CODE & FUNCTIONS

This code is modified version of the samples that comes along with Azure IoT hub C SDK .My friends in the Azure IoT team have done some excellent documentation on Azure IoT C SDK and I strongly recommend you to read those. Please don't miss "Use the IoTHub Client" and "Use the serializer" section as well. This link has some portion of code which we are using. These are the few AzureIoT libraries which you need to import: AzureIoTHub, AzureIoTProtocol_MQTT and AzureIoTUtility.Since we use JSON in this code we have ArduinoJson library which simplify JSON handling for us.

First, you need to create a handle using IoTHubClient_LL_CreateFromConnectionString method which initializes the library. Then you call read Message and passes a variable message count and a char array. We then call our wrapper send Message passing the handle and message we need to send.

In our wrapper we create a message handle and then use IoTHubClient_LL_SendEventAsync to send these data to Azure IoT. We also register a call back function sendCallback to be called when data is successfully sent. In this example we aren't using serializer library.

We need to pass the protocol in IoTHubClient_LL_CreateFromConnectionString method which is MQTT in this case. AQMP and HTTPS is also supported.

A few things to note here with C SDK, for Azure IoT hub. There are two sets of API methods one with LL and one without LL (low level). The corresponding methods are:

- IoTHubClient_LL_SendEventAsync
- IoTHubClient_LL_SetMessageCallback
- IoTHubClient_LL_Destroy
- IoTHubClient_LL_DoWork

When we use the method in LL API's we have explicit control over network transmissions. To actually send the data we use IoTHubClient_LL_DoWork
method while IoTHubClient_LL_SendEventAsync just places the message in a buffer. Last but not the least like with any native code don't forget to clean up resources using IoTHubMessage_Destroy method.

"W*hen you call **IoTHubClient_CreateFromConnectionString**, the underlying libraries create a new thread that runs in the background. This thread sends events to, and receives messages from, IoT Hub. No such thread is created when working with the **LL** APIs. The creation of the background*

*thread is a convenience to the developer. You don't have to worry about explicitly sending events and receiving messages from IoT Hub — it happens automatically in the background. In contrast, the **LL** APIs give you explicit control over communication with IoT Hub, if you need it* ".

I haven't tested non-LL methods in NodeMCU.My assumption is multiple threads doesn't make sense in ESP as long as it's single core or have a single core of computing. I may be wrong and don't know if anyone has ported it to FreeRTOS to support multi-threading yet.

The **IoTHubClient** library depends on other open-source libraries:

- The Azure C shared utility library, which provides common functionality for basic tasks (such as strings, list manipulation, and IO) needed across several Azure-related C SDKs.
- The Azure uAMQP library, which is a client-side implementation of AMQP optimized for resource constrained devices.
- The Azure uMQTT library, which is a general-purpose library implementing the MQTT protocol and optimized for resource constrained devices.

When you have a valid **IOTHUB_CLIENT_HANDLE**, you can start calling the APIs to send and receive messages to and from IoT Hub.

The lower-level APIs

The previous article described the basic operation of the **IotHubClient** within the context of the **iothub_client_sample_amqp** application. For example, it explained how to initialize the library using this code.

```
IOTHUB_CLIENT_HANDLE iotHubClientHandle;
iotHubClientHandle = IoTHubClient_CreateFromConnectionString (connectionString, AMQP_Protocol);
```

It also described how to send events using this function call.

```
IoTHubClient_SendEventAsync(iotHubClientHandle, message.messageHandle, SendConfirmationCallback, &message);
```

The article also described how to receive messages by registering a callback function.

```
int receiveContext = 0;
IoTHubClient_SetMessageCallback(iotHubClientHandle, ReceiveMessageCallback, &receiveContext);
```

The article also showed how to free resources using code such as the following.

```
IoTHubClient_Destroy(iotHubClientHandle);
```

There are companion functions for each of these APIs:

- IoTHubClient_LL_CreateFromConnectionString
- IoTHubClient_LL_SendEventAsync
- IoTHubClient_LL_SetMessageCallback
- IoTHubClient_LL_Destroy

These functions all include **LL** in the API name. Other the **LL** part of the name, the parameters of each of these functions are identical to their non-LL counterparts. However, the behavior of these functions is different in one important way.

When you call **IoTHubClient_CreateFromConnectionString**, the underlying libraries create a new thread that runs in the background. This thread sends events to, and receives messages from, IoT Hub. No such thread is created when working with the **LL** APIs. The creation of the background thread is a convenience to the developer. You don't have to worry about explicitly sending events and receiving messages from IoT Hub -- it happens automatically in the background. In contrast, the **LL** APIs give you explicit control over communication with IoT Hub, if you need it.

To understand this concept better, let's look at an example:

When you call **IoTHubClient_SendEventAsync**, what you're actually doing is putting the event in a buffer. The background thread created when you call **IoTHubClient_CreateFromConnectionString** continually monitors this buffer and sends any data that it contains to IoT Hub. This happens in the background at the same time that the main thread is performing other work.

Similarly, when you register a callback function for messages using **IoTHubClient_SetMessageCallback**, you're instructing the SDK to have the background thread invoke the callback function when a message is received, independent of the main thread.

The **LL** APIs don't create a background thread. Instead, a new API must be called to explicitly send and receive data from IoT Hub. This is demonstrated in the following example.

The **iothub_client_sample_http** application that's included in the SDK demonstrates the lower-level APIs. In that sample, we send events to IoT Hub with code such as the following:

```
EVENT_INSTANCE message;
sprintf_s(msgText, sizeof(msgText), "Message_%d_From_IoTHubClient_LL_Over_HTTP",
i);
message.messageHandle = IoTHubMessage_CreateFromByteArray((const unsigned
char*)msgText, strlen(msgText));

IoTHubClient_LL_SendEventAsync (iotHubClientHandle, message.messageHandle,
SendConfirmationCallback, &message)
```

The first three lines create the message, and the last line sends the event. However, as mentioned previously, sending the event means that the data is simply placed in a buffer. Nothing is transmitted on the network when we call **IoTHubClient_LL_SendEventAsync**. In order to actually ingress the data to IoT Hub, you must call **IoTHubClient_LL_DoWork**, as in this example:

```
while (1)
{
    IoTHubClient_LL_DoWork(iotHubClientHandle);
    ThreadAPI_Sleep(100);
}
```

This code (from the **iothub_client_sample_http** application) repeatedly calls **IoTHubClient_LL_DoWork**. Each time **IoTHubClient_LL_DoWork** is called, it sends some events from the buffer to IoT Hub and it retrieves a queued message being sent to the device. The latter case means that if we registered a callback function for messages, then the callback is invoked (assuming any messages are queued up). We would have registered such a callback function with code such as the following:

```
IoTHubClient_LL_SetMessageCallback (iotHubClientHandle, ReceiveMessageCallback,
&receiveContext)
```

The reason that **IoTHubClient_LL_DoWork** is often called in a loop is that each time it's called, it sends *some* buffered events to IoT Hub and retrieves *the next* message queued up for the device. Each call isn't guaranteed to send all buffered events or to retrieve all queued messages. If you want to send all events in the buffer and then continue on with other processing you can replace this loop with code such as the following:

```
IOTHUB_CLIENT_STATUS status;

while ((IoTHubClient_LL_GetSendStatus(iotHubClientHandle, &status) ==
IOTHUB_CLIENT_OK) && (status == IOTHUB_CLIENT_SEND_STATUS_BUSY))
{
    IoTHubClient_LL_DoWork(iotHubClientHandle);
    ThreadAPI_Sleep(100);
```

```
}
```

This code calls **IoTHubClient_LL_DoWork** until all events in the buffer have been sent to IoT Hub. Note this does not also imply that all queued messages have been received. Part of the reason for this is that checking for "all" messages isn't as deterministic an action. What happens if you retrieve "all" of the messages, but then another one is sent to the device immediately after? A better way to deal with that is with a programmed timeout. For example, the message callback function could reset a timer every time it's invoked. You can then write logic to continue processing if, for example, no messages have been received in the last *X* seconds.

When you're finished ingressing events and receiving messages, be sure to call the corresponding function to clean up resources.

```
IoTHubClient_LL_Destroy(iotHubClientHandle);
```

Basically, there's only one set of APIs to send and receive data with a background thread and another set of APIs that does the same thing without the background thread. A lot of developers may prefer the non-LL APIs, but the lower-level APIs are useful when the developer wants explicit control over network transmissions. For example, some devices collect data over time and only ingress events at specified intervals (for example, once an hour or once a day). The lower-level APIs give you the ability to explicitly control when you send and receive data from IoT Hub. Others will simply prefer the simplicity that the lower-level APIs provide. Everything happens on the main thread rather than some work happening in the background.

Whichever model you choose, be sure to be consistent in which APIs you use. If you start by calling **IoTHubClient_LL_CreateFromConnectionString**, be sure you only use the corresponding lower-level APIs for any follow-up work:

- IoTHubClient_LL_SendEventAsync
- IoTHubClient_LL_SetMessageCallback
- IoTHubClient_LL_Destroy
- IoTHubClient_LL_DoWork

The opposite is true as well. If you start with **IoTHubClient_CreateFromConnectionString**, then use the non-LL APIs for any additional processing.

In the Azure IoT device SDK for C, see the **iothub_client_sample_http** application for a complete example of the lower-level APIs.
The **iothub_client_sample_amqp** application can be referenced for a full example of the non-LL APIs.

Property handling

So far when we've described sending data, we've been referring to the body of the message. For example, consider this code:

```
EVENT_INSTANCE message;
sprintf_s(msgText, sizeof(msgText), "Hello World");
message.messageHandle = IoTHubMessage_CreateFromByteArray((const unsigned
char*)msgText, strlen(msgText));
IoTHubClient_LL_SendEventAsync(iotHubClientHandle, message.messageHandle,
SendConfirmationCallback, &message)
```

This example sends a message to IoT Hub with the text "Hello World." However, IoT Hub also allows properties to be attached to each message. Properties are name/value pairs that can be attached to the message. For example, we can modify the previous code to attach a property to the message:

```
MAP_HANDLE propMap = IoTHubMessage_Properties(message.messageHandle);
sprintf_s(propText, sizeof(propText), "%d", i);
Map_AddOrUpdate(propMap, "SequenceNumber", propText);
```

We start by calling **IoTHubMessage_Properties** and passing it the handle of our message. What we get back is a **MAP_HANDLE** reference that enables us to start adding properties. The latter is accomplished by calling **Map_AddOrUpdate**, which takes a reference to a MAP_HANDLE, the property name, and the property value. With this API we can add as many properties as we like.

When the event is read from **Event Hubs**, the receiver can enumerate the properties and retrieve their corresponding values. For example, in .NET this would be accomplished by accessing the Properties collection on the EventData object.

In the previous example, we're attaching properties to an event that we send to IoT Hub. Properties can also be attached to messages received from IoT Hub. If we want to retrieve properties from a message, we can use code such as the following in our message callback function:

```
static IOTHUBMESSAGE_DISPOSITION_RESULT
ReceiveMessageCallback(IOTHUB_MESSAGE_HANDLE message, void* userContextCallback)
{

    // Retrieve properties from the message
    MAP_HANDLE mapProperties = IoTHubMessage_Properties(message);
    if (mapProperties != NULL)
    {
        const char*const* keys;
        const char*const* values;
```

```
        size_t propertyCount = 0;
        if (Map_GetInternals(mapProperties, &keys, &values, &propertyCount) ==
MAP_OK)
        {
            if (propertyCount > 0)
            {
                printf("Message Properties:\r\n");
                for (size_t index = 0; index < propertyCount; index++)
                {
                    printf("\tKey: %s Value: %s\r\n", keys[index], values[index]);
                }
                printf("\r\n");
            }
        }
    }

    . . .
}
```

The call to **IoTHubMessage_Properties** returns the **MAP_HANDLE** reference. We then pass that reference to **Map_GetInternals** to obtain a reference to an array of the name/value pairs (as well as a count of the properties). At that point it's a simple matter of enumerating the properties to get to the values we want.

You don't have to use properties in your application. However, if you need to set them on events or retrieve them from messages, the **IoTHubClient** library makes it easy.

Message handling

Alternate device credentials

As explained previously, the first thing to do when working with the **IoTHubClient** library is to obtain a **IOTHUB_CLIENT_HANDLE** with a call such as the following:

```
IOTHUB_CLIENT_HANDLE iotHubClientHandle;
iotHubClientHandle = IoTHubClient_CreateFromConnectionString(connectionString,
AMQP_Protocol);
```

The arguments to **IoTHubClient_CreateFromConnectionString** are the device connection string and a parameter that indicates the protocol we use to communicate with IoT Hub. The device connection string has a format that appears as follows:

```
HostName=IOTHUBNAME.IOTHUBSUFFIX;DeviceId=DEVICEID;SharedAccessKey=SHAREDACCESSKEY
```

There are four pieces of information in this string: IoT Hub name, IoT Hub suffix, device ID, and shared access key. You obtain the fully qualified domain name (FQDN) of an IoT hub when you create your IoT hub instance in the Azure portal — this gives you the IoT hub name (the first part of the FQDN) and the IoT hub suffix (the rest of the FQDN). You get the device ID and the shared access key when you register your device with IoT Hub (as described in the previous article).

IoTHubClient_CreateFromConnectionString gives you one way to initialize the library. If you prefer, you can create a new **IOTHUB_CLIENT_HANDLE** by using these individual parameters rather than the device connection string. This is achieved with the following code:

```
IOTHUB_CLIENT_CONFIG iotHubClientConfig;
iotHubClientConfig.iotHubName = "";
iotHubClientConfig.deviceId = "";
iotHubClientConfig.deviceKey = "";
iotHubClientConfig.iotHubSuffix = "";
iotHubClientConfig.protocol = HTTP_Protocol;
IOTHUB_CLIENT_HANDLE iotHubClientHandle =
IoTHubClient_LL_Create(&iotHubClientConfig);
```

This accomplishes the same thing as **IoTHubClient_CreateFromConnectionString**.

It may seem obvious that you would want to use **IoTHubClient_CreateFromConnectionString** rather than this more verbose method of initialization. Keep in mind, however, that when you register a device in IoT Hub what you get is a device ID and device key (not a connection string). The *device explorer* SDK tool introduced in the previous article uses libraries in the **Azure IoT service SDK** to create the device connection string from the device ID, device key, and IoT Hub host name. So calling **IoTHubClient_LL_Create** may be preferable because it saves you the step of generating a connection string. Use whichever method is convenient.

Configuration options

So far everything described about the way the **IoTHubClient** library works reflects its default behavior. However, there are a few options that you can set to change how the library works. This is accomplished by leveraging the **IoTHubClient_LL_SetOption** API. Consider this example:

```
unsigned int timeout = 30000;
IoTHubClient_LL_SetOption(iotHubClientHandle, "timeout", &timeout);
```

There are a couple of options that are commonly used:

- **SetBatching** (bool) – If **true**, then data sent to IoT Hub is sent in batches. If **false**, then messages are sent individually. The default is **false**. Batching over AMQP / AMQP-WS, as well as adding system properties on D2C messages, is supported.
- **Timeout** (unsigned int) – This value is represented in milliseconds. If sending an HTTPS request or receiving a response takes longer than this time, then the connection times out.

The batching option is important. By default, the library ingresses events individually (a single event is whatever you pass to **IoTHubClient_LL_SendEventAsync**). If the batching option is **true**, the library collects as many events as it can from the buffer (up to the maximum message size that IoT Hub will accept). The event batch is sent to IoT Hub in a single HTTPS call (the individual events are bundled into a JSON array). Enabling batching typically results in big performance gains since you're reducing network round-trips. It also significantly reduces bandwidth since you are sending one set of HTTPS headers with an event batch rather than a set of headers for each individual event. Unless you have a specific reason to do otherwise, typically you'll want to enable batching.