScopeID”, “<your dps scope ID”]
 - IoT Hub Hostname → “AllowedConnections” entry
 - Azure Sphere Tenant GUID → “DeviceAuthentication” entry



```
app_manifest.json
Schema: ..\..\..\..\program%20files%20(x86)\microsoft%20visual%20studio\2019\community\common7\ide\commonextensions\microsoft.azure%20sphere\app_manifest_schema.json
1  {
2    "SchemaVersion": 1,
3    "Name": "AzureSpherePOC",
4    "ComponentId": "CF67D732-8BAC-4D36-89AD-4E9344FB8B1F",
5    "EntryPoint": "/bin/app",
6    "CmdArgs": [ "--ConnectionType", "DPS", "--ScopeID", "0ne0008CB04" ],
7    "Capabilities": {
8      "AllowedConnections": [ "global.azure-devices-provisioning.net", "BW-IoTHub.azure-devices.net" ],
9      "Gpio": [ "$SAMPLE_BUTTON_1", "$SAMPLE_LED" ],
10     "Uart": [ "$EXTERNAL_UART" ],
11     "DeviceAuthentication": "8d34f65c-532e-4dcf-a1d6-3e811c1e5c68"
12   },
13   "ApplicationType": "Default"
14 }
```


Run the application and confirm functionality

- Run the application
- Confirm that the application connects to the IoT Hub

If the application does not connect to your IoT Hub . . .

- Confirm the device has a wifi connection: **azsphere dev wifi show-status**
- Verify details entered into app_manifest.json
- Verify the Azure Resources

```
Show output from: Device Output
Remote debugging from host 192.168.35.1, port 58711
Azure IoT Application starting.
ConnectionType: DPS
ScopeID: 0ne0008CB04
Using DPS Connection: Azure IoT DPS Scope ID 0ne0008CB04
Opening SAMPLE_BUTTON_1 as input.
Opening SAMPLE_LED as output.
[Azure IoT] Using HSM cert at /run/daa/8d34f65c-532e-4dcf-a1d6-3e811c1e5c68
IoTHubDeviceClient_LL_CreateWithAzureSphereDeviceAuthProvisioning returned 'AZURE_SPHERE_PROV_RESULT_OK'.
Azure IoT connection status: IOTHUB_CLIENT_CONNECTION_OK
INFO: Azure IoT Hub client accepted request to report state '{"manufacturer":"Avnet","model":"Azure Sphere POC Device"}'.
Sending Azure IoT Hub telemetry: {"Temperature":49.70}.
INFO: IoTHubClient accepted the telemetry event for delivery.
INFO: Azure IoT Hub client accepted request to report state '{"StatusLED":false}'.
INFO: Azure IoT Hub Device Twin reported state callback: status code 204.
INFO: Azure IoT Hub send telemetry event callback: status code 0.
Sending Azure IoT Hub telemetry: {"Temperature":49.55}.
INFO: IoTHubClient accepted the telemetry event for delivery.
INFO: Azure IoT Hub Device Twin reported state callback: status code 204.
INFO: Azure IoT Hub send telemetry event callback: status code 0.
```

Connected to Azure IoT Hub

Device provisioned

Static Device Twin Properties sent to IoT Hub

Status Checkpoint

At this point, we've accomplished the following tasks

1. Modified the Azure IoT example for our custom application
2. Configured the application to connect to our Azure IoT Hub
3. Validated connectivity to IoT Hub

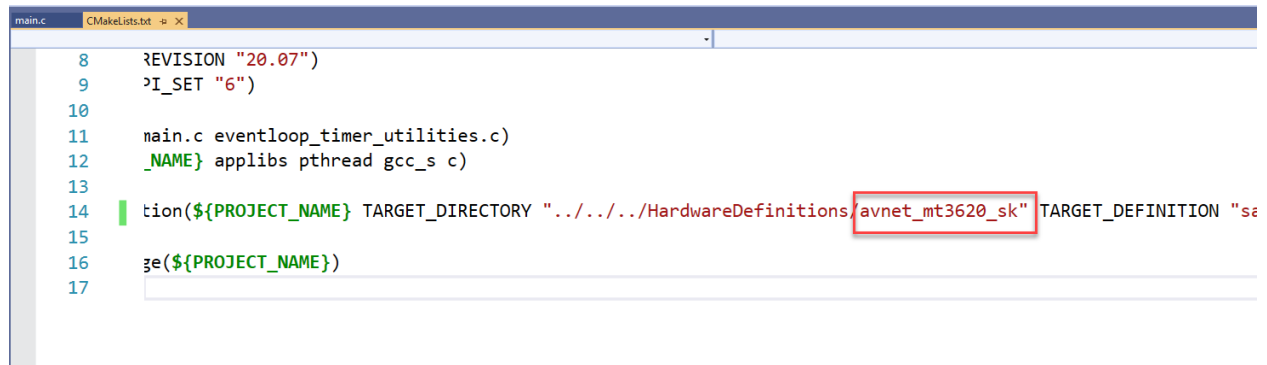
The next step is to port in the UART functionality and validate that it works

Load the UART_HighLevelApp example as a reference

- Open a second Visual Studio instance
- Open the CMakeLists.txt project from the new repo
 - <your repo directory>\azure-sphere-samples\Samples\UART\UART_HighLevelApp\CMakeLists.txt

Select the Avnet Azure Sphere Starter Kit Hardware definition

- Modify CMakeLists.txt to reference the Avnet Azure Sphere Starter Kit

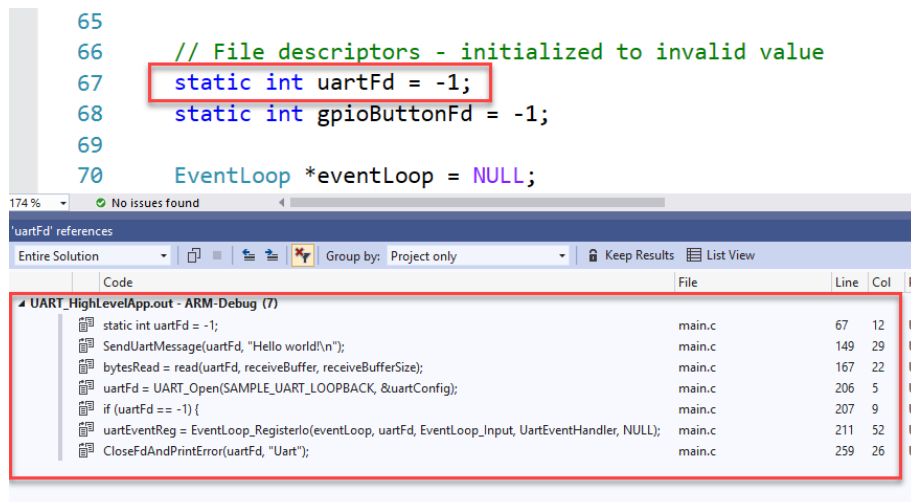


```
8 REVISION "20.07")
9 PI_SET "6")
10
11 main.c eventloop_timer_utilities.c)
12 _NAME} applibs pthread gcc_s c)
13
14 tion(${PROJECT_NAME} TARGET_DIRECTORY ".../.../HardwareDefinitions/avnet_mt3620_sk" TARGET_DEFINITION "se
15
16 ge(${PROJECT_NAME})
17
```

- Save the CMakeLists.txt file

Identify all the UART specific code

- Open main.c
- Find the uartFd variable declaration ~line 67
- Click on uartFd, then right-click, then “**Find All References**”
- A list is displayed identifying all locations in the project where uartFd is referenced
- Take a few minutes to review the UART implementation



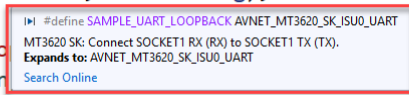
```
65
66 // File descriptors - initialized to invalid value
67 static int uartFd = -1;
68 static int gpioButtonFd = -1;
69
70 EventLoop *eventLoop = NULL;
```

| Code | File | Line | Col | Pr |
|--|--------|------|-----|----|
| static int uartFd = -1; | main.c | 67 | 12 | U |
| SendUartMessage(uartFd, "Hello world!\n"); | main.c | 149 | 29 | U |
| bytesRead = read(uartFd, receiveBuffer, receiveBufferSize); | main.c | 167 | 22 | U |
| uartFd = UART_Open(SAMPLE_UART_LOOPBACK, &uartConfig); | main.c | 206 | 5 | U |
| if (uartFd == -1) { | main.c | 207 | 9 | U |
| uartEventReg = EventLoop_RegisterIo(eventLoop, uartFd, EventLoop_Input, UartEventHandler, NULL); | main.c | 211 | 52 | U |
| CloseFdAndPrintError(uartFd, "Uart"); | main.c | 259 | 26 | U |

The UART implementation is straight forward, there are a few key items to notice

- The UART file descriptor “uartFd” is initialized in InitPeripheralsAndHandlers(void)
 - We’re opening the `SAMPLE_UART_LOOPBACK` ISU that evaluates to ISU0
 - Hover your mouse over `SAMPLE_UART_LOOPBACK` to see the constant evaluation

```
uartFd = UART_Open(SAMPLE_UART_LOOPBACK, &uartConfig);  
if (uartFd == -1) {  
    Log_Debug("ERROR: Could not open UART (errno: %d)", errno);  
    return ExitCode_Init_UartOpenFailed;  
}
```



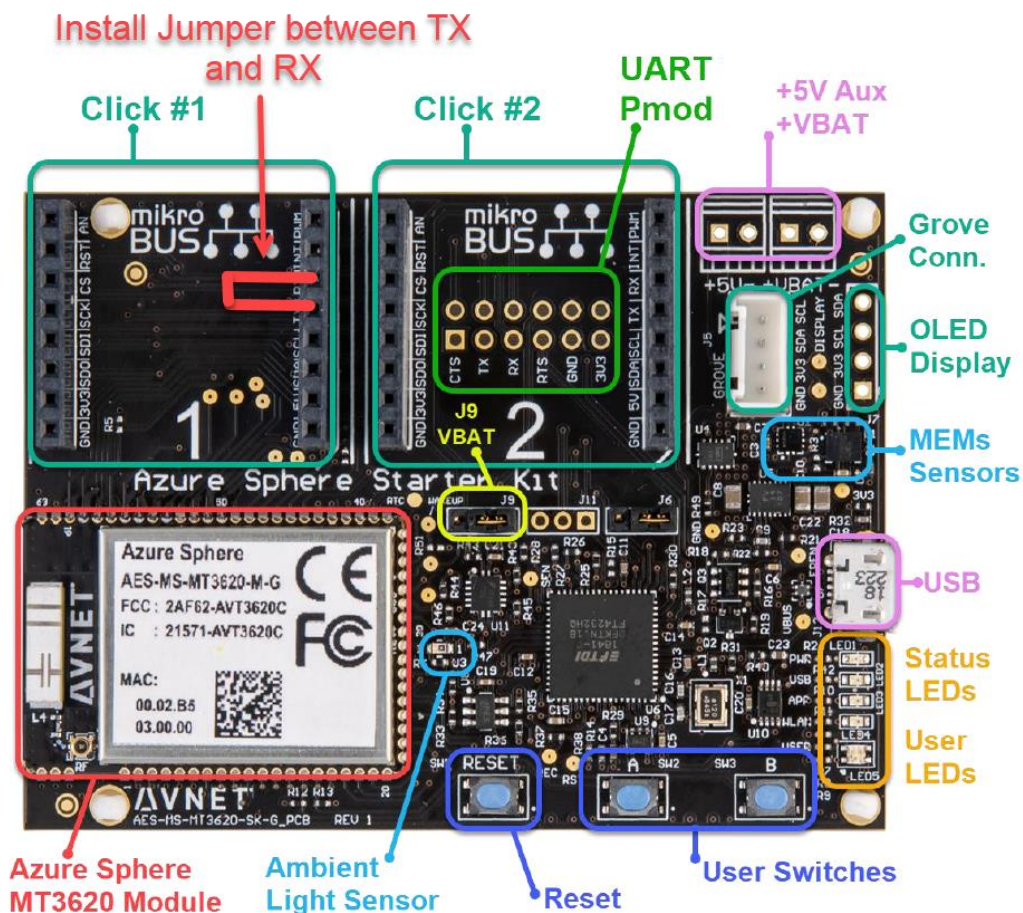
- We open the UART at a baud rate of 115200 with no Flow Control
- A UartEventHandler is associated with the uartFd and an eventLoop. When a message arrives on the UART the handler is instantiated to receive the data.
- The application sends a static message “Hello world!” over the UART from ButtonTimerEventHandler() when “the” button is pressed
- UartEventHandler reads the data on the receive side, puts it into a buffer, null terminates it and prints it to the debug window.
- uartFd is closed in ClosePeripheralsAndHandlers() when the application exits

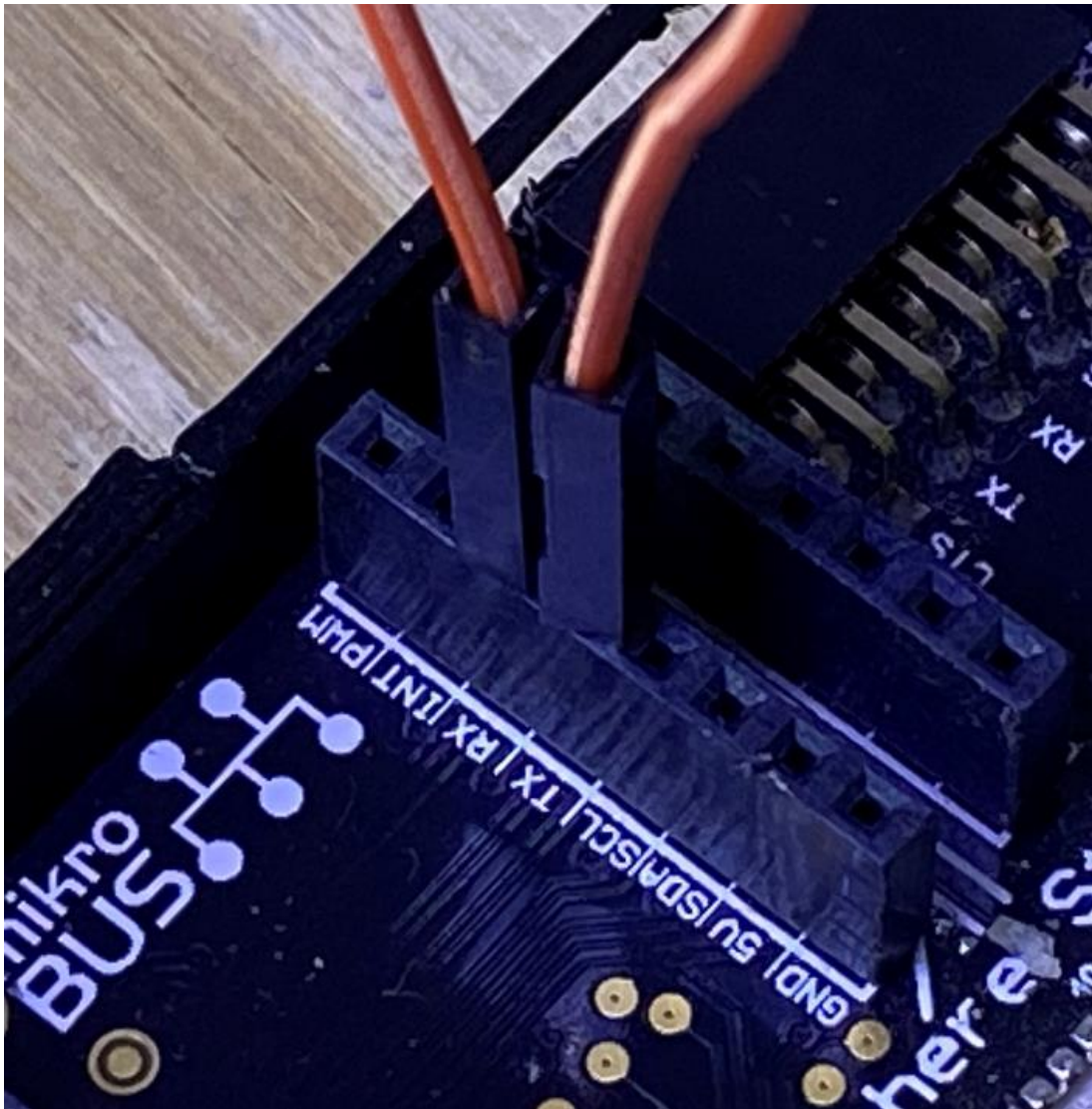
Build/run/validate the example using a loopback connection

Before we port the UART implementation into our application, let's validate that the example works. This will help us validate that our port also works. The example is written to use a simple loopback UART connection. We just need to connect ISU0 TX to ISU0 RX. Use the diagrams below to make this connection.

Click Socket #1

| Click1 Pin | Module Signal Name | Click1 Pin | Module Signal Name |
|------------|------------------------|------------|------------------------|
| AN | GPIO42_ADC1 | PWM | GPIO0_PWM0 |
| RST | GPIO16 | INT | GPIO2_PWM2 |
| CS | GPIO34_CSA1_CTS1 | RX | GPIO28_MISO0_RXD0_SDA0 |
| SCK | GPIO31_SCLK1_TX1 | TX | GPIO26_SCLK0_TXD0 |
| MISO | GPIO33_MISO1_RX1_DATA1 | SCL | GPIO37_MOSI2_RTS2_SCL2 |
| MOSI | GPIO32_MOSI1_RTS1_CLK1 | SDA | GPIO38_MISO2_RXD2_SDA2 |
| +3.3V | 3V3 | +5V | 5V |
| GND | GND | GND | GND |





- Run the application
- Press “the” button (button A, the center button on your Starter Kit)
- Note that 13 bytes are sent over the uart and 13 bytes are received 12 bytes at first, then the remaining 1 byte.

```
Output
Show output from: Device Output
Remote debugging from host 192.168.35.1, port 61022
UART application starting.
Opening SAMPLE BUTTON 1 as input.
Sent 13 bytes over UART in 1 calls.
UART received 12 bytes: 'Hello world!'
UART received 1 bytes: '
'
```


Port the UART functionality into the Azure IoT project

I'll leave the UART port as a hands on exercise. The approach I take is to move all the example UART code into my project, then try to build and run the application. The compiler will identify code that you missed. Remember to declare the UART in your app_manifest.json file.

Build/run/validate the application using a loopback connection

Once you have ported the UART changes into your POC project, run the application with the loopback connection installed on your board to verify that the UART Tx and Rx is working.

Review Changes

I've captured all the changes for this Lab in a gitHub repo. You can pull the Avnet repository using the details below.

- Clone the repo: <https://github.com/Avnet/azure-sphere-samples.git>
- Open the Samples/AvnetAzureIoTPOCExample/ high level application project
 - Note this is the completed POC example. Use the TAG references throughout this document to see specific changes as I developed the final application
- My changes can be reviewed by looking at the [POC Add UART Support tag](#).

Clean up the project

The next step is to remove any unwanted functionality and code from the example application.

Remove unneeded code

The Azure IoT example implements some features that we may want to remove/cleanup.

- Remove sending simulated temperature measurements
 - Find and remove all references to SendSimulatedTelemetry(void)
 - Do NOT remove the call to IoTHubDeviceClient_LL_DoWork(iothubClientHandle) in AzureTimerEventHandler()
- Remove all code and references to sendMessageButtonGpioFd
- Remove all code and references to the function IsButtonPressed()
- Remove all code and references to the function sendMessageButtonState()

Example application features that you may want to leave and enhance in future versions:

- Use device twins to control an LED on the device
 - Review DeviceTwinCallback()
 - Learn more about device twins [here](#)
- Use a Direct Method to receive a "Trigger Alarm" command from the Azure IoT Hub
 - Review DeviceMethodCallback()
 - Learn more about direct methods [here](#)

- Avnet has published a blog on direct methods [here](#)

Modify Application to request data every 10 seconds

The current implementation uses `buttonPollTimer` to poll the button GPIO, and if the button is pressed, it sends the UART message. The Guardian 100 (G100) hardware solution does not have a button, so let's change this functionality to send UART requests every 10 seconds, or whatever period you like. Later we can extend this functionality to allow cloud based control to set the period using a Device Twin.

The bullets below walk the user through a process where we will reuse the `buttonTimer` functionality code.

- Change all references of `ExitCode_Init_ButtonPollTimer` to `ExitCode_init_UartTxTimer`
- Change all references of `ExitCode_ButtonTimer_Consume` to `ExitCode_UartTimer_Consume`
- Rename all references of `buttonPollTimer` to `txUartMsgTimer`
- Change all references of `buttonPressCheckPeriod` to `txUartPeriod` and redefine to period to 10 seconds
- Rename all references of `ButtonPollTimerEventHandler` to `uartTxEventHandler`

```
431 // Set up a timer to periodically send UART messages
432 static const struct timespec txUartPeriod = {.tv_sec = 10, .tv_nsec = 1000 * 1000};
433 txUartMsgTimer =
434     CreateEventLoopPeriodicTimer(eventLoop, &uartTxMsgEventHandler, &txUartPeriod);
435 if (txUartMsgTimer == NULL) {
436     return ExitCode_init_UartTxTimer;
437 }
```

- At this point you can run the application on your Starter Kit and verify that the application is sending the canned UART messages every 10 seconds.

```
Output
Show output from: Device Output
Remote debugging from host 192.168.35.1, port 54845
Azure IoT Application starting.
ConnectionType: DPS
ScopeID: 0ne0008CB04
Using DPS Connection: Azure IoT DPS Scope ID 0ne0008CB04
Opening SAMPLE_LED as output.
[Azure IoT] Using HSM cert at /run/daa/8d34f65c-532e-4dcf-a1d6-3e811c1e5c68
IoTHubDeviceClient_LL_CreateWithAzureSphereDeviceAuthProvisioning returned 'AZURE_SPHERE_PROV_RESULT_OK'.
Sent 13 bytes over UART in 1 calls.
UART received 12 bytes: 'Hello world!'.
UART received 1 bytes: '
'.
Sent 13 bytes over UART in 1 calls.
UART received 12 bytes: 'Hello world!'.
UART received 1 bytes: '
'.
Sent 13 bytes over UART in 1 calls.
UART received 12 bytes: 'Hello world!'.
UART received 1 bytes: '
'.
Sent 13 bytes over UART in 1 calls.
UART received 12 bytes: 'Hello world!'.
UART received 1 bytes: '
'.
Sent 13 bytes over UART in 1 calls.
UART received 12 bytes: 'Hello world!'.
UART received 1 bytes: '
'.
```

Review Changes

- My changes can be reviewed by looking at the [POC Clean Up Example Code tag](#).

Status Checkpoint

At this point, we've accomplished the following tasks

1. Verified that the POC application connects to our Azure IoT Hub
2. Ported the UART example into our POC application
3. Verified that the UART port works
4. Cleaned up unwanted code from the POC application

The next step is to move the project onto our target hardware platform, the Avnet Guardian 100.

Port the project from the Avnet Starter Kit to the Avnet Guardian 100

We've gone as far as we can with the Starter Kit hardware. The next thing we want to do is to move over to our target hardware, the G100. This is a simple process, we need to complete the following tasks.

- Add G100 hardware definition files to the /HardwareDefinitions directory
- Update the CMakeLists.txt file to use the new hardware definition files
- Change all the hardware references in the application code to refer to the G100 hardware designations

Update the Target HW Definitions

The required G100 hardware definition folders/files are included in the [Avnet Azure Sphere Samples Repo](#)

- If you want to learn how to create your own custom hardware definitions, see the Azure Sphere documentation page [here](#).
- Open a file explorer and navigate to the project on your local drive
- Copy the azure-sphere-samples/HardwareDefinitions/avnet_g100 folder and place it into your working directory under the azure-sphere-samples/HardwareDefinitions folder
- Back in Visual Studio, open the CMakeLists.txt file
- Update the end of line 16 to reference the avnet_g100

```
16 | ${PROJECT_NAME} TARGET_DIRECTORY "../../HardwareDefinitions/avnet_g100" TARGET_DEFINITION "avnet_g100.json")
```

Update Remaining HW Definitions to reference the G100 hardware

You'll need to update two files.

- app_manifest.json
 - Remove the gpio reference to "\$SAMPLE_BUTTON_1"
 - Change "\$SAMPLE_LED" to "\$LED_1"

- Change "\$SAMPLE_UART_LOOPBACK" to "\$EXTERNAL_UART"

```

9 | | | | "Gpio": [ "$LED_1" ],
10 | | | | "Uart": [ "$EXTERNAL_UART" ],

```

- main.c

- Update the hardware specific #include

```

65 | | | | #include <hw/avnet_g100.h>

```

- Change the SAMPLE_LED reference to LED_1

```

409 | | | | GPIO_OpenAsOutput(LED_1, GPIO_OutputMode_PushPull, GPIO_Value_High);

```

- Change the SAMPLE_UART_LOOPBACK reference to EXTERNAL_UART

```

420 | | | | uartFd = UART_Open(EXTERNAL_UART, &uartConfig);

```

You should build and run your application to make sure everything still builds. The G100 uses a different UART ISU, so your loopback will NOT work. It should still connect to the Azure IoT Hub.

Review Changes

- My changes can be reviewed by looking at the [POC Add Guardian 100 Hw tag.](#)

Status Checkpoint

At this point, we've accomplished the following tasks . . .

- Changed the project to target the Guardian 100

Next let's start to work with the Guardian 100 and the Raspberry Pi

Bring up the Raspberry Pi surrogate device

The raspberry Pi is already configured with all the software and libraries needed to interoperate with your G100 and the POC application. For this lab you should not need to login to the Pi device. It will startup running our surrogate device application. If you want to access the Pi device or learn more about the surrogate application see the details in the Appendix.

If you don't have a raspberry pi already setup you can download my image from [here](#). Use the [balenaEtcher](#) application to flash it to a micro-sD card, and boot your Pi device.

Pi Details

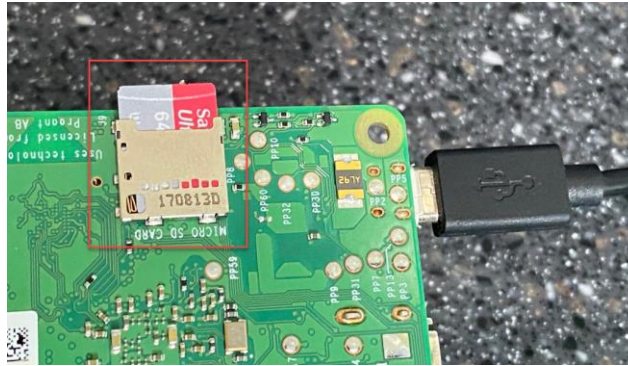
The Pi will run a python script at boot time (serialCmdResp.py) that's performing the following functionalities
...

- Opens the serial port to the G100
- Listens for commands
 - The application expects that all commands end with a '\n' (new line) character
- Supported commands
 - RebootCmd
 - Sends a “sudo reboot” command to the Pi OS
 - Returns: nothing
 - PowerdownCmd
 - Sends a “sudo shutdown -P” command to the Pi OS
 - Returns: nothing
 - ReadCPUTempCmd
 - Reads the Pi CPU temperature in Celsius
 - Returns: String with format “temp:xx.y\n”
 - `RX: temp:44.5|`
 - You should be able to affect the temperature by blowing across the CPU or holding a finger on it to heat it up more
 - IpAddressCmd
 - Reads all active network interfaces and returns the interface name and IPV4 IP address
 - Returns: One response for each active network port on the device
 - `RX: lo:127.0.0.1`
`RX: wlan0:192.168.8.208`
 - See the appendix to see the details for a pre-configured WiFi network

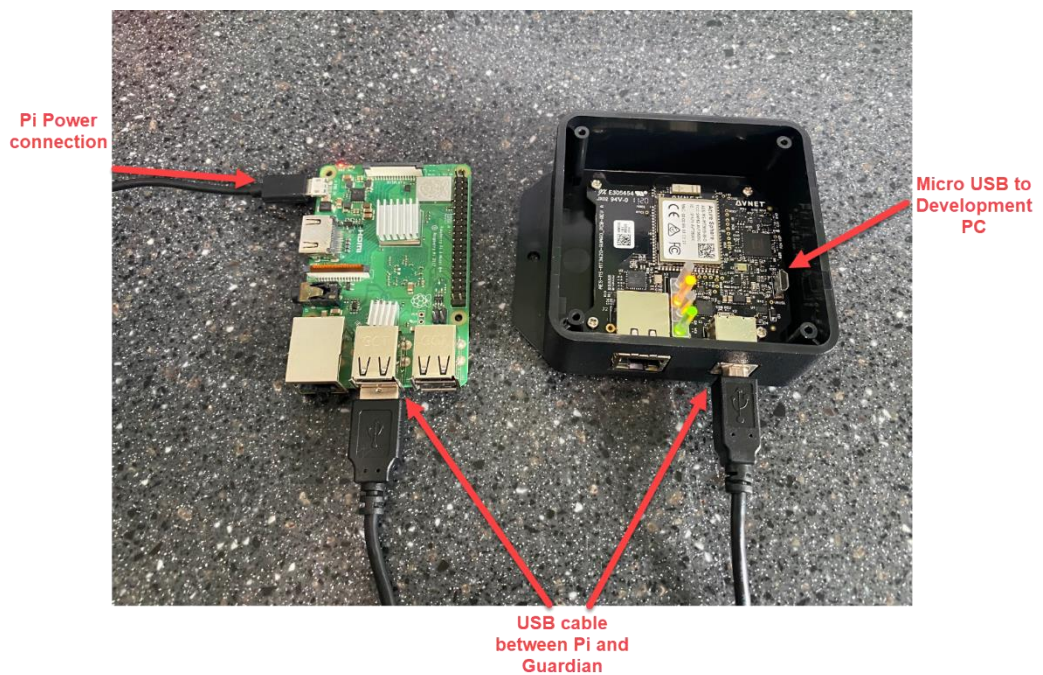
- You can also connect your Pi to a network that supports DHCP and you'll see an eth0 interface and associated IP Address

Connect devices and power on

- Verify that the microSD card is inserted all the way into your pi device



- Remove the cover of the G100 by removing the 4 Phillips (cross) screws
- Connect a micro USB cable from the G100 to your development PC
- Using the A to B USB cable that was included in your G100 box, connect the G100 to your Raspberry Pi
 - Note that in a deployment this cable can be used for Serial communications and can also power the Guardian 100 device.



- Connect the pi power supply to the micro USB connector on your pi and connect the power supply to wall power

Prepare the G100 for development

- Update the OS on your G100
 - **azsphere dev recover**
- Claim the G100 to your Azure Sphere Tenant
 - **azsphere tenant login**
 - **azsphere dev claim**
- Prepare your G100 for development
 - **azsphere dev enable-development**
- Add a WiFi network to your G100
 - **azsphere dev wifi add -s <your SSID> -p <your SSID password> -t**

Modify the app to send commands to the Pi

The azure sphere application is currently configured to send the “hello world” message over the UART. The pi application does not implement a hello world command. Update your application to send one of the supported commands.

- Open main.c and find the uartTxEventHandler() routine
- Modify the SendUardMessage(uartFd, “Hello world!”) line to send one of the supported pi commands
 - RebootCmd
 - `SendUardMessage(uartFd, “RebootCmd”)`
 - PowerdownCmd
 - `SendUardMessage(uartFd, “PowerdownCmd”)`
 - ReadCPUTempCmd
 - `SendUardMessage(uartFd, “ReadCPUTempCmd”)`
 - IpAddressCmd
 - `SendUardMessage(uartFd, “IpAddressCmd”)`
- Verify that the application receives responses from the pi
- Review the UartEventHandler() implementation
 - This implementation only receives uart response and prints what’s received

Modify the app to receive UART responses and output the responses

Next we need to modify the `UartEventHandler()` implementation to . . .

- Receive data from a single pi response that comes in as multiple UART receives
- Output the complete received message

Think about how you would implement this functionality. In the interest of time, I've developed some code that we can use to get this functionality working quickly.

- View the [TAG called POC_Uart_Rx_RingBuffer](#)
- Copy the contents of `UartEventHandler()` and paste them into your project
 - There may be a couple items outside that routine that you need, try to build your application and let the compiler identify the missing code

What I did was implement a ring buffer in a static array. It's not the most efficient implementation, but it's been tested and it seems pretty reliable.

The high level design:

- When data is received it's copied into the static `dataBuffer` starting at the next available location as identified by the `nextData` index
 - Check for buffer overflow
 - Allow the data to wrap from the end of the buffer to the beginning of the buffer
 - Update the `nextData` to point to the next available buffer location
- Each time data is received, check to see if a complete message has been received
 - A complete message is identified by a '\n' character at the end of the response
 - Start to look at index `currentData` until `nextData-1`
 - Account for messages that wrap from the end of the buffer to the beginning of the buffer
 - When a message is found, adjust the index variables to "free" the buffer where the message was found
 - Print the message to the debug window

This design lacks some robustness . . .

- There are no checks for data integrity
- If the buffer fills up, all the pointers are reset in an attempt to recover
- Other things that you'll think of . . .

Take a few minutes to port my changes into your application

Modify the app to send responses to Azure as telemetry

Now that we're receiving response messages, let's parse them and send the data to our Azure IoT Hub as telemetry.

The code required to send telemetry for the cpu temperature is shown below. Let's do a quick review:

```
// Define the {"key": value} Json string format for sending floating point data
static const char floatJsonObject[] = "{\"s\":\"%2.1f}\"";

// Allocate a buffer
#define JSON_BUFFER_SIZE 64
char *pjsonBuffer = (char *)malloc(JSON_BUFFER_SIZE);
if (pjsonBuffer == NULL) {
    Log_Debug("ERROR: not enough memory to send a message to Azure");
    return;
}

// Variable to hold the "key" string
char key[] = "temp";
// Variable to hold the "value" float
float value = 44.5;

// construct the telemetry message and send it
snprintf(pjsonBuffer, JSON_BUFFER_SIZE, floatJsonObject, key, value);
SendTelemetry(pjsonBuffer);

// Free the memory
free(pjsonBuffer);
```

Let's take some time to allow you to integrate this functionality into your application.

Review Changes

- My changes can be reviewed by looking at the [POC Parse Responses tag](#).

Status Checkpoint

At this point, we've accomplished the following tasks . . .

- Modified the application to receive long messages that are received across multiple read() calls
- Parsed out the key and value data from the UART messages
- Sent the data to the Azure IoT Hub

Next let's setup a OTA deployment for our application . . .

Deploy the POC application OTA

- Upload our application to the Azure Sphere Security Service
 - `azsphere image add --image <path to your image package>`
- Create a new product for our application
 - `azsphere product create --name AvnetPOC`
- Add our device to the device group and put it into production (enable-cloud-test) mode
 - `azsphere device enable-cloud-test --device-group "AvnetPOC/Production"`
- Create a deployment with our application package image
 - `azsphere device-group deployment create --device-group "AvnetPOC/Production" --images <Image ID from add image output above>`
- Reset your device
- Verify new image is installed OTA
 - **`azsphere device image list-installed`**

If you want to continue to add features to your application you'll need to put your G100 device back into development mode.

- `azsphere device enable-development`

Review other improvements

As the application stands right now, it's a viable IoT application. We're sending commands to a downstream device, receiving responses and sending data from the responses to our Azure IoT Hub as telemetry. The remaining sections of the lab make improvements to the application including . . .

- Use the G100 LEDs to show network and Azure IoT Hub connection status
- Send the read IP address and read CPU temperature commands at different intervals
- Control the interval that we read the CPU temperature using a Device Twin from the cloud
- Use direct method calls from the cloud to reset or power down the Pi device
- Use message properties and Azure message routing features to capture telemetry data in an Azure Storage account

Show connection status on LEDs

One basic troubleshooting step is to verify network connectivity and Azure connection status. I've added code that will use LEDs 1-3 on the G100 to indicate status. Only one LED will be lighted at a time.

LED1: On == The device does NOT have a WiFi/Network connection

LED2: On == The device is connected to Azure

LED3: On == The device is authenticated with the IoT Hub

Review Changes

- My changes can be reviewed by looking at the [POC Status LEDs TAG](#)

Create periodic events to request/send IP address updates

IP Address' should not change very often so we should not be updating these items frequently. We broke the UART Tx timer handler into two handlers so that they could run at different time intervals.

Use the current implementation as a reference and create two different periodic handlers.

Review Changes

- My changes can be reviewed by looking at the [POC Split UART TX Handler TAG](#)

Control Send Interval with Device Twin

When we did the application clean up step, I purposely left the code that catches and process device twin updates. Review that code and use it as a reference to configure the interval at which we send the read CPU command.

Review Changes

- My changes can be reviewed by looking at the [POC Add Device Twin TAG](#)

Reset or Shutdown the Pi device using Direct Methods

The Pi device is a linux system and should be shut down gracefully. I've implemented direct methods that can instruct the Pi to either reboot, or shutdown.

We added two direct methods

- RebootPi
 - Sends the RebootCmd message to the Pi
- PowerDownPi
 - Sends the PowerdownCmd message to the Pi

Review Changes

- My changes can be reviewed by looking at the [POC Add Direct Method Support TAG](#)

Add support to send data to Azure Storage

Azure Sphere IoT services can send data to an IoT Hub, but how do you migrate that data to an Azure Storage account? One way is to use message properties and Azure IoT message routes.

1. In the Azure Sphere Code add a property to the message

a. `IoTHubMessage_SetProperty(messageHandle, "log", "true");`

2:27:21 PM, 08/11/2020:

```
{
  "body": {
    "temp": 47.8
  },
  "enqueuedTime": "2020-08-11T18:27:21.287Z",
  "properties": {
    "log": "true"
  }
}
```

2. In the IoT Hub configure a message route

The screenshot shows the 'testRoute' configuration page in the Azure IoT Hub Message routing section. The route is named 'testRoute'. The endpoint is set to 'TelemetryStorage'. The data source is 'Device Telemetry Messages'. The route is enabled. A routing query is defined as 'log = 'true'', which is highlighted with a red box. A 'Test' button is visible at the bottom left, and a 'Save' button is at the bottom center.

Microsoft Azure

Dashboard > BW-IoTHub | Message routing >

testRoute
Route details

Name
testRoute

Endpoint * ⓘ
TelemetryStorage + Add

Data source * ⓘ
Device Telemetry Messages

Enable route * ⓘ
Enable Disable

Create a query to filter messages before data is routed to an endpoint. [Learn more](#)

Routing query ⓘ
1 log = 'true'

Test

Save

With a Custom Endpoint

[Dashboard](#) >

BW-IoTHub | Message routing 

×

»

Send data from your devices to endpoints that you choose.

[Routes](#) [Custom endpoints](#) [Enrich messages](#)

Choose which Azure services will receive your messages. You can add up to 10 endpoints to an IoT hub.

[+](#) Add [🔗](#) Synchronize keys [🗑️](#) Delete [🔄](#) Refresh

Event Hubs

Service Bus queue

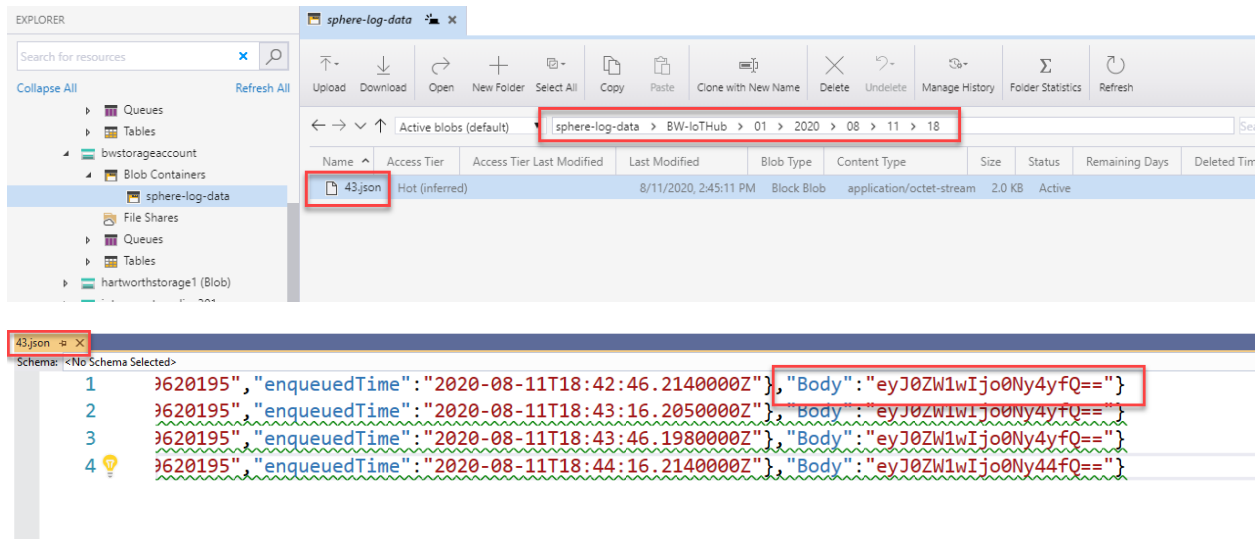
Service Bus topic

Storage

Recommended for archiving data.

| <input type="checkbox"/> Name | Container name | Encoding format | Batch frequency(sec... | Filename format | Authentication type | Status |
|---|-----------------|-----------------|------------------------|---------------------------|---------------------|-----------|
| <input type="checkbox"/> TelemetryStorage | sphere-log-data | JSON | 100 | (iohub)/(partition)/(Y... | Key-based | ● Unknown |

Telemetry is captured in Azure Storage as base64 encoded data.



The screenshot shows the Azure Storage Explorer interface. On the left, the 'EXPLORER' pane shows the hierarchy: Queues, Tables, Blob Containers, and a specific container named 'sphere-log-data'. The main pane shows the 'Active blobs (default)' view for this container. A single blob named '43.json' is listed. The blob's properties are: Access Tier: Hot (inferred), Last Modified: 8/11/2020, 2:45:11 PM, Blob Type: Block Blob, Content Type: application/octet-stream, Size: 2.0 KB, Status: Active. The blob's content is displayed in a text editor below, showing a JSON array of four telemetry messages. Each message is an object with 'enqueuedTime' and 'Body' fields. The 'Body' field contains a base64-encoded string. The messages are numbered 1 through 4 in the editor.

```
1  {"enqueuedTime": "2020-08-11T18:42:46.2140000Z", "Body": "eyJ0ZWlwIjo0Ny4yfQ=="}
```

```
2  {"enqueuedTime": "2020-08-11T18:43:16.2050000Z", "Body": "eyJ0ZWlwIjo0Ny4yfQ=="}
```

```
3  {"enqueuedTime": "2020-08-11T18:43:46.1980000Z", "Body": "eyJ0ZWlwIjo0Ny4yfQ=="}
```

```
4  {"enqueuedTime": "2020-08-11T18:44:16.2140000Z", "Body": "eyJ0ZWlwIjo0Ny44fQ=="}
```

https://www.base64decode.org

ere FAE R... Dashboard - Micros... General (Attabotics... Google Azure Sph

Decode from Base64 format

Simply enter your data then push the decode button.

eyJ0ZW1wljo0Ny44fQ==

For encoded binaries (like images, documents, etc.) use the file upload

UTF-8 Source character set.

☐ Decode each line separately (useful for multiple entries).

☒ Live mode OFF Decodes in real-time when you type or paste (

< DECODE > Decodes your data into the textarea below.

{"temp":47.8}

Review Changes

- My changes can be reviewed by looking at the [POC Add Message Property TAG](#)

Update the OTA Deployment

It's easy to update our production OTA deployment

- Before updating a deployment I recommend updating the app_manifest.json "Name": attribute. This string will be imbedded into the image package and will make it easier when managing multiple images.
 - Open app_manifest.json
 - Change the "Name" attribute. For example add a version



```
app_manifest.json  Diff - main.c:HEAD vs. main.c  main.c
Schemat: ..\..\..\..\program%20files%20(x86)\microsoft%20visual%20studio\2019\community\common7\ide\commonextensions\microsoft\azure%20sphere\
1  {
2    "SchemaVersion": 1,
3    "Name": "AzureSpherePOC_V1.1",
4    "ComponentId": "CF67D732-8BAC-4D36-89AD-4E9344FBBB1F",
5    "EntryPoint": "/bin/app",
6    "CmdArgs": [ "--ConnectionType", "DPS", "--ScopeID", "
```

- Upload our application to the Azure Sphere Security Service
 - `azsphere image add --image <path to your image package>`
- Create a deployment with our application package image
 - `azsphere device-group deployment create --device-group "AvnetPOC/Production" --images <Image ID from add image output above>`
- Put your device back into the device group and put it into production (enable-cloud-test) mode
 - `azsphere device enable-cloud-test --device-group "AvnetPOC/Production"`
 - When we put the device back into development mode, it was moved into the AvnetPOC/Development device group automatically
- You can add any device claimed to your Azure Sphere Tenant to this Product/Device group
 - Connect the new device to your development pc
 - `azsphere device-group deployment create --device-group "AvnetPOC/Production" --images <Image ID from add image output above>`

OR

- `azsphere device update --device <device ID> --device-group "AvnetPOC/Production"`
 - Add the **--device <device ID>** command line option to add devices by device ID if the device is not connected to your development PC

Wrap Up

This lab was written with the objective of showing the student how to create a proof of concept application using the Azure Sphere SDK and example applications.

We started with a new Azure Sphere application using the Microsoft Azure IoT example application in GitHub. From there we ported in the UART loopback example and then proceeded to build out the application with useful features including adding cloud based controls and routing our data to Azure Storage. We used a downstream device connected over the G100 UART that accepted commands and returned real data from the downstream device.

This POC application can be modified to interact with any downstream device that has a serial port that accepts commands and responds with data. The developer just needs to understand the data being returned from the new device so that it can be parsed for meaningful data.

Appendix

Details on the Raspberry Pi Implementation

pi image

Raspberry Pi image with all software and configurations can be downloaded from [here](#). Use the [balenaEtcher](#) application to flash it to a micro-sD card, and boot your Pi device.

pi scripts

The python script and launcher script is included in the POCLab branch in the piScripts folder

Script locations on the pi filesystem:

Guardian 100 script: /home/pi/python/serialCmdResp.py

Launcher script: /home/pi/python/launcher.sh

Launcher logs: /home/pi/python/cronlog

Pi script startup

The G100 script is automatically run at boot time using a cron job

To disable the auto run feature edit the cron tab

```
> sudo crontab -e
```

Find the line that calls the launcher.sh shell script and comment it out

To stop the script that's currently running

- Identify the Process ID
 - `ps -ef | grep python3`
- find the PID for the "python3 serialCmdResp.py" process
 - `sudo kill <PID>`

Raspberry Pi Information

- User credentials
 - user: pi
 - password: 12345678
- SSH is enabled
- VNC is enabled
- There is a WiFi network configured in the image
 - SSID: IoTDemo
 - SSID password: IoTDemo2001

Revision History

| Date | Version | Revision |
|-----------------|---------|--|
| 11 August 2020 | 00 | Preliminary release |
| 12 October 2020 | 01 | Improvements based on user feedback |
| 9 April 2021 | 02 | Removed source control steps, cleaned up TAG links |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |