Application Note

Renesas RA Family

RA MQTT/TLS Azure Cloud Connectivity Solution - Ethernet

Introduction

This application note describes IoT Cloud connectivity solutions in general and introduces you briefly to the IoT Cloud solution provider, Microsoft Azure. It covers the RA FSP MQTT/TLS module along with the Azure IoT SDK for embedded C.

This application project is built with the integrated "Azure IoT SDK for Embedded C" package which allows small embedded (IoT) devices like Renesas RA family of MCUs RA6M3/RA6M4/RA6M5 to communicate with Azure IoT services.

The application example uses Azure IoT DPS (Device Provisioning Service) to provision, register the IoTdevice, and send and receive data to and from the development kit.

This application note enables you to effectively use the RA FSP modules in your own design with the FSP-integrated Azure IoT SDK. Upon completion of this guide, you will be able to add the FSP modules to your own design, configure it correctly with Azure IoT SDK for the target application, and write code using the included application example code as a reference and efficient starting point. References to more detailed API descriptions and sample code, that demonstrate advanced usage of FSP modules are available in the *RA FSP Software Package (FSP) User's Manual* (see Next Steps section) and serve as valuable resources in creating more complex designs. Explaining the underlying operation of Azure IoT SDK for Embedded C is beyond the scope of this document. Users should refer to the documentation from Microsoft for education on topics related to the Azure IoT SDK section: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide- sdks

In this release, the CK-RA6M5 v2 kit is used for the application project.

Required Resources

To build and run the MQTT/TLS application example, you need:

Development Tools and Software

- e^2 studio version: v2024-04 or later
- RA FSP Software Package (FSP) v5.3.0
- SEGGER J-Link® RTT viewer version: 7.96a or later
- Azure IoT explorer 0.14.13.0 or later (PC tool for validating the Cloud side). Download Link: Releases · Azure/azure-iot-explorer (github.com)
- Azure CLI 2.44 or later (Azure command-line interface is a set of commands used to create and manage Azure resources) Download Link: How to install the Azure CLI | Microsoft Learn
- Access to Azure Cloud Connectivity Portal (https://portal.azure.com/#home) to create IoT Devices (If you are new to Azure IoT)

Hardware

- Renesas CK-RA6M5 v2 kit (CK-RA6M5 - Cloud Kit Based on RA6M5 MCU Group | Renesas)
- PC running Windows® 10, Tera Term console or similar application, and an installed web browser (Google Chrome, Internet Explorer, Microsoft Edge, Mozilla Firefox, or Safari).
- Micro USB cable
- USB-C cable
- Ethernet cable (CAT5/6)
- Router with an ethernet port or ethernet switch to connect to the router for Internet connectivity.

Prerequisites and Intended Audience

This application note assumes that you have some experience with the Renesas e^2 studio ISDE and RA Flexible Software Package (FSP). Before you perform the procedures in this application note, follow the procedure in the *FSP User Manual* to build and run the Blinky project. Doing so enables you to become familiar with the e^2 studio and the FSP and also validates that the debug connection to your board functions properly. In addition, this application note assumes you have some knowledge of MQTT/TLS and its communication protocols.

The intended audience are users who want to connect to Azure Cloud using the Azure IoT SDK for Embedded C on the Renesas RA/RA6 MCU Series.

Note: If you are a first-time user of e^2 studio and FSP, we highly recommend you install e^2 studio and FSP on your system in order to run the Blinky Project and to get familiar with the e^2 studio and FSP development environment before proceeding to the next sections.

Note: If you are new to Azure Internet of Things, we recommend you get started with Introduction to the AzureIoT https://learn.microsoft.com/en-us/azure/iot/iot-introduction

Prerequisites

- Access to online documentation available for Azure in the Cloud Connectivity References section.
- Access to the latest documentation for the identified Renesas Flexible Software Package.
- Prior knowledge of operating e^2 studio and built-in (or standalone) RA Configurator.
- Access to associated hardware documentation such as User Manuals and Schematics.

Using this Application Note

Section 1 of this document covers the General Overview of the Cloud Connectivity, Azure IoT Solution using IoT Central, Azure DPS, MQTT, and TLS Protocols, and Device certificates and Keys used in the Cloud Connectivity.

Section 2 covers the modules provided by RA FSP to establish connectivity to Cloud service providers and the features supported by the module.

Section 3 covers the architecture of the reference application project, an overview of the software components included, and step-by-step guidelines for recreation using the FSP configurator. It also covers setting up the Azure IoT Hub, creating the self-signed certificates, storing the certificates in the flash using the application CLI.

Section 4 covers Importing, building, and running the Application project.

Note: We recommend that you operate with your own Microsoft Azure Cloud credentials and use your created Cloud configurations to run the application. The default sample configuration detailed in this project is for reference only and may have access issues to Azure since the application is communicating with a test account.

Note: For a quick validation using the provided application project, you can skip sections 1 to 2 and go to sections 3 and 4 for instructions on setting up the Azure IoT Hub, creating the self-signed certificates, storing the certificates in the flash using the application CLI, and running the application project on the CK-RA6M5 v2 board.

Contents

1. Introduction to Cloud Connectivity

1.1 Cloud Connectivity Overview

Internet of Things (IoT) is a sprawling set of technologies described as connecting everyday objects, like sensors or smartphones, to the World Wide Web. IoT devices are intelligently linked together to enable newforms of communication between things and people, and among things.

These devices, or things, connect to the network. Using sensors, they provide the information they gather from the environment or allow other systems to reach out and act on the world through actuators. In the process, IoT devices generate massive amounts of data, and Cloud computing provides a pathway, enabling data to travel to its destination.

The IoT Cloud Connectivity Solution includes the following major components:

1. Devices or Sensors
2. Gateway
3. IoT Cloud services
4. End-user application/system

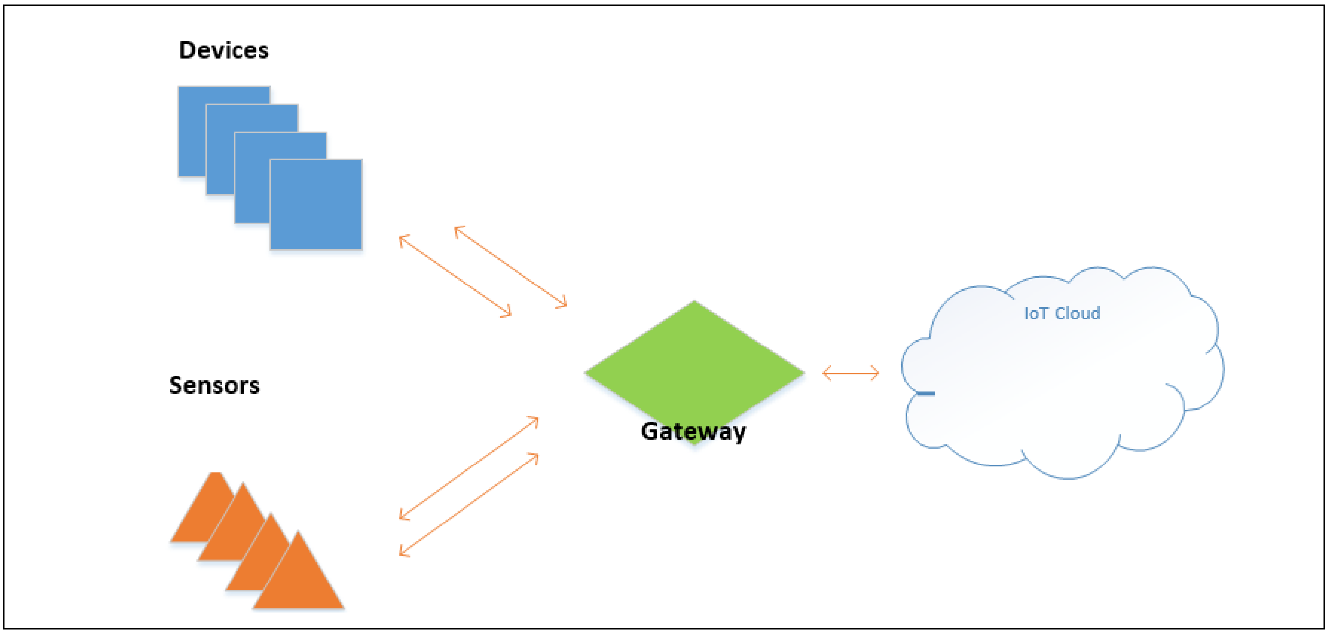

Figure 1. IoT Cloud Connectivity Architecture

Devices or Sensors

A device includes hardware and software that interacts directly with the world. Devices connect to a network to communicate with each other, or to centralized applications. Devices may connect to the Internet either directly or indirectly.

Gateway

A gateway enables devices that are not directly connected to the Internet to reach Cloud services. The data from each device is sent to the Cloud platform, where it is processed and combined with data from other devices, and potentially with other business-transactional data. Most of the common communication gateways support one or more communication technologies such as Wi-Fi, Ethernet, or Cellular to connect to the IoT Cloud service provider.

IoT Cloud

Many IoT devices produce lots of data. You need an efficient, scalable, affordable way to manage those devices, handle all that information, and make it work for you. When it comes to storing, processing, and analyzing data, especially big data, it is hard to surpass the Cloud.

1.2 Microsoft Azure IoT Solution

1.2.1 Overview

Microsoft's end-to-end IoT platform is a complete IoT offering so that enterprises can build and realize value from IoT solutions quickly and efficiently. Azure IoT Central solutions are used with backend support from the Azure IoT Hub Device Provisioning Service.

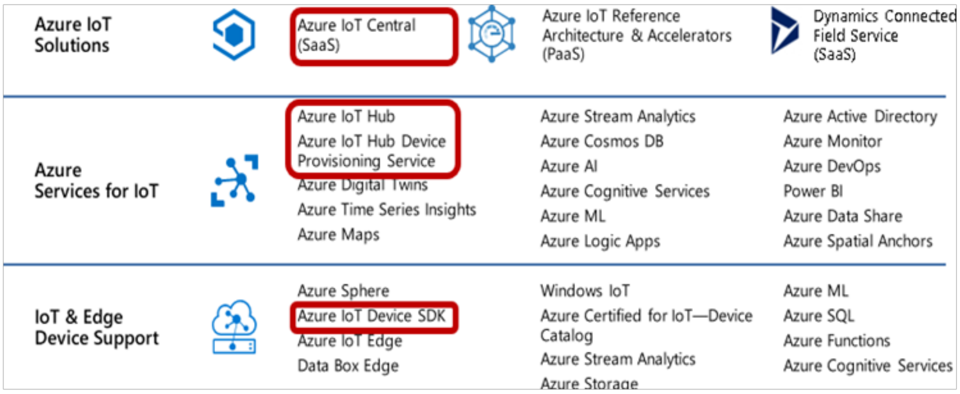

Figure 2. Microsoft Azure IoT Cloud Solution

1.2.2 IoT Hub Device Provisioning Service

1.2.2.1 Azure IoT Hub and IoT Hub Device Provisioning Service (DPS)

IoT Hub provides built-in support for the MQTT v3.1.1 protocol. See the following webpage for more understanding of the IoT Hub and Device Provisioning Services (DPS):
https://docs.microsoft.com/en-us/azure/iot-dps/

(1) Device Provisioning Service

A high-level sequence of events to connect a Device to the IoT Hub is as follows:

1. After the device is manufactured, the device enrollment information is added to the DPS. This is the only manual step in the process.
2. At some point afterward, which could be a day or several months, the device goes online and connects to DPS to find its IoT solution home.
3. DPS and the device go through an attestation handshake using the device enrollment information. DPS proves the device's identity.
4. DPS registers the device to the IoT hub and populates the initial desired device state.
5. IoT hub returns the connection info for the device.
6. DPS gives the device its IoT Hub connection information.
7. The device now establishes a connection with IoT Hub and retrieves its initial configuration from IoT Hub and makes any changes/updates, as needed.
8. The device starts sending telemetry to the IoT Hub.

(2) Embedded C SDK

The Embedded C SDK, the newer addition to the Azure SDKs family, was designed to allow embedded IoT devices to leverage Azure services, like device to Cloud telemetry, Cloud to device messages, direct methods, device twin, device provisioning, and IoT Plug and play, all while maintaining a minimal footprint.

It allows full control over memory allocation and the flexibility to bring your own MQTT client, TLS, and Socket layers.

Written in C, this version of the SDK is optimized to be used on small and embedded devices with limited capabilities and resources.

The Azure IoT SDK is open source and is published on GitHub (https://github.com/Azure/azure-sdk-for-c). This is also distributed with FSP version 5.3.0 and above.

1.2.3 Authentication Methods

Security is a critical concern when deploying and managing IoT devices. IoT Hub offers the security features described in the following sections.

1.2.3.1 X.509

The communication path between devices and Azure IoT Hub, or between gateways and Azure IoT Hub, is secured using the industry-standard Transport Layer Security (TLS) with Azure IoT Hub, authenticated using the X.509 standard.

To protect devices from unsolicited inbound connections, Azure IoT Hub does not open any connection to the device. The device initiates all connections.

1.2.3.2 Per-Device Key Authentication

Figure 3 shows authentication in the IoT Hub using security tokens.

Figure 3. Authentication using Security Tokens

1. The device prepares a shared access signature (SAS) token using the device endpoint, device id,and primary key (generated as part of the device addition to the IoT Hub).
2. When connecting to the IoT Hub, the device presents the SAS token as the password in the MQTT CONNECT message. The username content is the combination of the device endpoint and device name along with the additional Azure-defined string.
3. The IoT Hub verifies the SAS token and registers the device and connection is established.
4. IoT Hub provides a Symmetric key for Data encryption.
   Note: The connection is closed when the SAS token expires.

1.3 MQTT Protocol Overview

MQTT stands for **Message Queuing Telemetry Transport**. MQTT is a Client Server publish-subscribe messaging transport protocol. It is an extremely lightweight, open, simple messaging protocol, designed forconstrained devices, as well as low-bandwidth, high-latency, or unreliable networks. These characteristics make it ideal for use in many situations, including constrained environments, such as communication in machine-to-machine (M2M) and IoT contexts, where a small code footprint is required, and/or network bandwidth is at a premium.

An MQTT client can publish information to other clients through a broker. A client, if interested in a topic, can subscribe to the topic through the broker. A broker is responsible for authentication and authorization of clients, as well as delivering published messages to any of its clients who subscribe to the topic. In this publisher/subscriber model, multiple clients may publish data on the same topic. A client will receive the messages published if the client subscribes to the same topic.

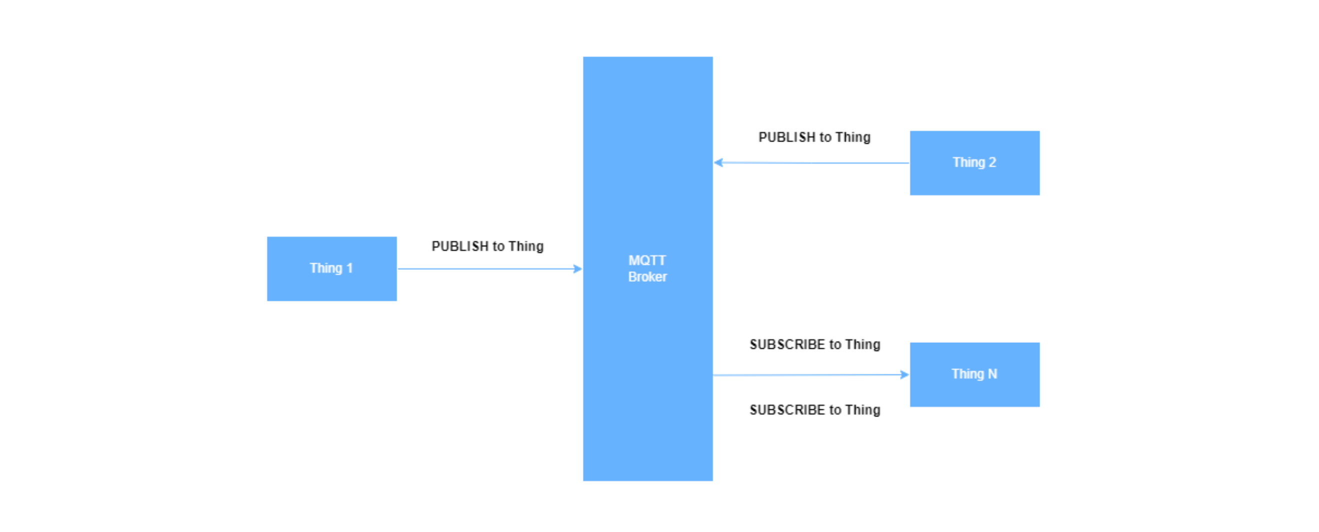

Figure 4. MQTT Client Publish/Subscribe Model

In the Pub/Sub model used by MQTT, there is no direct connection between a publisher and the subscriber.To handle the challenges of a Pub/Sub system, the MQTT generally uses quality of service (QoS) levels.

There are three QoS levels in MQTT:

- At most once (0)
- At least once (1)
- Exactly once (2)

At most once (0)

A message will not be acknowledged by the receiver or stored and redelivered by the sender.

At least once (1)

It is guaranteed that a message will be delivered at least once to the receiver. But the message can also bedelivered more than once. The sender will store the message until it gets an acknowledgment in the form of a PUBACK command message from the receiver.

Exactly once (2)

It guarantees that each message is received only once by the counterpart. It is the safest and the slowest QoS level.

1.4 TLS Protocol Overview

Transport Layer Security (TLS) protocol and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols that provide communications security over a computer network.

The TLS/ SSL protocol provides privacy and reliability between two communicating applications. It has the following basic properties:

Encryption: The messages exchanged between communicating applications are encrypted to ensure that the connection is private. A symmetric cryptography mechanism such as AES (Advanced Encryption Standard) is used for data encryption.

Authentication: A mechanism to check the peer’s identity using certificates.

Integrity: A mechanism to detect message tampering and forgery ensures that connection is reliable. A Message Authentication Code (MAC), such as the Secure Hash Algorithm (SHA), ensures message integrity.

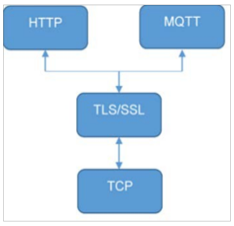

Figure 5. SSL/TLS Hierarchy

1.4.1 Device Certificates and Keys

Device certificates, public and private keys, and the ways they can be generated are discussed in this section.

Security is a critical concern when deploying and managing IoT devices. In general, each of the IoT devices needs an identity before they can communicate with the Cloud. Digital certificates are the most common method for authenticating a remote host in TLS. Essentially, a digital certificate is a document with specific formatting that provides identity information for a device.

TLS normally uses a format called X.509, a standard developed by the International Telecommunication Union (ITU), though other formats for certificates may apply if TLS hosts can agree on a format to use. X.509 defines a specific format for certificates and various encodings that can be used to produce a digital document. Most X.509 certificates used with TLS are encoded using a variant of ASN.1, which is another telecommunication standard. Within ASN.1 there are various digital encodings, but the most common encoding for TLS certificates is the Distinguished Encoding Rules (DER) standard. DER is a simplified subset of the ASN.1.

Though DER-formatted binary certificates are used in the actual TLS protocol, they may be generated and stored in a number of different encodings, with file extensions such as `.pem`, `.crt`, and `.p12`. The most common of the alternative certificate encodings is Privacy-Enhanced Mail (PEM). The PEM format is a base-64 encoded version of the DER encoding.

Depending on your application, you may generate your own certificates, be provided certificates by a manufacturer or government organization, or purchase certificates from a commercial certificate authority.

Loading Certificates onto your Device

To use a digital certificate in your NetX™ Secure application, you must first convert your certificate into a binary DER format, and optionally convert the associated private key into a binary format, typically, a PKCS#1-formatted, DER-encoded RSA key. Once converted, it is up to you how to load the certificate and the private key onto the device. Possible options include using a flash-based file system or generating a C array from the data (using a tool such as xxd from Linux® with the -i option) and compiling the certificate and key into your application as constant data.

Once your certificate is loaded on the device, you can use the TLS API to associate your certificate with a TLS session.

1.4.2 Device Security Recommendations

The following security recommendations are not enforced by Cloud IoT Core but will help you secure your devices and connections.

- The private key of the device should be kept secret.
- Use the latest version of TLS (v1.2 or above) when communicating with IoT Cloud and verify that the server certificate is valid using trusted root certificate authorities.
- Each device should have a unique public/private key pair. If multiple devices share a single key and one of those devices is compromised, an attacker could impersonate all the devices that have been configured with that one key.
- Keep the public key secure when registering it with Cloud IoT Core. If an attacker can tamper with the public key and trick the provisioner into swapping the public key and registering the wrong public key, the attacker will subsequently be able to authenticate on behalf of the device.
- The key pair is used to authenticate the device to Cloud IoT Core and should not be used for other purposes or protocols.

- Depending on the device's ability to store keys securely, key pairs should be rotated periodically. When practical, all keys should be discarded when the device is reset.
- If your device runs an operating system, make sure you have a way to securely update it. Android Things provides a service for secure updates. For devices that don't have an operating system, ensurethat you can securely update the device's software if security vulnerabilities are discovered after deployment.

2. RA FSP MQTT/TLS Cloud Solution

2.1 MQTT Client Module Introduction

The NetX Duo MQTT Client module provides high-level APIs for a Message Queuing Telemetry Transport (MQTT) protocol-based client. The MQTT protocol works on top of TCP/IP and therefore the MQTT client is implemented on top of NetX Duo IP and NetX Duo Packet pool. NetX Duo IP attaches itself to the appropriate link layer frameworks, such as Ethernet, Wi-Fi, or Cellular.

The NetX Duo MQTT client module can be used in normal or in secure mode. In normal mode, the communication between the MQTT client and broker is not secure. In secure mode, the communication between the MQTT client and broker is secured using the TLS protocol.

2.1.1 Design Considerations

- By default, the MQTT client does not use TLS; communication is not secure between a MQTT client and broker.
- The RA FSP Azure RTOS NetX Duo IoT middleware module provides the NetX Duo TLS session block. It adds Azure RTOS NetX Secure block. This block defines/controls the common properties of NetX secure.

2.1.2 Supported Features

NetX Duo MQTT Client supports the following features:

- Compliant with OASIS MQTT version 3.1.1 Oct 29, 2014. The specification can be found at http://mqtt.org/.
- Provides an option to enable/disable TLS for secure communication using NetX Secure in FSP.
- Supports QoS and provides the ability to choose the levels that can be selected while publishing the message.
- Internally buffers and maintains the queue of received messages.
- Provides a mechanism to register callback when a new message is received.
- Provides a mechanism to register callback when the connection with the broker is terminated.

2.2 TLS Session Module Introduction

The NetX Duo TLS session module provides high-level APIs for the TLS protocol-based client. It uses services provided by the RA FSP Crypto Engine (SCE) to carry out hardware-accelerated encryption anddecryption.

The NetX Duo TLS Session module is based on Azure RTOS NetX Secure which implements the SecureSocket Layer (SSL) and its replacement, TLS protocol, as described in RFC 2246 (version 1.0) and 5246(version 1.2). NetX Secure also includes routines for the basic X.509 (RFC 5280) format. NetX Secure is intended for applications using ThreadX RTOS in the project.

2.2.1 Design Considerations

- NetX Secure TLS performs only basic path validation on incoming server certificates.
  Once the basic path validation is complete, TLS then invokes the certificate verification callback supplied by the application.
- It is the responsibility of the application to perform any additional validation of the certificate.
  To help with the additional validation, NetX Secure provides X.509 routines for common validation operations, including DNS validation and Certificate Revocation List checking.
- Software-based cryptography is processor-intensive.
  NetX Secure software-based cryptographic routines have been optimized for performance but depending on the capabilities of the target processor, performance may result in very long operations. When hardware-based cryptography is available, it should be used for optimal performance of the NetX secure TLS.

- Due to the nature of embedded devices, some applications may not have the resources to support the maximum TLS record size of 16 KB.
  NetX Secure can handle 16 KB records on devices with sufficient resources.

2.2.2 Supported Features

- Support for RFC 2246 Transport Layer Security (TLS) Protocol Version 1.0
- Support for RFC 5246 TLS Protocol Version 1.2
- Support for RFC 5280 X.509 PKI Certificates (v3)
- Support for RFC 3268 Advanced Encryption Standard (AES) Cipher suites for TLS
- RFC 3447 Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version2.1
- RFC 2104 HMAC: Keyed-Hashing for Message Authentication
- RFC 6234 US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)
- RFC 4279 Pre-Shared Key Cipher suites for TLS

2.3 Azure IoT Device SDK Module Introduction

The Azure IoT device SDK is a set of libraries designed to simplify the process of developing IoT applications for Azure Cloud to make sending and receiving messages easy from the Azure IoT Hub service. There are different variations of the SDK, each targeting a specific platform, but in this application note, we will describe the Azure IoT device SDK for C.

The Azure IoT device SDK for C is written in ANSI C (C99) to maximize portability. This feature makes the libraries well-suited to operate on multiple platforms and devices, especially where minimizing disk and memory footprint is a priority.

In this application note, we will cover how to initialize the device library, send data to the IoT Hub, and receive messages from it.

More details on the Azure IoT Device SDK can be found in the reference link Azure IoT Hub device and service SDKs | Microsoft Learn

2.3.1 Design Considerations

The Azure IoT Device SDK is integrated with FSP and is available for customers to use. To add the SDKto the application, users are required to use the **Stacks** tab and select **Networking > Azure RTOS NetX Duo IOT Middleware.**

When the components are selected using the **Stacks** tab, and the project is created, the SDK and libraries can be seen under the `ra/microsoft/azure-rtos/netxduo/addons/azure_iot` and `ra/microsoft/azure-rtos/netxduo/addons/cloud` folders.

Note: In the following sections, the step-by-step procedure for adding the Azure IoT middleware is explained in detail.

2.3.2 Supported Features

Table 1. IoT SDK Supported features

Features	Descriptions
Send device-to-cloud messages	Send device-to-cloud messages to IoT Hub with the option to add custom message properties.
Receive cloud-to-device messages	Receive cloud-to-device messages and associated properties from IoT Hub.
Device twins	IoT Hub persists a device twin for each device that you connect to IoT Hub. The device can perform operations such as retrieving twin documents and subscribing to desired property updates.
Direct methods	IoT Hub gives you the ability to invoke direct methods on devices from the Cloud.
Device Provisioning Service (DPS)	This SDK supports connecting your device to the Device Provisioning Service, for example, through individual enrollment using an X.509 leaf certificate.
Protocol	The Azure SDK for Embedded C supports only MQTT.

Features	Descriptions
Retry policies	The Azure SDK for Embedded C provides guidelines for retries, but actual retries should be handled by the application.
IoT plug-and-play	IoT Plug and Play enables solution builders to integrate smart devices with their solutions without any manual configuration.

3. MQTT/TLS Application Example

3.1 Application Overview

This application project demonstrates the Renesas RA IoT Cloud Connectivity solution using the FSP and uses Microsoft® Azure as the cloud provider. Ethernet is the primary communication interface between the MQTT device and the Azure IoT Services.

The CK-RA6M5 v2 kit acts as an MQTT node and connects to the Azure IoT service using MQTT/TLS protocol over the Ethernet interface. The application periodically reads the onboard sensor values and publishes this information to the Azure IoT Hub. It also subscribes to a User LED state MQTT topic. You can turn the User LEDs ON/OFF by publishing the LED state remotely. This application reads the updated LED state and turns the User LEDs ON/OFF.

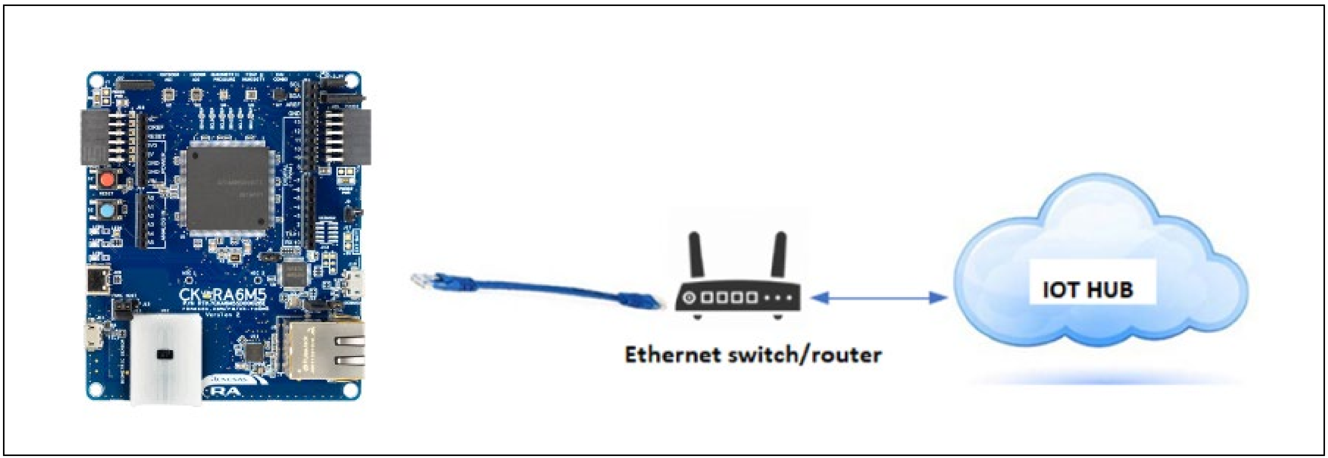

Figure 6. RA MQTT/TLS Application HW Connection Overview

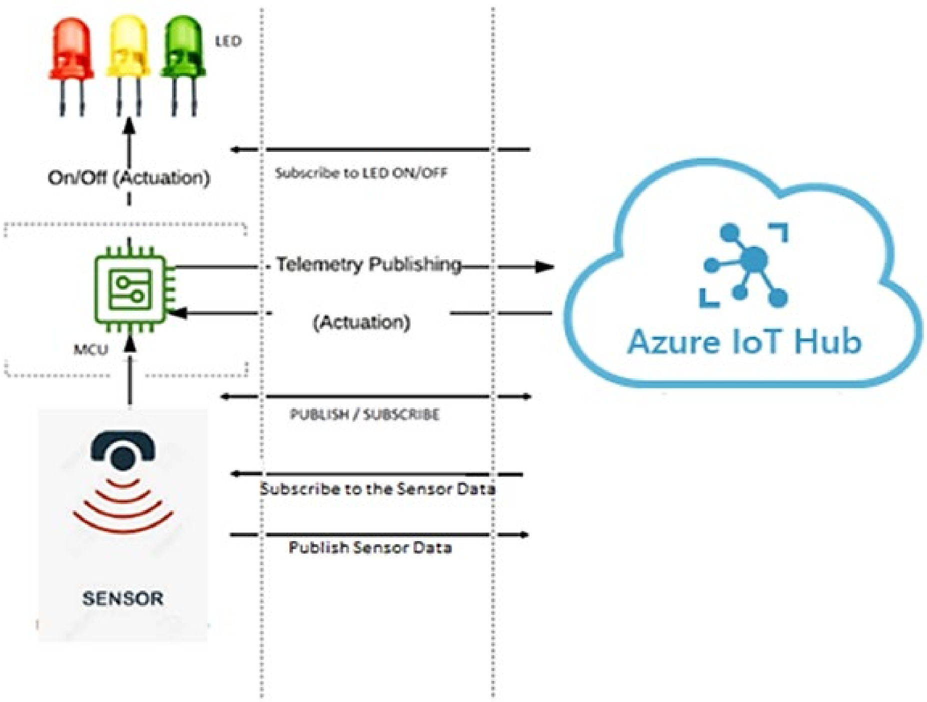

Figure 7. MQTT Publish/Subscribe to/from Azure IoT Central

The following files from this application project serve as a reference.

Table 2. Files Used in Application Project

No.	Filename	Purpose
1.	`src/application_thread_entry.c`	Contains initialization code and has the main thread used in the Cloud Connectivity application.
2.	`src/common_init.h`	Contains macros, data structures, and functions prototypes used to initialize common peripherals across the project.
3.	`src/common_utils.c`	Contains macros, data structures, and functions commonly used across the project.
4.	`src/common_utils.h`	Contains macros, data structures, and functions prototypes commonly used across the project.
5.	`src/Console_Thread_entry.c`	Contains the code for the command line interface and flash memory operations.
6.	`src/ICM42605.c`	Contains the code for the 6-Axis MEMS Motion Tracking™ Sensor (Gyroscope, Accelerometer)
7.	`src/ICM42605.h`	Contains the Data structure function prototypes for the 6-Axis MEMS Motion Tracking™ Sensor (Gyroscope, Accelerometer)
8.	`src/RA_ICM42605.c`	Contains codes for 6 Axis sensor (Gyroscope, Accelerometer) sensor's initialization and measurement.

No.	Filename	Purpose
9.	`src/icm.h`	Contains user-defined data types and function prototypes which have implementation in `RA_ICM42605.c`
10.	`src/ICP_20100.c`	Contains the code for Barometric Pressure and Temperature Sensor
11.	`src/ICP_20100.h`	Contains the Data structure and function prototypes for Barometric Pressure and Temperature Sensor
12.	`src/RA_ICP20100.c`	Contains codes for Barometric Pressure and Temperature sensor's initialization and measurement.
13.	`src/icp.h`	Contains user-defined data types and function prototypes which have an implementation in `RA_ICP20100.c`
14.	`src/Sensor_Thread_entry.c`	Contains the Code to access the Sensor data from the different sensors and order topic to publish
15.	`src/OB_1203_Thread_entry.c`	Contains the code for Heart Rate, Blood Oxygen Concentration, Pulse Oximetry, Proximity, Light and Color Sensor
16.	`src/oximeter.c`	Contains data structures and functions used for the oximeter sensor
17.	`src/oximeter.h`	Contains the Data structure and function prototypes for the oximeter sensor
18.	`src/r_typedefs.h`	Contains the common derived data types
19.	`src/RA_HS3001.c`	Contains the code for the Renesas Relative Humidity and Temperature Sensor
20.	`src/RA_HS3001.h`	Contains function prototypes for Relative Humidity and Temperature Sensor
21.	`src/RA_ZMOD4XXX_Common.c`	Contains the common code for Renesas ZMOD sensors
22.	`src/RA_ZMOD4XXX_Common.h`	Contains the common data structure's function prototypes for the Renesas ZMOD sensors
23.	`src/RA_ZMOD4XXX_IAQ1stGen.c`	Contains the common code for the Renesas ZMOD Internal Air Quality sensors
24.	`src/RA_ZMOD4XXX_OAQ1stGen.c`	Contains the common code for the Renesas ZMOD Outer Air Quality sensors
25.	`src/RmcI2C.c`	Contains the I2C wrapper functions for the third-party sensors not integrated with FSP
26.	`src/RmcI2C.h`	Contains the I2C function prototypes for wrapper functions for the third-party sensors not integrated with FSP
27.	`src/user_choice.h`	Contains the Function prototypes for the Sensor and its user configuration for the different sensors and its data accessibility.
28.	`src/usr_config.h`	To customize the user configuration to run the application.
29.	`src/usr_hal.c`	Contains data structures and functions used for the Hardware Abstraction Layer (HAL) initialization and associated utilities.
30.	`src/usr_hal.h`	Accompanying header for exposing functionality provided by `usr_hal.c`.

No.	Filename	Purpose
31.	src/usr_network.c	Contains data structures and functions used tooperate the NetX Duo TCP/IP and Ethernet Module. This file is for Ethernet-specific usage.
32.	src/usr_network.h	Accompanying header for exposing functionality provided by usr_network.c. This file is for Ethernet-specific use.
33.	src/ZMOD4410_Thread_entry.c	Contains the code for the indoor air quality sensor
34.	src/sample_pnp_environmental_sensor_component.c	PNP Telemetry for HS3001 Temperature sensor data
35.	src/sample_pnp_gas_component.c	PNP Telemetry for ZMOD4410 IAQ Sensor Data
36.	src/sample_pnp_barometric_pressure_sensor_component.c	PNP Telemetry for ICP20100 Pressure Sensor data
37.	src/sample_pnp_inertial_sensor_component.c	PNP Telemetry for ICM42605 Inertial Sensor data
38.	src/sample_pnp_gas_oaq.c	PNP Telemetry for ZMOD4510 OAQ Sensor Data
39.	src/sample_pnp_biometric_sensor_component.c	PNP Telemetry for OB1203 Biometric Sensor Data
40.	src/ZMOD4510_Thread_entry.c	Reading Outdoor Air Quality Data
41.	src/console_menu/console.c	Contains data structures and functions used to print data on the console using UART
42.	src/console_menu/console.h	Contains the Function prototypes used to print data on the console using UART
43.	src/console_menu/menu_flash.c	Contains data structures and functions used to provide CLI flash memory-related menu
44.	src/console_menu/menu_flash.h	Contains the Function prototypes and macros used to provide CLI flash memory-related menu
45.	src/console_menu/menu_kis.c	Contains functions to get the FSP version, get UUID, and help option for the main menu on the CLI
46.	src/console_menu/menu_kis.h	Contains the Function prototypes and macros used to get the FSP version, get UUID, and help option for the main menu on the CLI
47.	src/console_menu/menu_main.c	Contains data structures and functions used to provide CLI main menu options
48.	src/console_menu/menu_main.h	Contains the Function prototypes and macros used to provide CLI main menu options
49.	src/flash/flash_hp.c	Contains data structures and functions used to perform flash memory-related operations
50.	src/flash/flash_hp.h	Contains the function prototypes and macros used to perform flash memory-related operations
51.	src/ob1203_bio/KALMAN/kalman.c	Contains algorithm for Heart Rate, Blood Oxygen Concentration, Pulse Oximetry, Proximity, Light and Color Sensor sample calculations
52.	src/ob1203_bio/KALMAN/kalman.h	
53.	src/ob1203_bio/SAVGOL/SAVGOL.c	
54.	src/ob1203_bio/SAVGOL/SAVGOL.h	
55.	src/ob1203_bio/SPO2/SPO2.c	
56.	src/ob1203_bio/SPO2/SPO2.h	

No.	Filename	Purpose
57.	src/ob1203_bio/ob1203_bio.c	Contains codes for the OB1203 sensor's implementation to use with FSP stacks.
58.	src/ob1203_bio/ob1203_bio.h	Contain user data structure and function prototypes used in ob1203_bio.c
59.	src/SEGGER_RTT/SEGGER_RTT.c	Implementation of SEGGER real-time transfer (RTT) which allows real-time communication on targets which support debugger memory accesses while the CPU is running.
60.	src/SEGGER_RTT/SEGGER_RTT.h	
61.	src/SEGGER_RTT/SEGGER_RTT_Conf.h	
62.	src/SEGGER_RTT/SEGGER_RTT_printf.c	
63.	src/nx_azure_iot_cert.c	Azure IoT Interface code. These have the reference to the working sample implementation and other features such as Device Twin and Direct Method. These files can be used as a reference for developing the application
64.	src/nx_azure_iot_cert.h	
65.	src/nx_azure_iot_ciphersuites.c	
66.	src/nx_azure_iot_ciphersuites.h	
67.	src/sample_azure_iot_embedded_sdk.c	
68.	src/sample_config.h	
69.	src/usr_app.c	Contains data structures and functions used to operate the user application functions.
70.	src/usr_app.h	Accompanying header for exposing functionalityprovided by usr_app.c.
71.	src/base64_decode.c	Contains function used for BASE64 to Hex Conversion
72.	src/base64.h	Contains function prototype used for BASE64 to Hex Conversion
73.	src/c2d_thread_entry.c	Contains data structures functions and main thread used in Cloud to Device message handling.
74.	src/hal_entry.c	Auto-generated unused file for Non-RTOS thing.
75.	src/commandRX_Thread_entry.c	Cloud to Device Commands reception

3.2 Creating the Application Project using the FSP Configurator

Note: Skip this section, if you are planning to import, build, and run the project attached with this App note.

Complete the steps to create the project from the start using the e^2 studio and FSP configurator. The following table shows the step-by-step process of creating the project. It is assumed that the user is familiar with the e^2 studio and FSP configurator. Launch the installed e^2 studio for the FSP.

Table 3. Step-by-step Details for Creating the Application Project

	Steps	Intermediate Steps
1	Project Creation:	**File → New → Renesas C/C++ Project → Renesas RA**
2	Project Template: Templates for Renesas RA Project	**Renesas RA C/C++ Project → Next**
3	e^2 studio - Project Configuration: Renesas RA C/C++ Project Project Name and Location	**Project Name (Name for the project of your choice) → Next**
4	**Device and Tools Selection**	
	Device Selection	**FSP Version: 5.3.0 (or higher)**
		Board: CK-RA6M5 V2
		Device: R7FA6M5BH3CFC
		Language: C
5	**Toolchains**	**Toolchain: GNU ARM Embedded** (Default)
		Toolchain version: 13.2.1.arm-13-7
		Debugger: J-Link ARM **→ Next**
6	**Project Type Selection**	**Flat (Non-TrustZone) Project** **→ Next**
7	**Build Artifact and RTOS Selection**	**Build Artifact Selection: Executable** **RTOS Selection: Azure RTOS ThreadX (v6.2.1+fsp.5.3.0)** **→ Next**
8	**Project Template Selection**	**Azure RTOS ThreadX – Minimal → Finish**
9	**Clock**	**HOCO 20MHz →PLL Src: HOCO → PLL Div/2 →PLL Mul x20.0 → PLL200MHz**
10	**Stacks** tab (Part of the **FSP Configurator**)	**Threads → New Thread**
11	Configure **Properties → Thread**	**Symbol: application_thread**
		Name: Application Thread
		Stack size (bytes): 0x2400
		Priority: 1
		Auto start: Disabled
		Time slicing interval (ticks): 25
		Note: The stack size of the application thread needs to be a minimum of 0x1000 bytes or greater. This is the requirement for theNetX Duo Crypto use.
12	Adding the NetX DHCP, IoT Middleware, SNTP Clients, and Packet Pool to the Application Thread Keep the default names **g_dhcp_client0**, **g_dns0**, **g_sntp_client0**. The default configuration providedby FSP configurator is used, so there is no need to change any of the specific configurations in the **Property** window.	
	Adding DHCP Client	
	New Stack	**Networking → Azure RTOS NetX Duo DHCP IPv4 Client**
	Adding Packet Pool for the DHCP Client	Click on **Add NetX Duo Packet Pool → Use→ g_packet_pool0 Azure RTOS NetX Duo Packet Pool Instance**

	Adding NetX Duo Network Driver	Click on **Add NetX Duo Network Driver → New → NetXDuo Ethernet Driver**
		g_ether_phy0 Ethernet →PHY-LSI Address →5
	Property Settings for g_ether_phy0 Ethernet → **Pins**	**ET0_LinkSTA: None** **ET0_WOL: None**
	Modifying the **BSP** tab → **Properties** → **RA Common** (for Main stack and Heap Settings)	
	Property settings for **RA Common**	**Main stack size(bytes): 0x1000**
		Heap size (bytes): 0x1000
	Adding Azure RTOS NetX Duo IoT Middleware	
	New Stack	**Networking → Azure RTOS NetX Duo IoT Middleware**
	Adding NetX Duo IP instance for DNS Client	Click on **Add NetX Duo IP Instance → Use → g_ip0 Azure RTOS NetXDuo IP Instance**
	Adding Packet Pool for the DNS Client	Click on **Add NetX Duo Packet Pool →Use → g_packet_pool0 Azure RTOS NetX Duo Packet Pool Instance**
13	Note: After the Azure IoT Middleware is added, the configurator reports following errors when you hover over the red Blocks. **Error: NetX Duo Azure IoT Middleware Requires NetX Secure to be enabled.** **Error: NetX Duo Azure IoT Middleware Requires IP Packet Filter to be enabled.** **Error: NetX Duo Azure IoT Middleware Requires MQTT Cloud to be enabled.** **Error: A NetX Crypto Implementation must be added.** Note: To fix these errors, enable them as explained in the following steps	
	Enable the NetX Secure	**g_dns0 Azure RTOS NetX Duo DNS Client →Property → Common → MQTT → Client → NX Secure: Enable**
	Enable MQTT Cloud	**g_dns0 Azure RTOS NetX Duo DNS Client →Property → Common → MQTT → Client → Cloud Enable: Enable**
	Enable IP Packet Filter	**g_dns0 Azure RTOS NetX Duo DNS Client →Property → Common → Common → IP Packet Filter: Enabled**
	Add NetX Crypto Implementation	**Click on Add NetX Crypto SW Only or HW/SW Implementation →** **New → Azure RTOS NetX Crypto HW Acceleration**
	Enable the Extended Notify Support	**g_dns0 Azure RTOS NetX Duo DNS Client →Property →Common → Common →Extended Notify Support: Enabled**
14	NetX Secure Component is added from the HW Crypto perspective. IoT SDK also works with SW crypto. But in this application, the HW Crypto Accelerators are used.	
	Configure **Azure RTOS NetX Secure** property values (Only values that changed from the default are shown here)	
	PSK Cipher Suite	**Enable**
	ECC Cipher Suite	**Enable**
	TLSv1.0	**Enable**
	TLSv1.1 Legacy Mode	**Enable**
	TLSV1.1	**Enable**
	TLSV1.3	**Disable**
	Server Mode	**Enable**
	Configure **Azure RTOS NetX Crypto HW Acceleration** property values (Only values that changed from the default are shown here)	
	Common→Hardware Acceleration→Public Key Cryptography (PKC)→ RSA→RSA	**Use Hardware**
	Common→Hardware Acceleration→ Public Key Cryptography (PKC)→ RSA→RSA 3072 Verify/Encryption (HW)	**Enabled**

	Common→Hardware Acceleration → Public Key Cryptography (PKC) → RSA → RSA 4096 Verify/Encryption (HW)	**Enabled**
	Common→Hardware Acceleration → Public Key Cryptography (PKC) → RSA → RSA Scratch Buffer Size	**Disabled (HW)**
	Common→ Standalone Usage	**Use with TLS**
	Note: Increase the Stack size in the BSP Tab to get rid of the error in configurator for NetX Crypto HW Acceleration	Refer to the Modifying the **BSP** tab → **Properties** → **RA Common** for (Main stack and Heap Settings) section in step 11 of this table Note: For crypto operation it is recommended to have a stacksize of 4K or more**.**
	Adding SNTP Client	
	New Stack	**Networking → Azure RTOS NetX Duo SNTP Client**
	Adding NetX Duo IP instance for SNTP Client	Click on **Add NetX Duo IP Instance →Use → g_ip0 Azure RTOS NetXDuo IP Instance**
	Adding Packet Pool for the SNTP Client	Click on **Add NetX Duo Packet Pool →Use → g_packet_pool0 Azure RTOS NetX Duo Packet Pool Instance**
15	Increase the **Number of Packets in Pool**	
		Click on **g_packet_pool0 Azure RTOS NetX Duo Packet Pool Instance → Property → Module g_packet_pool0 Azure RTOS NetX Duo Packet Pool Instance → Number of Packets in Pool.** Change from **16** to **50** (To allow enough buffer for the packets). This can be tuned based on the frequency and size
	Note: After adding the SNTP the configurator reports the following errors when you hover over the red Blocks. **Error: Maximum time adjustment (milliseconds) should be greater than unicast poll interval (seconds).** Note: To fix these errors, enable them as explained in the following steps	
	Reduce the starting poll interval for unicast update request (seconds)	**g_sntp_client0 Azure RTOS NetX Duo SNTP Client → Property → Common → SNTP → Client →Starting poll interval for unicast update request (seconds): 36**
16	Add Cloud to Device Processing Thread to the Application	
	Stacks tab (Part of the **FSP Configurator**)	**Threads → New Thread**
	Configure Thread Properties	
	Symbol	**c2d_thread**
	Name	**Cloud2Device Thread**
	Stack size (bytes)	**2048**
	Priority	**1**
	Auto start	**Disabled**
	Time slicing interval (ticks)	**25**
17	Adding the HAL Modules as required for the Application Project: GPT Timer0 for control publishing sensor value into MQTT and display to console.	
	HAL/Common Stacks → New Stack	**Timers → Timer, General PWM on r_gpt**
	Property Settings for r_gpt → **General**	**Name: gpt**
		Channel: 0
		Mode: Periodic
		Period: 1
		Period Unit: Seconds
	Interrupts:	**Callback: g_gpt_timer_cb**

		Overflow/Crest Interrupt Priority: Priority 10
18	Adding Azure RTOS Objects for the Application (Topic Queue needs to be created for the application –Message Queue)	
	Stacks Tab → Objects	**New Object → Queue**
	Property Settings for the Queue	**Name: Topic Queue**
		Symbol: g_topic_queue
		Message Size (Words): 16
		Queue Size (Bytes): 64
	Stacks Tab → Objects	**New Object → Mutex**
		Name: consolprint_mutex
		Symbol: consolprint_mutex
		Priority Inheritance: Disabled
	Stacks Tab → Objects	**New Object → Queue**
	Property Settings for the Queue	**Name: HS3001 Queue**
		Symbol: g_hs3001_queue
		Message Size (Words): 2
		Queue Size (Bytes): 8
	Stacks Tab → Objects	**New Object → Queue**
	Property Settings for the Queue	**Name: ZMOD4410 Queue**
		Symbol: g_iaq_queue
		Message Size (Words): 3
		Queue Size (Bytes): 12
	Stacks Tab → Objects	**New Object → Queue**
	Property Settings for the Queue	**Name: ICM Queue**
		Symbol: g_icm_queue
		Message Size (Words): 12
		Queue Size (Bytes): 48
	Stacks Tab → Objects	**New Object → Queue**
	Property Settings for the Queue	**Name: OB1203 Queue**
		Symbol: g_ob1203_queue
		Message Size (Words): 3
		Queue Size (Bytes): 12
	Stacks Tab → Objects	**New Object → Queue**
	Property Settings for the Queue	**Name: ZMOD4510 Queue**
		Symbol: g_oaq_queue
		Message Size (Words): 1
		Queue Size (Bytes): 4
	Stacks Tab → Objects	**New Object → Queue**
	Property Settings for the Queue	**Name: ICP Queue**
		Symbol: g_icp_queue
		Message Size (Words): 4
		Queue Size (Bytes): 16
19	Add Sensor Thread, this thread used to access sensor's values of HS3001, ICP-20100 and ICM-42605	
	Stacks tab (Part of the **FSP Configurator**)	**Threads → New Thread**
	Configure Thread Properties	
	Symbol	**Sensor_Thread**
	Name	**Sensor_Thread**
	Stack size (bytes)	**8192**

	Priority	**3**
	Auto start	**Disabled**
	Time slicing interval (ticks)	**200**
20	Adding the HS300X Temperature/Humidity Sensor Module to the Sensor Thread	
	New Stack →	**Sensor → HS300X Temperature/Humidity Sensor**
	Config HS300X sensor→	Name: g**_hs300x_sensor0**
		Callback: **hs300x_callback**
	Under **I2C Shared Bus → Add I2C Communications Peripheral →**	**New → I2C Master(r_iic_master)**
	Config for I2C Master →	Name: **g_i2c_master0**
		Channel: **0**
		Rate: **Fast-mode**
		Interrupt Priority Level: **Priority 12**
21	Adding ICP-20100 and ICM-42605 sensors to the Sensor Thread. Note: FSP doesn't provide an integrated module for ICP-20100 and ICM-42605 sensors. This needs to be integrated via the i2c communication device and external IRQ manually. Also, its related sensor driver code needs to be added to the src folder.	
	New Stack →	**Connectivity → I2C Communication Device**
	Config I2C Communication Device →	Name: **g_comms_i2c_device4**
		Slave Address: **0x63**
		Callback: **ICP_comms_i2c_callback**
	Under the **I2C Communication Device → Add I2C Shared Bus →**	**Use → g_comms_i2c_bus0 I2C Shared Bus**
	New Stack →	**Input → External IRQ**
	Config for External IRQ	Name: **g_external_irq6**
		Channel: **6**
		Trigger: **Falling**
		Callback: **ICP_IRQ_CALLBACK**
22	Adding I2C Communication Device and External IRQ for ICM-42605 into Sensor Thread	
	New Stack →	**Connectivity → I2C Communication Device**
	Config I2C Communication Device →	Name: **g_comms_i2c_device5**
		Slave Address: **0x68**
		Callback: **ICM_comms_i2c_callback**
	Under the **I2C Communication Device → Add I2C Shared Bus →**	**Use → g_comms_i2c_bus0 I2C Shared Bus**
	New Stack →	**Input → External IRQ**
	Config for External IRQ	Name: **g_external_irq3**
		Channel: **3**
		Trigger: **Falling**
		Callback: **ICM_42605_Callback2**
	New Stack →	**Input → External IRQ**
	Config for External IRQ	Name: **g_external_irq12**
		Channel: **12**
		Trigger: **Falling**
		Callback: **ICM_42605_Callback1**
23	Add ZMOD4410 Sensor (IAQ) Processing Thread to the Application	
	Stacks tab (Part of the **FSP Configurator**)	**Threads → New Thread**
	Configure Thread Properties	
	Symbol	**ZMOD4410_Thread**
	Name	**ZMOD4410_Thread**
	Stack size (bytes)	**2048**

	Priority	**3**
	Auto start	**Disabled**
	Time slicing interval (ticks)	**1**
24	Adding ZMOD4XXX Gas Sensor Module to ZMOD4410_Thread	
	New Stack →	**Sensor → ZMOD4XXX Gas Sensor**
	Config ZMOD4XXX Properties→	**Add Requires ZMOD Library→ New→ZMOD4410 IAQ 1st Generation**
		Add I2C Shared Bus→ New→ I2C Shared Bus
		Add I2C Communications→New→ I2C Master (r_iic_master)
		Add IRQ Driver for Measurement →New→ External IRQ
	Module g_zmod4xxx_sensor0	**Name: g_zmod4xxx_sensor0**
		Comms I2C callback: zmod4xxx_comms_i2c_callback
		IRQ Callbacks: zmod4xxx_irq0_callback
	Under the **ZMOD4410 IAQ 1st Generation → I2C Communication Device →**	Name: **g_comms_i2c_device1**
	Config I2C Shared bus →	Name: **g_comms_i2c_bus2**
	Config I2C Master →	Name: **g_i2c_master2**
		Channel: **2**
		Rate: **Fast-mode**
		Interrupt Priority Level: **Priority 12**
	Config External IRQ→	**Name: g_external_irq4**
		Channel :4
		Trigger: Falling
		Pin Interrupt Priority: Priority 3
		Pins→IRQ04: (Navigate to IRQ04): P402
25	Add ZMOD4510 Sensor (OAQ) Processing Thread to the Application	
	Stacks tab (Part of the **FSP Configurator)**	**Threads → New Thread**
	Configure Thread Properties	
	Symbol	**ZMOD4510_Thread**
	Name	**ZMOD4510_Thread**
	Stack size (bytes)	**2048**
	Priority	**2**
	Auto start	**Disabled**
	Time slicing interval (ticks)	**1**
26	Adding ZMOD4XXX Gas Sensor Module to ZMOD4510_Thread	
	New Stack →	**Sensor → ZMOD4XXX Gas Sensor**
	Config ZMOD4XXX Gas Sensor Properties→	**Add Required ZMOD Library→ New→ZMOD4510 OAQ 1st Generation**
		Add I2C Shared Bus→Use→g_comms_i2c_bus2 I2C Shared Bus
		Add IRQ Driver for Measurement→New→ External IRQ
	Module g_zmod4xxx_sensor1	**Name: g_zmod4xxx_sensor1**
		Comms I2C callback: zmod4xxx_comms_i2c1_callback
		IRQ Callbacks: zmod4xxx_irq1_callback
	Module g_comms_i2c_device2 I2C Communication Device (rm_comms_i2c)	**Name: g_comms_i2c_device2**
	Config External IRQ→	**Name: g_external_irq15**
		Channel :15
		Trigger: Falling
		Pin Interrupt Priority:12
		Pins→IRQ15: (Navigate to IRQ15): P404

27	Add OB1203 (optical biosensor) Processing Thread to the Application	
	Stacks tab (Part of the **FSP Configurator**)	**Threads → New Thread**
	Configure Thread Properties	
	Symbol	**OB_1203_Thread**
	Name	**OB_1203_Thread**
	Stack size (bytes)	**2048**
	Priority	**2**
	Auto start	**Disabled**
	Time slicing interval (ticks)	**25**
28	Add the OB1203 sensor module, PPG mode to the OB_1203_Thread.	
	New Stack →	**Sensor → OB1203 Light/Proximity/PPG Sensor**
	Config OB1203 Light/Proximity/PPG Sensor →	Name: **g_ob1203_sensor0**
	Under the OB1203 Light/Proximity/PPG Sensor → Add Requires OB1203 Operation mode →	**New → OB1203 PPG mode**
	Under the OB1203 PPG mode → I2C Communication Device →	Name: **g_comms_i2c_device3**
	Under the I2C Communication Device → Add I2C Share Bus →	**New → I2C Shared Bus**
	Config **I2C Shared Bus →**	Name: **g_comms_i2c_bus1**
	Under **I2C Shared Bus → Add I2C Communications Peripheral →**	**New → I2C Master (r_iic_master)**
	Config **I2C Master →**	Name: **g_i2c_master1**
		Channel: **1**
		Rate: **Standard**
		Interrupt Priority Level: **Priority 12**
	Under the OB1203 Light/Proximity/PPG Sensor → Add IRQ Driver for measurement →	**New → External IRQ**
	Config for External IRQ →	Name: **g_external_irq14**
		Channel: **14**
		Trigger: **Falling**
		Pins→IRQ14: (Navigate to IRQ14): P403
29	Add the OB1203 sensor module, Proximity mode to the OB_1203_Thread.	
	New Stack →	**Sensor → OB1203 Light/Proximity/PPG Sensor**
	Config OB1203 Light/Proximity/PPG Sensor →	Name: **g_ob1203_sensor1**
	Under the OB1203 Light/Proximity/PPG Sensor → Add Requires OB1203 Operation mode →	**New → OB1203 Proximity mode**
	Under the OB1203 Proximity mode → I2C Communication Device →	Name: **g_comms_i2c_device6**
	Under the I2C Communication Device → Add I2C Share Bus →	**Use → g_comms_i2c_bus1 I2C Shared Bus**
	Under the OB1203 Light/Proximity/PPG Sensor → Add IRQ Driver for measurement →	**Use → g_external_irq14 External IRQ**
30	Add Cloud to Device Processing Thread to the Application	
	Stacks tab (Part of the **FSP Configurator**)	**Threads → New Thread**
	Configure Thread Properties	

	Symbol	**Console_Thread**
	Name	**Console_Thread**
	Stack size (bytes)	**4096**
	Priority	**4**
	Auto start	**Enabled**
	Time slicing interval (ticks)	**50**
31	Add Cloud to Device Command Reception Thread to the Application	
	Stacks tab (Part of the **FSP Configurator**)	**Threads → New Thread**
	Configure Thread Properties	
	Symbol	**CommandRX_Thread**
	Name	**CommandRX_Thread**
	Stack size (bytes)	**2048**
	Priority	**4**
	Auto start	**Disabled**
	Time slicing interval (ticks)	**40**
32	Adding UART to Console_Thread	
	New Stack →	**Connectivity→ UART**
	Config Common →	**FIFO Support: Enable**
		DTC Support: Enable
		Flow Control Support: Enable
	Config General →	**Name: g_console_uart**
		Channel: 5
		Data Bits: 8bits
		Parity: None
		Stop Bits: 1bit
	Config Baud→	**Baudrate: 115200**
	Config Interrupts →	**Callback: user_uart_callback**
	Config Pins →	**TXD: P501**
		RXD: P502
33	Adding Flash to Console_Thread	
	New Stack →	**Storage→ Flash (r_flash_hp)**
		Name: user_flash
		Data Flash Background Operation: Disabled
		Callback: flash_callback
		Flash Ready Interrupt Priority: Priority 6
		Flash Error Interrupt Priority: Priority 6
34	Enable "Use float with nano printf" to print float values.	
	Project → Properties → C/C++ Build → Settings → Tool Settings tab → GNU ARM Cross C Linker → Miscellaneous →Check the box	Use float with nano printf (-u _printf_float)
35	Add "--specs=rdimon.specs" to Other linker flags	
	Project → Properties → C/C++ Build → Settings → Tool Settings tab → GNU ARM Cross C Linker → Miscellaneous → Other linker flags →	Add **--specs=rdimon.specs → Apply → Apply and Close**

The above configuration is a prerequisite to generate the required stack and features for the Cloud connectivity application provided with this application note. Once the **Generate Project Content** button is clicked,e^2 studio generates the source code for the project. The generated source code contains the required drivers, stacks, and middleware. The user application files must be added to the src folder.

For the validation of the created project, the same source files listed in the section 3, MQTT/TLS Application Example, Table 2, may be added. This is the quickest way to create and build the application without writing the code for the configuration created in the above section.

Note: After you follow the instructions in section 3.2 to recreate the Application project using the FSP configuratorand add the src code to the project, the project is ready for building.

Note: If you get an error while assigning PIN to External IRQ, go to **Pin Configuration > Pin Number** and select the IRQ function for that pin number for example, for External IRQ channel number 4, you can select Function IRQ14 for Pin Number 4.

Note: As part of the manual creation of this project, you might encounter known issues/pin errors with the Pin configurator while selecting the peripherals. We recommended selecting the operation mode, disabling/enabling, and selecting the pins. You can also refer to the attached project as a working reference.

3.3 Install Azure CLI

To prepare Azure Cloud resources and connect a device to Azure, you can use Azure CLI. Azure CLI can be installed locally on your PC.

1. Azure CLI can be downloaded from the Microsoft site (https://learn.microsoft.com/en-us/cli/azure/install-azure-cli)
2. The installer name will be similar to `azure-cli-2.44.x.msi`. or later. Click on the installer and the install shield will guide you through the installation process. Install it to your desired directory. For example, `c:\AzureCLI`
3. Install the current release of the Azure CLI. After the installation is complete, you will need to close and reopen any active Windows Command Prompt or PowerShell windows to use the Azure CLI.
4. After the Azure CLI installation is successful, open and launch the Windows PowerShell to use the Azure CLI. A screenshot of the Windows PowerShell is shown below.

Figure 8. Windows Power Shell

5. If you already have Azure CLI installed locally, go to the directory of the installed AzureCLI and run *az - - version* to check the version. This application note requires Azure CLI 2.44.0 or later.

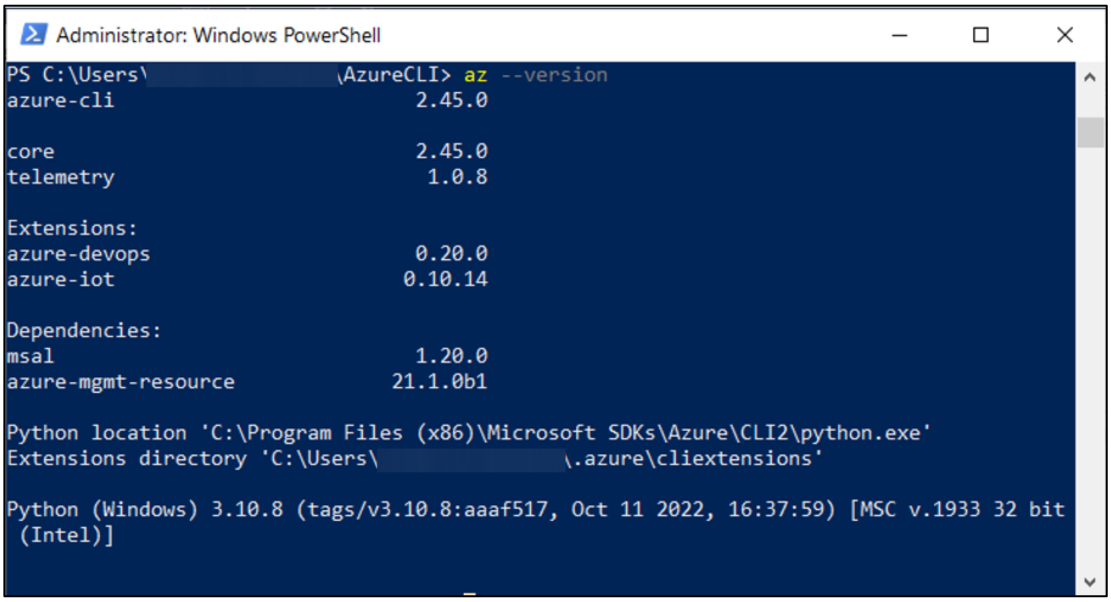

Figure 9. Azure CLI Version

3.4 Create an IoT Hub

You can use Azure CLI to create an IoT hub that handles events and messaging for your device.

Note 1: Before you start creating the IoT Hub, you are required to have a login to your Azure Portal via a webbrowser. If not logged in, then you may notice an error that you are not logged in, while creating the IoT Hub:
https://portal.azure.com/

Note 2: If you do not have an Azure Account, you can create one that is valid for 12 months with limited features from the following link:
https://azure.microsoft.com/en-us/free/

To create an IoT hub:

Note 3: Some of the user parameters while creating the IoT Hub need to be unique. Users are required to take care of this while creating the IoT Hub credentials.

1. In your CLI console, run the “az extension add” command to add the Microsoft Azure IoT Extension for Azure CLI to your CLI shell. The IoT Extension adds IoT Hub, IoT Edge, and IoT Device Provisioning Service (DPS) specific commands to Azure CLI.
   — `az extension add --name azure-iot`

Note 4: When you run the command for the first time you may not notice output on the console as shown below. It just accepts the command.

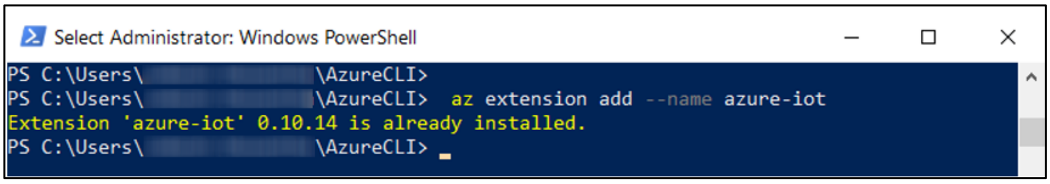

Figure 10. Add Extension for Azure CLI

2. Run the `az login` command to login to the Azure account. Running the `az login` command opens the browser for login. You can enter the login credentials to login to the Azure account. You will notice a similar message on the browser on successful login.

   Note: You can find more info on the Azure CLI at [Overview of the Azure CLI | Microsoft Docs]

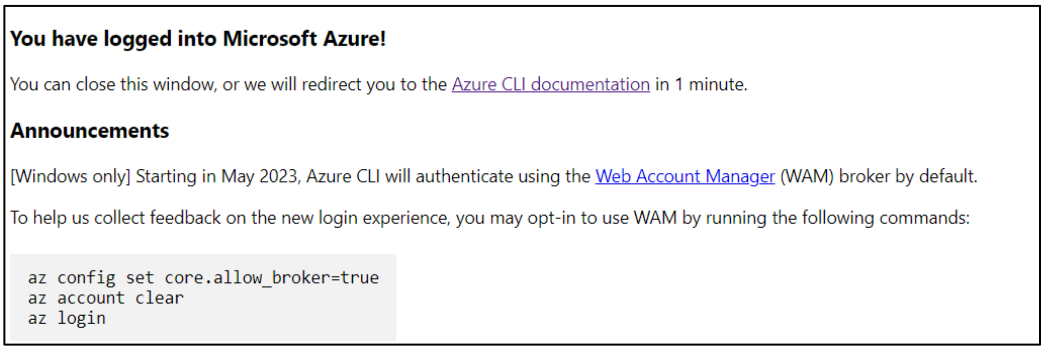

Figure 11. Successful Login to the Azure Account

3. Run the `az group create` command to create a resource group. The following command creates a resource group named `MyRAResourceGroup` in the `westus` region.
4. Optionally, to set an alternate `location`, run `az account list-locations` to see available locations. Then specify the alternate location in the following command in place of `westus`.
   `az group create --name MyRAResourceGroup --location westus`

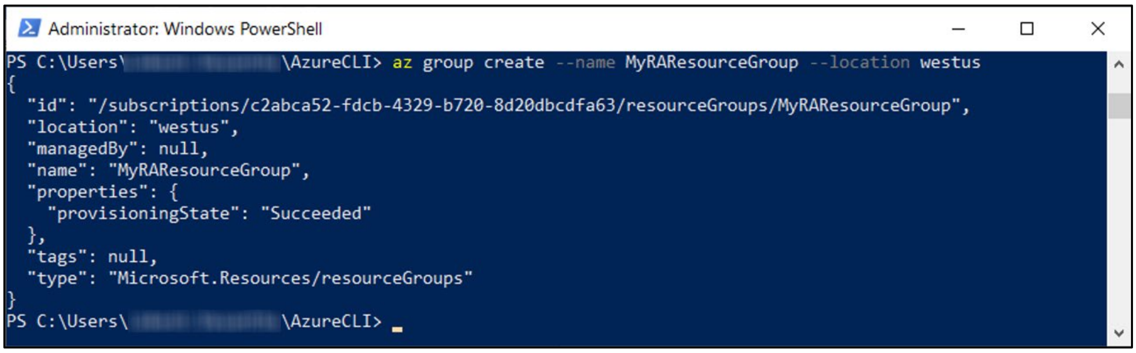

Figure 12. Create Resource Group

5. Run the `az iot hub create` command to create an IoT hub. It might take a few minutes to create an IoT Hub.
   Replace the *`YourIotHubName`* placeholder below with the name you chose for your IoT hub. An IoT hub name must be globally unique in Azure. This placeholder is used in the rest of this tutorial to represent your unique IoT hub name. Use any command given below.
   - `az iot hub create --resource-group MyRAResourceGroup --name {YourIoTHubName}`
     OR
   - `az iot hub create --resource-group MyRAResourceGroup --name {YourIoTHubName} --location {YourLocation}`

Note: It may take a few minutes to create the IoT Hub. In this case, the IoT Hub name used is `RACLOUDHUB`.

Note: Microsoft recommends creating a new IoT Hub. The IoT Hub created previously (2-3 years old) may not work as desired. So, we recommend to create a new IoT Hub to run the application to yield the proper results

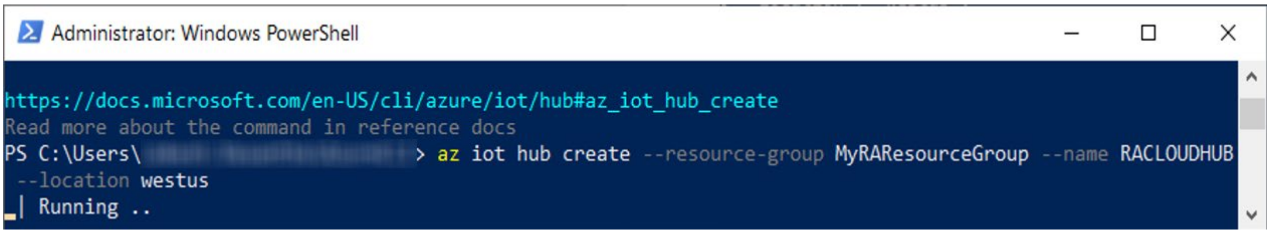

Figure 13. IoT Hub Creation in Progress

6. After the IoT Hub is created, view the JSON output in the console, and copy the `hostName` value to a safe place. You use this value in a later step. The `hostname` value looks like the following example:
   — `{Your IoT hub name}.azure-devices.net`

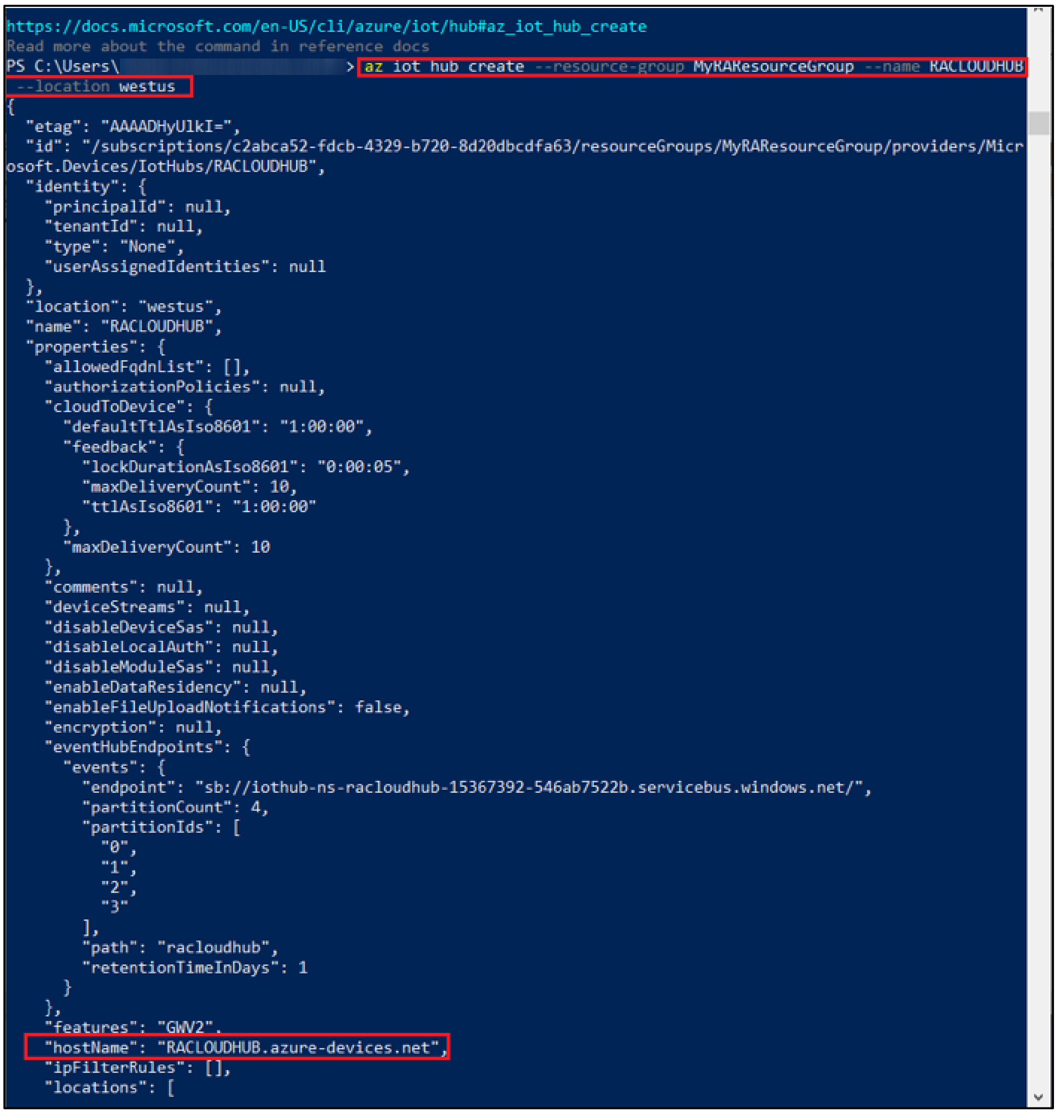

Figure 14. JSON Output after IoT Hub Creation

3.5 Certificate Creation Process

You can use GIT Bash utility for this process. If not installed on your computer, you can download and install it.(Git for Windows or Git for Windows (github.com)).

1. Install Git for Windows.
2. Launch the Git Bash.
3. Create a directory of your choice (for example, `mkdir Azure`).
4. Go to the directory and create the configuration. This created directory is the place where your self-signed certificate is created and stored.
5. Copy and paste the configuration listed below to create `x509_config.cfg` as shown in the below figure.

```
cat > x509_config.cfg <<EOT
[req]
req_extensions = client_auth
distinguished_name = req_distinguished_name

[req_distinguished_name]
```

```
[ client_auth ]
basicConstraints = CA:FALSE
keyUsage = digitalSignature, keyEncipherment
extendedKeyUsage = clientAuthEOT
```

Note: All OpenSSL commands and self-signed certificate creation process is given at this link.

Steps are as follows:

1. Set x509 configuration file for common name in cert.

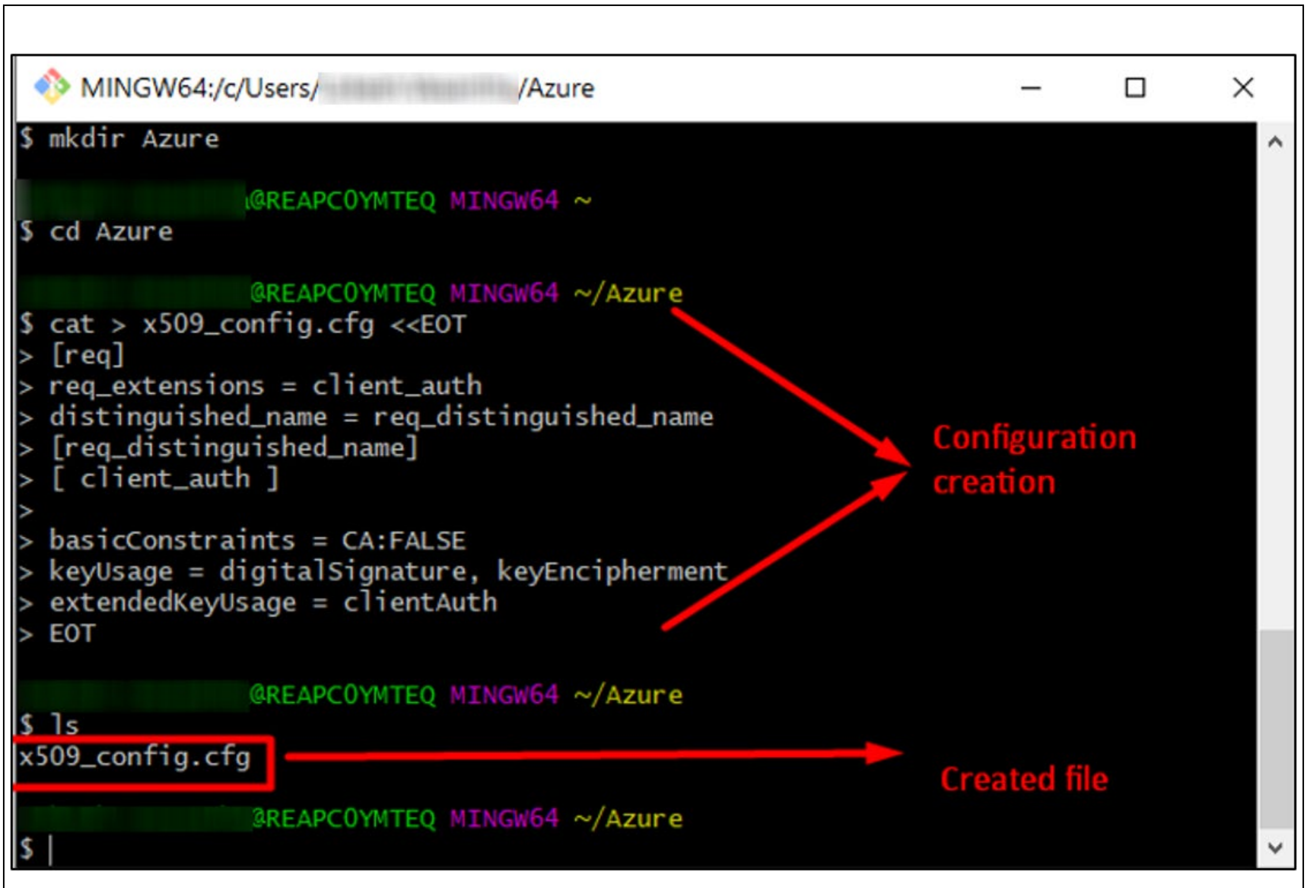

Figure 15. Set X509 Configuration File

2. Create RSA self-signed certificate.
   Generate private key and certificate (public key) using the command as shown in the snapshot “`openssl genrsa -out privkey.pem 2048`”

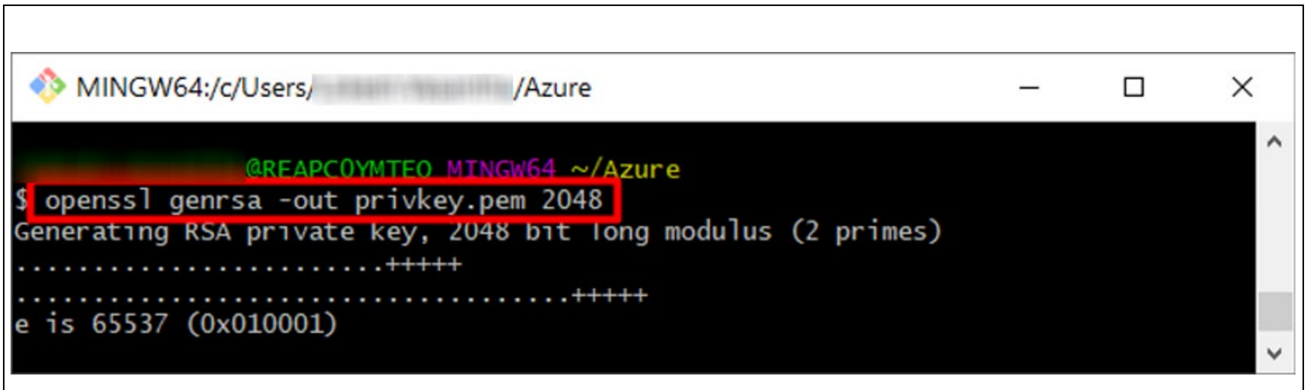

Figure 16. Generate Private Key and Certificate (public key)

3. Embed Device ID in certificate.
   This command will not give you any response if successfully executed.
   `openssl req -new -days 365 -nodes -x509 -key privkey.pem -out cert.pem -config x509_config.cfg -subj "//CN=<Same as device Id>"`
   Note: In this example, the device ID name "CK_RA6M5_X509" is used. Note down this Device ID. This will be used in future steps. Use your own Device ID to make it unique across your system.

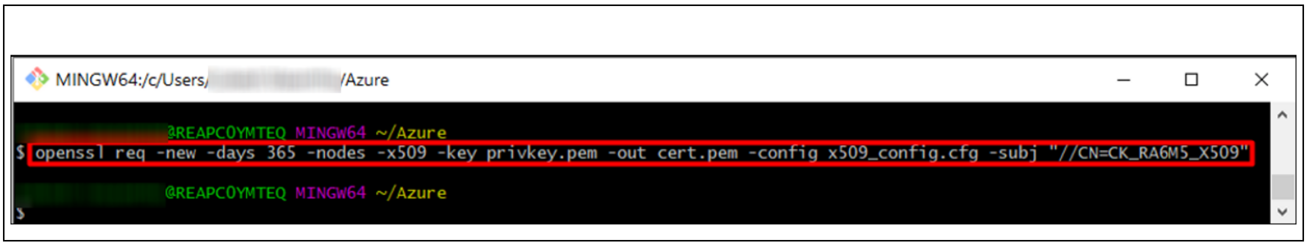

Figure 17. Embed Device ID in Certificate

4. Run command to convert format of key from pem to der.
   `openssl rsa -outform der -in privkey.pem -out privkey.der`
   Here you get response "`writing RSA key`"

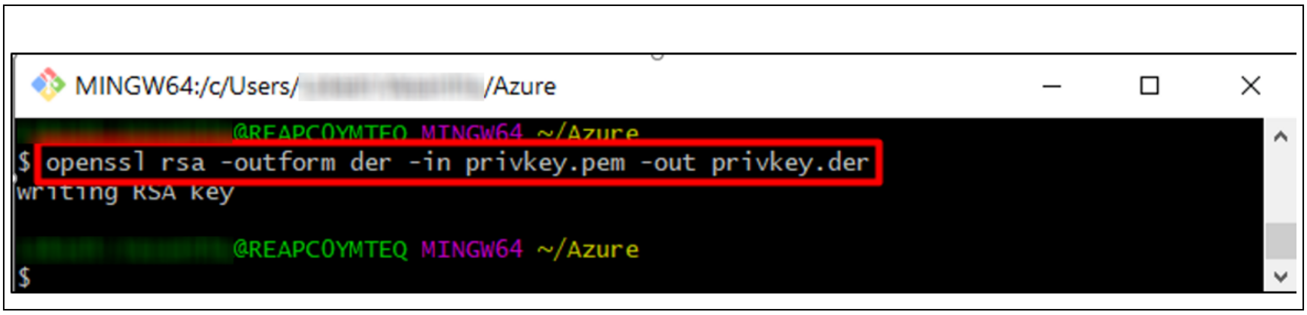

Figure 18. Convert Format from key to der

5. Run command to convert format of cert from pem to der.
   `openssl x509 -outform der -in cert.pem -out cert.der`
   This command will not give you any response if successfully executed.

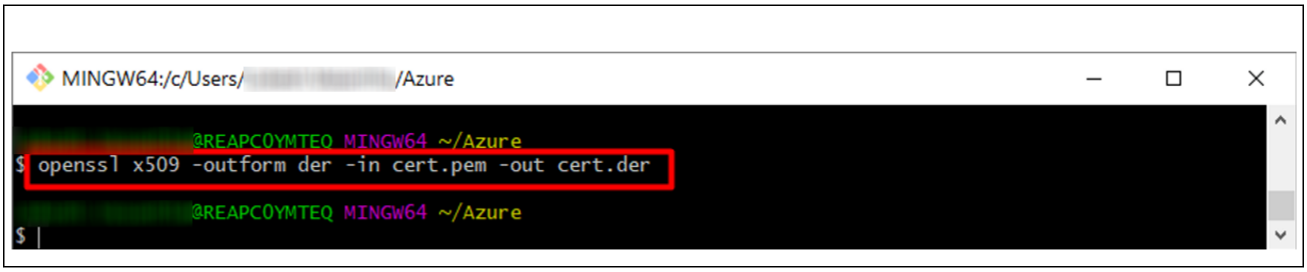

Figure 19. Convert Format of cert from pem to der

6. Convert der to hex array and set them in `sample_device_identity.c` file in the project.
   For easier access, the command text is given as follows. User can copy paste text in the command line to create `sample_device_identity.c`.

```
echo "#include \"nx_api.h\"
/**
device cert (`openssl x509 -in cert.pem -fingerprint -noout | sed 's/://g' `) :
`cat cert.pem`

device private key:
`cat privkey.pem`
*/
" > sample_device_identity.c
```

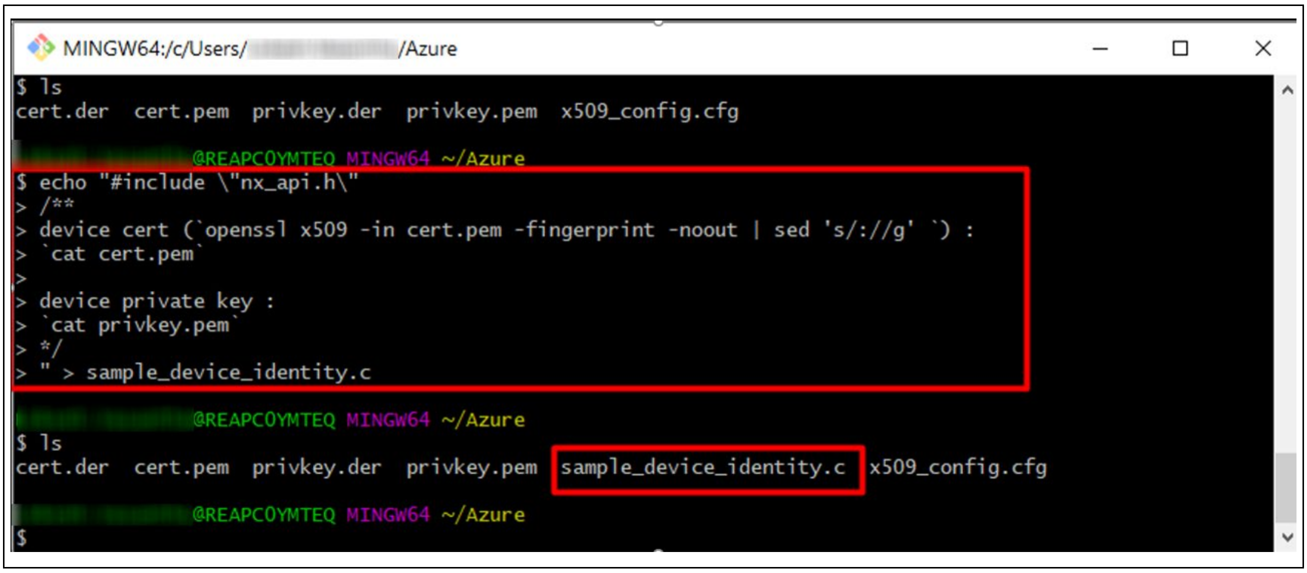

Figure 20. Convert der to Hex Array and Set them in sample_device_identity.c

7. Run "ls" command to check whether `sample_device_identity.c` is created.
8. Run the following commands to produce `sample_device_cert_ptr` and `sample_device_private_key_ptr` array containing device certificate and private key equivalent hex values along with length.

```
xxd -i cert.der | sed -E "s/(unsigned char) (\w+)/\1
sample_device_cert_ptr/g; s/(unsigned int) (\w+)_len/\1
sample_device_cert_len/g" >> sample_device_identity.c

xxd -i privkey.der | sed -E "s/(unsigned char) (\w+)/\1
sample_device_private_key_ptr/g; s/(unsigned int) (\w+)_len/\1
sample_device_private_key_len/g" >> sample_device_identity.c
```

These commands will not give you any response if successfully executed.

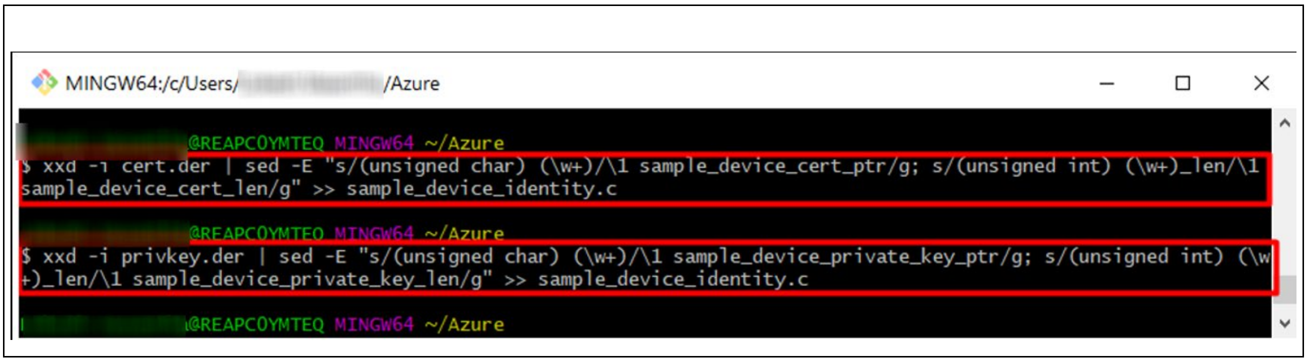

Figure 21. Producing arrays containing hex values

Check the content of `sample_device_identity.c` with cat command. In this file, you will get the Device certificate along with SHA1 fingerprint, Device Private Key, `sample_device_cert_ptr` and `sample_device_private_key_ptr` array along with their length. You will also notice the Fingerprint; you need to use this fingerprint as "thumbprint" in device creation process using the IoT Explorer in later sections. Please note down this Fingerprint.

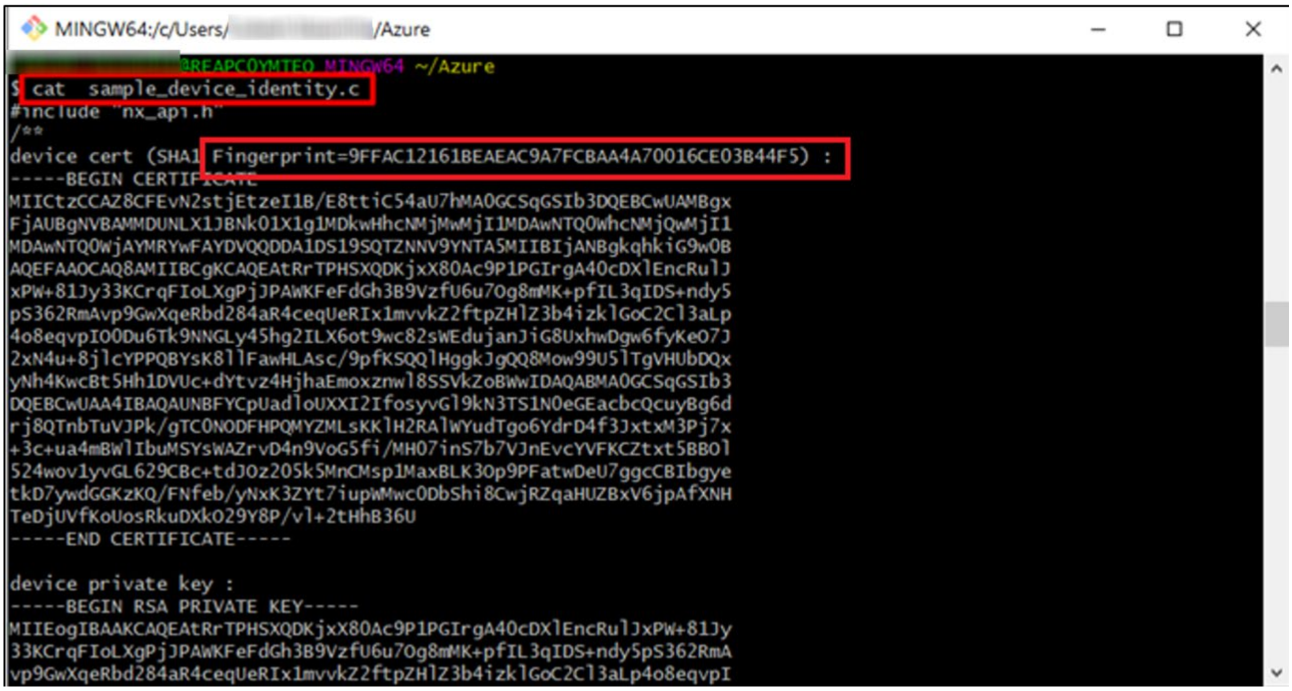

Figure 22. Check the Content of sample_device_identity.c

3.6 View Device Properties

You can use the Azure IoT Explorer (Install and use Azure IoT explorer - Azure IoT | Microsoft Learn) to view and manage the properties of your devices. In the following steps, you will add a connection to your IoT Hub in IoT Explorer. With the connection, you can view properties for devices associated with the IoT Hub.

Download and install the latest (above v0.15.6.0) Azure IoT Explorer from: https://github.com/Azure/azure-iot-explorer/releases

Note: Click and install the downloaded msi file `Azure.IoT.Explorer.Preview.0.15.6.msi` or a newer version of the downloaded file. The install shield guides you through the installation process.

3.7 Set IoT Hub

To add a connection to your IoT Hub:

1. In your Azure CLI console, run the `az iot hub connection-string show` command to get the connection string for your IoT Hub.
   — `az iot hub connection-string show -n {YourIoTHubName}`
   Note: See section Create an IoT Hub for the IoT Hub Name.

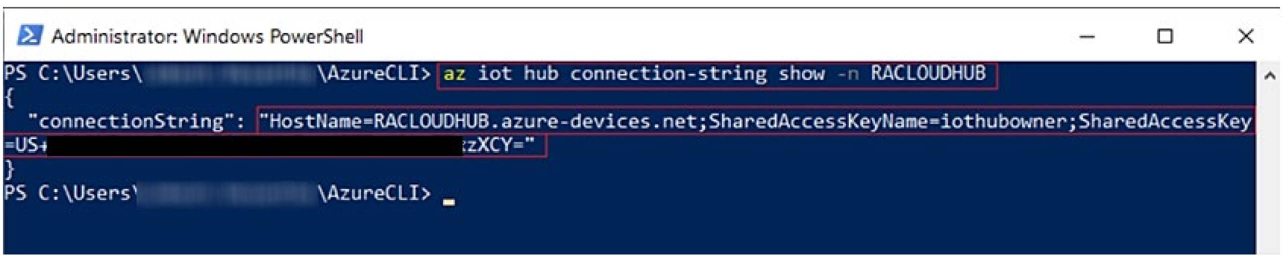

Figure 23. Connection String

2. Copy the connection string.
3. Open the Azure IoT Explorer and select **IoT hubs > Add connection**.
4. Paste the connection string into the **Connection string** box.
5. Select **Save**.

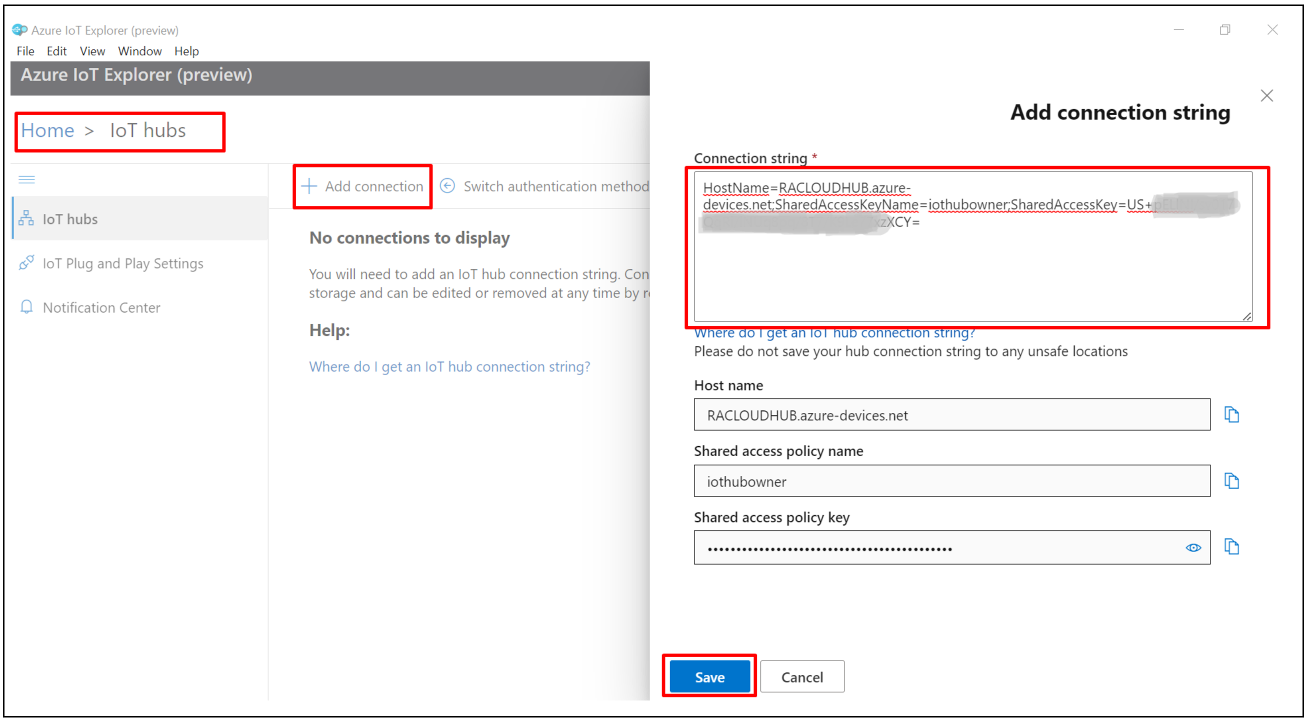

Figure 24. Adding Connection String

Note: In some cases, Azure IoT Explorer may report an error that the default port that IoT Explorer is trying to use is being used by another application. In order to overcome this error, you can add a different port number for the Azure IoT Explorer shown as follows.

Note: In some cases, Azure IoT Explorer may report an error that "*Failed to retrieve device list: request to https://raxxxxxx.azure-devices.net/devices%2Fquery?api-version=2020-09-30 failed, reason: unable to get local issuer certificate.*" This error is due to Zscaler tool running on your PC set by IT. To overcome this error, you try running the IOT Explorer on a PC without Zscaler or Lab machine.

Reference: https://github.com/Azure/azure-iot-explorer/issues/604

On your PC, edit the system environmental variables as shown in the following screenshots.

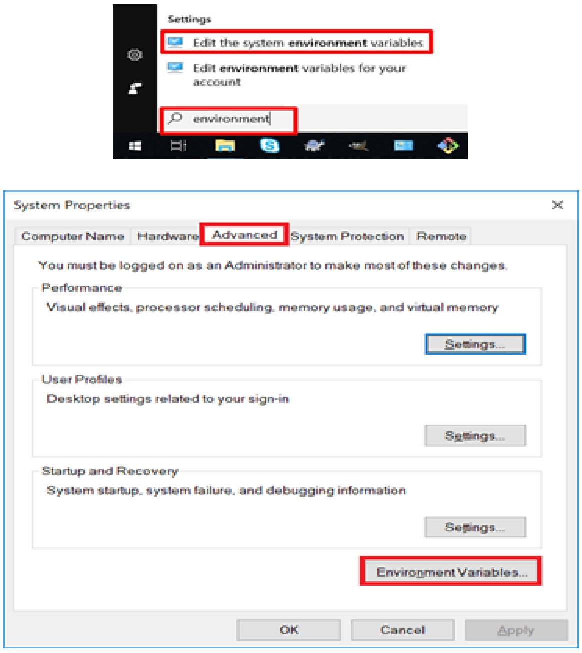

Figure 25. Editing System Environment Variable

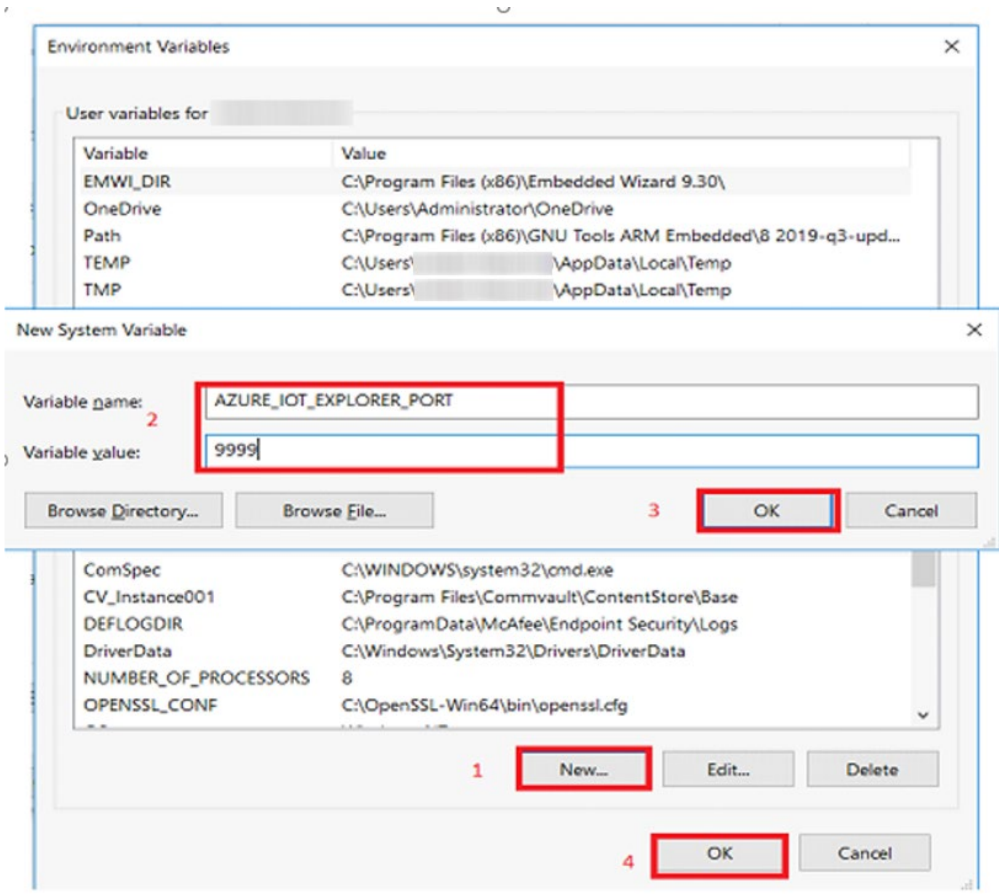

Figure 26. Adding System Environment Variable for Alternate Port - Azure IoT Explorer

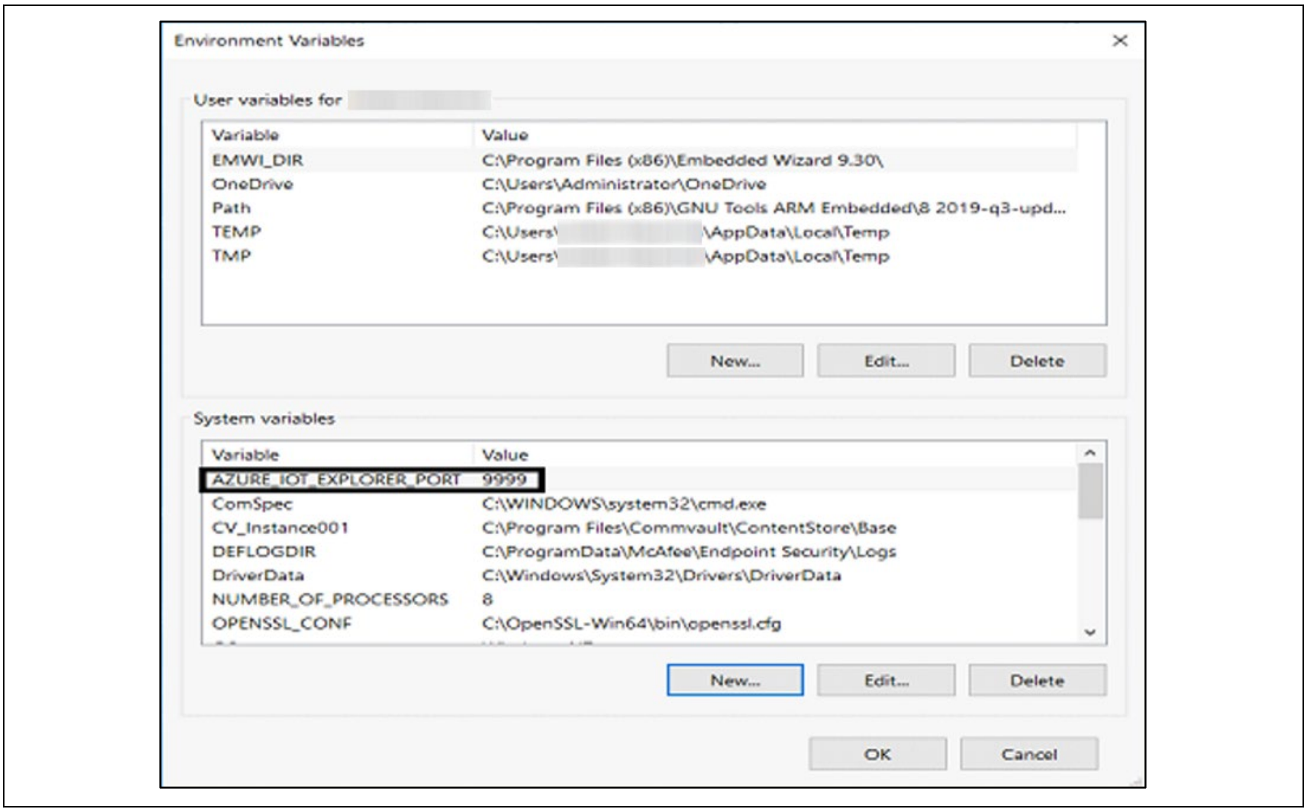

Figure 27. Added Alternate Port for Azure IoT Explorer

If the connection succeeds, the Azure IoT Explorer switches to a Devices view and lists your device.

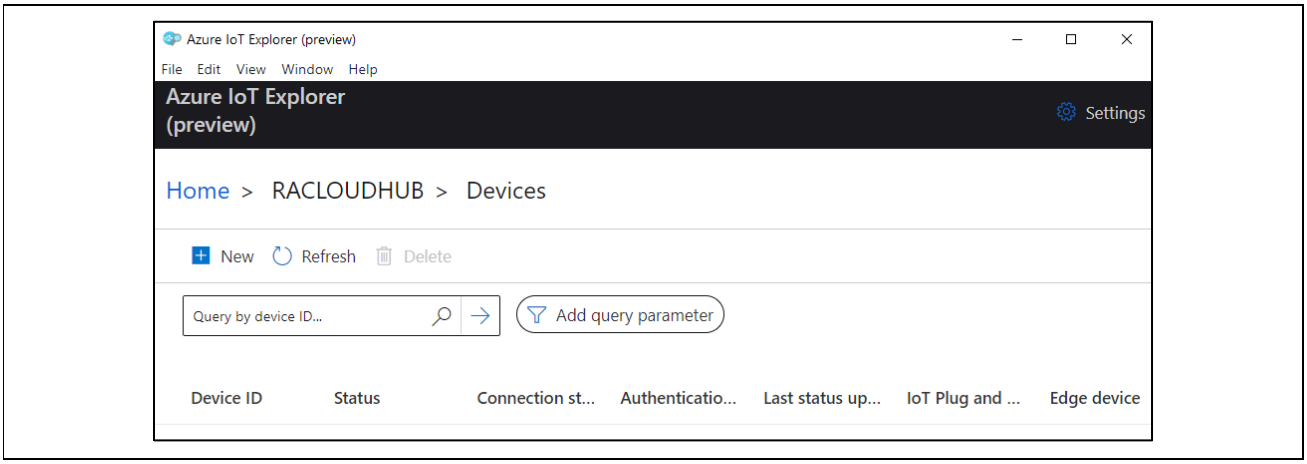

Figure 28. Listed Devices

3.8 Register an IoT Hub Device

In this section, you create a new device instance and register it with the IoT Hub you created. You will use the connection information for the newly registered device to securely connect your physical device in a later section.

To register a device:

1. You can create a device with help of Azure IoT Explorer shown as follows.
   Click on **New**.

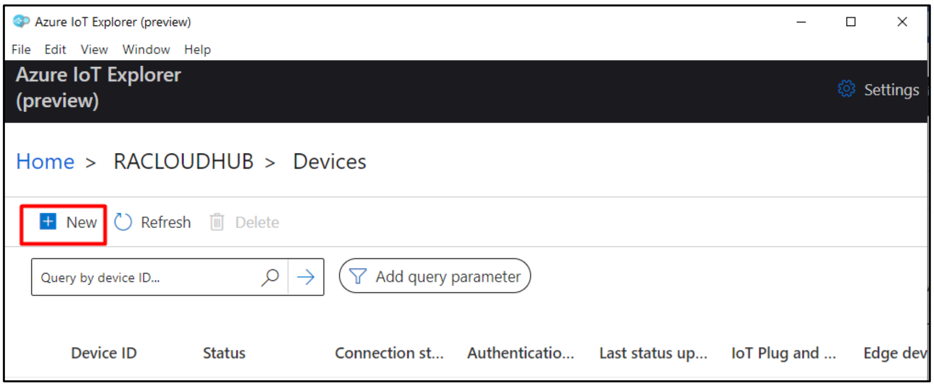

Figure 29. New Device Creation Process with Azure IoT Explorer

2. In this stage, you have to enter the Device ID, Authentication type, Primary thumbprint, Secondary thumbprint then click on **Create**. Use fingerprint generated in Figure 22 in the section 3.5. Certificate Creation Process, for the primary and secondary thumbprints. Follow steps 1-5 numbered in the Figure 30, to create the device.

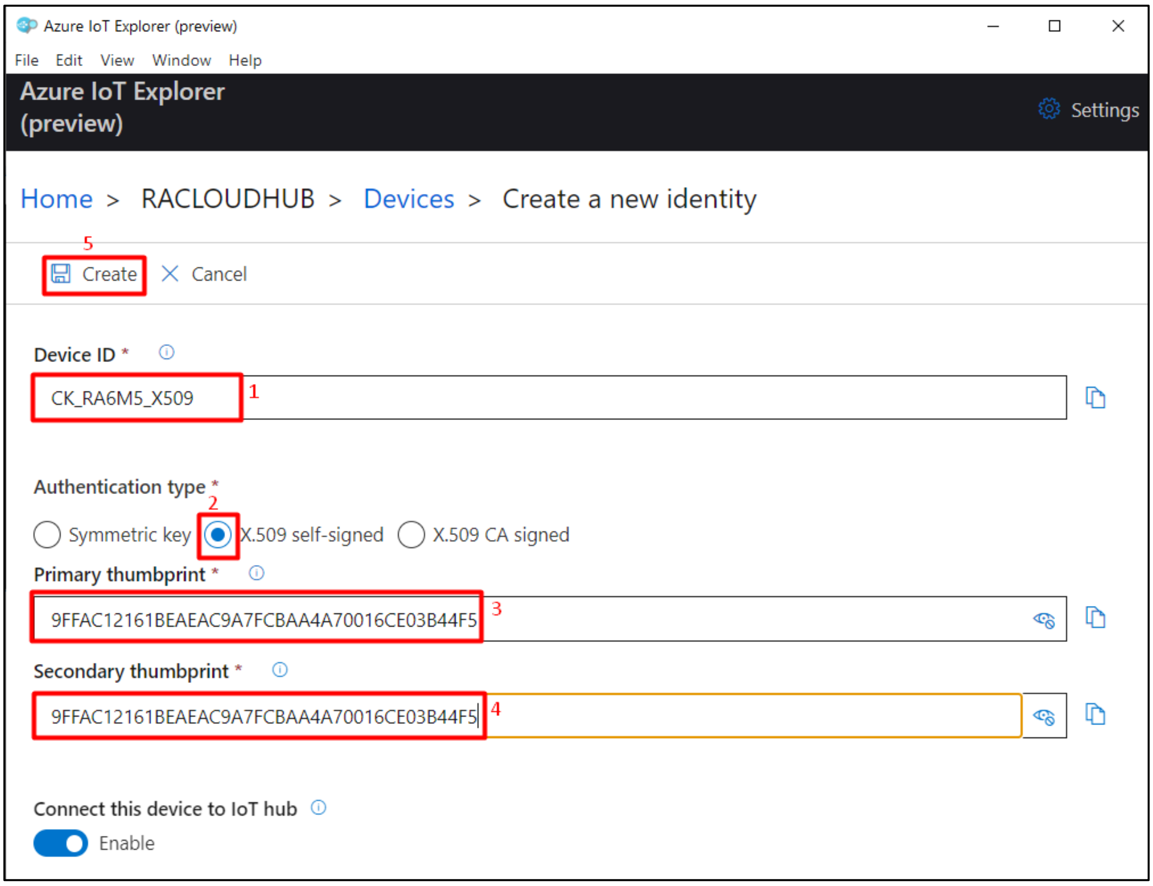

Figure 30. Naming, Authentication type, and Thumbprints

3. You can see your created device in Devices section of Azure IoT Explorer.

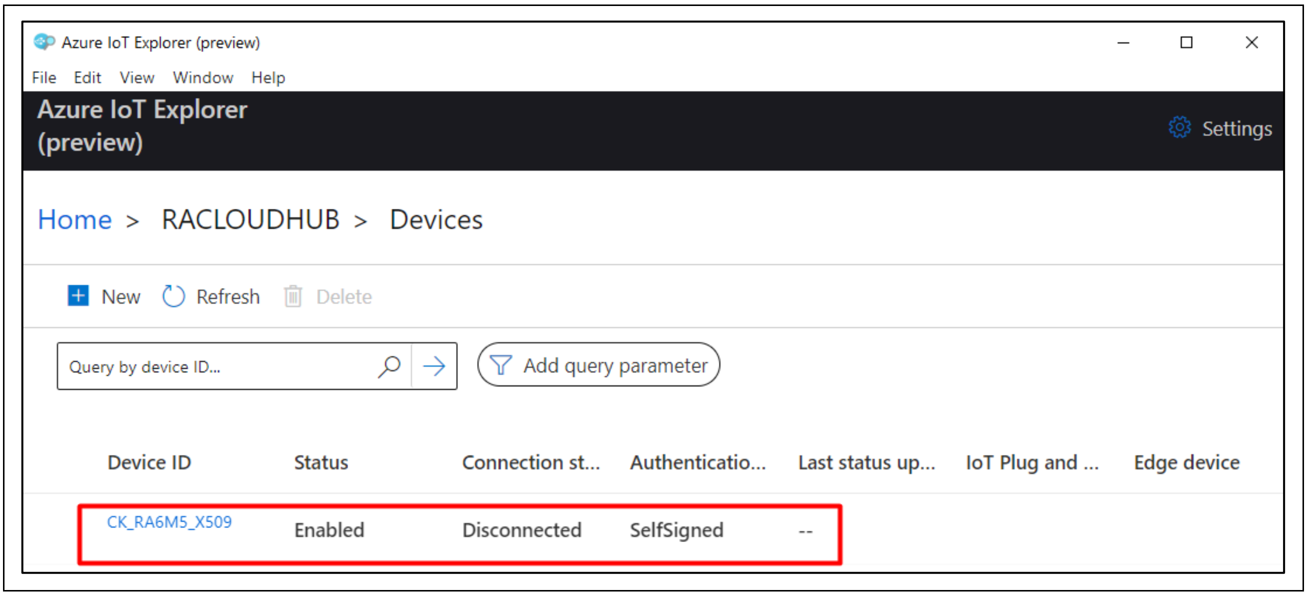

Figure 31. Newly Created Device

3.9 Prepare the Device

To connect the device to Azure, modify a configuration file for Azure IoT settings (of your Device ID and Hostname), and build and flash the image to the device.

Add configuration

1. Import the application project into an empty e[2] studio. Open `sample_config.h` and make the changes to the configuration as shown in the snapshot with the option USE_DEVICE_CERTIFICATE.

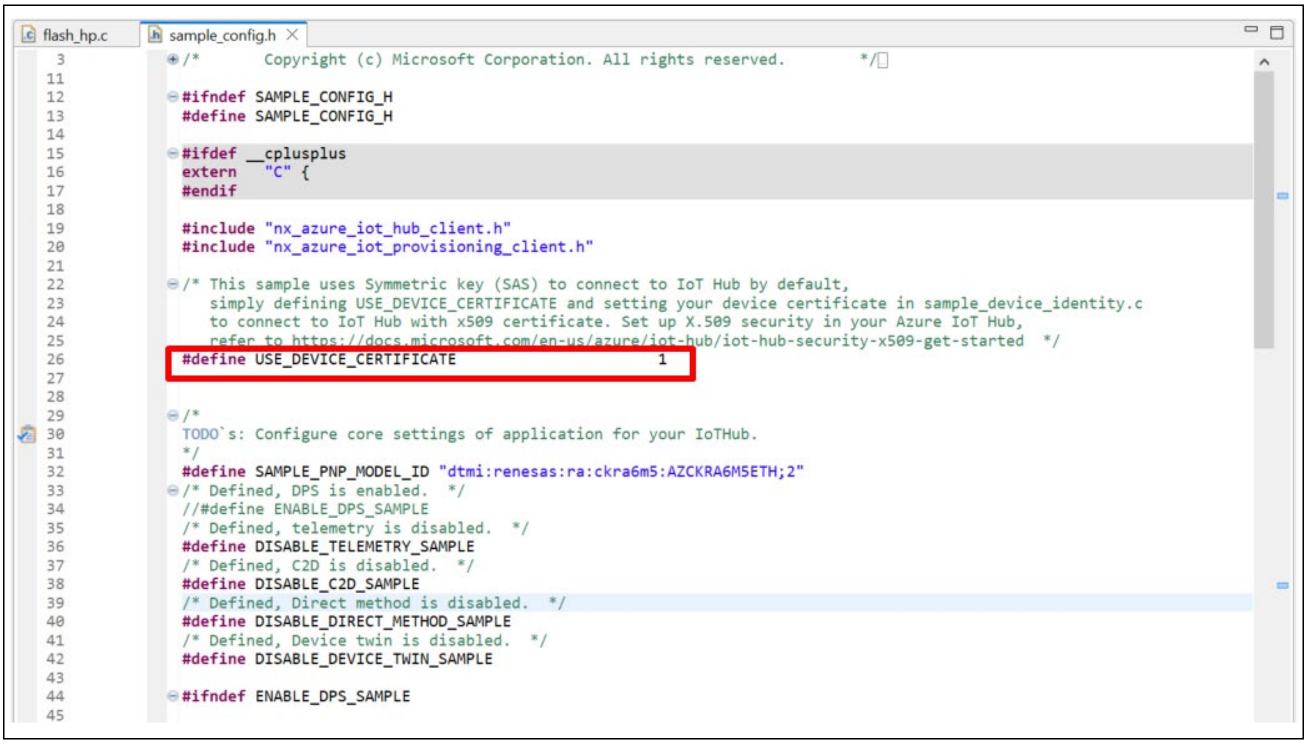

Figure 32. Configuration Changes to sample_config.h

Constant name	Value
`USE_DEVICE_CERTIFICATE`	1

2. Open nx_azure_iot_cert.c to check the root CA data following the Azure IoT Hub. This application is migrated to use root CA “DigiCert Global Root G2”

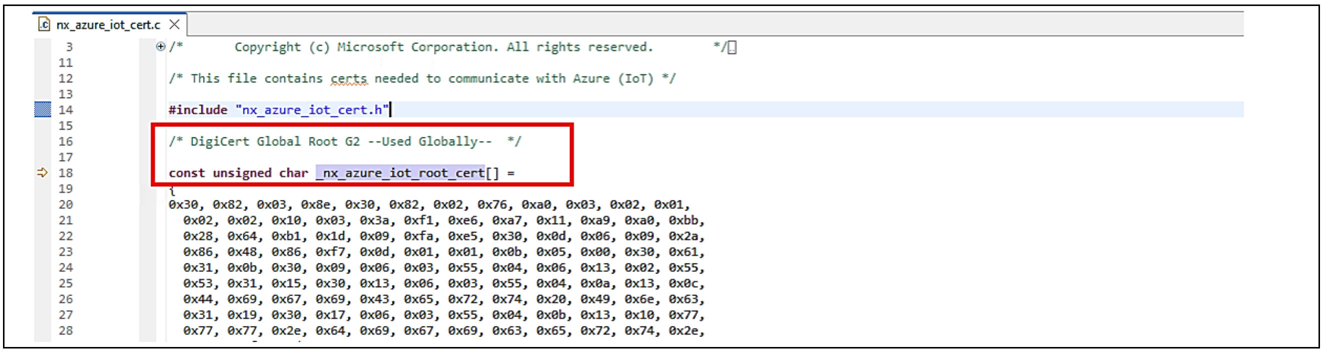

Figure 33. Root CA certificate in this project

Note: IoT Hub in Azure Cloud can change the root CA in the future. So please check and update the new root CA at [How to migrate hub root certificate - Azure IoT Hub | Microsoft Learn] if you cannot connect to Azure IoT Hub due to the expiration of the root CA issue.

You can download the root CA file at: [DigiCert Root Certificates - Download & Test | DigiCert.com]

Step to change the root CA data in this project:

1. Download the root CA.
2. Using command “$xxd -i <file.cert> >> <output.c>” to convert file .pem to array in C.
3. Copy value into src/nx_azure_iot_cert.c

3.10 Building and Running the Application

The project is now ready to be compiled. Press the **Build** (hammer icon) to start building the project.

Figure 34. Starting to Build the Project

The toolchain will report compilation and build status to the console pane in the lower-right corner of e² studio. When the building has been completed, confirm that there are zero errors and few warnings. Warnings, if any, may result from highly restrictive compilation warnings settings being applied by e² studio to third-party code.

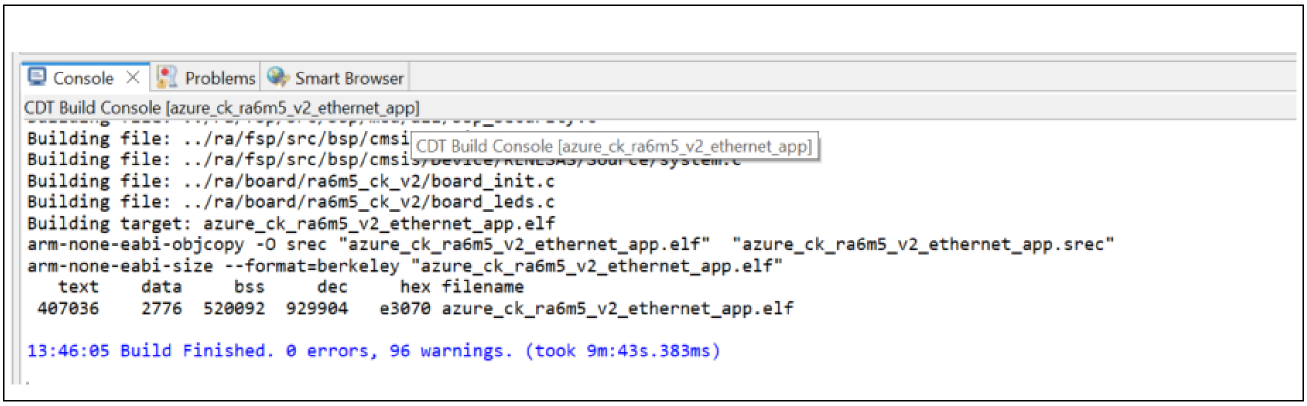

Figure 35. Compilation and Build Status Report

3.11 Download and Run the Project

1. To connect power to the board, connect the USB cable to the CK-RA6M5 v2 board's J28 connector (USBC) and the other end to the PC USB port.
2. Connect the second USB cable to J10 connector of the CK-RA6M5 v2 board and other end to the second USB port of the PC (this will be the console port for the application). Users are required to use the Command Line Interface (CLI) to configure and run the application.
3. Make sure the Ethernet cable is connected to the RJ-45 connector (J5) of the board and other end to the router/switch as applicable for the internet access.
4. In e[2] studio, open the **Debug Configurations** dialog and launch the **azure_ck_ra6m5_v2_ethernet_app.elf** debug configuration.

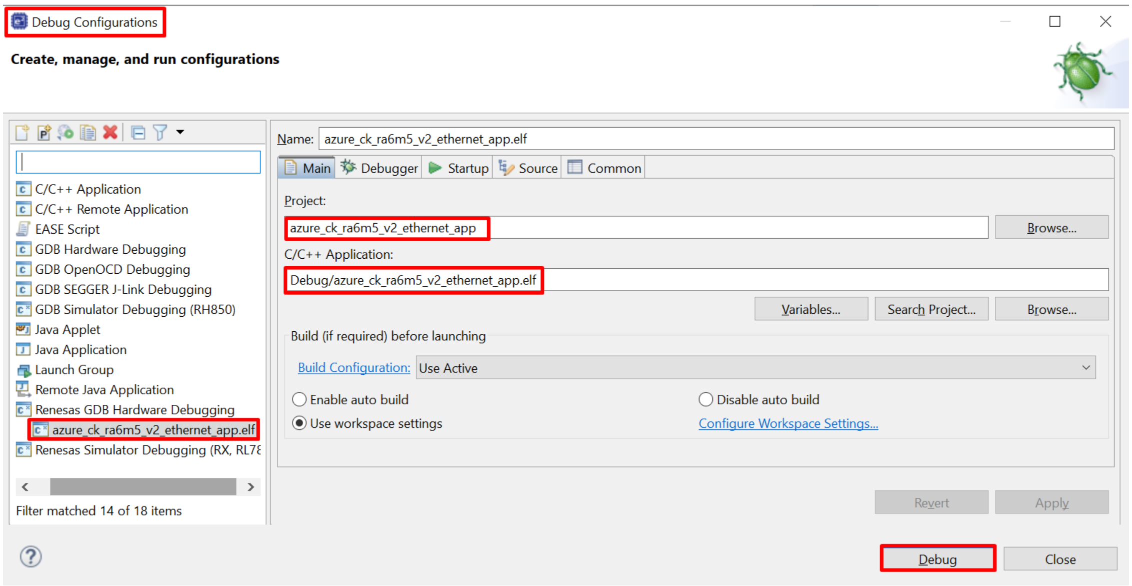

Figure 36. Start Debug

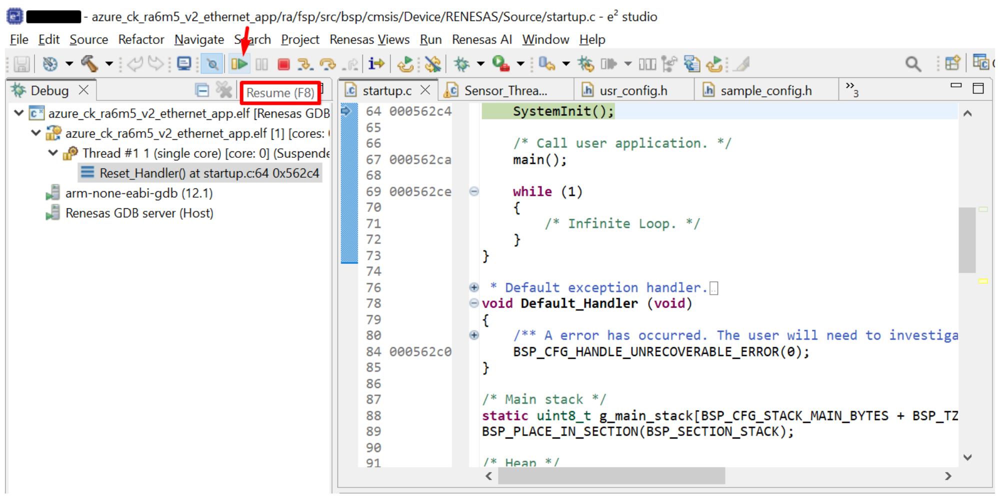

Figure 37. Resume the Debug

5. To view output, you have to use serial terminal like tera term. To know your COM port, on the host PC, open the Windows Device Manager. Expand **Ports (COM & LPT),** locate **JLink CDC UART Port (COMxx)** and note down the COM port number for reference in the next step.
   Note: JLink CDC UART drivers are required to communicate between the CK-RA6M5 v2 board and the terminal application on the host PC.

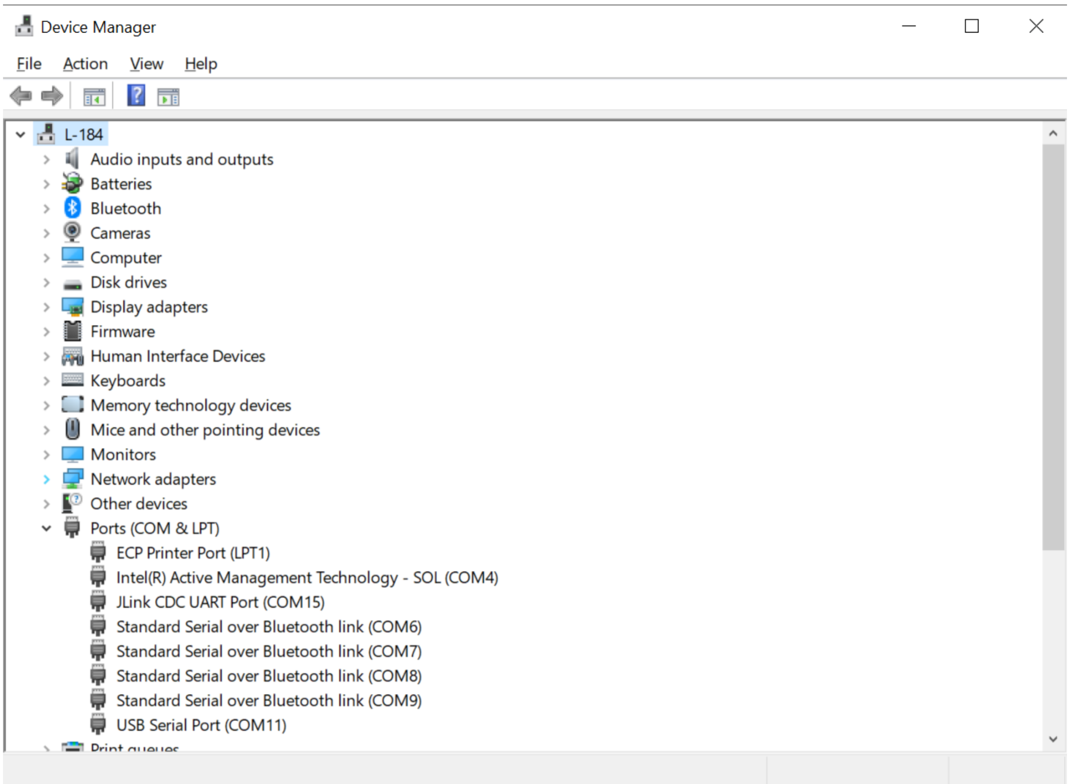

Figure 38. JLink CDC UART in Windows Device Manager

6. Open Tera Term, select **New connection,** and select **Serial,** and for the port, enter **COMxx: JLink CDC UART Port (COMxx)** and click **OK**.

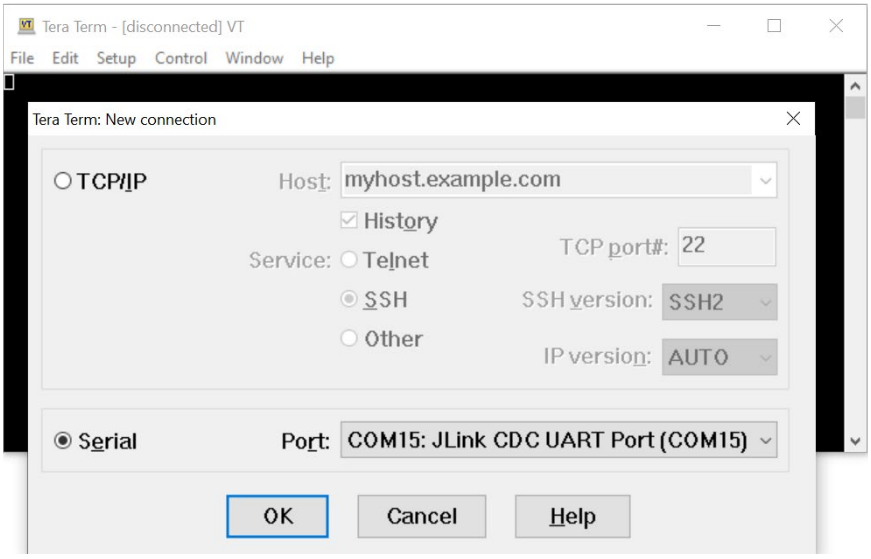

Figure 39. Selecting the UART Port on Tera Term

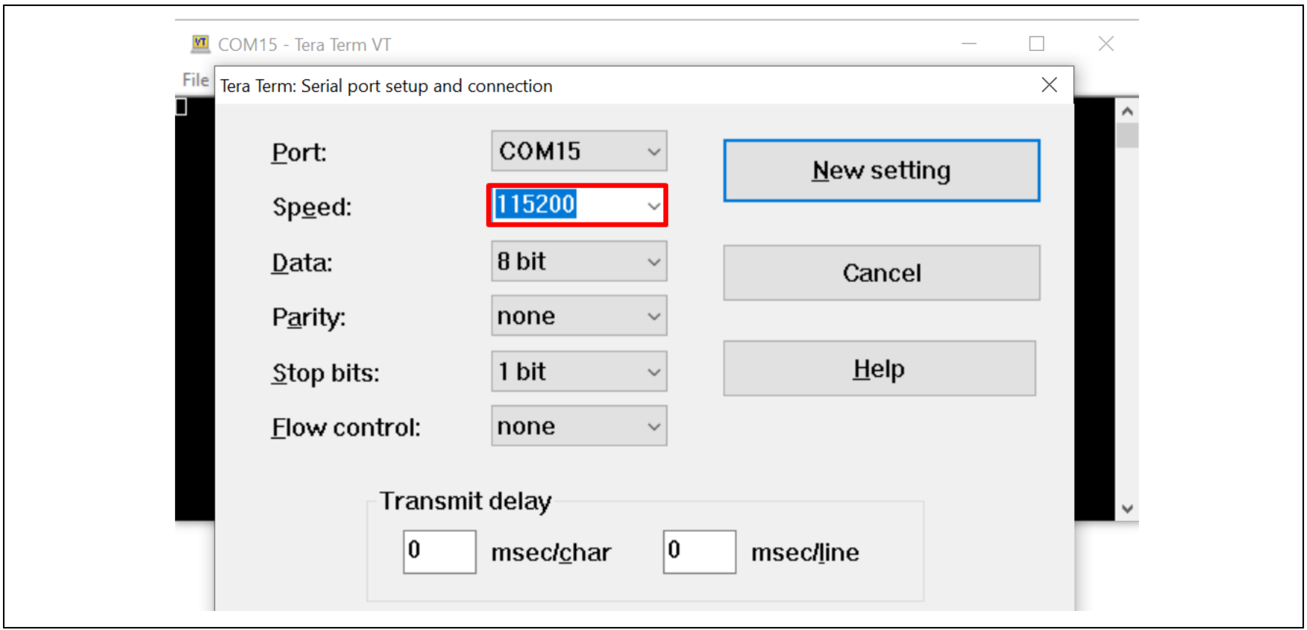

Figure 40. Select 115200 on the Speed Pulldown

7. Using the setup menu pull-down, select **Serial port**… and ensure that the speed is set to **115200**, shown as follows.
8. Complete the connection. The Configuration CLI menu will be displayed on the console shown as follows.
   Note: Please reset the board by pressing the S1 user switch if the menu is not displayed.

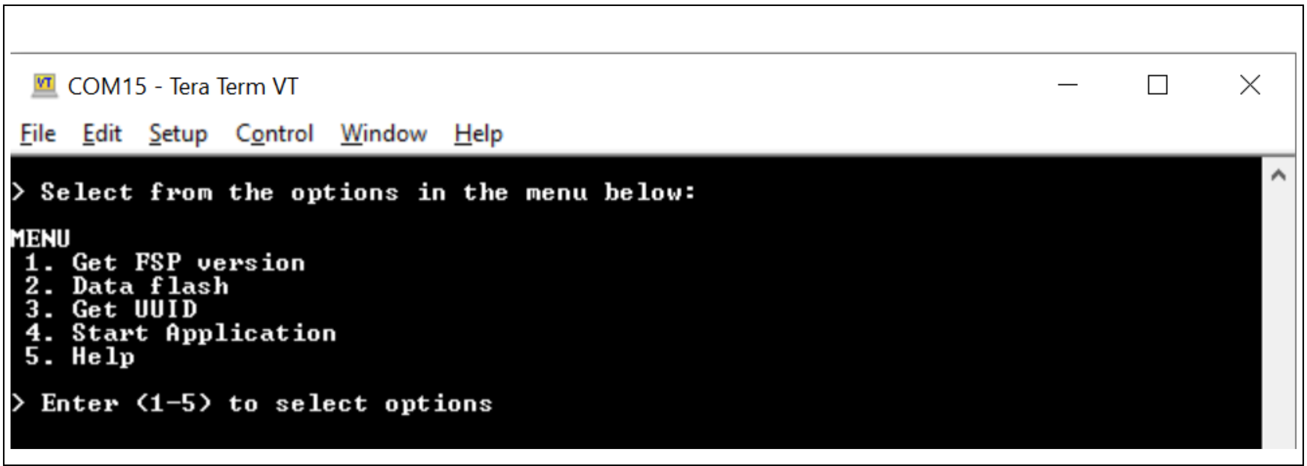

Figure 41. Main Menu

9. Here, you can select options from the menu by pressing key **1 to 5**. Press spacebar to go to previous menu FSP version and UUID details as follows.

Figure 42. FSP Version Information

Figure 43. Getting Board UUID Information

3.12 Storing Device Certificate, Host Name, Device ID

Please reset the board by pressing the S1 user switch if the menu is not displayed.

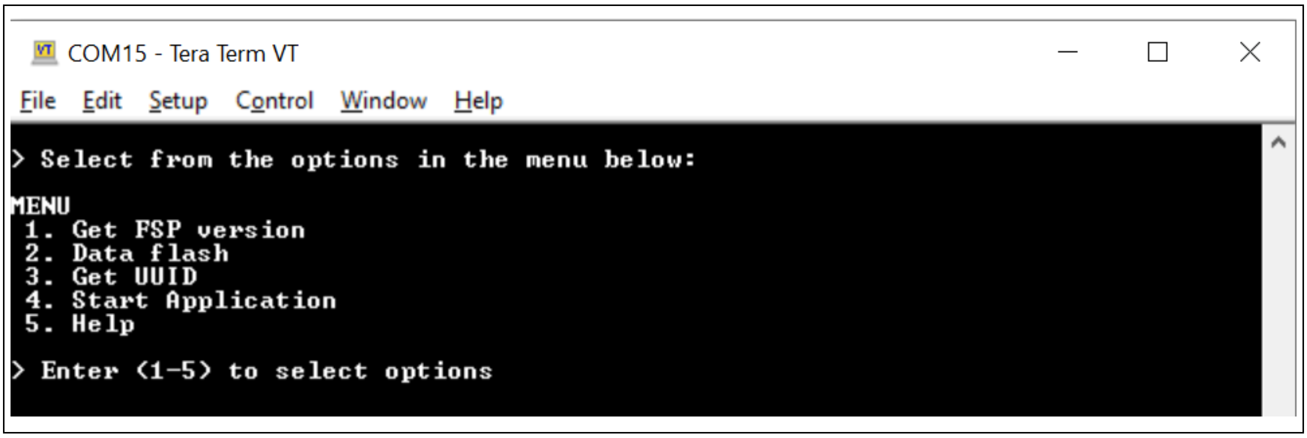

Figure 44. Main Menu

1. Press **2** on the Main Menu to display Data Flash related commands as shown in the following screenshots. This sub menu has commands to store, read, and validate the data.

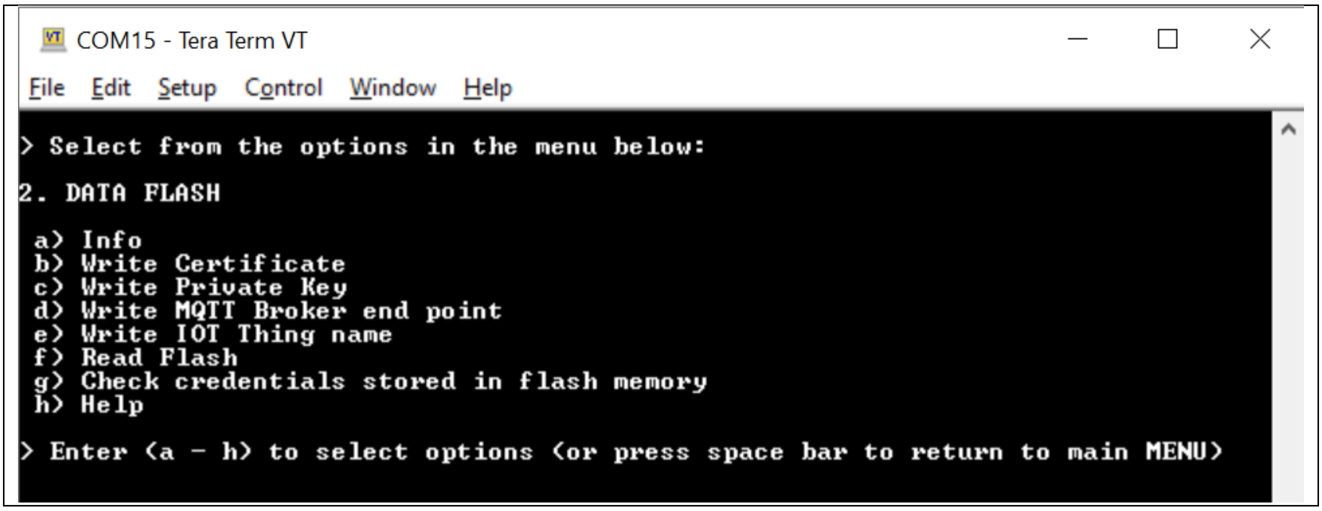

Figure 45. Data Flash Menu

2. Press **b** for Write Certificate.

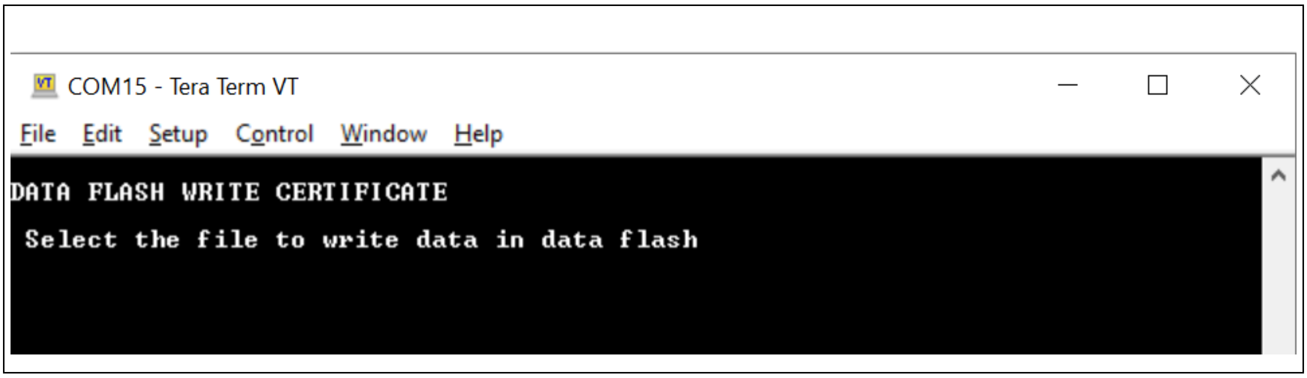

Figure 46. Select File to Write Data in Data Flash

3. Go to **Tera Term > File > Send file**

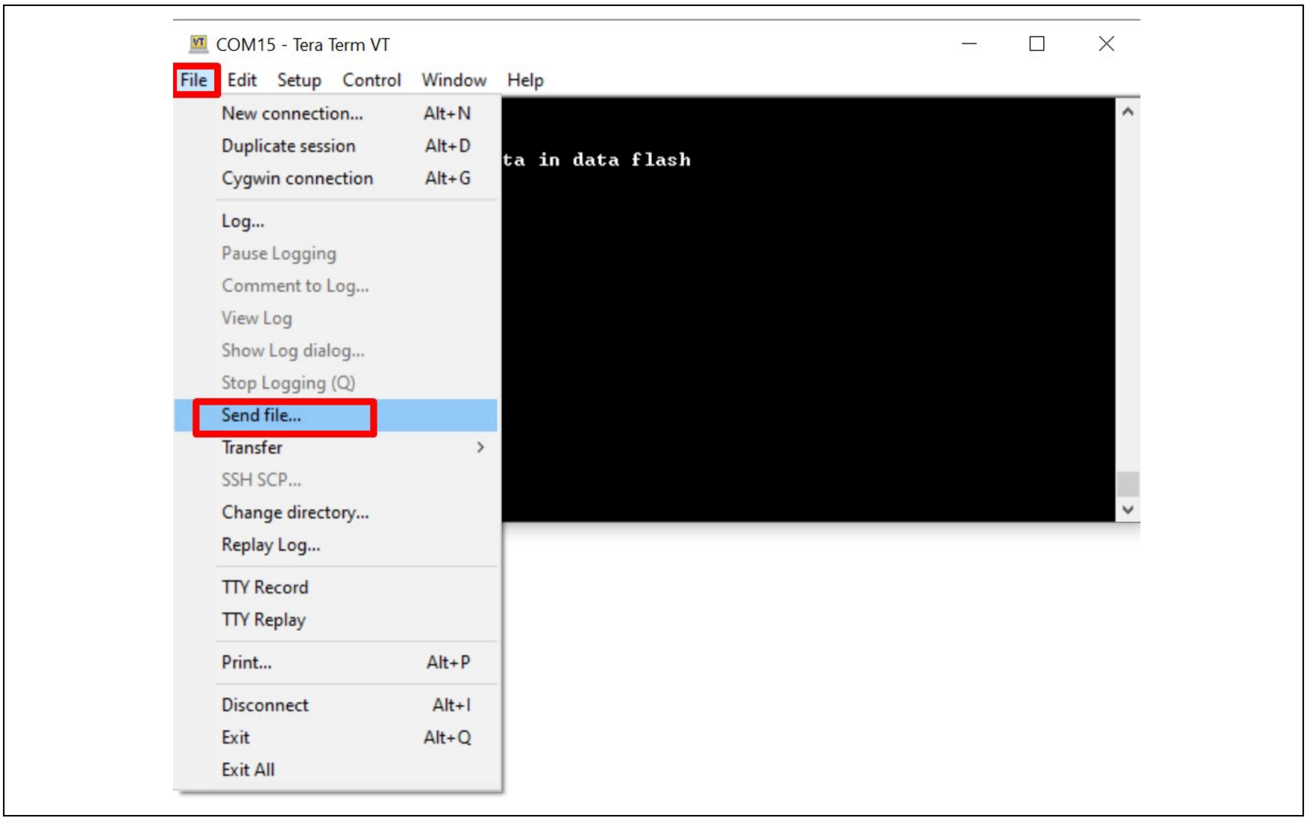

Figure 47. Send File Option in File Menu

4. Browse to the folder where X509 certificates are generated as part of section 3.5, Certificate Creation Process. Select **cert.pem**. Press **Open**.

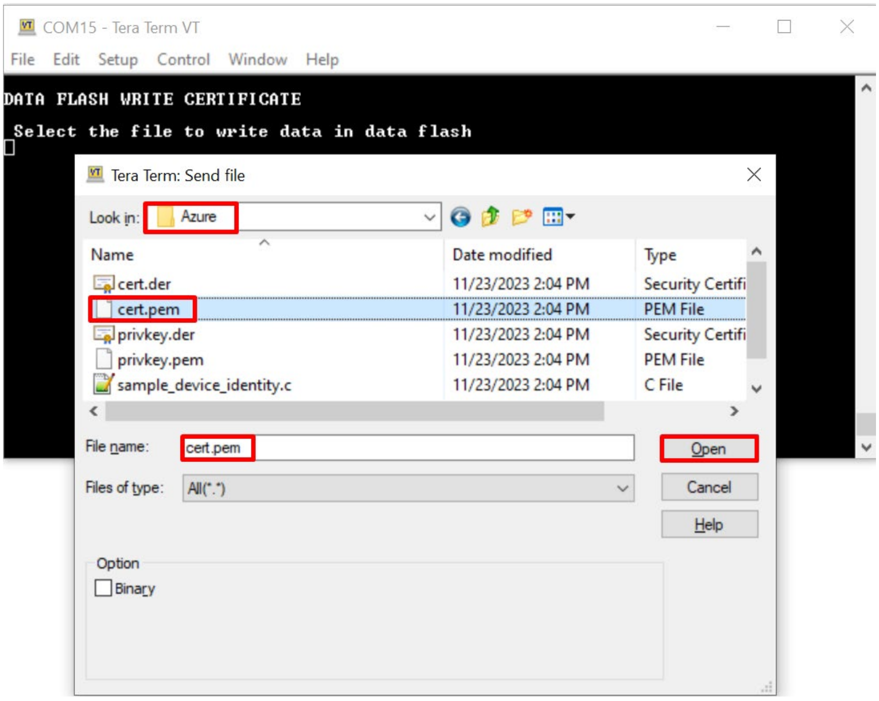

Figure 48. Browse, Select and Open the File to be Written

5. Status of Device Certificate Downloading is as follows.

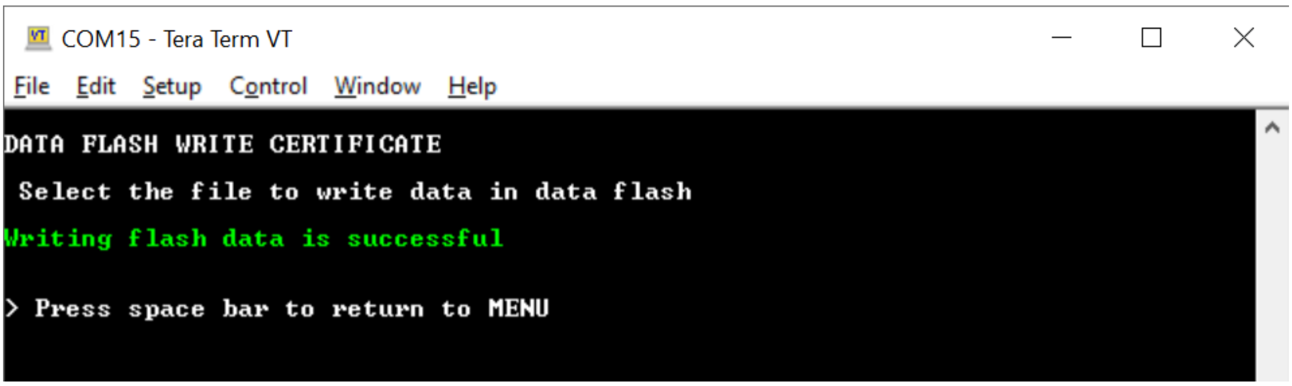

Figure 49. Status of File Writing Process

6. To store the device's private key, go back to the data flash menu by pressing the space bar key. **Press c** in **Data Flash menu**, go to **Tera Term > File > Send file**, select file **privkey.pem** from the folder where you have generated certificates.
7. To store MQTT Broker End point, that is, **Host Name,** first copy Host Name without double quotes then **press d** in **Data Flash** menu, go to **Tera Term > Edit > Paste <CR>;** you will get the copied Host Name in the clipboard. Please verify and confirm it and press **OK.**

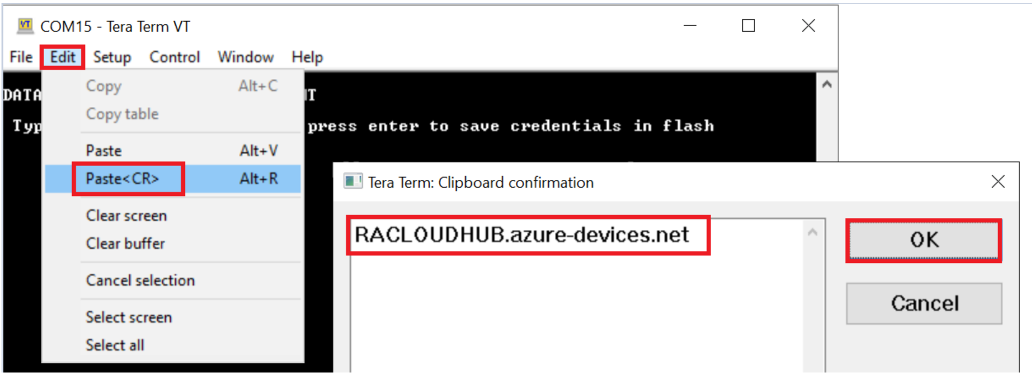

Figure 50. Input MQTT Broker End point aka Host Name

8. To store IoT Thing Name, that is, **DEVICE ID**, first copy the DEVICE ID created without double quotes, **press e** in **Data Flash** menu and follow the procedure in step 5.

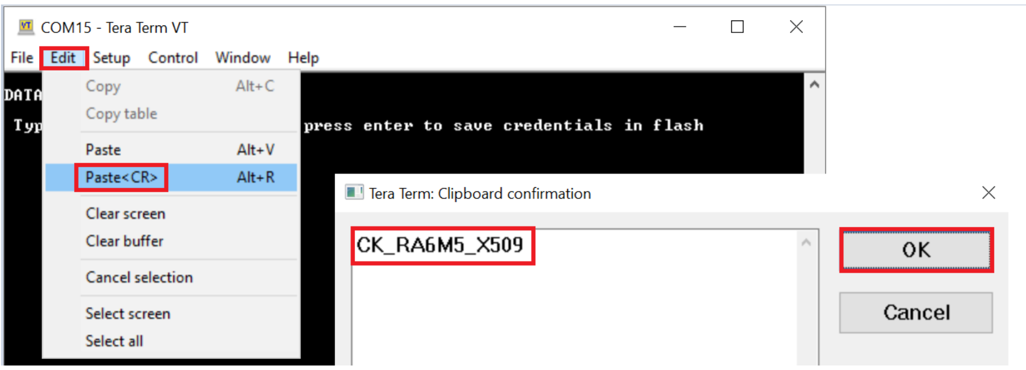

Figure 51. Input Device ID aka IoT Thing name

9. To verify the data stored in Data Flash, **press f** in Data Flash menu, scroll down to see data**.**

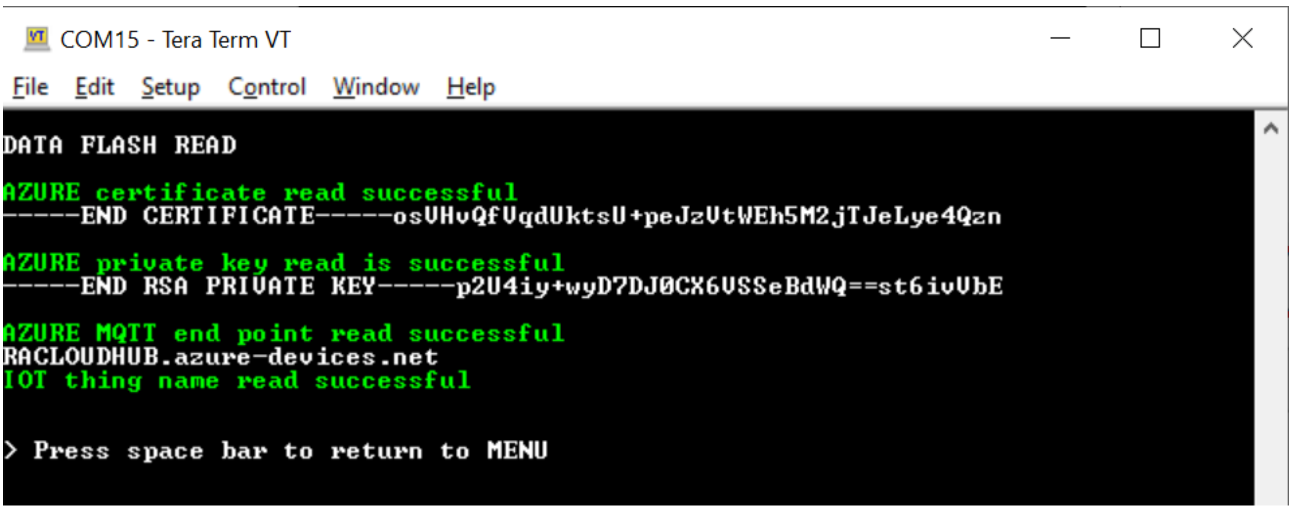

Figure 52. Scroll Down and Verify the Data Stored in Data Flash

10. To check the credentials stored in Data Flash, press **g.**
11. Press spacebar to go to previous menu or main menu.
12. Press **4** to start the application from the main menu.
13. Serial terminal output on successful start of application.

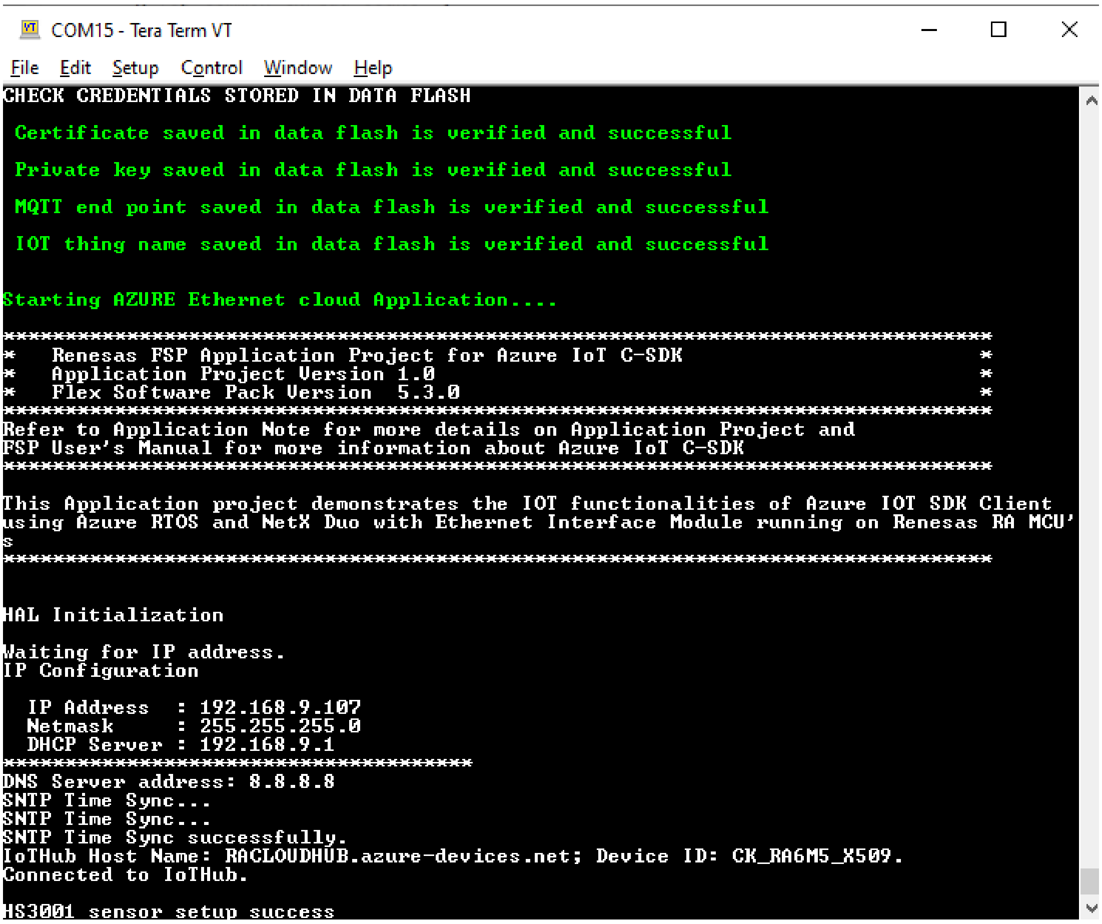

Figure 53. Device Connected to Azure IoT Hub

14. Sensor data output on serial terminal.

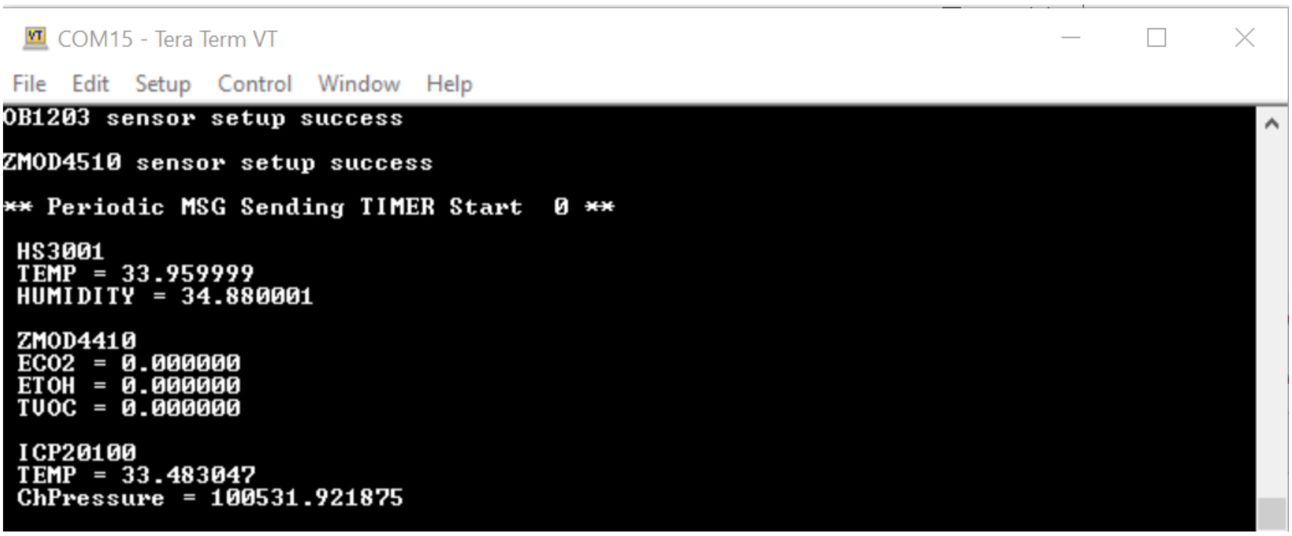

Figure 54. Sensor Data on Serial Terminal

3.13 Send Device to Cloud Message

With Azure IoT Explorer, you can view the flow of telemetry from your device to the Cloud. To view telemetry in Azure IoT Explorer:

1. In IoT Explorer, select your created IoT Hub, and click on **view devices in this hub**, click on the created device (Device ID). Finally select the **Telemetry (Home > RACLOUDHUB > Devices > CK_RA6M5_X509 >Telemetry)**. Confirm that **use built-in event hub** is set to **Yes**.
2. Select **Start**.
3. View the telemetry as the device sends messages to the Cloud.

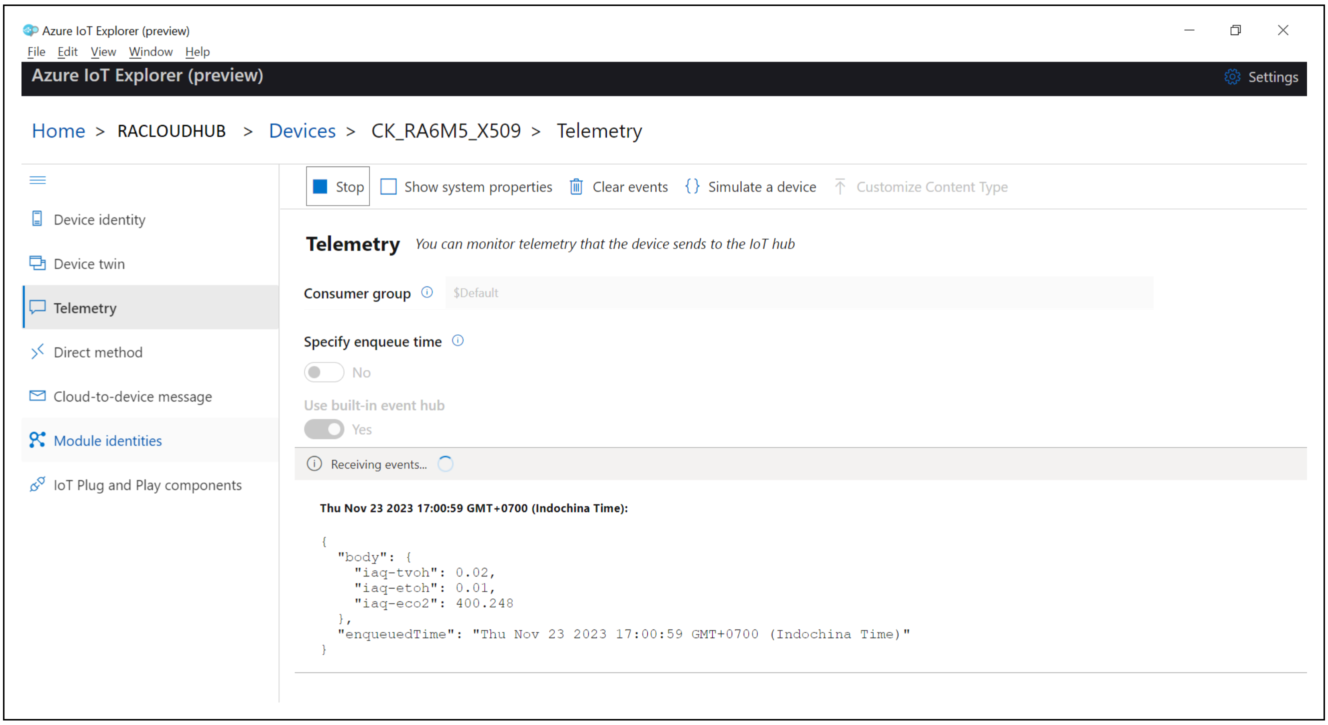

Figure 55. Device Telemetry Details

3.14 Send Cloud-to-Device Message

To send a Cloud-to-device message in Azure IoT Explorer:

1. In IoT Explorer, select **Cloud-to-device message**.
2. Enter the message in the **Message body** = "**LED**", **Key** = **LED**, **Value** = **Given in Table**
3. **Check** Add timestamp to message body.
4. Select **Send message to device**.

LED On Board	Value
LED2 (Tri Color LED)	TC_GREEN_ON, TC_RED_ON, TC_BLUE_ON TC_GREEN_OFF, TC_RED_OFF, TC_BLUE_OFF
LED4 BLUE	BLUE_ON, BLUE_OFF

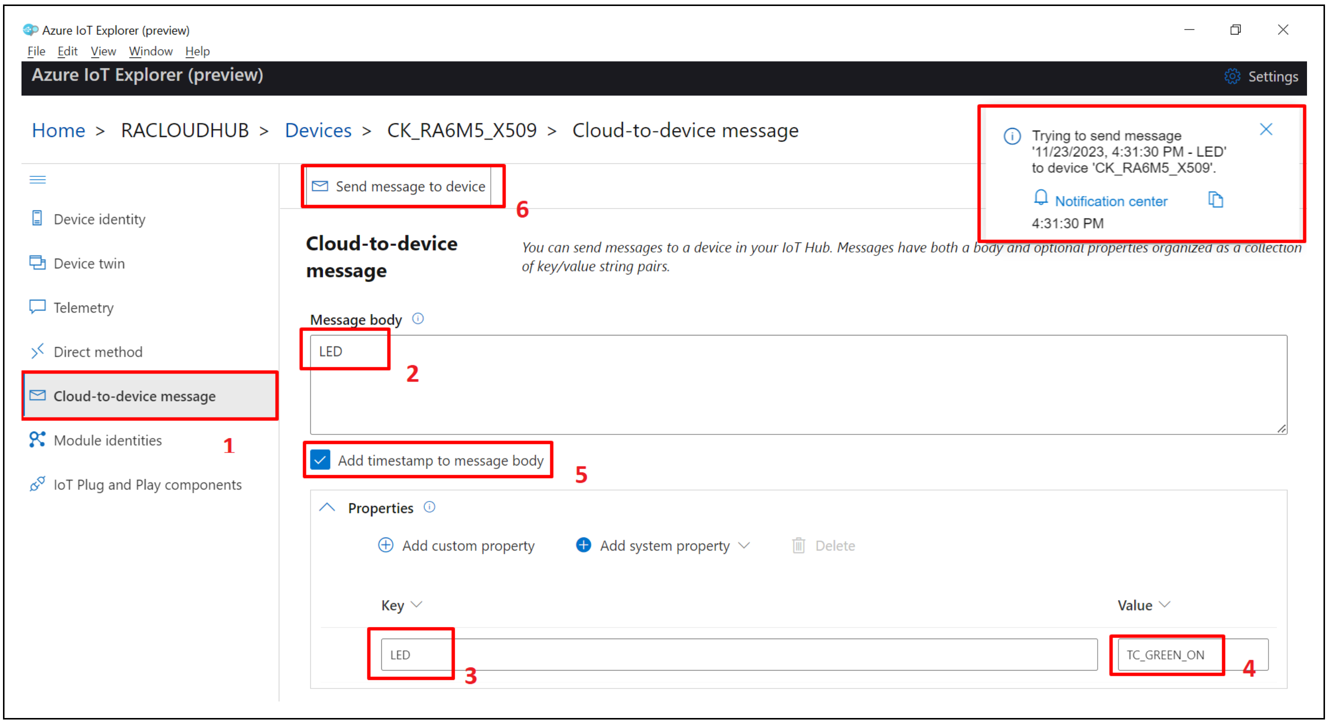

Figure 56. Device Telemetry Details

5. In the terminal window, you can see that the message is received by the IoT Device.

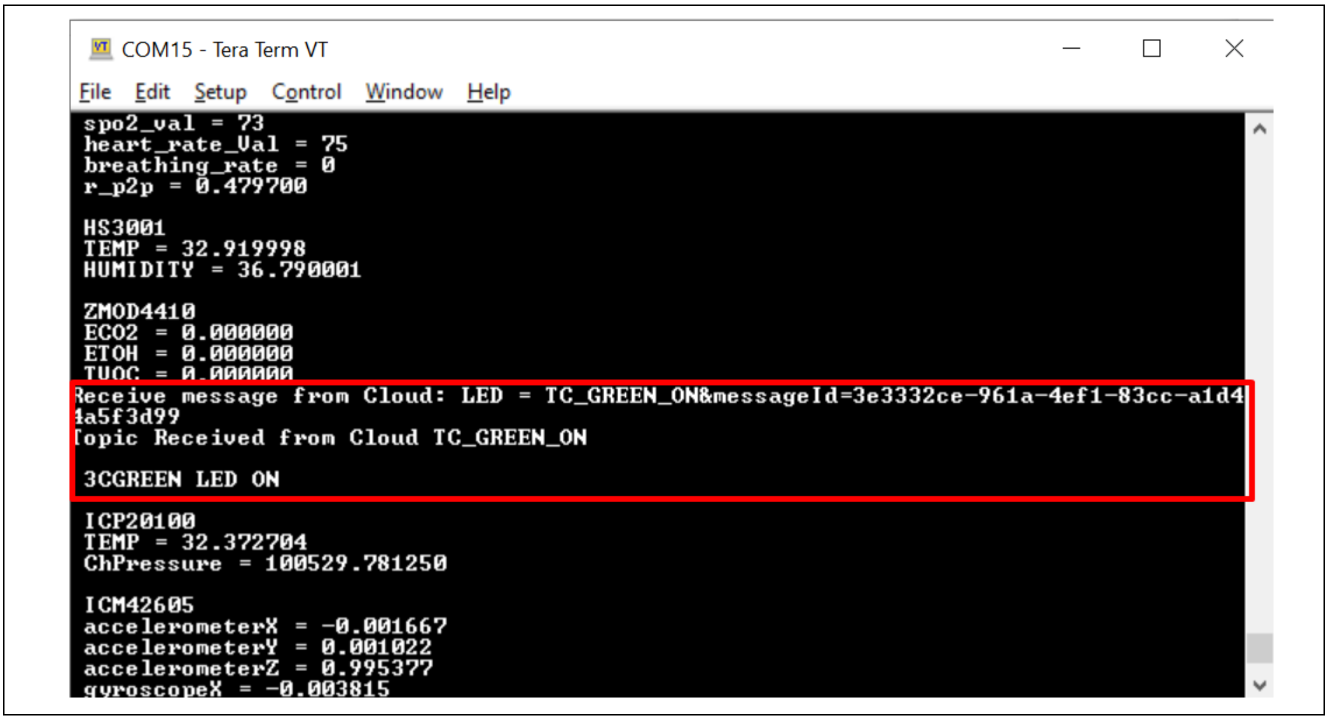

Figure 57. Serial Terminal Output

4. Importing, Building and Loading the Project

For a quick validation of this application project, import and build the project. The following steps show how to import, build, and download the project.

Note: To run the application project successfully and to communicate to the Cloud, follow the instructions for setting up the Cloud interface as described in section 3.3, which details making changes to the credentials and creating your own cloud devices, running and validating the application.

4.1 Importing

The application project bundled as part of this app note can be imported into e^2 studio using instructions provided in the *RA FSP User's Manual*. See Section *Starting Development > e^2 studio ISDE User Guide > Importing an Existing Project into e^2 studio ISDE.*

4.2 Building the Latest Executable Binary

Upon successfully importing and/or modifying the project into e^2 studio IDE, follow instructions provided in the *RA FSP User's Manual* to build an executable `binary/hex/mot/elf` file. See Section *Starting Development > e^2 studio ISDE User Guide > Tutorial: Your First RA MCU Project > Build the Blinky Project.*

4.3 Loading the Executable Binary into the Target MCU

The executable file may be programmed into the target MCU through any one of three means.

4.3.1 Using a Debugging Interface with e^2 studio

Instructions on how to program the executable binary are found in the latest *RA FSP User Manual* (https://www.renesas.com/us/en/software-tool/flexible-software-package-fsp). See section*Starting Development > e^2 studio ISDE User Guide > Tutorial: Your First RA MCU Project > Debug theBlinky Project.*

This is the preferred method for programming as it allows for additional debugging functionality available through the on-chip debugger.

4.3.2 Using J-Link Tools

SEGGER J-Link Tools such as J-Flash, J-Flash Lite, and J-Link Commander can be used to program the executable binary into the target MCU. Refer to User Manuals UM08001 and UM08003 on www.segger.com.

4.3.3 Using Renesas Flash Programmer

The Renesas Flash Programmer (https://www.renesas.com/us/en/software-tool/renesas-flash-programmer-programming-gui) provides usable and functional support for programming the on-chip flash memory of Renesas microcontrollers in each phase of development and mass production. The software supports all RA MCUs, and the software user's manual is available on renesas.com.

5. Next Steps and References

- Refer to the following GitHub repository for various FSP modules example projects and application projects (https://github.com/renesas/ra-fsp-examples/)
- Refer to *Establishing and Protecting Device Identity using SCE7 and Security MPU* (R11AN0449) on renesas.com
- Refer to *Securing Data at Rest Utilizing the RA Security MPU* (R11AN0416) on renesas.com
- Refer to the Azure GitHub link for more details on Azure SDK for Embedded C (https://github.com/Azure/azure-sdk-for-c)

6. MQTT/TLS References

- FSP v5.3.0 User's Manual (Flexible Software Package (FSP) | Renesas).
- Azure IoT documentation (https://docs.microsoft.com/en-us/azure/iot-hub/)

7. Known Issues and Limitations

1. Occasional outages in Cloud connectivity may be noticed during the demonstration due to changes in the Cloud server. Contact the Renesas support team for questions.
2. Currently, there is no support for direct device-to-device communications with Azure IoT Hub.
3. Device will reconnect after 65 minutes due to the SAS token refresh. Currently, it is under SDK control. Usersneed to know this when developing the application.
4. When running debug on e^2 studio, if the application is rerun multiple times, it might randomly occur issue with i2c communication of the OB1203 sensor. Users need to reconnect the micro-USB cable (J10) and USB-C cable (J28) to reset the OB1203 sensor and run the application again.

Website and Support

Visit the following vanity URLs to learn about key elements of the RA family, download components and related documentation, and get support.

CK-RA6M5 v2 Kit Information	renesas.com/ra/ck-ra6m5
RA Cloud Solutions	renesas.com/cloudsolutions
RA Product Information	renesas.com/ra
RA Product Support Forum	renesas.com/ra/forum
RA Flexible Software Package	renesas.com/FSP
Renesas Support	renesas.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Mar.22.23	—	Initial release
1.01	May.05.23		Corrected the document number in the document footer
1.10	Dec.22.23		Updated to FSP 5.0.0
1.20	Sept.09.24		Updated to FSP 5.3.0

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit: www.renesas.com/contact/.